INFORMATION-FLOW PRESERVATION IN COMPILER TRANSFORMATIONS

FRÉDÉRIC BESSON
ALEXANDRE DANG
THOMAS JENSEN

INRIA RENNES

CSF 2019

INTRODUCTION

COMPILATION AND SECURITY

Semantic correctness at the core of compilers

- Optimizing compilers like gcc or LLVM
- Formally verified: CompCert, Vellvm, CakeML …

Correctness is not enough for security [1]

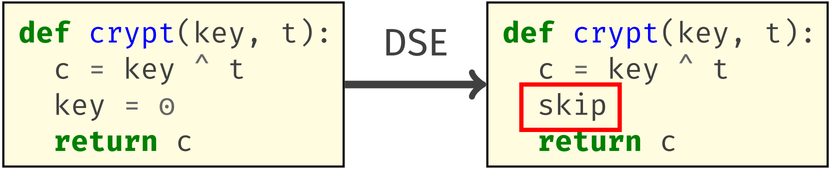
- Not suited against side-channel attacks
- Timing, power analysis, data remanence …

---

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

DEAD STORE ELIMINATION IS NOT SECURE[1]

- Sensitive data should not remain in memory
- Erasure is performed on sensitive data
- *Dead Store Elimination* (DSE) may break erasure
- Bug reports of LLVM, gcc, OpenSSL …



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

DSE

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

---

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

INFORMATION-FLOW PRESERVATION

Goal

Attackers should not learn more information from the transformed program than from the source program

Contributions and content of the talk

- Formal definition of an IFP[1] transformation
- Proof technique to certify that a transformation is IFP
- Implementation of an IFP *Register Allocation*

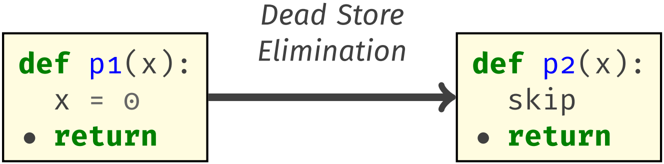
---

[1] Information-Flow Preserving

Getting familiar with IFP

Effects we want to avoid:

- **Data remanence**



```
def p1(x):
  x = 0
• return
```

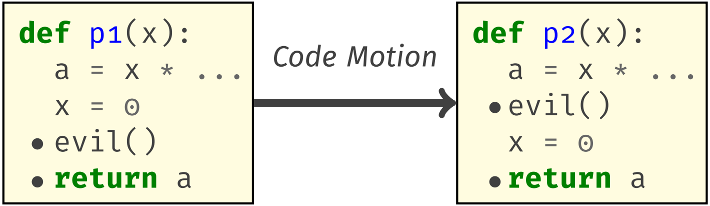
Dead Store Elimination

```
def p2(x):
  skip
• return
```

Getting familiar with IFP

Effects we want to avoid:

- Data remanence
- **Lifetime extension**



```
def p1(x):
   a = x * ...
   x = 0
 • evil()
 • return a
```

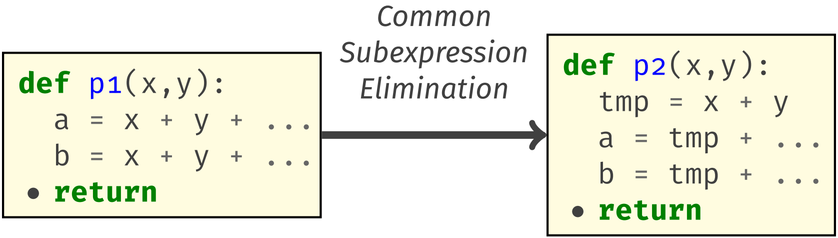
Code Motion

```
def p2(x):
   a = x * ...
 • evil()
   x = 0
 • return a
```

GETTING FAMILIAR WITH IFP

Effects we want to avoid:

- Data remanence
- Lifetime extension
- **Worsening of leakage**



```
def p1(x,y):
  a = x + y + ...
  b = x + y + ...
• return
```

