Secure Optimization Through Opaque Observations

Son Tuan Vu    Karine Heydemann
Sorbonne Université
Laboratoire d’Informatique de Paris 6

Arnaud de Grandmaison
Arm

Albert Cohen
Google AI

16 March 2021
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```
secret_key ⊕ m

m ←

secret_key (leaked)

n ←

secret_key ⊕ n

secret_key ⊕ m

n ←

secret_key ⊕ m ⊕ n

m ←

secret_key ⊕ n
```
Assuming a functionally-correct, well-defined program

Mismatch between

1. Behavior intended by the programmer (source code)
2. What is actually executed by the processor (machine code)

Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = (mk ^ n) ^ m;
    return mk;
}
```
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = (mk ^ n) ^ m;
    return mk;
}
```

- Re-masking of secret key with new mask $n$
- De-masking of old mask $m$

SECRET KEY $\oplus$ $m$
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk += (mk ^ n) ^ m;
    return mk;
}
```

Underlying property of protection: Re-masking before De-masking

De-masking of old mask \( m \)
Re-masking of secret key with new mask \( n \)
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = (mk ^ n) ^ m;
    return mk;
}
```

Expression reordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = (mk ^ m) ^ n;
    return mk;
}
```
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = (mk ^ n) ^ m;
    return mk;
}
```

```
int mask_swap(int mk, int m) {
    int n = rand();
    mk = (mk ^ m) ^ n;
    return mk;
}
```

*Property not respected*

*Expression reordering*
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

Use of temporary variable to fix evaluation order
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

Temporary variable optimized out + Expression reordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = mk ^ m ^ n;
    return mk;
}
```
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

```c
int mask_swap(int mk, int m) {
    int n = rand();
    mk = mk ^ m ^ n;
    return mk;
}
```

*Temporary variable optimized out + Expression reordering*
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

Coding trick: volatile + asm

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

```c
int mask_swap(int mk, int m) {
    int n = rand();
    volatile int tmp = mk ^ n;
    __asm__ __volatile__
        ("":::"memory");
    mk = tmp ^ m;
    return mk;
}
```
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing
  (so that observable results are statistically uncorrelated to secret data)

Coding trick: `volatile + asm`

Fragile and not portable: `volatile int` may be ignored

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

```c
int mask_swap(int mk, int m) {
    int n = rand();
    volatile int tmp = mk ^ n;
    __asm__ __volatile__(""
        : "memory"
    :
    :
    );
    mk = tmp ^ m;
    return mk;
}
```
Background and Motivation: WYSINWYX phenomenon

- Assuming a functionally-correct, well-defined program
- Mismatch between
  1. Behavior intended by the programmer (source code)
  2. What is actually executed by the processor (machine code)
- Open issue for security engineering: e.g. cryptographic mask changing (so that observable results are statistically uncorrelated to secret data)

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

How to reliably prevent the compiler from optimizing out `tmp` thus respect the evaluation order?
Problem Statement

Approach: make the underlying properties of security countermeasures explicit and instruct the compiler to preserve it

Objective: preserving properties throughout the optimizing compilation flow

Constraint: aim for the least intrusive mechanism in order to implement in production compilers
Property Preservation: Intuition and Challenges

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = observe(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = observe(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = observe(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = observe(mk ^ n);
    mk = tmp ^ m;
    return mk;
}

- Observation semantics?

- Constraints induced by observations on program transformations?

- Preservation of observations and induced constraints: how to make them transformation-independent?
Program Operational Semantics

- State $\sigma = (\{SSAV\text{Values}, References, Memory\}, \text{ProgramCounter})$

- Event $e = \sigma \xleftarrow{i} \sigma', i = \text{Inst}(e)$

- Program semantics $C[P]() = \text{function mapping inputs to outputs}$

- Input and output operations are conducted through I/O events

- I/O events from the same I/O stream are totally ordered

- Execution for input $I \mathcal{E}[P](I) = \sigma_0 e_0 \sigma_1 e_1 \sigma_2 \cdots$

  $\Rightarrow$ induces a partial ordering relation $\rightarrow_{io}$ on I/O events
Observation Semantics

