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
What is the expected guarantee?

Semantic preservation

If \( \text{beh}(S) \neq \emptyset \) Then \( \text{beh}(T) \subseteq \text{beh}(S) \).

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

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

Functional properties are preserved.

⇒ I can reason at source level!
Compilers may enhance security

- Shadow stack
- Canaries
- Security instrumentation

Compilers may also break security counter-measures\(^1\)

- Introduction of \texttt{jump} breaks CT-programming
- Associativity of \texttt{xor} breaks \emph{masking}
- CSE breaks Fault-Injection protection
- (Dead) code removal breaks \emph{CFI}; breaks \emph{safe erasure}

\[\Rightarrow \text{Security people do not trust compilers.}\]

\(^1\)\textit{The Correctness-Security Gap in Compiler Optimization}, D’Silva et al. [2015]
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.
Security Enhancement: Software Fault Isolation
Property: Integrity of a host running untrusted code

Security Preservation: Information Flow Preservation
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
"Run safely binary code of untrusted origin"

A modified Compiler (∉ TCB) masks memory accesses.

Binary verifier (∈ TCB) checks masking is correct.

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

\[ C \rightarrow \text{CLANG} \rightarrow \text{LLVM} \rightarrow \text{SFI} \rightarrow \text{binary} \]

A verified verifier [RockSalt, PLDI’12]

\[ \text{verifier : binary } \rightarrow \mathbb{B} \]
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**).
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
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

<table>
<thead>
<tr>
<th>sb</th>
<th>TAGOOO ... TAGFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>XYZTVU</td>
</tr>
<tr>
<td>msk(A)</td>
<td>TAGTUV</td>
</tr>
</tbody>
</table>

Masking pointer arithmetic has no C semantics.

Security: Vacuously true

Safety: Vacuously false
# Masking of Memory Accesses

<table>
<thead>
<tr>
<th>sb</th>
<th>TAGOOO ... TAGFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>XYZTUV</td>
</tr>
<tr>
<td>A &amp; OxFFF</td>
<td>OOOTUV</td>
</tr>
<tr>
<td>( msk(A) )</td>
<td>TAGTU V</td>
</tr>
</tbody>
</table>
# Masking of Memory Accesses

<table>
<thead>
<tr>
<th>sb</th>
<th>TAGOOO</th>
<th>...</th>
<th>TAGFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>XYZTU</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>A &amp; OxFFF</td>
<td>OOOTU</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>(A &amp; OxFFF)</td>
<td>TAGOOO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>msk(A)</td>
<td>TAGTU</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

Masking pointer arithmetic has no C semantics

Security: Vacuously true

Safety: Vacuously false

\[
\text{msk}(A) = (A \& (\text{two.osf} - \text{one.osf})) | \text{sb}
\]
## Masking of Memory Accesses

<table>
<thead>
<tr>
<th>sb</th>
<th>TAGOOO ... TAGFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>XYZTUVOOTUV</td>
</tr>
<tr>
<td>A &amp; 0xFFF</td>
<td>0000TU</td>
</tr>
<tr>
<td>(A &amp; 0xFFF)</td>
<td>TAGOOO</td>
</tr>
<tr>
<td>$msk(A)$</td>
<td>TAGTUV</td>
</tr>
</tbody>
</table>

$$msk(A) = (A \& (2^k - 1)) \& sb$$
Masking of Memory Accesses

\[
\text{msk}(A) = (A \& (2^k - 1)) | \& \text{sb}
\]

Masking pointer arithmetic has no C semantics

\textbf{Security} \quad \text{Vacuously true}

\textbf{Safety} \quad \text{Vacuously false}
Our solution: Pointer-free Transformation

Pointers are compiled into their numeric value

\[
sfi(\&sb) = \text{tag} \times 2^k
\]

\[
\ldots
\]

\[
sfi(\ast(e)) = \ast(sfi(e)\&msk + \&sb)
\]

\[
sb[2^k] = \{5; \ldots \};
\]

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

C \rightarrow \text{CompCertSfi} \rightarrow \text{CM} \rightarrow \text{CompCert}

\text{CompCert} \rightarrow \text{GCC} \rightarrow \text{NaCl}

\text{GCC} \rightarrow \text{Clang} \rightarrow \text{PNacl}


- **GCCSFI/CLANGSFI** very competitive
- **COMPCERTSFI** average overhead 9% (removing outliers)

⇒ Optimisations improve SFI
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
Our Information-Flow Preservation property aims at protecting against:

- **Data remanence**
- **Lifetime extension**
- **Increased information leakage**
- **Duplication of information**
Dead Store Elimination (DSE) is not secure\(^1\)

---

\(^1\) *Dead Store Elimination (Still) Considered Harmful*, Yang et al. [2017]
Code motion is not secure.

