Information-Flow Preservation in Compiler Transformations

Frédéric Besson
Alexandre Dang
Thomas Jensen

Celtique/Inria/Univ Rennes

Gdr SSLR, November 2019

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 is free of bugs (e.g., LLVM)

Hyp2 : My program has no undefined behaviour (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*

$\Rightarrow$ Cryptographers do not trust compilers.

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

Long-term goal: a Secure Compiler

> *A secure compiler does not break/remove security counter-measures.*

Attackers do not get an advantage at attacking the target.
Research Agenda

- Define classes of attackers.
- Revisit/Patch existing compiler passes.

Today

Information-Flow Preservation

Attackers should not learn more information from the Target than from the Source.

Attacker model

Passive observation of (arbitrary) memory content.

Contributions

- Formal definition of an IFP[1]
- Sufficient condition to ensure IFP
- Application to *Register Allocation*

[1] Information-Flow Preserving

Getting Familiar with IFP

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

```
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]

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

Code motion is not secure.

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

→

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

Sec. Req. 3: Limit Leakage of Information

Common Expression Elimination is not secure.

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

→

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

Register Allocation is not secure.

```
def p1(x):
  ...
• return
```

→

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

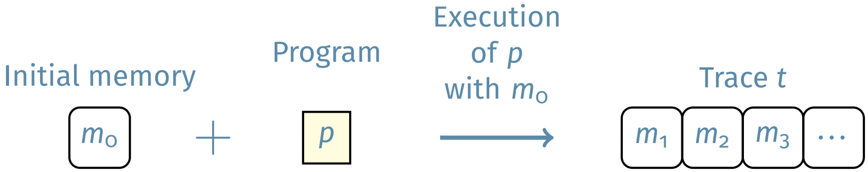
Information-Flow Preservation

IFP protects against:

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

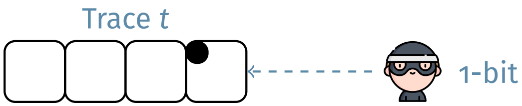
Formal Definition of IFP

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



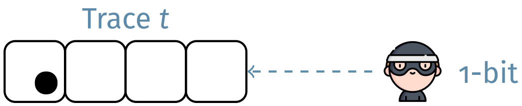
Attacker model

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

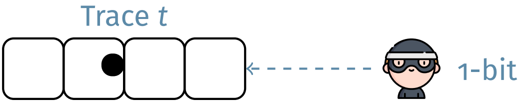


Attacker model

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

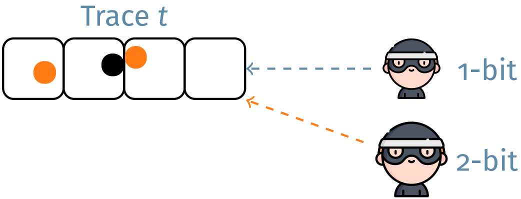


- Attackers know the code
- Attackers observe n bits in the trace



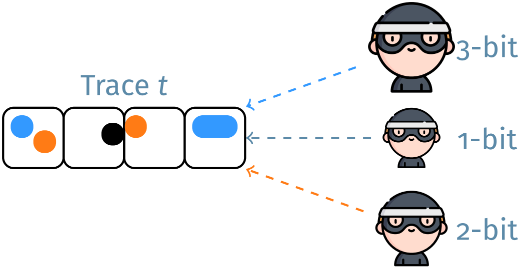
Attacker model

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



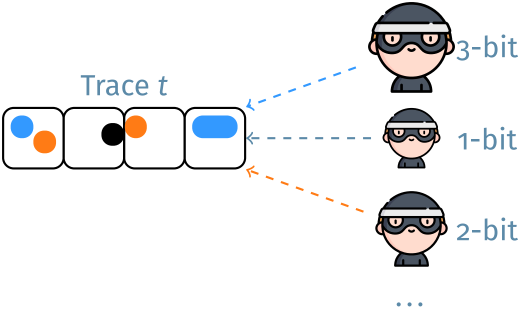
Attacker model

- 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
```