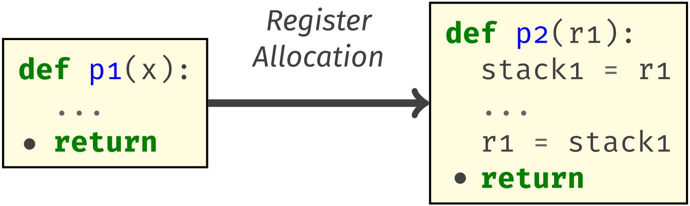
Common Subexpression Elimination

```
def p2(x,y):
  tmp = x + y
  a = tmp + ...
  b = tmp + ...
• return
```

GETTING FAMILIAR WITH IFP

Effects we want to avoid:

- Data remanence
- Lifetime extension
- Worsening of leakage
- **Duplication**



```
def p1(x):
   ...
 • return
```

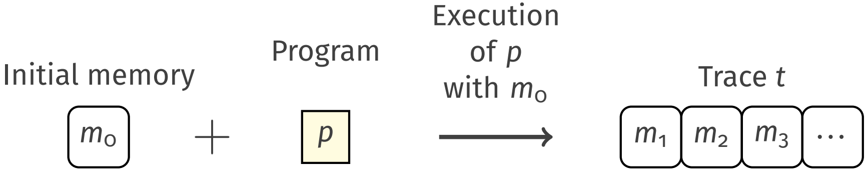
Register Allocation

```
def p2(r1):
  stack1 = r1
  ...
  r1 = stack1
 • return
```

DEFINITION OF IFP

EXECUTION MODEL

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



Initial memory

Program

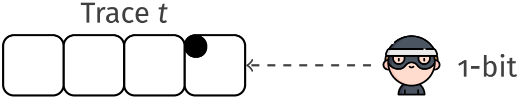
Execution of p with m_0

Trace t

$$\boxed{m_0} + \boxed{p} \longrightarrow \boxed{m_1}\boxed{m_2}\boxed{m_3}\boxed{\cdots}$$

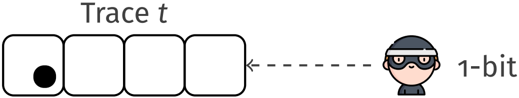
ATTACKER MODEL

- Attackers have access to program's code
- Attackers observe *n* bits in the trace



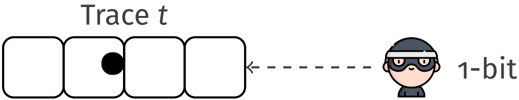
ATTACKER MODEL

- Attackers have access to program's code
- Attackers observe *n* bits in the trace



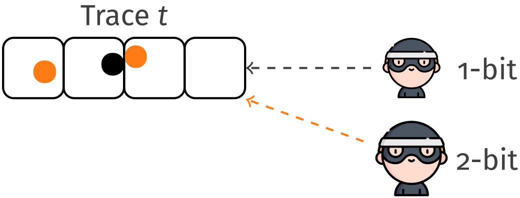
ATTACKER MODEL

- Attackers have access to program's code
- Attackers observe *n* bits in the trace



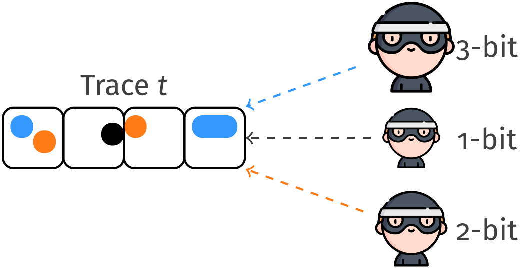
ATTACKER MODEL

- Attackers have access to program's code
- Attackers observe *n* bits in the trace



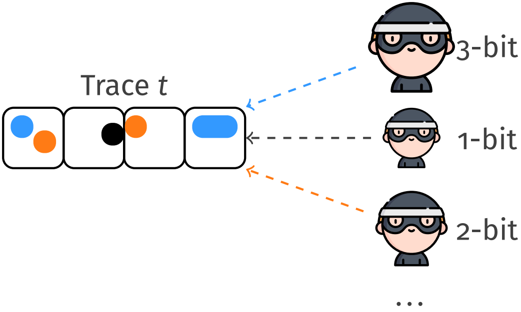
ATTACKER MODEL

- Attackers have access to program's code
- Attackers observe *n* bits in the trace

