Secure Compilation: Software Fault Isolation and Information Flow Preservation

F. Besson, S. Blazy, **A. Dang**, T. Jensen, P. Wilke

Celtique/Inria/Univ Rennes

Gdr MfSec, Mars 2021

Are Compilers Trustworthy?

What is the expected guarantee?

Semantic preservation

If $beh(S) \neq \emptyset$ Then $beh(T) \subseteq beh(S)$.

1. If source is deterministic, target has same behaviour.
2. If source has undefined behaviour, all bets are off.

Beware: aggressive optimisations exploit undefined behaviours[1].

Formal verification: CompCert, Vellum, CakeML

[1] Undefined behavior: what happened to my code?, Wang et al. [2012]

Functional Correctness of Target Code

Hyp1 : My compiler has no bug (e.g., LLVM)

Hyp2 : My program has no UB (e.g., Linux kernel)

Functional properties are preserved.

$\Rightarrow$ I can reason at source level!

Security Properties of Target Code?

Compilers may enhance security

- Shadow stack
- Canaries
- Security instrumentation

Compilers may also break security counter-measures[1]

- Introduction of `jump` breaks CT-programming
- Associativity of `xor` breaks *masking*
- CSE breaks Fault-Injection protection
- (Dead) code removal breaks *CFI*;breaks *safe erasure*

$\Rightarrow$ Security people do not trust compilers.

[1] *The Correctness-Security Gap in Compiler Optimization*, D'Silva et al. [2015]

Goal: a Security Enhancing Compiler

> *A secure compiler inserts security counter-measures (in the source) and preserves them (in the assembly).*

Attackers get a **disadvantage** at attacking the target.

Research Agenda

- Define classes of properties/attackers.
- Revisit/patch existing compiler passes.

Today

Security Enhancement: Software Fault Isolation[1]
Property: Integrity of a host running untrusted code

Security Preservation: Information Flow Preservation[2]
Property: Preservation of lifetime of secrets (secure erasure)

[1] Compiling Sandboxes: Formally Verified Software Fault Isolation, ESOP 2019
[2] Information-Flow Preservation in Compiler Optimisations, CSF 2019

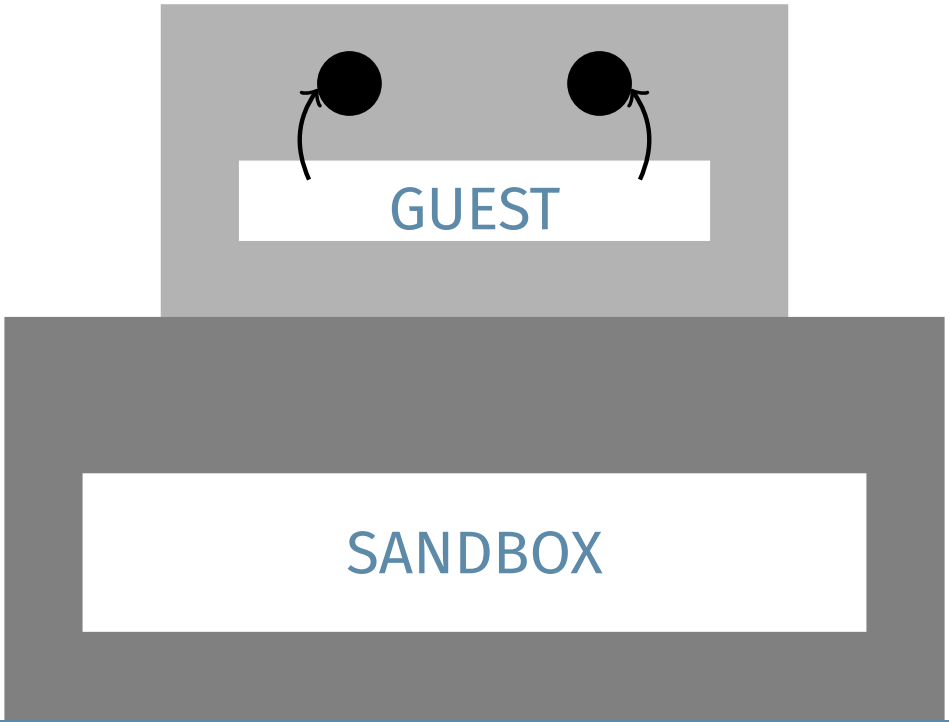
Software Fault Isolation

Software Fault Isolation (SFI)

A trusted host wishes to run untrusted guest plugins

Full speed & Full security

- Speed: native code, same address space
- Security: strong isolation
  - Code: calls limited to host API
  - Data: memory accesses limited to sandbox



Traditional SFI [Wahbe et al., SOSP'93]

"Run safely binary code of untrusted origin"

A modified Compiler ($\notin$ TCB) masks memory accesses.

Binary verifier ($\in$ TCB) checks masking is correct.

Ex: (P)NaCl (Google Chrome) [S&P'09, USENIX Sec'10, CACM'10]
A compiler:

$$C \rightarrow \boxed{\text{CLANG}} \rightarrow \mathit{LLVM} \rightarrow \boxed{\text{SFI}} \rightarrow \mathit{binary}$$

A **verified** verifier [RockSalt, PLDI'12]

$$\mathit{verifier} : \mathit{binary} \rightarrow \mathbb{B}$$

Portable Software Fault Isolation [Kroll *et al.*, CSF'14]

$$C \to \boxed{\text{Frontend}} \to Cm \to \boxed{\text{SFI}} \to Cm' \to \boxed{\text{Backend}} \to Asm$$

Property (SFI Security)

*A program P is **SFI-secure** if all its memory accesses are within the sandbox memory region.*

Property (Safety)

*A program P is **safe** if all its behaviours are **defined** i.e. not stuck*

Transfer of security from Cm' down-to Asm.

Let P be a program that is both **SFI-secure** and **safe**.
Let $B \in behave(ccomp(P))$.
By semantic preservation, $B \in behave(P)$ (P is **safe**)
B is a secure behaviour (P is **SFI-secure**). □

Our work: Fully Verified SFI within CompCert

Machined-checked proof of **SFI-security** and **Safety**

- Security: see [Kroll *et al.*]
- **Safety**: Re-design of the SFI transformation

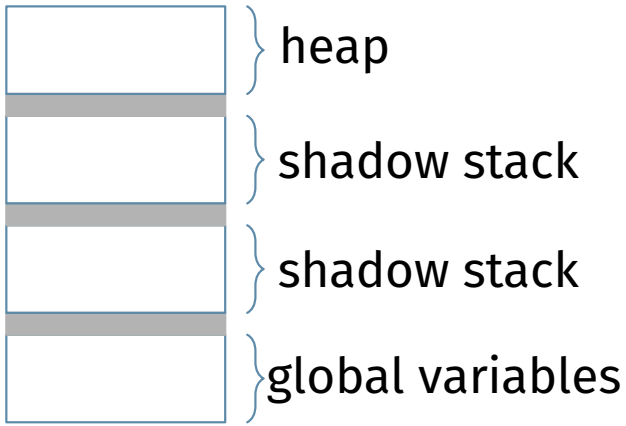
Reduced TCB (no axiom for)

- Sandboxing memory accesses
  Low-level pointer arithmetic
- Control-flow integrity
  Trampoline indirect function calls

Other features

- Support for multi-threading
- Trusted Runtime

Sandbox Relocation



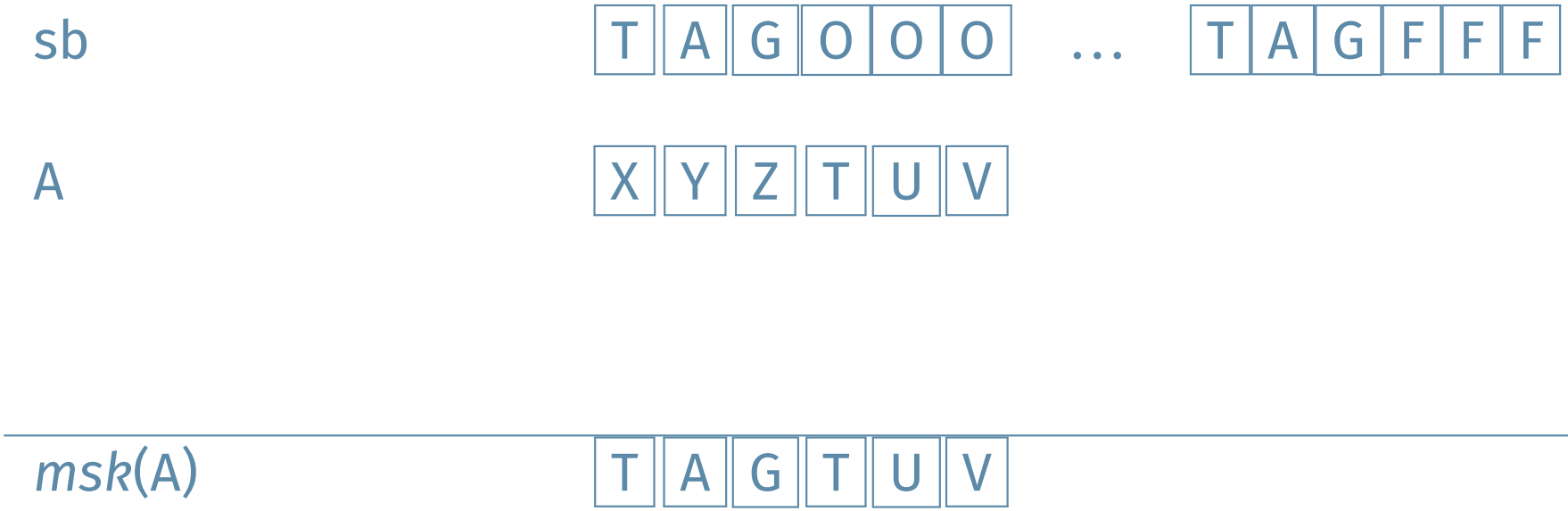

```
g = 5;

long foo(bar: int -> int -> unit){
  stk[8];
  *bar(g, &stk);
  return(*(&stk));
}
```

```
  sb[2^k]= {5;...};

  long foo(sp:int,
           bar:int -> int -> unit){
    sp1=sp + 8 ;
    *bar(sp1,*(&sb),sp);
    return(*sp);
  }
```