• Observation is event associated with the execution of instruction
  \text{snapshot}(v_1, v_2, \ldots, v_n)
  \rightarrow \text{captures the observed values } v_1, v_2, \ldots, v_n \text{ into a partial observation state}
  \rightarrow \text{can be traced down to machine code for verification, debugging, monitoring, etc.}
Observation Semantics

- Observation is event associated with the execution of instruction
  \texttt{snapshot(v1, v2, \ldots, vn)}
  \implies captures the \textit{observed values} v1, v2, \ldots, vn into a \textit{partial observation state}
  \implies can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
Observation Semantics

- Observation is event associated with the execution of instruction
  `snapshot(v1, v2, ..., vn)`
  → captures the *observed values* $v_1, v_2, ..., v_n$ into a *partial observation state*
  → can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
  - `observe-from $\rightarrow$`: data dependences over events defining observed values and the observation of these values
Observation Semantics

- Observation is an event associated with the execution of an instruction `snapshot(v1, v2, ..., vn)` → captures the *observed values* v1, v2, ..., vn into a *partial observation state*

  → can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
  - `observe-from` →: data dependences over events defining observed values and the observation of these values
  - `observation ordering` →: data or control dependences over observations

<table>
<thead>
<tr>
<th>(1)</th>
<th><code>a = b ^ c</code>; <strong>observe-from</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td><code>snapshot(a)</code>; <strong>observation ordering</strong></td>
</tr>
<tr>
<td>(3)</td>
<td><code>a = a + 42</code>;</td>
</tr>
<tr>
<td>(4)</td>
<td><code>snapshot(a)</code>;</td>
</tr>
</tbody>
</table>
Observation Semantics

- Observation is event associated with the execution of instruction `snapshot(v1, v2, ..., vn)`
  → captures the *observed values* `v1, v2, ..., vn` into a *partial observation state*
  → can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
  - `observe-from \( \rightarrow_f \)`: data dependences over events defining observed values and the observation of these values
  - `observation ordering \( \rightarrow_o \)`: data or control dependences over observations

- Observation preservation = preserving partial states, \( \rightarrow_f \) and \( \rightarrow_o \)
  → preserving observations induces additional constraints on program transformations
Program Transformations

- Transformation $\tau$ induces an event map $\propto_\tau$ relating events before and after transformation.

- Valid transformation preserves program semantics $C[P]() = C[\tau(P)]()$ (i.e. preserves I/O events and their partial ordering relations $\overset{\text{io}}{\rightarrow}$).
Transformation $\tau$ induces an event map $\propto_\tau$ relating events before and after transformation.

Valid transformation preserves program semantics $C[P]() = C[\tau(P)]()$ (i.e. preserves I/O events and their partial ordering relations $\rightarrow_{io}$)

Assuming the compiler implements valid transformations, how to make them observation-preserving (i.e. preserving partial states, $\rightarrow_{of}$ and $\rightarrow_{oo}$)?
Opacification is event associated with the execution of instruction 
\[ v_1' = \text{opacify}(v_1, v_2, \ldots, v_n) \]
→ captures the *observed values* \( v_1, v_2, \ldots, v_n \) into a *partial observation state*
→ returns a value \( v_1' = v_1 \), but the compiler does not know about it

\[ v_1' \] opaque to program analyses and transformations
  → compiler sees a statically unknown yet functionally deterministic value
  → compiler does not assume any relation with the original value \( v_1 \)
Transformed Opacification

Given a program $P$, an input $I$, an opacification $e_{op} \in \mathcal{E}[P](I)$, $\text{Inst}(e_{op}) = (v_1' = \text{opacify}(v_1, \ldots, v_n))$, and a valid transformation $\tau$. Let $\xrightarrow{\text{dep}}$ denote a data or control dependence relation between two events.

Given an event $e \in \mathcal{E}[P](I)$ such that $e_{op} \xrightarrow{\text{dep}} e$. 

Transformed Opacification

Given a program $P$, an input $I$, an opacification $e_{op} \in \mathcal{E}[P](I)$, $\text{Inst}(e_{op}) = (v1' = \text{opacify}(v1, \ldots, vn))$, and a valid transformation $\tau$. Let $\xrightarrow{\text{dep}}$ denote a data or control dependence relation between two events.