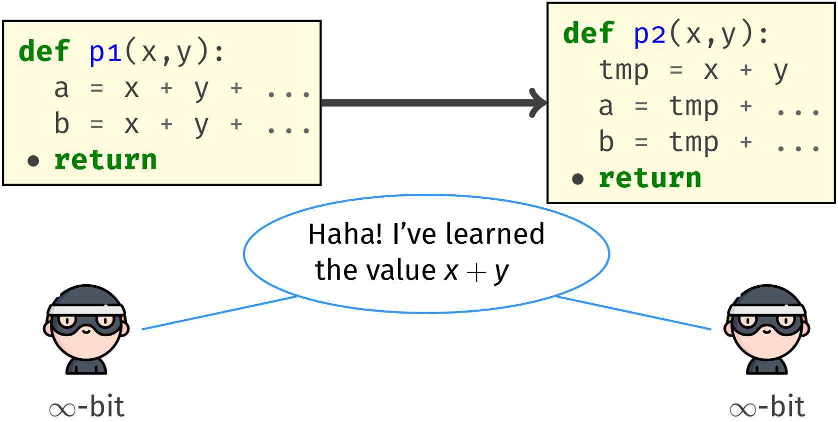


ATTACKER MODEL

- Attackers have access to program's code
- Attackers observe *n* bits in the trace



RATIONALE FOR MULTIPLE ATTACKERS



```
def p1(x,y):
   a = x + y + ...
   b = x + y + ...
 • return
```

```
def p2(x,y):
   tmp = x + y
   a = tmp + ...
   b = tmp + ...
 • return
```

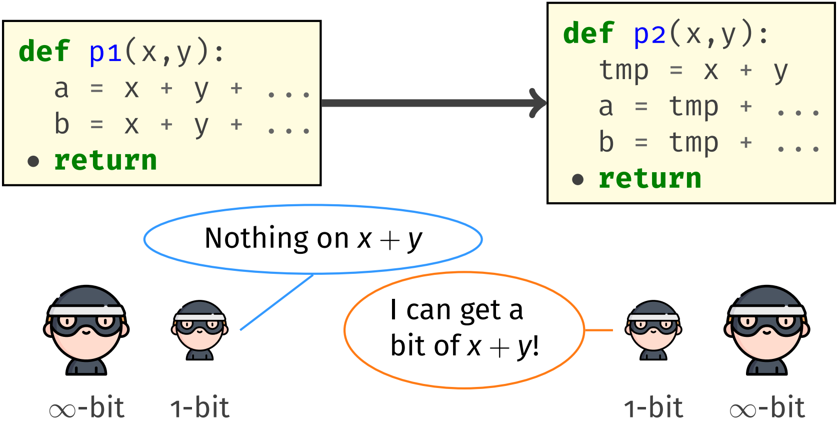
Haha! I've learned the value $x + y$

∞-bit

∞-bit

- equally insecure for a strong attacker

RATIONALE FOR MULTIPLE ATTACKERS



```
def p1(x,y):
   a = x + y + ...
   b = x + y + ...
 • return
```

```
def p2(x,y):
   tmp = x + y
   a = tmp + ...
   b = tmp + ...
 • return
```

Nothing on $x + y$

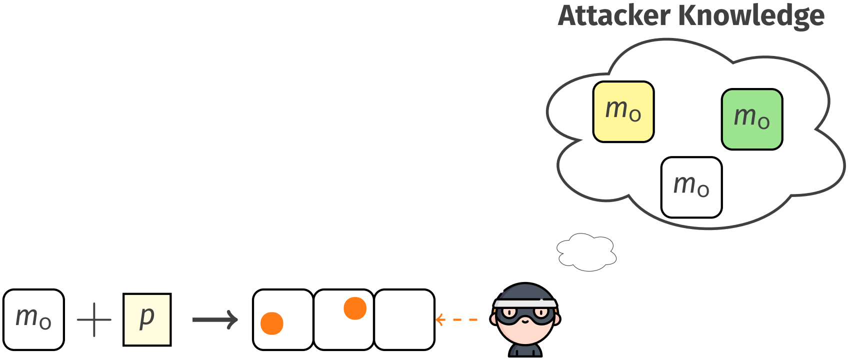
I can get a bit of $x + y$!