Masking of Memory Accesses



Masking of Memory Accesses

sb	T	A	G	O	O	O	...	T	A	G	F	F	F
A	X	Y	Z	T	U	V							
A & OxFFF	O	O	O	T	U	V							
msk(A)	T	A	G	T	U	V							

Masking of Memory Accesses

sb	T	A	G	O	O	O	...	T	A	G	F	F	F
A	X	Y	Z	T	U	V						
A & 0xFFF	O	O	O	T	U	V						
(A & 0xFFF)|TAGOOO	T	A	G	T	U	V						
msk(A)	T	A	G	T	U	V						

Masking of Memory Accesses

sb	T	A	G	O	O	O	...	T	A	G	F	F	F
A	X	Y	Z	T	U	V						
A & OxFFF	O	O	O	T	U	V						
(A & OxFFF)|TAGOOO	T	A	G	T	U	V						
msk(A)	T	A	G	T	U	V						

$$msk(A) = (A\&(2^k - 1))|\&sb$$

Masking of Memory Accesses

sb	T	A	G	O	O	O	…	T	A	G	F	F	F
A	X	Y	Z	T	U	V						
A & OxFFF	O	O	O	T	U	V						
(A & OxFFF)|TAGOOO	T	A	G	T	U	V						
msk(A)	T	A	G	T	U	V						

$$msk(A) = (A\&(2^k - 1))|\&sb$$

Masking pointer arithmetic has no C semantics

Security Vacuously true

Safety Vacuously false

Our solution: Pointer-free Transformation

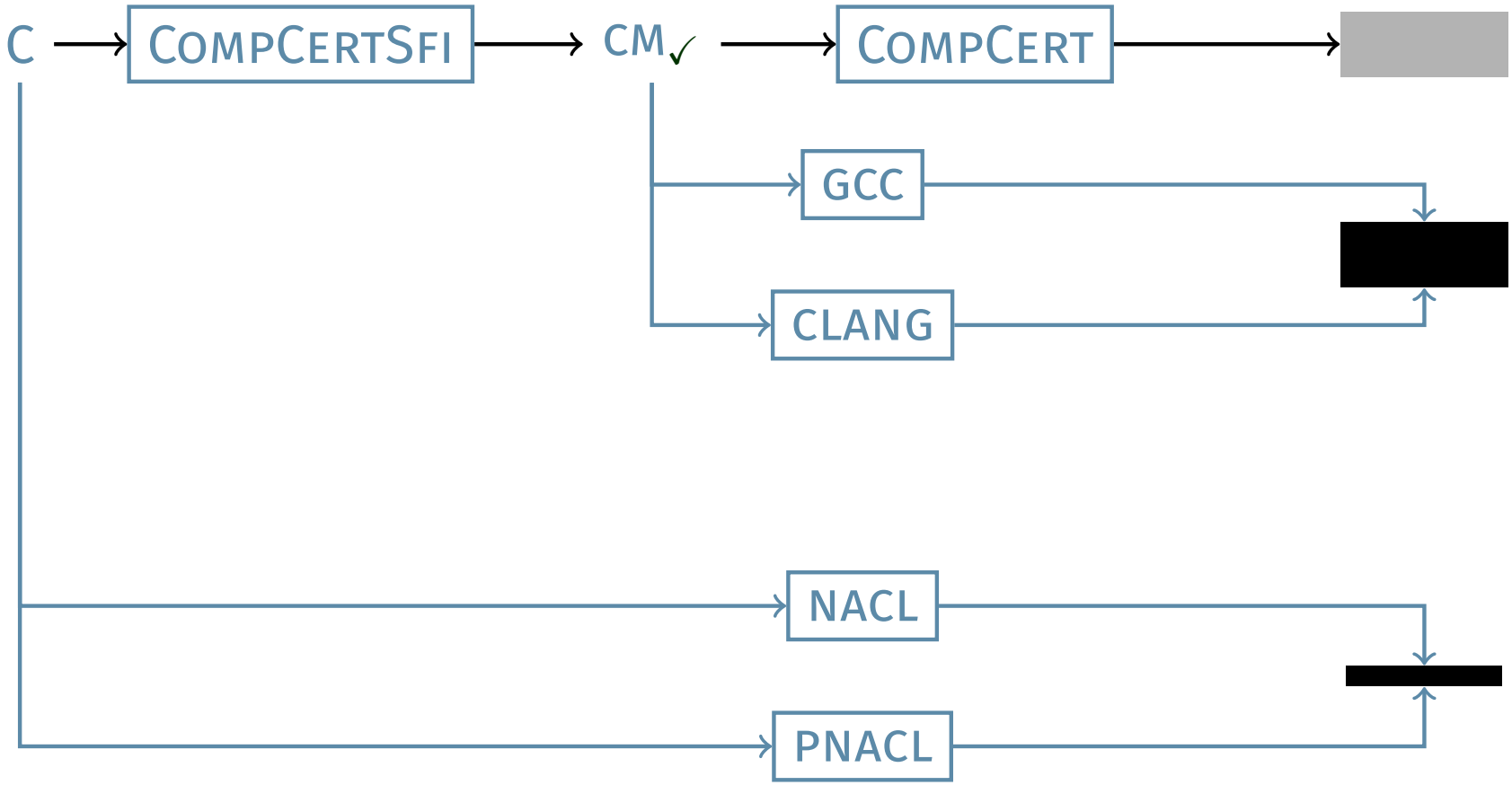
Pointers are compiled into their numeric value

$$
\begin{aligned}
sfi(\&sb) &= tag \times 2^k \\
&\dots \\
sfi(*(e)) &= *(sfi(e)\&msk + \&sb)
\end{aligned}
$$

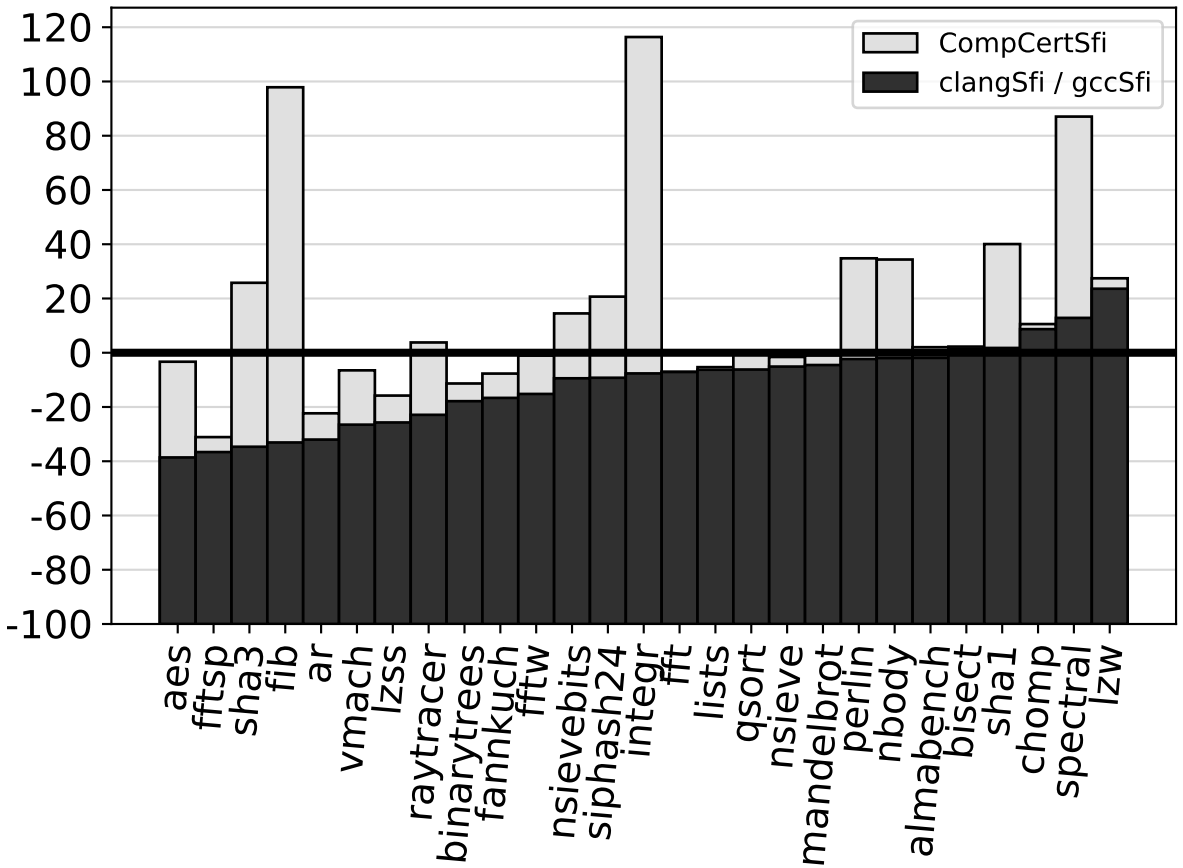
```
sb[2^k]= {5;...};

long foo(sp:int,bar:int -> int -> unit){
  sp1=sp + 8 ;
  *bar(sp1,*(&sb),sp);
  return(*(sp&msk + &sb));
}
```

Experiments with CompCert Benchmarks



Comparison with (P)NaCl



- gccSfi/clangSfi very competitive
- CompCertSfi average overhead 9% (removing outliers)

$\Rightarrow$ Optimisations improve SFI

Conclusion 1

Our source SFI pass deviates existing binary instrumentations:

- Masking without bitwise pointer arithmetics
- Source level control-flow integrity

Compilers can be used for **Security**

Price Guarantee only holds for **safe** programs

Limitation Compilers only preserve **observable** behaviours

But, security is not always reducible to safety.