Let

Given an event $e \in \mathcal{E}[P](I)$ such that $e_{op} \xrightarrow{\text{dep}} e$.

1. $\exists e' \in \mathcal{E}[\tau(P)](I),\ e \propto_{\tau} e' \implies \exists e'_{op} \in \mathcal{E}[\tau(P)](I),\ e_{op} \propto_{\tau} e'_{op} \land e'_{op} \xrightarrow{\text{dep}} e'$

preservation of $e$ dependent on $e_{op}$ implies preservation of $e_{op}$
Transformed Opacification

Given a program $P$, an input $I$, an opacification $e_{op} \in \mathcal{E}[P](I)$, $\text{Inst}(e_{op}) = (v_1' = \text{opacify}(v_1, \ldots, v_n))$, and a valid transformation $\tau$. Let $\xrightarrow{\text{dep}}$ denote a data or control dependence relation between two events.

Let an event $e \in \mathcal{E}[P](I)$ such that $e_{op} \xrightarrow{\text{dep}} e$.

1. $\exists e' \in \mathcal{E}[\tau(P)](I), \ e \xrightarrow{\tau} e' \implies \exists e'_{op} \in \mathcal{E}[\tau(P)](I), \ e_{op} \xrightarrow{\tau} e'_{op} \land e'_{op} \xrightarrow{\text{dep}} e'$
   
   preservation of $e$ dependent on $e_{op}$ implies preservation of $e_{op}$

2. $\exists e'_{op} \in \mathcal{E}[\tau(P)](I), \ e_{op} \xrightarrow{\tau} e'_{op} \implies e'_{op}$ is also an opacification

if preserved, opacifications are always transformed into opacifications
Transformed Opacification

Given a program $P$, an input $I$, an opacification $e_{op} \in \mathcal{E}[P](I)$, $\text{Inst}(e_{op}) = (v1' = \text{opacify}(v1, \ldots, vn))$, and a valid transformation $\tau$. Let $\rightarrow_{\text{dep}}$ denote a data or control dependence relation between two events.

Given an event $e \in \mathcal{E}[P](I)$ such that $e_{op} \rightarrow_{\text{dep}} e$.

1. $\exists e' \in \mathcal{E}[\tau(P)](I), e \propto_{\tau} e' \implies \exists e'_{op} \in \mathcal{E}[\tau(P)](I), e_{op} \propto_{\tau} e'_{op}$ and $e'_{op} \rightarrow_{\text{dep}} e'$
   preservation of $e$ dependent on $e_{op}$ implies preservation of $e_{op}$

2. $\exists e'_{op} \in \mathcal{E}[\tau(P)](I), e_{op} \propto_{\tau} e'_{op} \implies e'_{op}$ is also an opacification
   if preserved, opacifications are always transformed into opacifications

3. $\exists e'_{op} \in \mathcal{E}[\tau(P)](I), e_{op} \propto_{\tau} e'_{op} \implies v1, \ldots, vn$ are also preserved in $\tau(P)$
   all values used by opacification (i.e. observed values) are always preserved
Transformed Opacification

Given a program $P$, an input $I$, an opacification $e_{op} \in E[P](I)$, $\text{Inst}(e_{op}) = (v_1' = \text{opacify}(v_1, \ldots, v_n))$, and a valid transformation $\tau$. Let $\xrightarrow{\text{dep}}$ denote a data or control dependence relation between two events.

Let $\xrightarrow{\text{dep}}$ denote a data or control dependence relation between two events.

Given an event $e \in E[P](I)$ such that $e_{op} \xrightarrow{\text{dep}} e$.