∞-bit 1-bit 1-bit ∞-bit

- 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



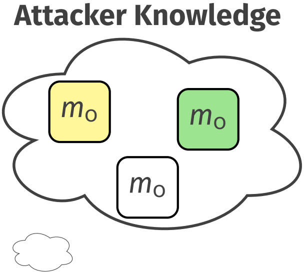
---

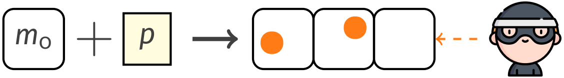
[1] *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*, Askarov and Sabelfeld [2007]

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_o





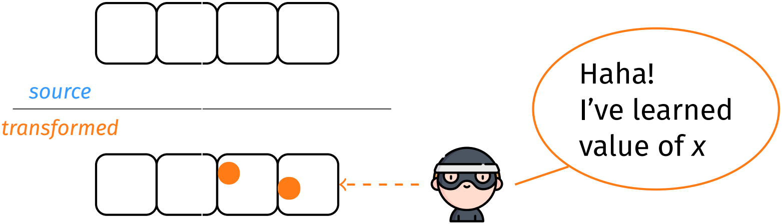
---

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

IFP TRANSFORMATION (1/2)

Intuition

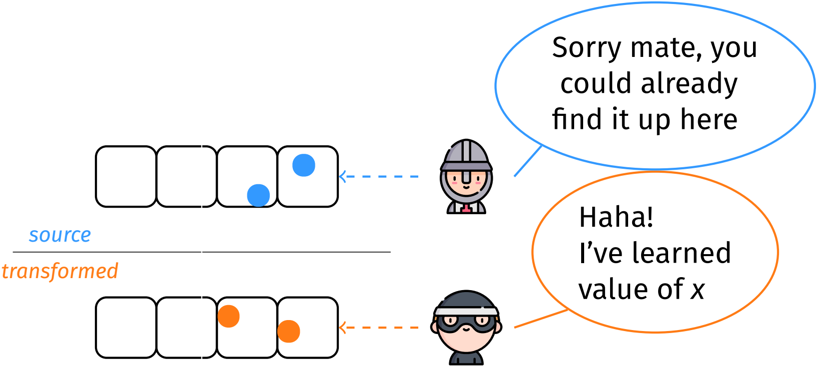
Any information that can be learned with a trace observation of the transformed program can also be learned with the source program