Information Flow Preservation

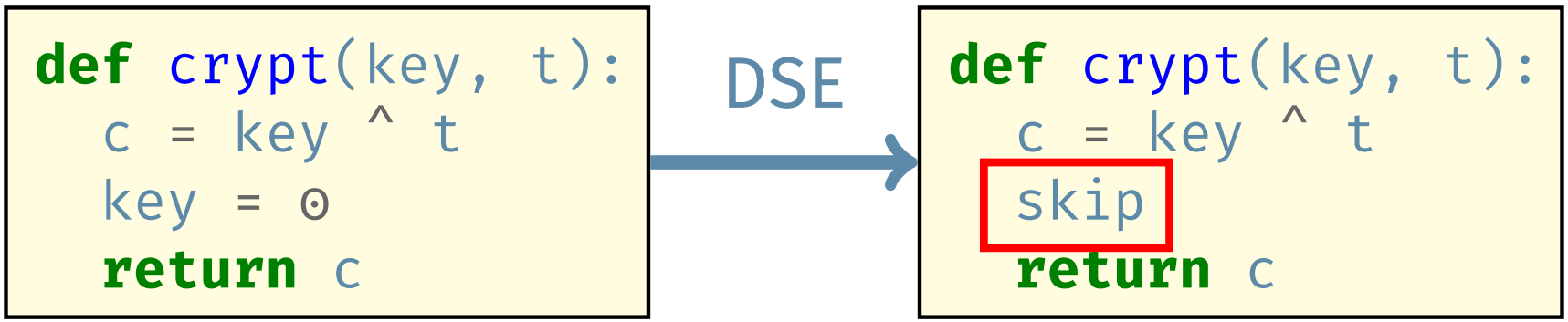
Information-Flow Preservation

Our Information-Flow Preservation property aims at protecting against:

- **Data remanence**
- **Lifetime extension**
- **Increased information leakage**
- **Duplication of information**

Sec. Req. 1.: Erase Sensitive Data

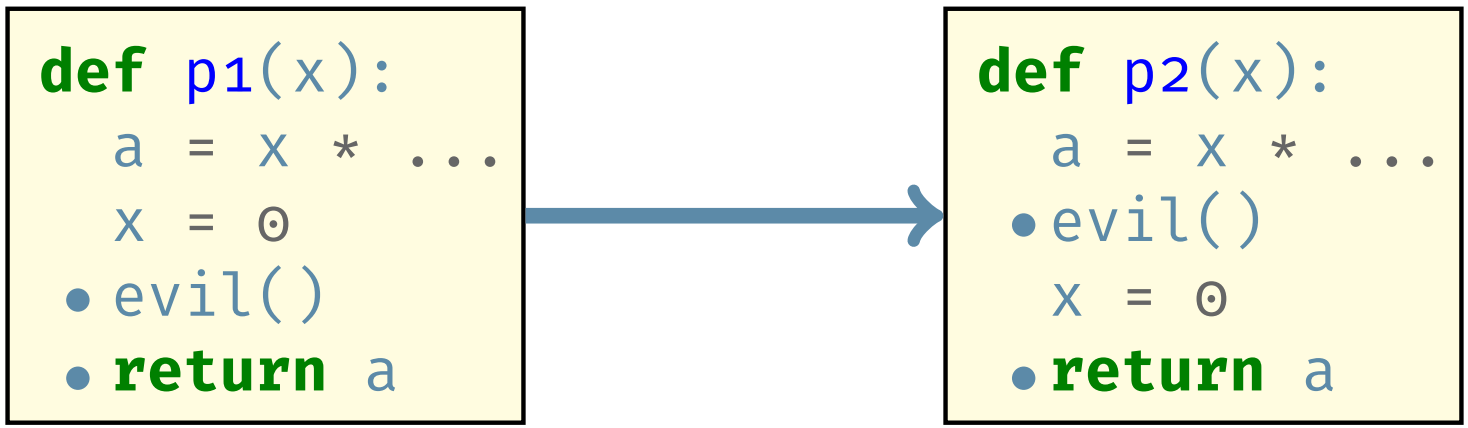
Dead Store Elimination (DSE) is not secure[1]



[1] *Dead Store Elimination (Still) Considered Harmful*, Yang et al. [2017]

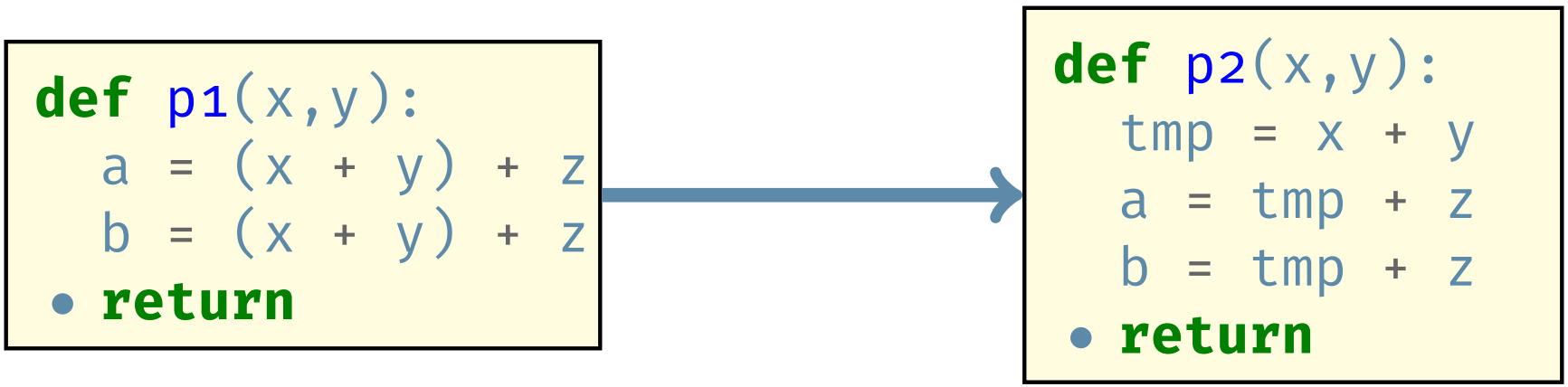
Sec. Req. 2: Reduce the Lifetime of Sensitive Data

Code motion is not secure.



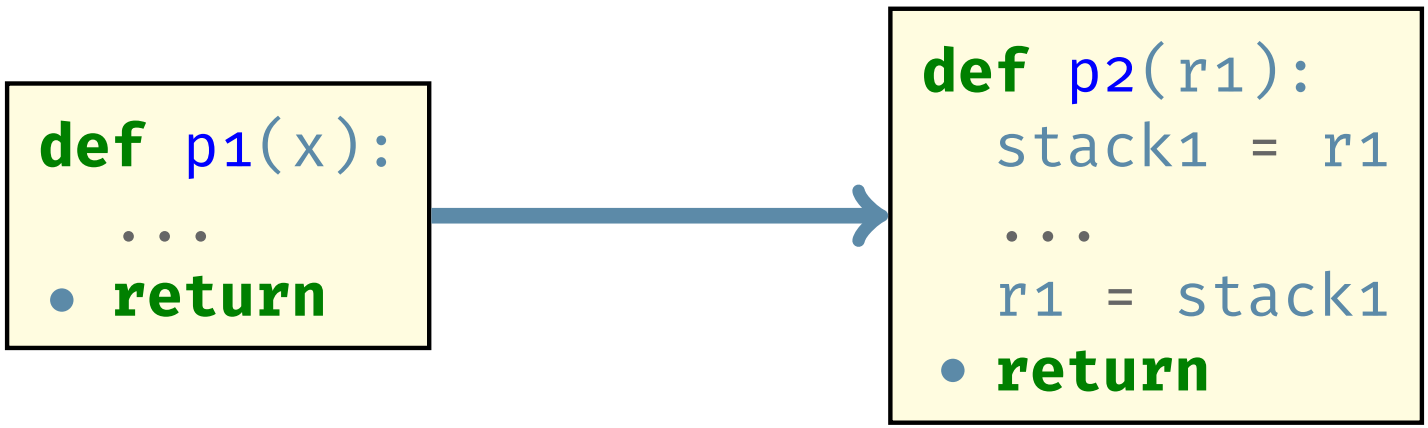
Sec. Req. 3: Limit Leakage of Information

Common Expression Elimination is not secure.



Sec. Req. 4: Do not Duplicate Sensitive Data

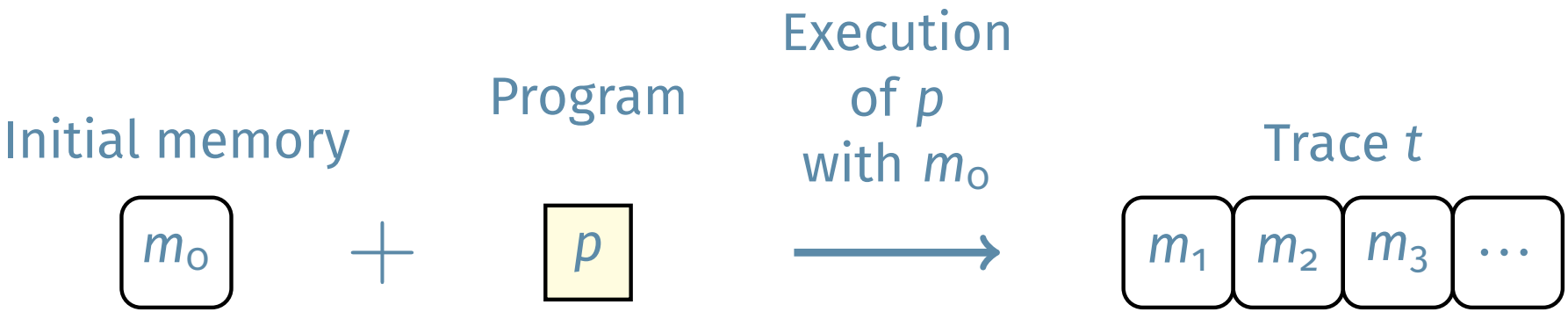
Register Allocation is not secure.