Haha! I've learned the value $key = c \hat{} t$

∞-bit

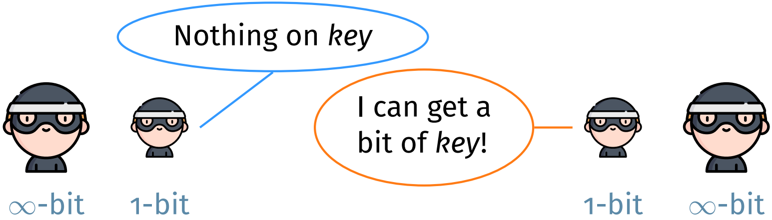
∞-bit

- equally insecure for a strong attacker

Rationale for hierarchy of attackers

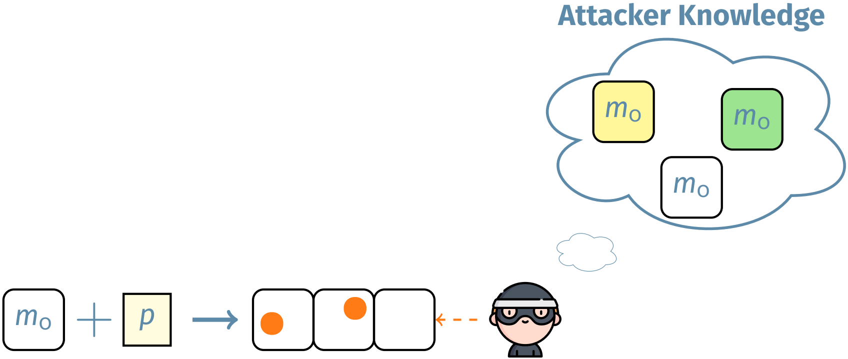
```
def crypt(key, t):
  c = key ^ t
  key = 0
 •return c
```

```
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

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



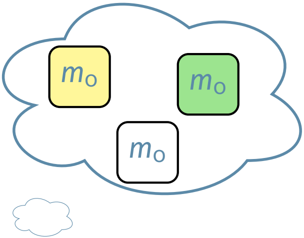
[1] *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*, Askarov and Sabelfeld [2007]

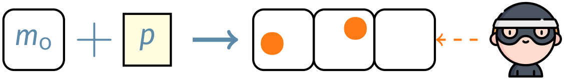
Attacker Knowledge [1]

- 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

Attacker Knowledge



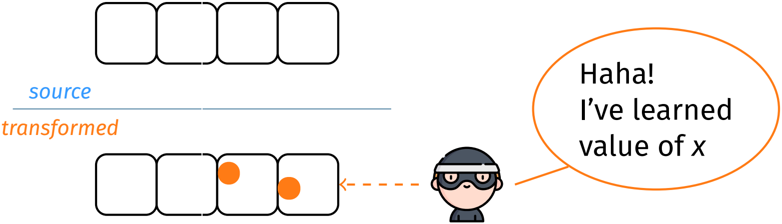


[1] *Gradual Release: Unifying Declassification, Encryption and Key Release Policies*, Askarov and Sabelfeld [2007]

IFP TRANSFORMATION (1/2)

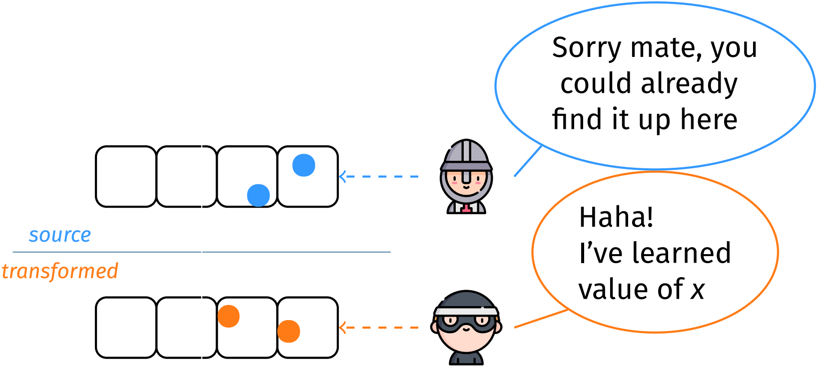
Intuition

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



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

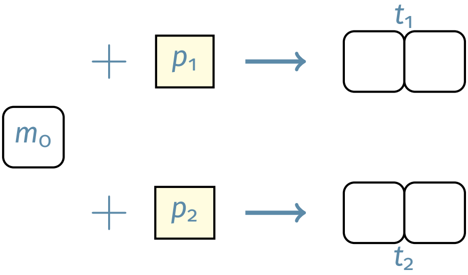
p_1

p_2

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