1. $\exists e' \in E[\tau(P)](I), e \propto_\tau e' \implies \exists e'_{op} \in E[\tau(P)](I), e_{op} \propto_\tau e'_{op} \land e'_{op} \xrightarrow{\text{dep}} e'$
   
   preservation of $e$ dependent on $e_{op}$ implies preservation of $e_{op}$

2. $\exists e'_{op} \in E[\tau(P)](I), e_{op} \propto_\tau e'_{op} \implies e'_{op}$ is also an opacification
   
   if preserved, opacifications are always transformed into opacifications

3. $\exists e'_{op} \in E[\tau(P)](I), e_{op} \propto_\tau e'_{op} \implies v_1, \ldots, v_n$ are also preserved in $\tau(P)$
   
   all values used by opacification (i.e. observed values) are always preserved

$\implies$ properties directly induced by the definition of “opacity”
Opaque Chain:

- Used to enforce opacification preservation
  ⇒ preserving observations and partial states

- Used to enforce opacification ordering preservation
  ⇒ preserving $\circ_f$ and $\circ\circ$ relations
Opaque Chain:

- Used to enforce opacification preservation
  ⇒ preserving observations and partial states

- Used to enforce opacification ordering preservation
  ⇒ preserving $\circ f$ and $\circ\circ$ relations

→ Opaque Chain = Opacifications in Dependence Chain + Opacity-Preserving Instruction
Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation
  ⇒ preserving observations and partial states
- Used to enforce opacification ordering preservation
  ⇒ preserving $\rightarrow^f$ and $\rightarrow^\circ$ relations

$\rightarrow$ Opaque Chain = Opacifications in Dependence Chain
  + Opacity-Preserving Instruction

```c
int main() {
    int a = get_int();
    int opaque_a = opacify(a);
    int b = opaque_a + 1;
    return b;
}
```
Opaque Chain:

- Used to enforce opacification preservation
  \[ \Rightarrow \text{preserving observations and partial states} \]
- Used to enforce opacification ordering preservation
  \[ \Rightarrow \text{preserving } \rightarrow_{\text{of}} \text{ and } \rightarrow_{\text{oo}} \text{ relations} \]

\[ \rightarrow \text{Opaque Chain} = \text{Opacifications in Dependence Chain} + \text{Opacity-Preserving Instruction} \]

```c
int main() {
    int a = get_int();
    int opaque_a = opacify(a);
    int b = opaque_a + 1;
    return b;
}
```
Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation
  \[ \Rightarrow \text{preserving observations and partial states} \]

- Used to enforce opacification ordering preservation
  \[ \Rightarrow \text{preserving } \circ_f \text{ and } \circ_0 \text{ relations} \]

\[ \rightarrow \text{Opaque Chain} = \text{Opacifications in Dependence Chain} \]
\[ + \text{Opacity-Preserving Instruction} \]

```c
int main() {
    int a = get_int();
    int opaque_a = opacify(a);
    int b = opaque_a * 0;
    return b;
}
```
Opaque Chain:

- Used to enforce opacification preservation
  \[ \Rightarrow \text{preserving observations and partial states} \]

- Used to enforce opacification ordering preservation
  \[ \Rightarrow \text{preserving } \xrightarrow{\text{of}} \text{ and } \xrightarrow{\text{o}} \text{ relations} \]

\[ \rightarrow \text{Opaque Chain } = \text{Opacifications in Dependence Chain} \]
\[ + \text{Opacity-Preserving Instruction} \]
Opaque Chain:

- Used to enforce opacification preservation
  \[ \Rightarrow \text{preserving observations and partial states} \]

- Used to enforce opacification ordering preservation
  \[ \Rightarrow \text{preserving } \circ \text{ and } \circ \circ \text{ relations} \]

\[ \rightarrow \text{Opaque Chain} = \text{Opacifications in Dependence Chain} \]
\[ + \text{Opacity-Preserving Instruction} \]

\[ \rightarrow \text{If the tailing instruction is preserved, the opaque chain will also be preserved} \]

Opaque Chain preserved \[ \Rightarrow \text{Opacifications } + \text{Ordering preserved} \]
Putting it to Work

Implementation in latest LLVM with minimal changes to individual passes → transformation-independent and future-proof mechanism

Observation extra info = LLVM metadata

→ no additional instructions generated in machine code
Applications

- Enforcing countermeasures requiring value preservation
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int res = a + 42;
    return res;
}
```

Redundant computation, commonly-used technique against fault injections
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = a;
    int res = a + 42;

    return res;
}
```