Formal Definition of IFP

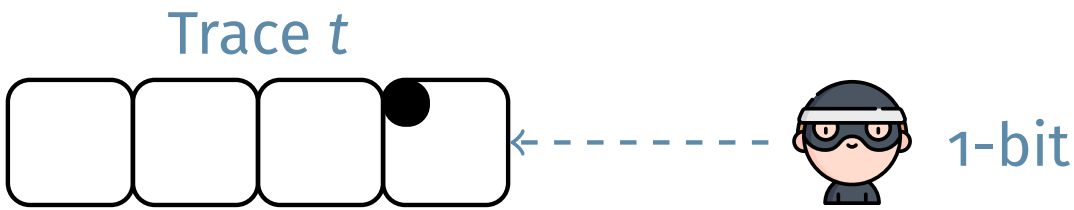
Execution model

- Trace based execution model
- Memory states: data observable by attackers



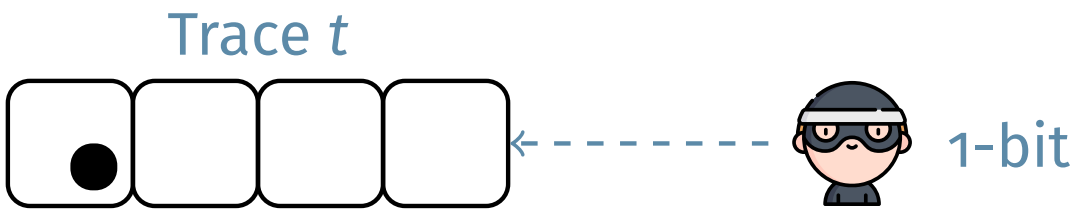
Attacker model

- Attackers know the code
- Attackers observe n bits in the trace



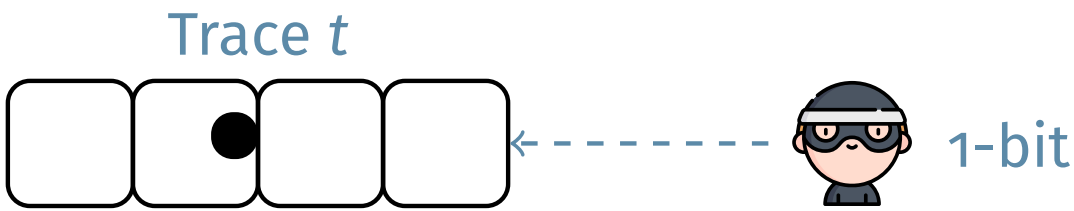
Attacker model

- Attackers know the code
- Attackers observe n bits in the trace



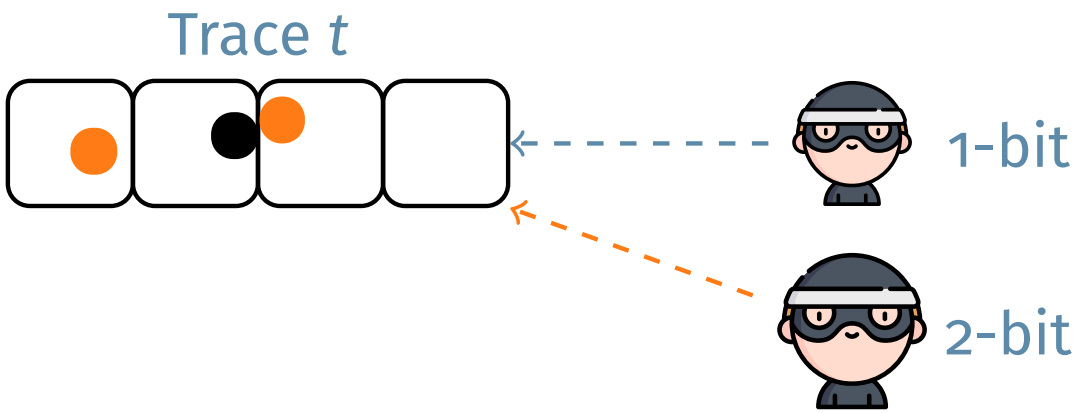
Attacker model

- Attackers know the code
- Attackers observe n bits in the trace



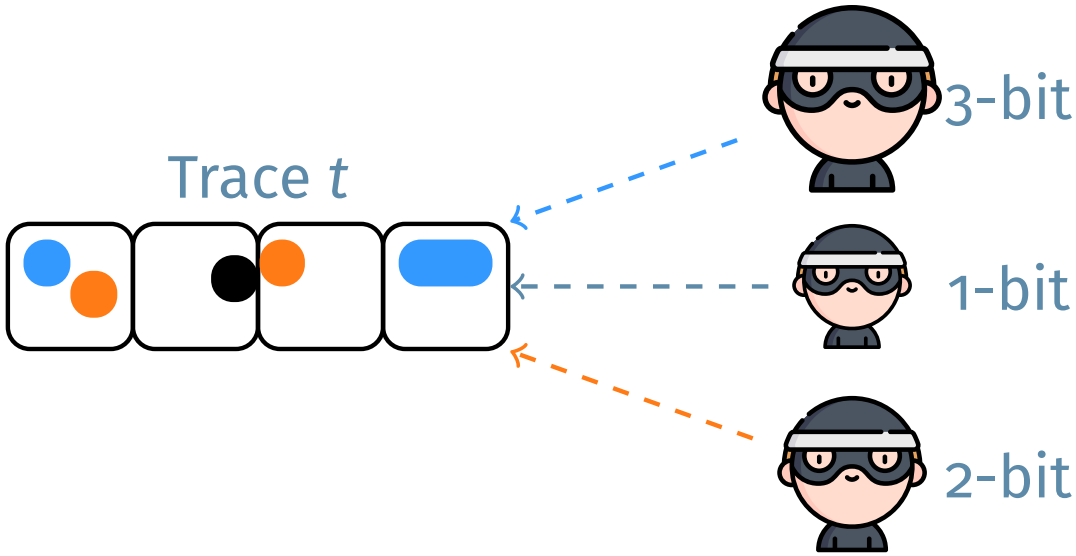
Attacker model

- Attackers know the code
- Attackers observe n bits in the trace



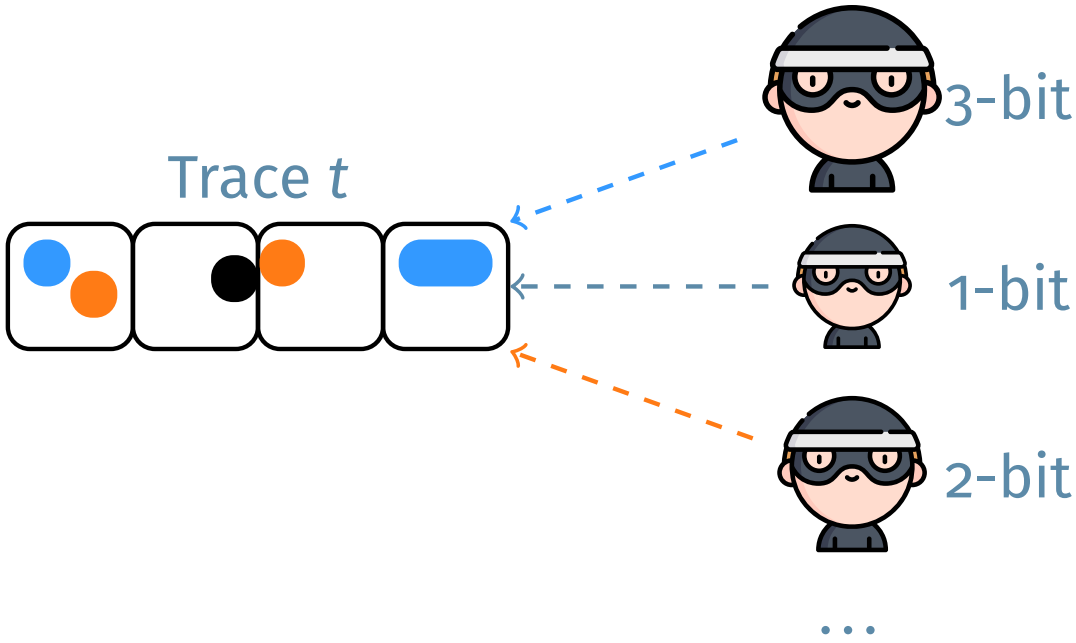
Attacker model

- Attackers know the code
- Attackers observe n bits in the trace



Attacker model

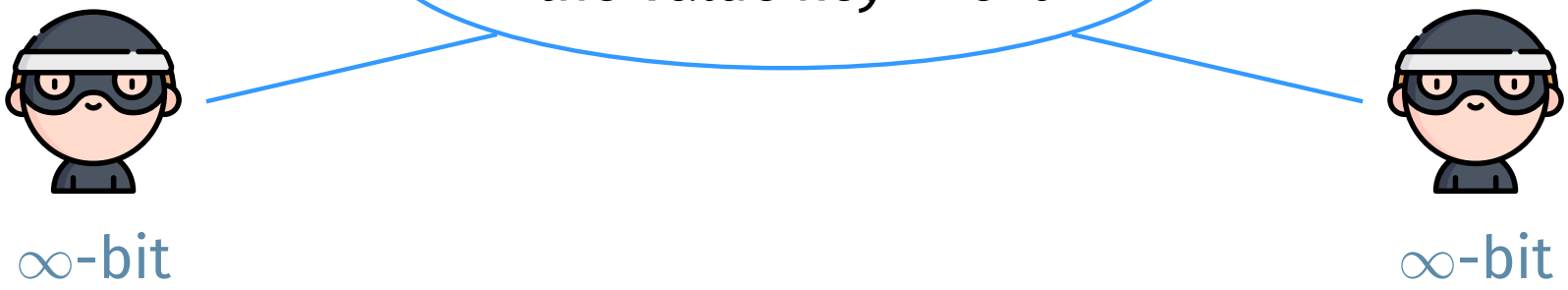
- Attackers know the code
- Attackers observe n bits in the trace



Rationale for hierarchy of attackers

```
def crypt(key, t):
  c = key ^ t
  key = 0
 •return c
```

```
def crypt(key, t):
  c = key ^ t
  skip
 • return c
```