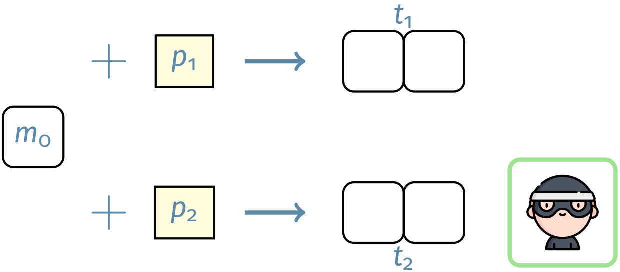


IFP TRANSFORMATION (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

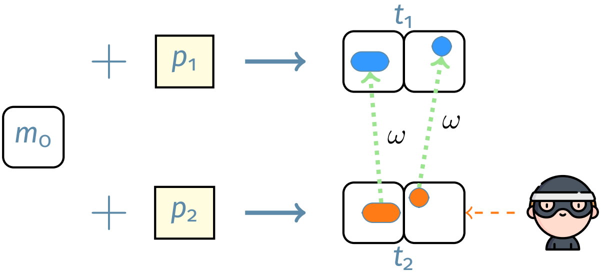
For attackers with any observation capabilities



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

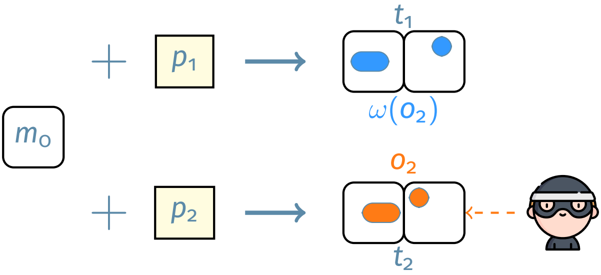


IFP transformation (2/2)

A transformation from p_1 to p_2 is IFP iff:

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

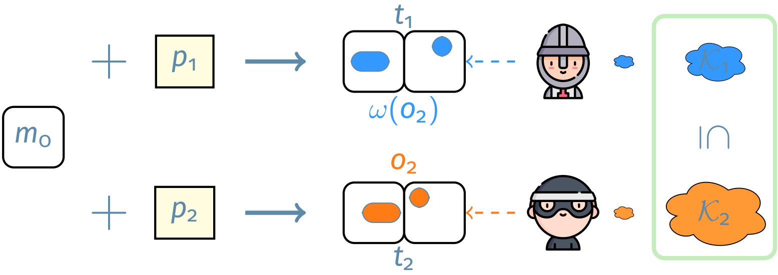
For any observation o_2 of size n on the trace t_2



A transformation from p_1 to p_2 is IFP iff:

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

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

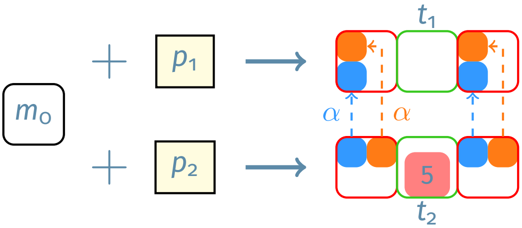


Proof technique

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*

- 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
```

- 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:

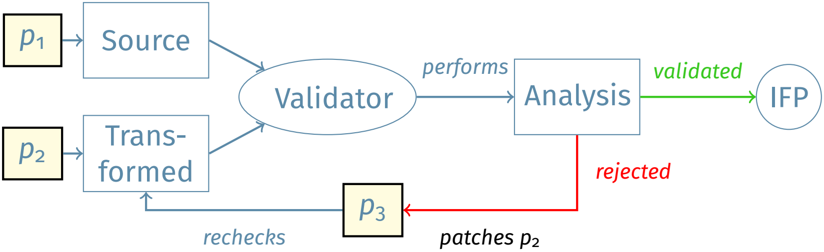
```
def p1(k,t,salt):
  tmp = t + salt
  k = tmp
  return k
```

```
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



- 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
```