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

def p2(x):
    a = x * ...
    evil()
    x = 0
    return a
```
Common Expression Elimination is not secure.

```python
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.

```python
def p1(x):
    ...  
    • return

def p2(r1):
    stack1 = r1  
    ...  
    r1 = stack1  
    • return
```
FORMAL DEFINITION OF IFP
Trace based execution model
Memory states: data observable by attackers
ATTKER 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
Attackers know the code
Attackers observe $n$ bits in the trace

Trace $t$
ATTACKER MODEL

- Attackers know the code
- Attackers observe $n$ bits in the trace
Attackers know the code
- Attackers observe $n$ bits in the trace
Rationale for hierarchy of attackers

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

Haha! I’ve learned the value `key = c^t`

- equally insecure for a strong attacker
def crypt(key, t):
    c = key ^ t
    key = ⊥
    • return c

Nothing on key

I can get a bit of key!

无限-位 1-位

- 无限-位 1-位 无限-位

- 1-位 无限-位

- 同样不适合强大的攻击者
- p1 对于1-位攻击者是安全的
**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

---

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 $m_0$

---

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.

source

transformed

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:

$$\forall (m_0, t_1, t_2). \ \forall n. \ \exists \omega \in \Omega(t_1, t_2). \ \forall o_2. \ \mathcal{K}_n^t(p_1, \omega(o_2)) \subseteq \mathcal{K}_n^t(p_2, o_2)$$
A transformation from $p_1$ to $p_2$ is IFP iff:

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

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

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

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

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

For attackers with any observation capabilities
IFP TRANSFORMATION (2/2)

A transformation from $p_1$ to $p_2$ is IFP iff:

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

 Exists lockstep pairings of observations from $t_2$ to $t_1$
A transformation from $p_1$ to $p_2$ is IFP iff:

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

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

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

\( \mathcal{K}_1 \) derived from \( \omega(o_2) \) is a subset of \( \mathcal{K}_2 \) derived from \( o_2 \)
TRANSLATION VALIDATION FOR REGISTER ALLOCATION
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
```
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 = r1 + r1
    stack_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
build pairings from address of $p_2$ to address/constant

\[
\begin{align*}
def p1(k, t, salt): & \\
& \bullet \text{ tmp } = t + \text{ salt} \\
& \quad \text{ } k = \text{ tmp } + k \\
& \bullet \text{ return } k
\end{align*}
\]

\[
\begin{align*}
def p2(r1, r2, stack\_salt): & \\
& \bullet \text{ stack}\_k = r1 \\
& \quad \text{ } r1 = stack\_salt \\
& \quad \text{ } r1 = r2 + r1 \\
& \quad \text{ } r2 = stack\_k \\
& \quad \text{ } r2 = r1 + r2 \\
& \bullet \text{ return } r2
\end{align*}
\]
Computingpairingsfromaddressof$p_2$toaddress/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
```

- $k \leftarrow r_1$
- $t \leftarrow r_2$
- $salt \leftarrow stack\_salt$
- $k \leftarrow stack\_k$
**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
salt ← r1
t ← 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
    tmp = t + salt
    k = tmp + k
    # return k
    return k
```

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

```python
tmp ← r1
mit ← 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
```

```
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
```

\[
\text{tmp} \leftarrow r1 \\
\text{k} \leftarrow r2 \\
\text{salt} \leftarrow \text{stack\_salt} \\
? \leftarrow \text{stack\_k}
\]
build pairings from address of $p_2$ to address/constant

def $p_1(k,t,salt)$:
  • $tmp = t + salt$
  • $k = tmp + k$
  • return $k$

def $p_2(r_1,r_2,stack\_salt)$:
  • $stack\_k = r_1$
  • $r_1 = stack\_salt$
  • $r_1 = r_2 + r_1$
  • $r_2 = stack\_k$
  • $r_2 = r_1 + r_2$
  • return $r_2$

Leakage

tmp ← $r_1$
k ← $r_2$
salt ← $stack\_salt$
？ ← $stack\_k$
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

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

- Time overhead
- Executed instructions overhead

Percentage

Executed instructions overhead

- fannkuch
- fftw
- nsieve
- mandelbrot
- bisect
- vmach
- aes
- nbody
- sha1
- binarytrees
- siphash24
- fft
- spectral
- sha3
- lists
- knucleotide
- chomp
- nsieve
- fib
- fftsp
- almbench
- perlin
RELATED WORK AND CONCLUSION
RELATED WORK

- 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 4}
  - 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]
\textsuperscript{4}Formal verification of a constant-time preserving C compiler, Barthe et al. [2020]
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!
⇒ Optimisations need to be carefully reviewed

An opportunity for secure (and verified) compilation?