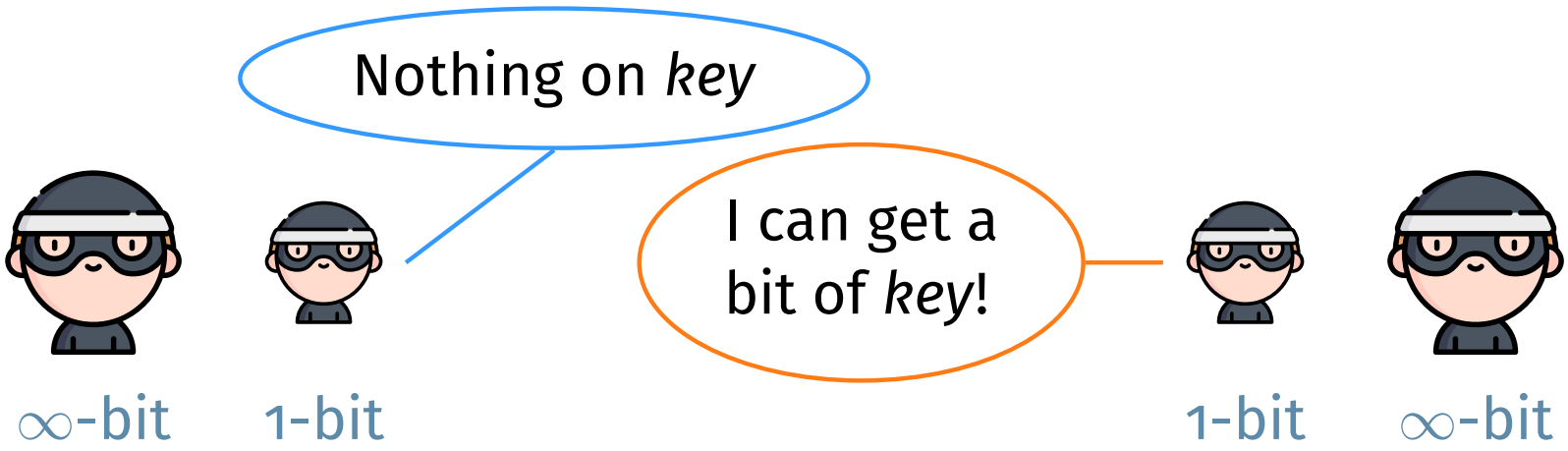


- equally insecure for a strong attacker

Rationale for hierarchy of attackers

```
def crypt(key, t):
  c = key ^ t
  key = 0
 •return c
```

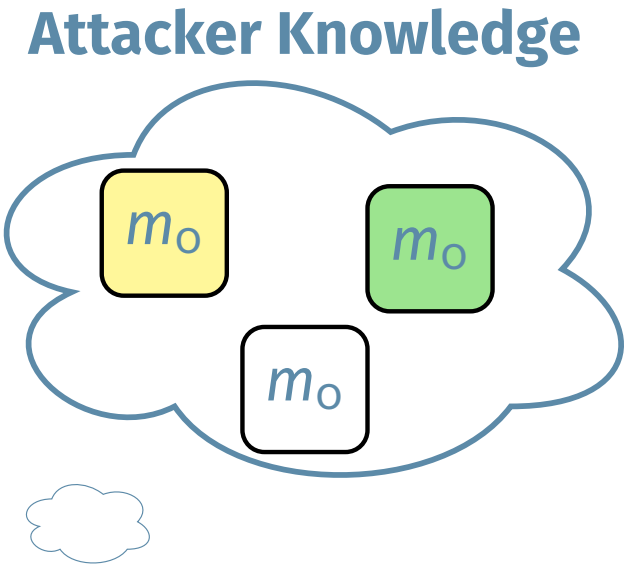
```
def crypt(key, t):
  c = key ^ t
  skip
 • return c
```

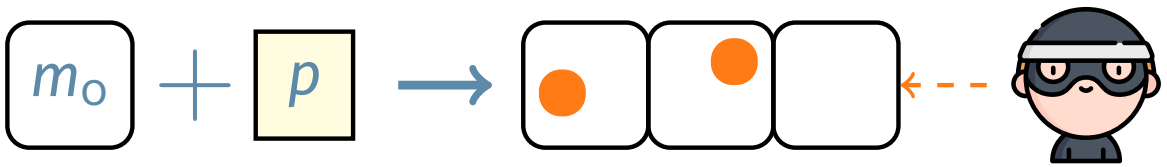


- equally insecure for a strong attacker
- *p*1 is secure for the 1-bit attacker

Attacker Knowledge [1]

- Attackers try to guess the initial memory used
- Possible initial memories matching its observations



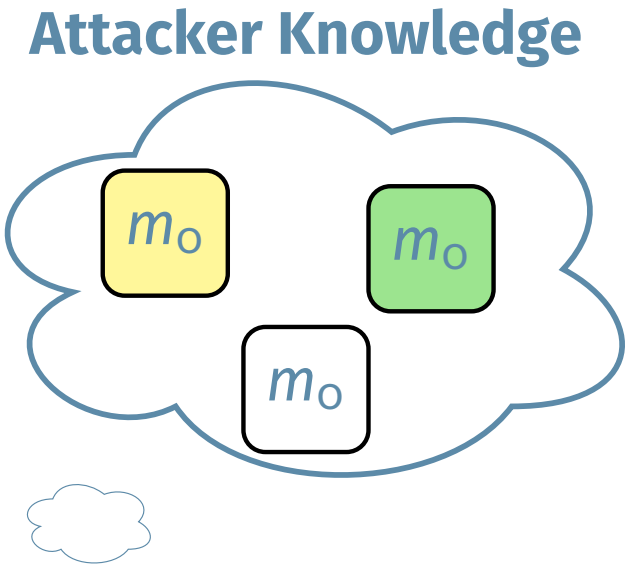


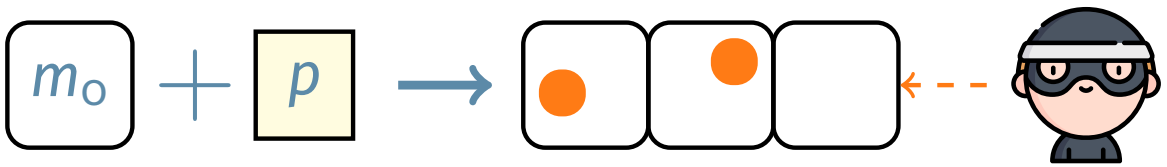
[1] *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*, Askarov and Sabelfeld [2007]

Attacker Knowledge [1]

- Attackers try to guess the initial memory used
- Possible initial memories matching its observations

Remark:
Big/coarse attacker knowledge means that there is few information on m_0



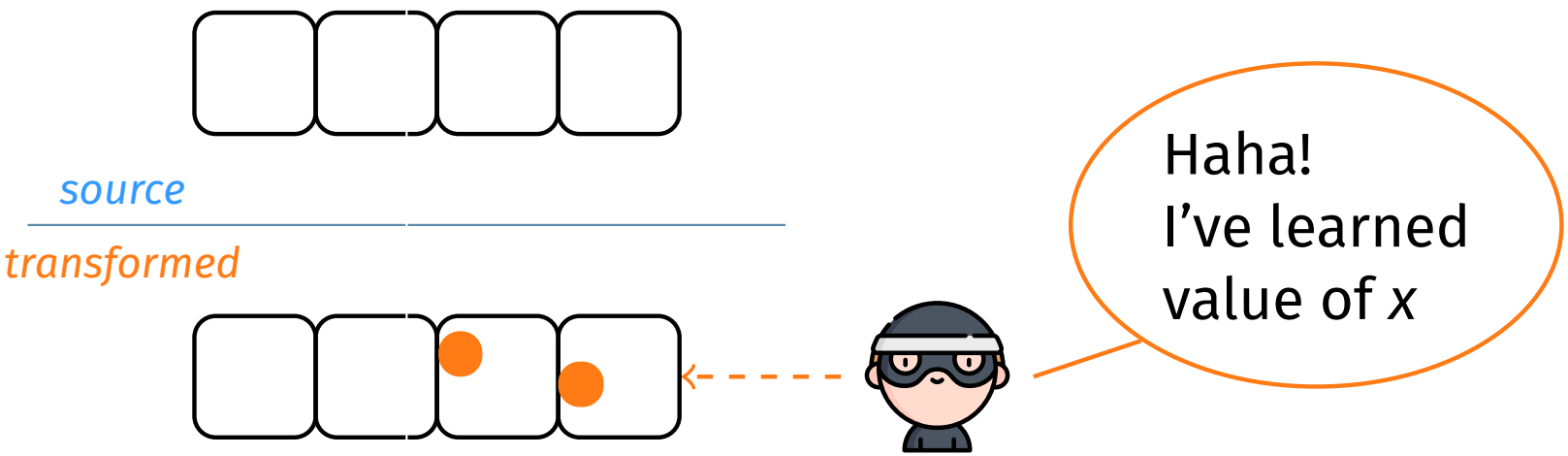


[1] *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*, Askarov and Sabelfeld [2007]

IFP transformation (1/2)