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

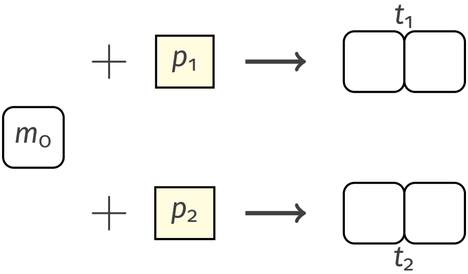
p_2

IFP TRANSFORMATION (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

For any execution from the same initial memory m_0

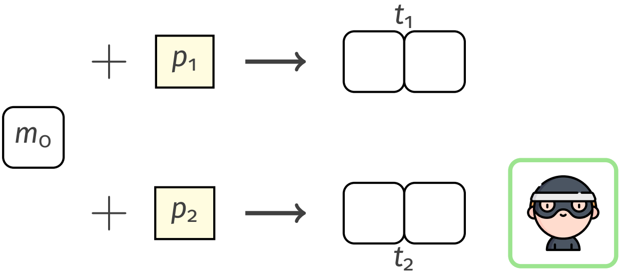


IFP TRANSFORMATION (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

For attackers with any observation capabilities

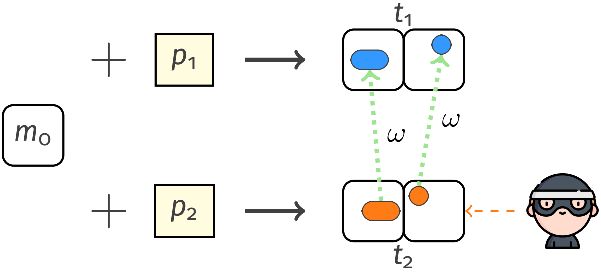


IFP TRANSFORMATION (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

Exists lockstep pairings of observations from t_2 to t_1

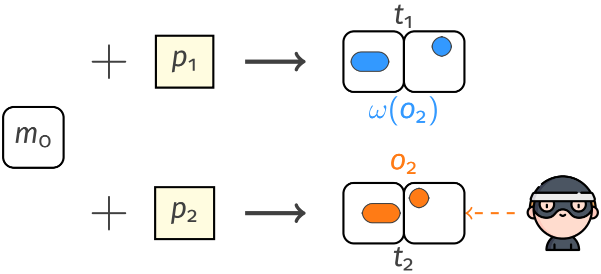


IFP TRANSFORMATION (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

For any observation o_2 of size n on the trace t_2

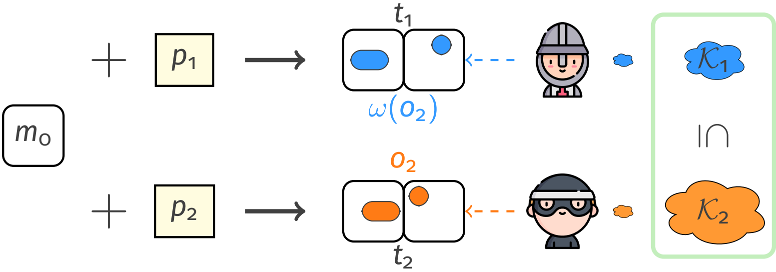


IFP TRANSFORMATION (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

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

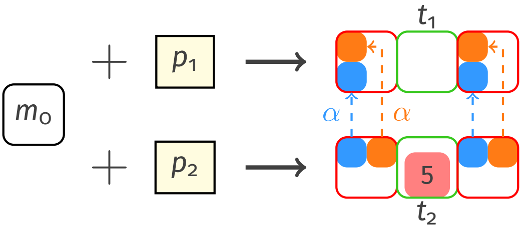


PROOF TECHNIQUE

SUFFICIENT CONDITION FOR AN IFP TRANSFORMATION

- Lockstep pairings from memory address of the trace t_2
- Each address of t_2 is paired to:
  - a lockstep address of t_1 OR
  - a constant

$$\exists\alpha.\,\forall(m_0,t_1,t_2).\,\forall a_2,i.\quad t_2[i](a_2)=\begin{cases} t_1[i](\alpha_i(a_2)) & \text{if } \alpha_i(a_2)\in \mathit{Address}\\ \alpha_i(a_2) & \text{if } \alpha_i(a_2)\in \mathit{Bit}\end{cases}$$



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:

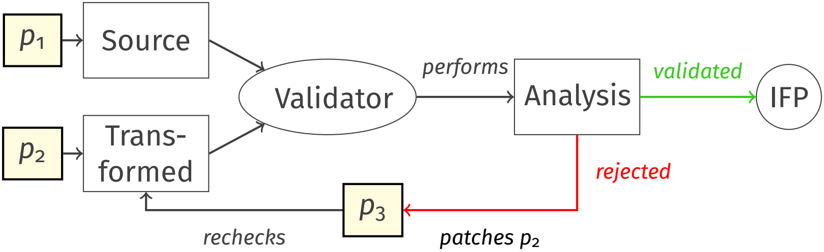
```
def p1(k,t,salt):
  tmp = t + salt
  k = tmp
  return k
```

```
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
```

$$
\begin{array}{ccc}
k & \leftarrow & r1 \\
t & \leftarrow & r2 \\
salt & \leftarrow & stack\_salt
\end{array}
$$

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
```

$k \leftarrow r1$

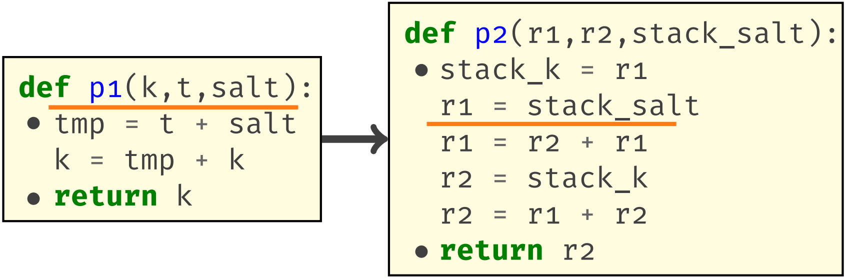
$t \leftarrow r2$

$salt \leftarrow stack\_salt$

$k \leftarrow stack\_k$

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
```