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

- 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
```

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

- 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*

- 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*

- 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
```

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

- 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
```

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

Leakage

Leakage are patched with constant values

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

→

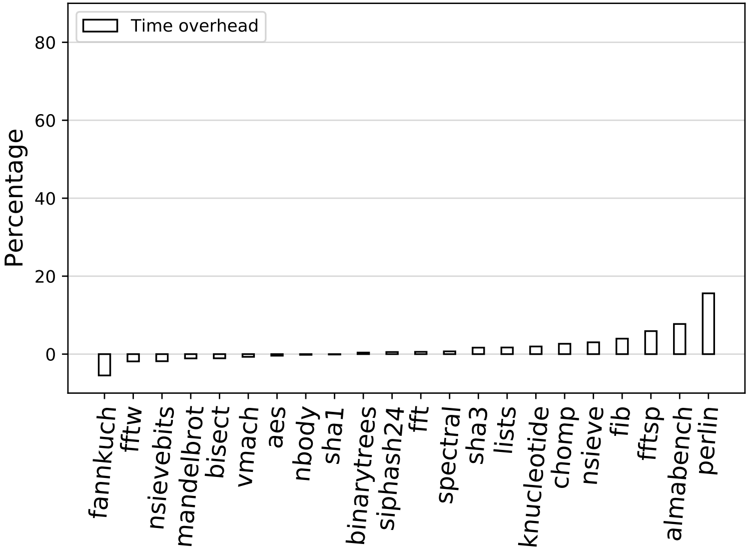
```
def p2(r1,r2,stack_salt):
• stack_k = r1
  r1 = stack_salt
  r1 = r2 + r1
  r2 = stack_k
  r2 = r1 + r2
  stack_k = 0
• return r2
```

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

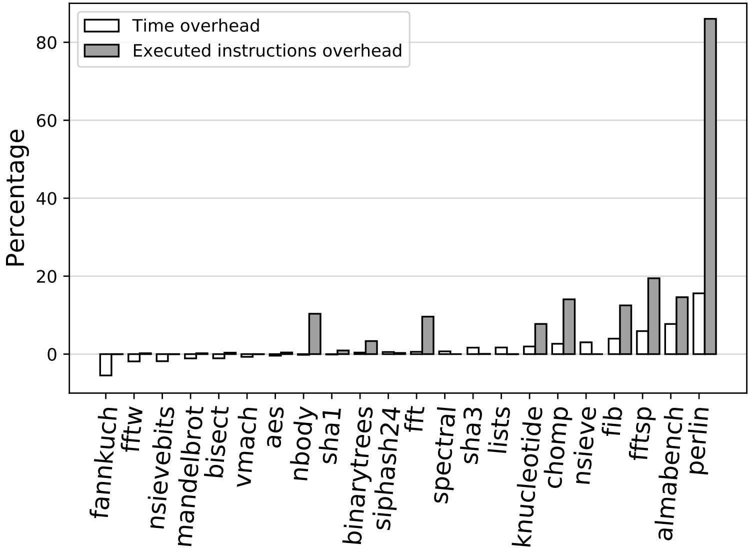
- Observation points are placed at function calls and returns
- On the verified compiler CompCert[1]
- We measure the impact of patching on the programs
- Correctness is ensured by CompCert original validator
- Patching of duplication was not implemented here

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

Measuring impact of patching



Measuring impact of patching



Related work and Conclusion

- 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

- Towards a secure IFP compiler
  - More compilation passes
  - Better performance of patching
- Refine our IFP property
  - Current property is bound by observation points
  - Could attackers observe at any time?
- Other Models of Attackers
  - Speculative Attackers
  - Hamming Weight Model

Thank you for listening

Contact me!

alexandre.dang@inria.fr