Intuition

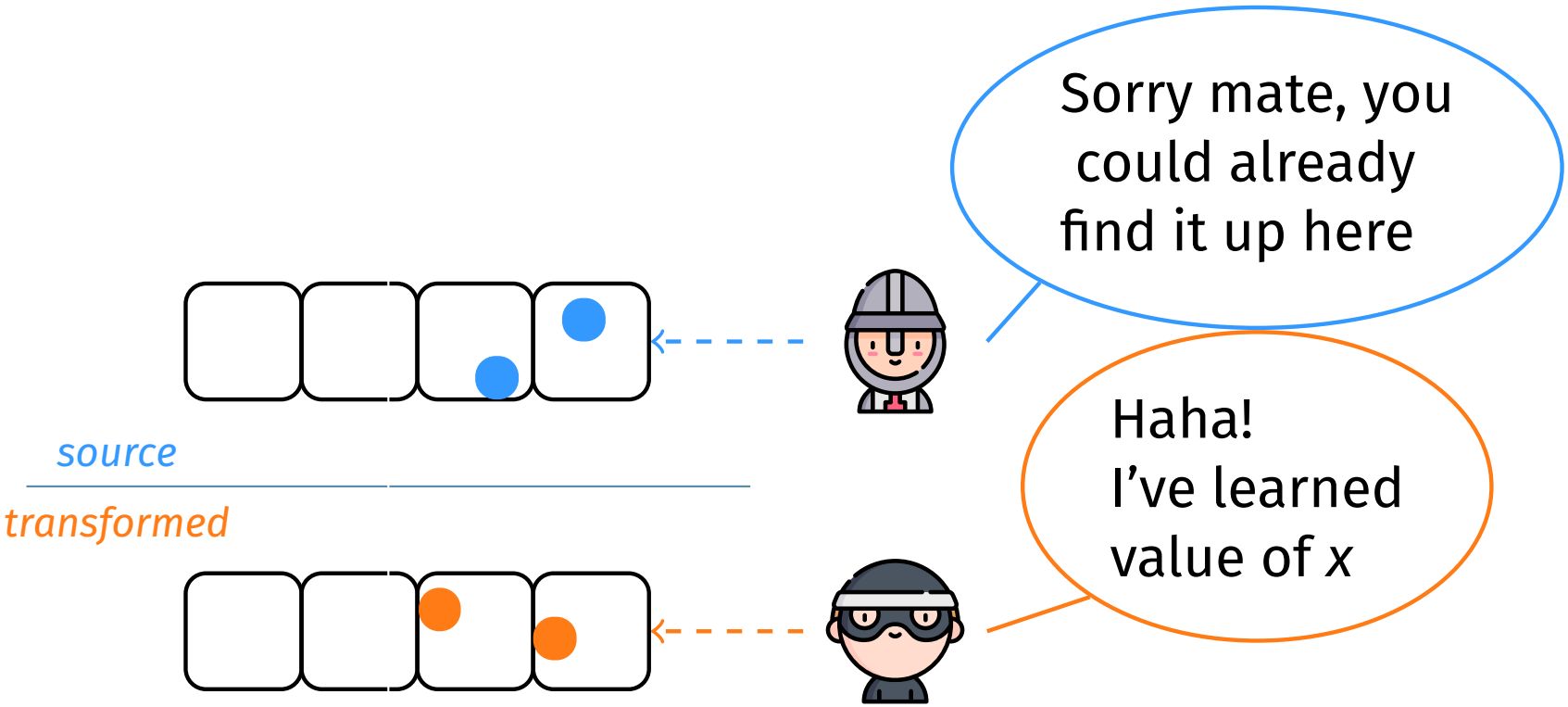
Any information that can be learned with a trace observation of the transformed program can also be learned with the source program



IFP transformation (1/2)

Intuition

Any information that can be learned with a trace observation of the transformed program can also be learned with the source program



IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$$\forall (m_0, t_1, t_2).\ \forall n.\ \exists \omega \in \Omega(t_1, t_2).\ \forall o_2. \quad \mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$$

IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$$\forall (m_0, t_1, t_2).\ \forall n.\ \exists \omega \in \Omega(t_1, t_2).\ \forall o_2. \quad \mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$$

Source program p_1
Transformed program p_2

p_1

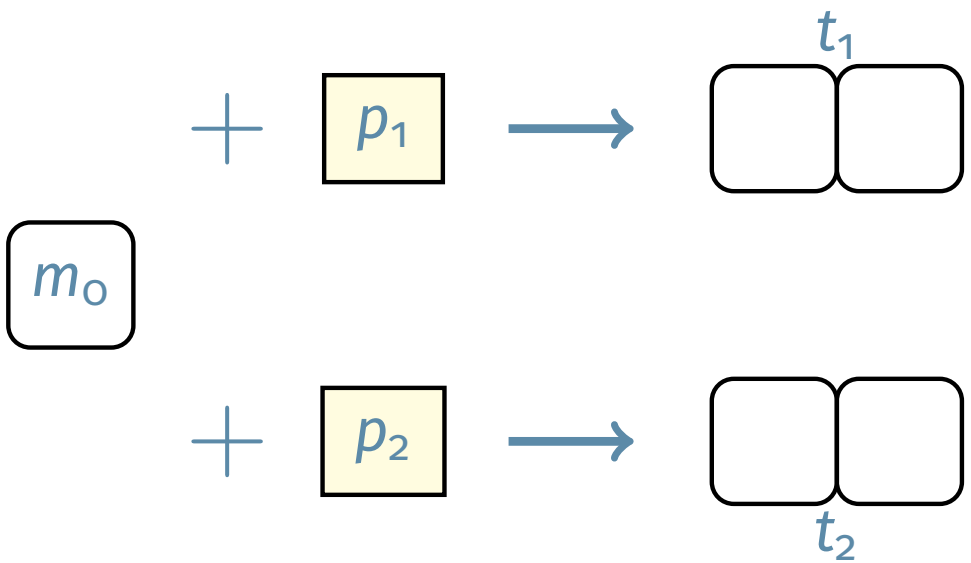
p_2

IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$\forall(m_0, t_1, t_2).\ \forall n.\ \exists \omega \in \Omega(t_1, t_2).\ \forall o_2.\quad \mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$

For any execution from the same initial memory m_0

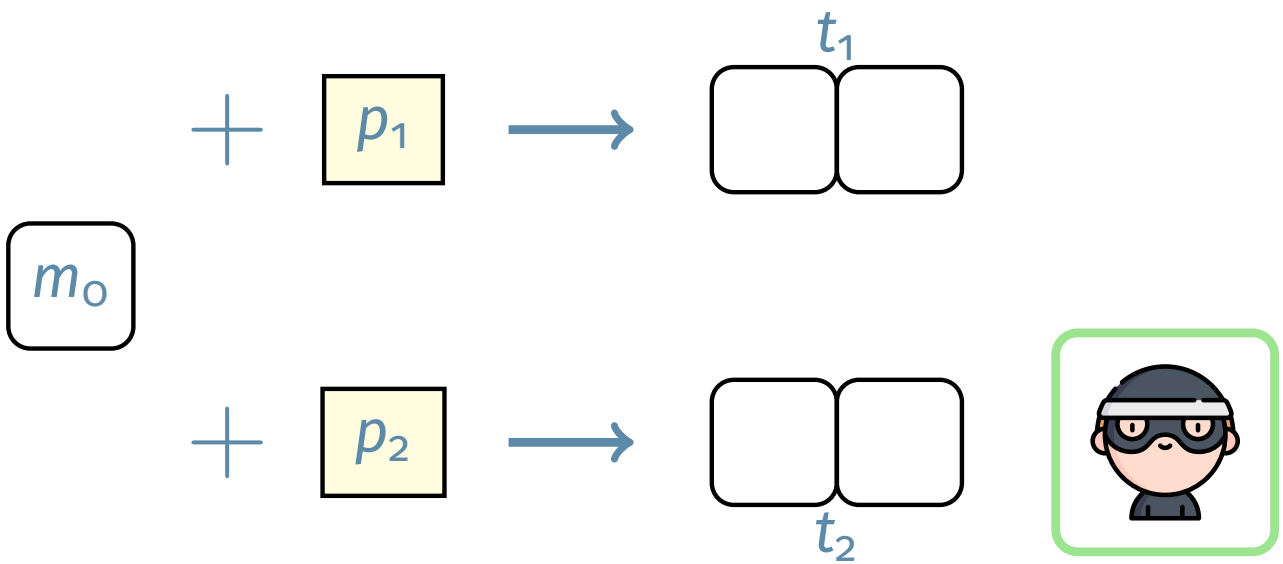


IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$$\forall (m_0, t_1, t_2).\ \forall n.\ \exists \omega \in \Omega(t_1, t_2).\ \forall o_2.\quad \mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$$

For attackers with any observation capabilities

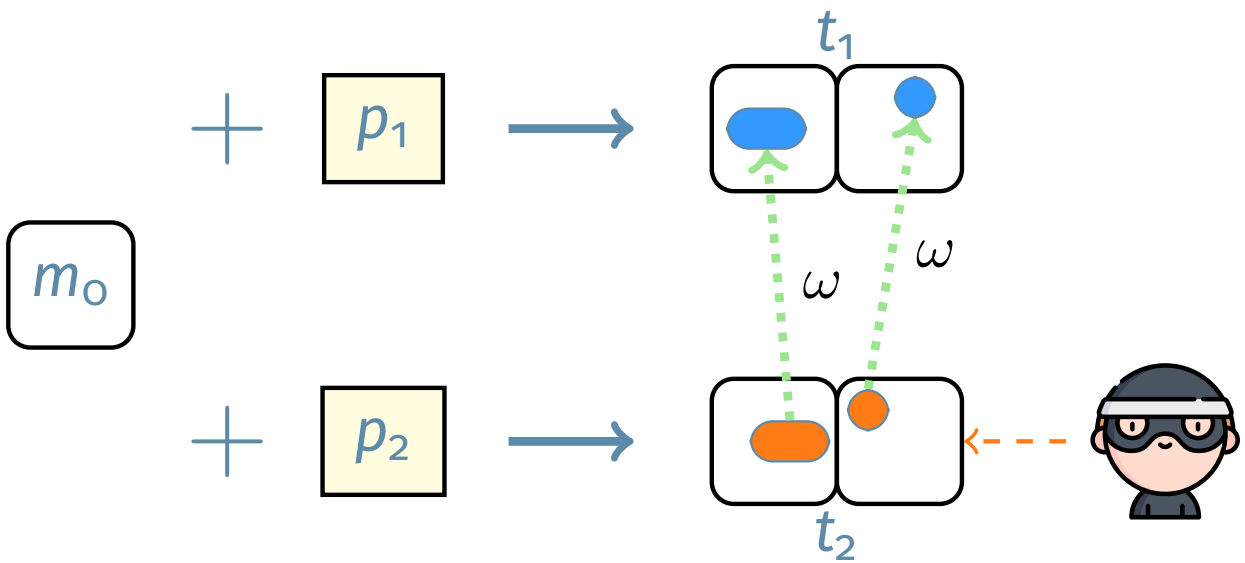


IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$$\forall (m_0, t_1, t_2).\ \forall n.\ \exists \omega \in \Omega(t_1, t_2).\ \forall o_2.\quad \mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$$