$salt \leftarrow r1$

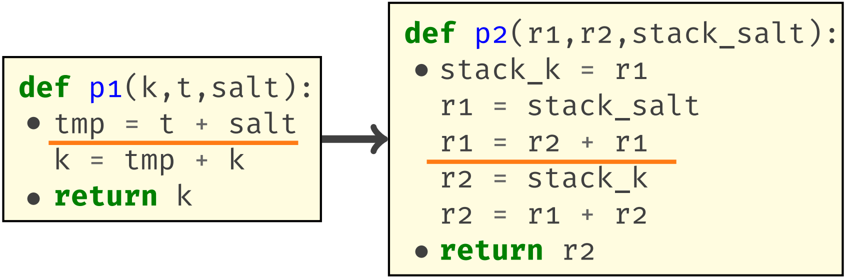
$t \leftarrow r2$

$salt \leftarrow stack\_salt$

$k \leftarrow stack\_k$

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
```

$tmp \leftarrow r1$

$t \leftarrow r2$

$salt \leftarrow stack\_salt$

$k \leftarrow stack\_k$

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
```

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

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
```

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

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
```

$tmp \leftarrow r1$

$k \leftarrow r2$

$salt \leftarrow stack\_salt$

$0 \leftarrow stack\_k$

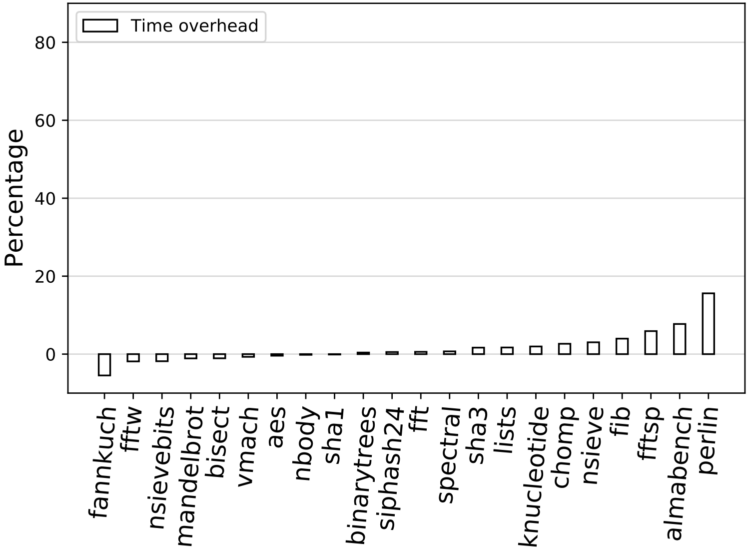
EXPERIMENTS

- 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
- Patching of duplication was not implemented here

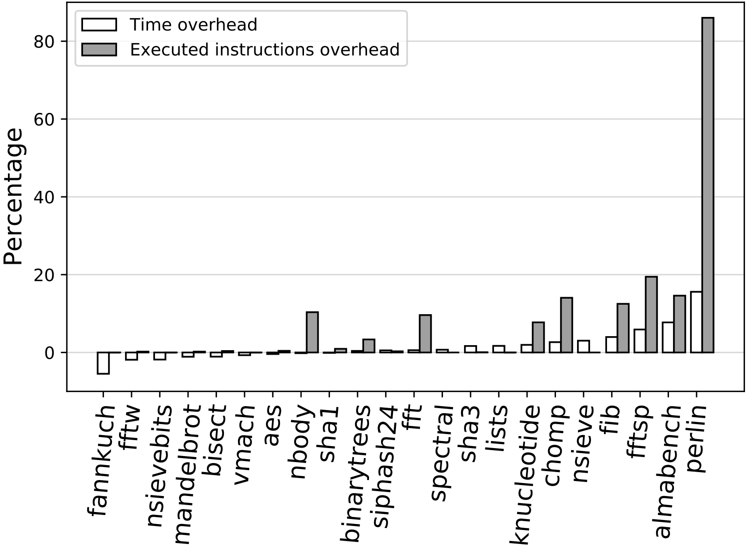
---

[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]
  - 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]

FUTURE WORK

- Development
    - Extend our property to other compilation passes
    - Improve performance with more precise patching

- Improve IFP property
    - current property is bound by observation points
    - extend to attackers that can make observations at any time

Thank you for listening

Contact me!

alexandre.dang@inria.fr