Redundant computation, commonly-used technique against fault injections
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = a;
    int res = a + 42;
    int res_dup = a_dup + 42;

    return res;
}
```

Redundant computation, commonly-used technique against fault injections
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = a;
    int res = a + 42;
    int res_dup = a_dup + 42;
    if (res != res_dup)
        fault_handler();
    return res;
}
```

Redundant computation, commonly-used technique against fault injections
Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = opacify(a);
    int res = a + 42;
    int res_dup = a_dup + 42;
    if (res != res_dup)
        fault_handler();
    return res;
}
```

Redundant computation, commonly-used technique against fault injections
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = opacify(a);
    int res = a + 42;
    int res_dup = a_dup + 42;
    if (res != res_dup)
        fault_handler();
    return res;
}
```

```c
int ct_sel(bool b, int x, int y) {
    return b ? x : y;
}
```

Selecting between two values without jump conditioned by secret value

Redundant computation, commonly-used technique against fault injections
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = opacify(a);
    int res = a + 42;
    int res_dup = a_dup + 42;
    if (res != res_dup)
        fault_handler();
    return res;
}
```

```c
int ct_sel(bool b, int x, int y) {
    signed m = 0 - b;
    return (x & m) | (y & ~m);
}
```

Selecting between two values without jump conditioned by secret value

Redundant computation, commonly-used technique against fault injections
Applications

- Enforcing countermeasures requiring value preservation

```c
int redundant_add(int a) {
    int a_dup = opacify(a);
    int res = a + 42;
    int res_dup = a_dup + 42;
    if (res != res_dup)
        fault_handler();
    return res;
}
```

Redundant computation, commonly-used technique against fault injections

```c
int ct_sel(bool b, int x, int y) {
    signed m = opacify(0 - b);
    return (x & m) | (y & ~m);
}
```

Selecting between two values without jump conditioned by secret value
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering
Applications

- Enforcing countermeasures requiring value preservation

- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = mk ^ n;
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations

```c
int add(int x, int y) {
    int res = x;
    res += y;
    return res;
}
```

Enforcing proper interleaving of counter incrementation and original code
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations

```c
int add(int x, int y) {
    int cnt = 0;
    int res = x;
    res += y;
    return res;
}
```

Enforcing proper interleaving of counter incrementation and original code
**Applications**

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations

```c
int add(int x, int y) {
    int cnt = 0;
    int res = x;
    cnt++;
    res += y;
    cnt++;
    return res;
}
```

Enforcing proper interleaving of counter incrementation and original code
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations

```c
int add(int x, int y) {
    int cnt = 0;
    int res = x;
    cnt++;
    res += y;
    cnt++;
    if (cnt != 2)
        fault_handler();
    return res;
}
```

Enforcing proper interleaving of counter incrementation and original code
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations

```c
int add(int x, int y) {
    int cnt = 0;
    int res = opacify(x, cnt);
    cnt = opacify(cnt, res) + 1;
    res = opacify(res, cnt) + opacify(y, cnt);
    cnt = opacify(cnt, res) + 1;
    if (cnt != 2)
        fault_handler();
    return res;
}
```

Enforcing proper interleaving of counter incrementation and original code
Applications

- Enforcing countermeasures requiring value preservation
- Enforcing computation ordering

```c
int mask_swap(int mk, int m) {
    int n = rand();
    int tmp = opacify(mk ^ n);
    mk = tmp ^ m;
    return mk;
}
```

Enforcing specific evaluation order of associative operations