Exists lockstep pairings of observations from t_2 to t_1

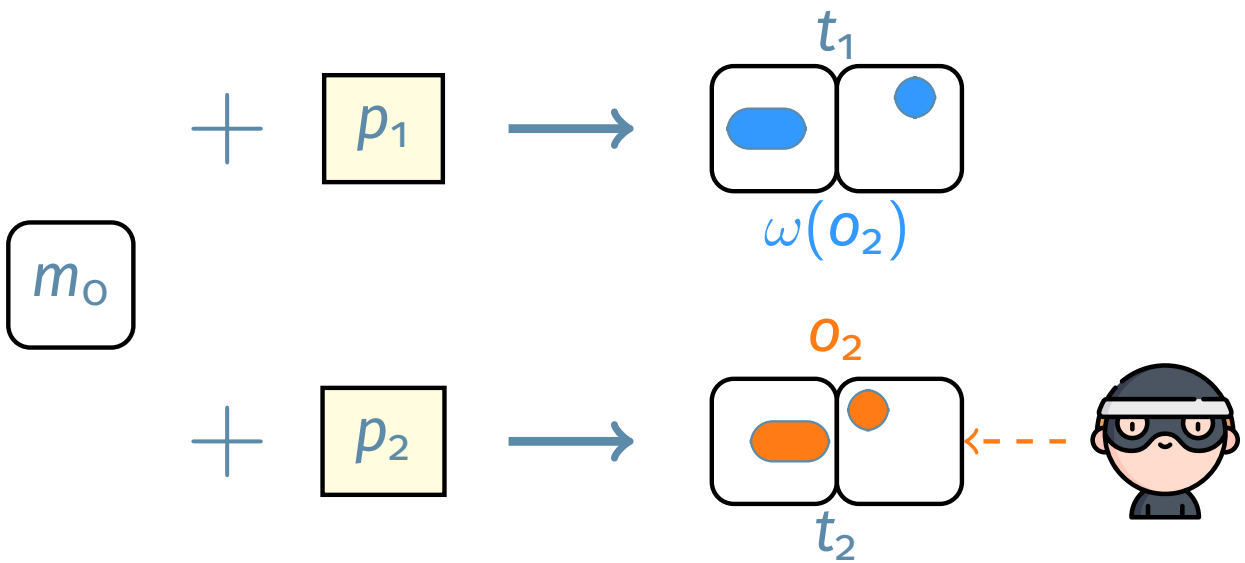


IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$$\forall (m_0, t_1, t_2).\ \forall n.\ \exists \omega \in \Omega(t_1, t_2).\ \forall o_2.\quad \mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$$

For any observation o_2 of size n on the trace t_2

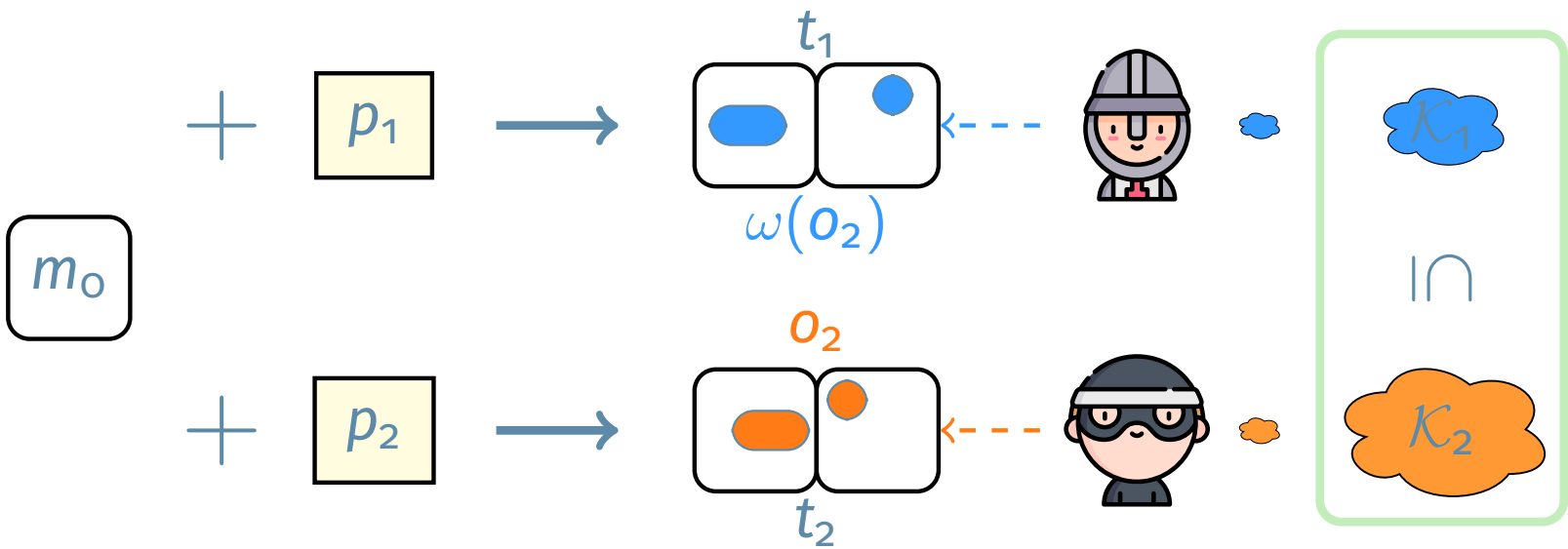


IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

$\forall(m_0, t_1, t_2).\ \forall n.\ \exists\omega \in \Omega(t_1, t_2).\ \forall o_2.$ $\mathcal{K}_n^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^{t_2}(p_2, o_2)$

$\mathcal{K}_1$ derived from $\omega(o_2)$ is a subset of $\mathcal{K}_2$ derived from o_2



Translation Validation for *Register Allocation*

Register Allocation

- Introduce spilling of values in the stack
- Usually not IFP:
  - Duplication on both stack and registers
  - Erasure may not be applied to both locations

Example with a 2-register machine:

```
def p1(k,t,salt):
  tmp = t + salt
  k = tmp + k
  return k
```

```
def p2(r1,r2,stack_salt):
  stack_k = r1
  r1 = stack_salt
  r1 = r2 + r1
  r2 = stack_k
  r2 = r1 + r2
  return r2
```

Register Allocation

- Introduce spilling of values in the stack
- Usually not IFP:
  - Duplication on both stack and registers
  - Erasure may not be applied to both locations

Example with a 2-register machine:

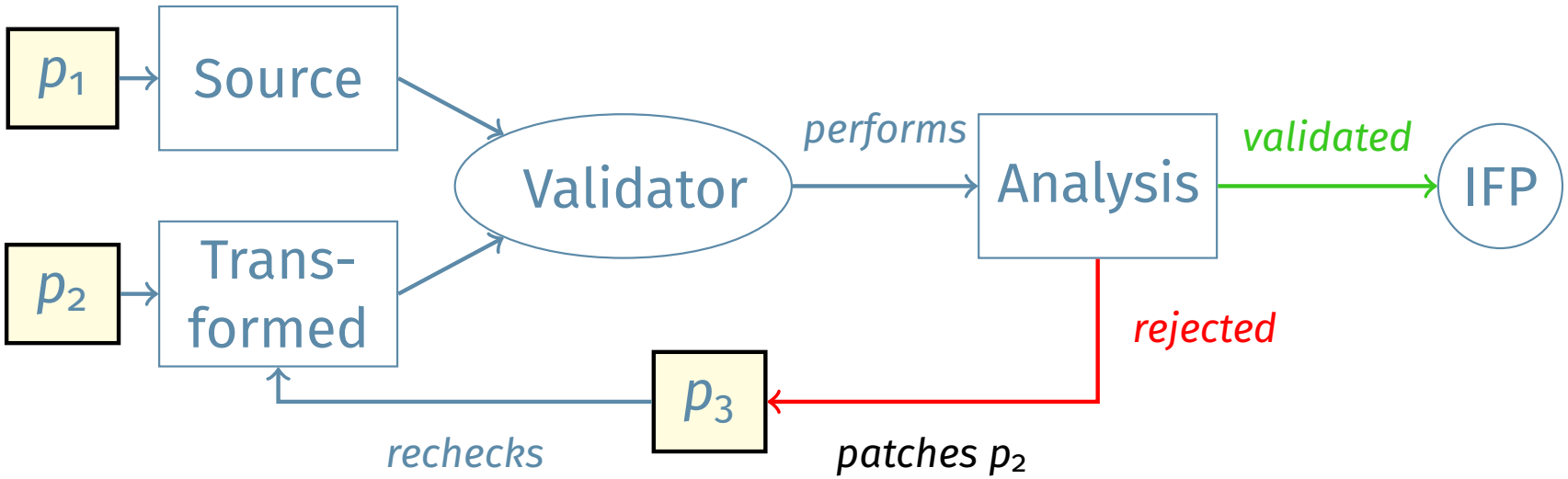
```
def p1(k,t,salt):
  tmp = t
  k = tmp
  return
```

```
def p2(r1,r2,stack_salt):
  stack_k = r1
  r1 = stack_salt
  r1
  k_k
  r2 = r1 + r2
  return r2
```

Secret value is duplicated
and not erased on the stack

Validation and patching toolchain

- Validator verifies the sufficient condition
- Detected leakage are patched



Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
  • tmp = t + salt
    k = tmp + k
  • return k
```

→

```
def p2(r1,r2,stack_salt):
  • stack_k = r1
    r1 = stack_salt
    r1 = r2 + r1
    r2 = stack_k
    r2 = r1 + r2
  • return r2
```

k	$\leftarrow$	$r1$
t	$\leftarrow$	$r2$
$salt$	$\leftarrow$	$stack\_salt$

Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
 • return r2
```

k	$\leftarrow$	$r1$
t	$\leftarrow$	$r2$
$salt$	$\leftarrow$	$stack\_salt$
k	$\leftarrow$	$stack\_k$

Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
 • return r2
```

salt	←	*r1*
t	←	*r2*
salt	←	*stack_salt*
k	←	*stack_k*

Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
 • return r2
```

tmp	←	*r1*
t	←	*r2*
salt	←	*stack_salt*
k	←	*stack_k*

Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
 • return r2
```

tmp	←	*r1*
k	←	*r2*
salt	←	*stack_salt*
k	←	*stack_k*

Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
 • return r2
```

tmp	←	*r1*
k	←	*r2*
salt	←	*stack_salt*
?	←	*stack_k*

Computing pairings

- build pairings from address of p_2 to address/constant

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
 • return r2
```

tmp	←	*r1*
k	←	*r2*
salt	←	*stack_salt*
?	←	*stack_k*

Leakage

Patching leakage

Leakage are patched with constant values

```
def p1(k,t,salt):
 • tmp = t + salt
   k = tmp + k
 • return k
```

```
def p2(r1,r2,stack_salt):
 • stack_k = r1
   r1 = stack_salt
   r1 = r2 + r1
   r2 = stack_k
   r2 = r1 + r2
   stack_k = 0
 • return r2
```

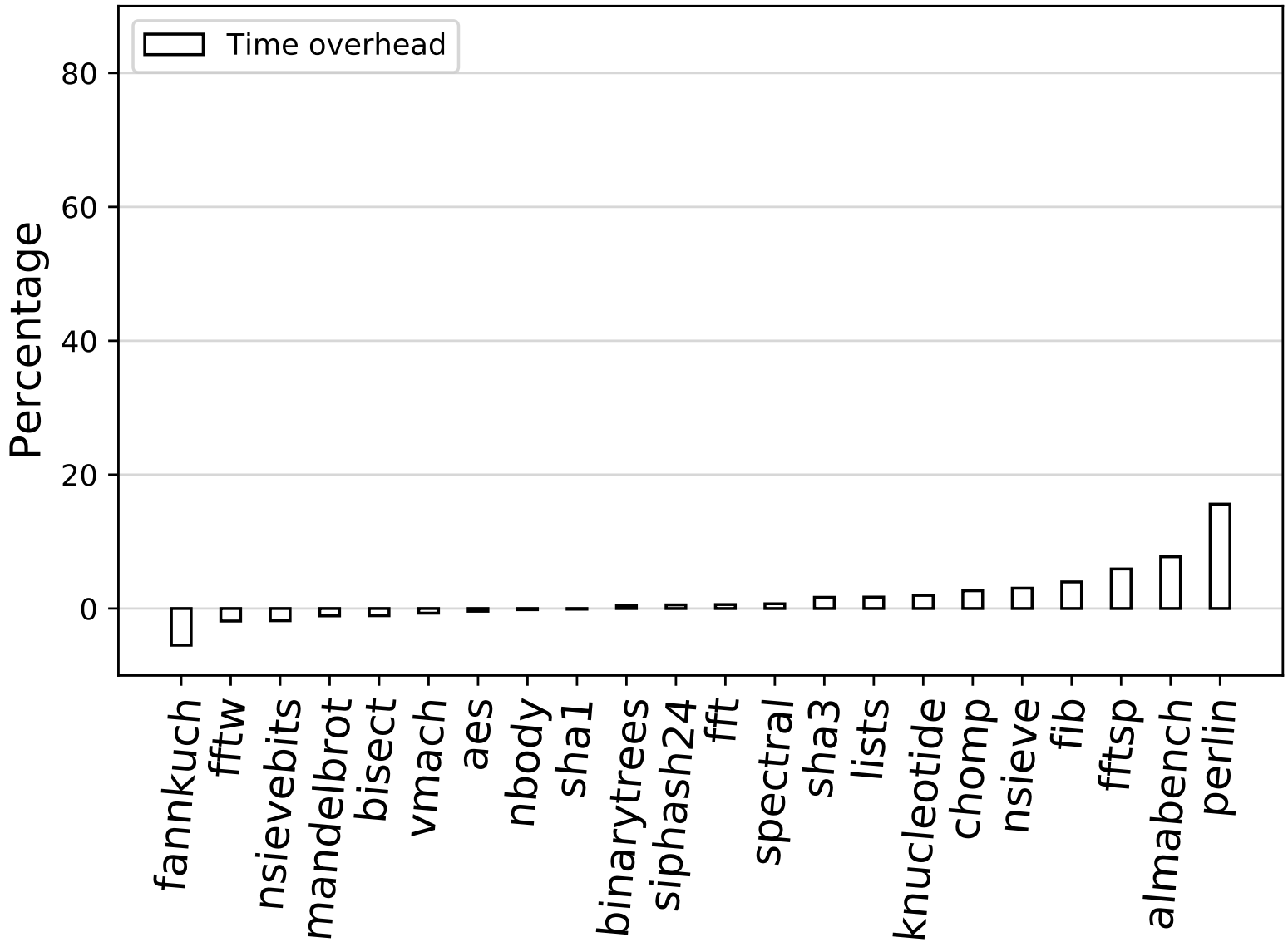
tmp	←	*r1*
k	←	*r2*
salt	←	*stack_salt*
0	←	*stack_k*

Experiments

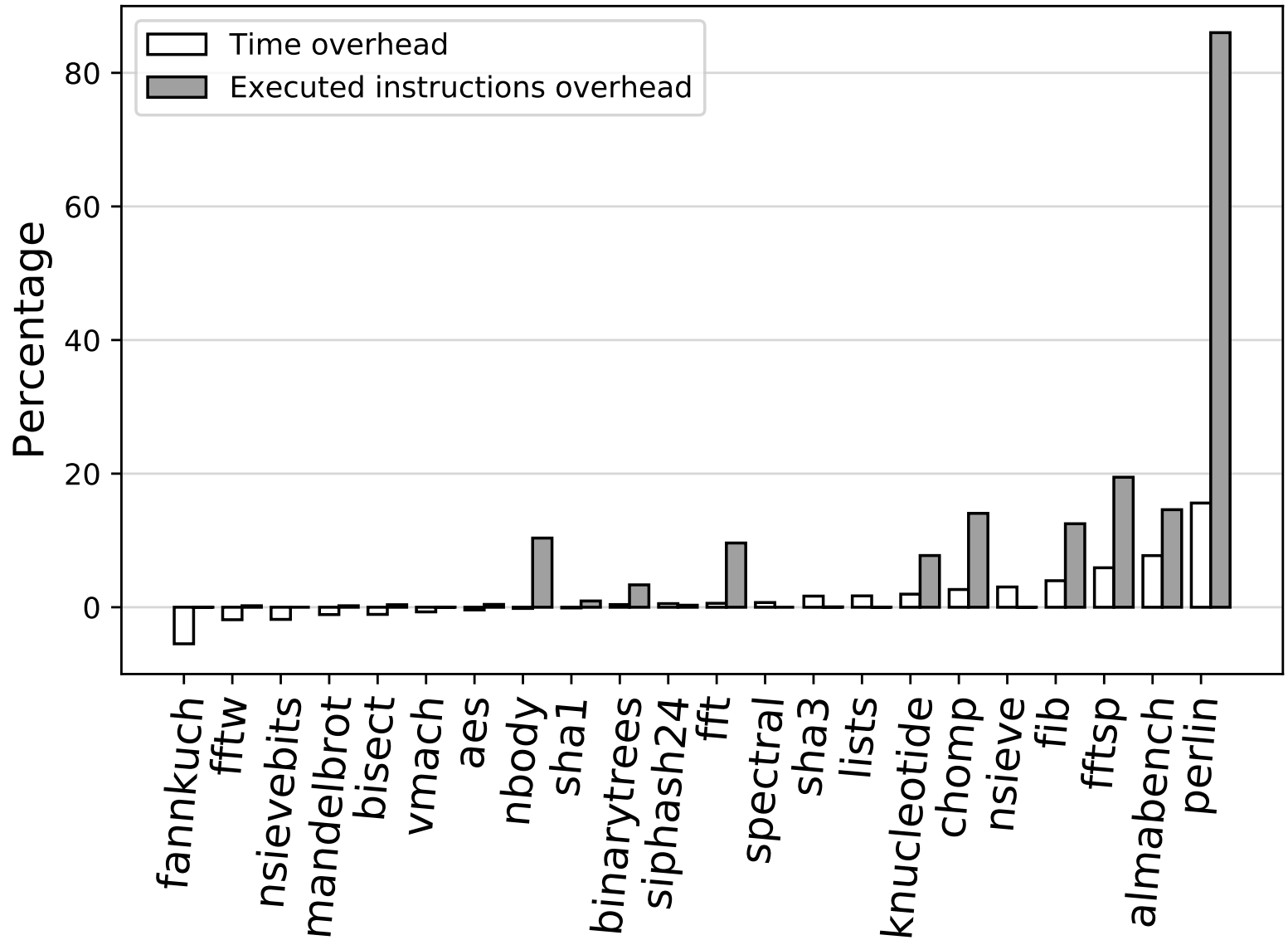
- Observation points are placed at function calls and returns
- On the verified compiler CompCert[1]
- We measure the impact of patching on the programs
- Correctness is ensured by CompCert original validator

[1] *Formal Certification of a Compiler Back-end*, Leroy [2006]

Measuring impact of patching



Measuring impact of patching



Related work and Conclusion

Related work

- Securing a compiler transformation[1][2]
  - preserve programs that do not leak
  - does not differentiate between degrees of leakage

- Preservation of side-channel countermeasures[3] [4]
  - framework to preserve security properties
  - different leakage model
  - use a 2-simulation property

---

[1] *Securing a Compiler Transformation*, Deng and Namjoshi [2016]

[2] *Securing the SSA Transform*, Deng and Namjoshi [2017]

[3] *Secure Compilation of Side-Channel Countermeasures*, Barthe et al. [2018]

[4] *Formal verification of a constant-time preserving C compiler*, Barthe et al. [2020]

Conclusion

General purpose compilers are not designed for security

They aim at preserving **observable** behaviours

- Software Fault Isolation ✓
- Information Flow ✗

In theory, compiler may not preserve information flow
In practice, they do break security of the source code

The **best** compilers are the **least** secure!
$\Rightarrow$ Optimisations need to be carefully reviewed

An opportunity for secure (and verified) compilation?