INFORMATION-FLOW PRESERVATION IN COMPILER TRANSFORMATIONS

FRÉDÉRIC BESSON
ALEXANDRE DANG
THOMAS JENSEN

INRIA RENNES

CSF 2019
INTRODUCTION
Semantic correctness at the core of compilers

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

Correctness is not enough for security

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

---

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

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

---

1 Dead Store Elimination (Still) Considered Harmful, Yang et al. [2017]
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
Effects we want to avoid:

- Data remanence
Effects we want to avoid:

- Data remanence
- Lifetime extension
Effects we want to avoid:
- Data remanence
- Lifetime extension
- Worsening of leakage

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

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
- Trace based execution model
- Memory states: data observable by attackers
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
Attackers have access to program’s code

- Attackers observe $n$ bits in the trace
RATIONAL FOR MULTIPLE ATTACKERS

\[
\text{def } p_1(x, y):
\begin{align*}
  a &= x + y + \ldots \\
  b &= x + y + \ldots \\
  &\quad \text{\textbullet return}
\end{align*}
\]

\[
\text{def } p_2(x, y):
\begin{align*}
  \text{tmp} &= x + y \\
  a &= \text{tmp} + \ldots \\
  b &= \text{tmp} + \ldots \\
  &\quad \text{\textbullet return}
\end{align*}
\]

Haha! I’ve learned the value $x + y$

- equally insecure for a strong attacker

\[\infty\text{-bit}\]
def $p_1(x,y)$:
  a = $x + y + ...$
  b = $x + y + ...$
  • return

def $p_2(x,y)$:
  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**

- 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.

**Attacker Knowledge**

- 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 $m_0$

$\overset{1}{\text{Gradual Release: Unifying Declassification, Encryption and Key Release Policies, Askarov and Sabelfeld [2007]}}$
Intuition

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

Haha! I’ve learned value of x.
Intuition

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

Sorry mate, you could already find it up here.

Haha! I’ve learned value of x.
A transformation from $p_1$ to $p_2$ is IFP iff:
\[
\forall (m_0, t_1, t_2). \forall n. \exists \omega \in \Omega(t_1, t_2). \forall o_2. \ \mathcal{K}_{n}^{t_1}(p_1, \omega(o_2)) \subseteq \mathcal{K}_{n}^{t_2}(p_2, o_2)
\]
A transformation from $p_1$ to $p_2$ is IFP iff:

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

**Source program $p_1$**

**Transformed program $p_2$**
A transformation from $p_1$ to $p_2$ is IFP iff:

$$\forall (m_0, t_1, t_2), \forall n. \exists \omega \in \Omega(t_1, t_2). \forall o_2. \quad K_{n_1}^{t_1}(p_1, \omega(o_2)) \subseteq K_{n_2}^{t_2}(p_2, o_2)$$

For any execution from the same initial memory $m_0$
A transformation from $p_1$ to $p_2$ is IFP iff:

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

For attackers with any observation capabilities
A transformation from $p_1$ to $p_2$ is IFP iff:

$$\forall (m_0, t_1, t_2). \forall n. \exists \omega \in \Omega(t_1, t_2). \forall o_2. \quad K_{n_1}^t(p_1, \omega(o_2)) \subseteq K_{n_2}^t(p_2, o_2)$$

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

For any observation $o_2$ of size $n$ on the trace $t_2$
A transformation from $p_1$ to $p_2$ is IFP iff:

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

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

```python
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:

```python
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 + r1
    r2 = r1 + r2
    return r2
```

Secret value is duplicated and not erased on the stack
- Validator verifies the sufficient condition
- Detected leakage are patched
Compute pairings from address of $p_2$ to address/constant.

```python
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
```

$k \leftarrow r1$
$t \leftarrow r2$
$salt \leftarrow stack_{\text{salt}}$
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
```

$k \leftarrow r1$
$t \leftarrow r2$
$salt \leftarrow stack\_salt$
$k \leftarrow stack\_k$
build pairings from address of $p_2$ to address/constant

def $p_1(k, t, \text{salt})$:
- $\text{tmp} = t + \text{salt}$
- $k = \text{tmp} + k$
- return $k$

def $p_2(r_1, r_2, \text{stack\_salt})$:
- $\text{stack\_k} = r_1$
- $r_1 = \text{stack\_salt}$
- $r_1 = r_2 + r_1$
- $r_2 = \text{stack\_k}$
- $r_2 = r_1 + r_2$
- return $r_2$
Computing Pairings

- build pairings from address of $p_2$ to address/constant

```python
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

```python
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
```

```plaintext
tmp ← r1  
k ← r2  
salt ← stack_salt  
k ← stack_k
```
Computing pairings

- build pairings from address of $p_2$ to address/constant

```python
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
```

```plaintext
tmp ← r1
k ← r2
salt ← stack_salt
? ← stack_k
```
Build pairings from address of $p_2$ to address/constant

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

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

Leakage:
- $tmp \leftarrow r1$
- $k \leftarrow r2$
- $salt \leftarrow stack\_salt$
- $? \leftarrow stack\_k$
Leakage are patched with constant values

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

```python
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
```
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

\(^1\text{Formal Certification of a Compiler Back-end, Leroy [2006]}\)
Measuring impact of patching

![Graph showing time overhead for various benchmarks.](image-url)
Measuring impact of patching

![Graph showing percentage of time overhead and executed instructions overhead across different benchmarks.](image-url)
RELATED WORK AND CONCLUSION
Securing a compiler transformation\textsuperscript{12}
  ▶ preserve programs that do not leak
  ▶ does not differentiate between degrees of leakage

Preservation of side-channel countermeasures\textsuperscript{3}
  ▶ framework to preserve security properties
  ▶ different leakage model
  ▶ use a 2-simulation property

\textsuperscript{1}Securing a Compiler Transformation, Deng and Namjoshi [2016]
\textsuperscript{2}Securing the SSA Transform, Deng and Namjoshi [2017]
\textsuperscript{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