```c
int add(int x, int y) {
    int cnt = 0;
    int res = opacify(x, cnt);
    cnt = opacify(cnt, res) + 1;
    res = opacify(res, cnt)
    + opacify(y, cnt);
    cnt = opacify(cnt, res) + 1;
    if (cnt != 2)
        fault_handler();
    return res;
}
```

Enforcing proper interleaving of counter incrementation and original code
## Validation and Performance Evaluation

<table>
<thead>
<tr>
<th>Attack</th>
<th>Side-channel</th>
<th>Data remanence</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Masking of secret data</td>
<td>Inserting code to erase secret data</td>
<td>Inserting redundant data and/or protection code</td>
</tr>
<tr>
<td></td>
<td>Constant-time selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Instruction ordering in masking operations</td>
<td>Presence of sensitive memory data erasure</td>
<td>Interleaving of functional and protection code</td>
</tr>
<tr>
<td></td>
<td>No jump conditioned by secret value</td>
<td></td>
<td>Presence of redundant data detecting fault injections</td>
</tr>
<tr>
<td>Application</td>
<td>mask-aes mask-swap</td>
<td>erasure-rsa-enc erasure-rsa-dec</td>
<td>sci-pin sci-aes</td>
</tr>
<tr>
<td></td>
<td>ct-rsa ct-montgomery</td>
<td></td>
<td>loop-pin</td>
</tr>
</tbody>
</table>

### Targets:
x86-64 + ARMv7-M/Thumb-2, compiled at -01/2/3/s/z
Validation and Performance Evaluation

<table>
<thead>
<tr>
<th>Attack</th>
<th>Side-channel</th>
<th>Data remanence</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Masking of secret data</td>
<td>Constant-time selection</td>
<td>Inserting redundant data and/or protection code</td>
</tr>
<tr>
<td>Property</td>
<td>Instruction ordering in</td>
<td>No jump conditioned by secret value</td>
<td>Interleaving of functional and protection code</td>
</tr>
<tr>
<td></td>
<td>masking operations</td>
<td></td>
<td>Presence of redundant data detecting fault injections</td>
</tr>
<tr>
<td>Application</td>
<td>mask-aes, mask-swap</td>
<td>ct-rsa, ct-montgomery</td>
<td>sci-pin, sci-aes</td>
</tr>
<tr>
<td></td>
<td>erasure-rsa-enc, erasure-rsa-dec</td>
<td></td>
<td>loop-pin</td>
</tr>
</tbody>
</table>

Targets: x86-64 + ARMv7-M/Thumb-2, compiled at -01/2/3/s/z

- Validation:
  - automated checking of observation integrity and ordering
  - manual inspection of security countermeasure integrity
### Validation and Performance Evaluation

<table>
<thead>
<tr>
<th>Attack</th>
<th>Side-channel</th>
<th>Data remanence</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Masking of secret data</td>
<td>Inserting code to erase secret data</td>
<td>Inserting redundant data and/or protection code</td>
</tr>
<tr>
<td>Property</td>
<td>Instruction ordering in masking operations</td>
<td>Presence of sensitive memory data erasure</td>
<td>Interleaving of functional and protection code Presence of redundant data detecting fault injections</td>
</tr>
<tr>
<td>Application</td>
<td>mask-aes mask-swap</td>
<td>erasure-rsa-enc erasure-rsa-dec</td>
<td>sci-pin sci-aes loop-pin</td>
</tr>
</tbody>
</table>

**Targets:** x86-64 + ARMv7-M/Thumb-2, compiled at -01/2/3/s/z

- **Validation:**
  - automated checking of observation integrity and ordering
  - manual inspection of security countermeasure integrity

- **Performance Evaluation:** comparison with other solutions:
### Validation and Performance Evaluation

<table>
<thead>
<tr>
<th>Attack</th>
<th>Side-channel</th>
<th>Data remanence</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Masking of secret data</td>
<td>Constant-time selection</td>
<td>Inserting redundant data and/or protection code</td>
</tr>
<tr>
<td>Property</td>
<td>Instruction ordering in</td>
<td>No jump conditioned by secret value</td>
<td>Interleaving of functional and protection code</td>
</tr>
<tr>
<td></td>
<td>masking operations</td>
<td></td>
<td>Presence of redundant data detecting fault injections</td>
</tr>
<tr>
<td>Application</td>
<td>mask-aes mask-swap</td>
<td>ct-rsa ct-montgomery</td>
<td>sci-pin sci-aes</td>
</tr>
<tr>
<td></td>
<td>erasure-rsa-enc</td>
<td>erasure-rsa-dec</td>
<td>loop-pin</td>
</tr>
</tbody>
</table>

**Targets:** x86-64 + ARMv7-M/Thumb-2, compiled at -01/2/3/s/z

- **Validation:**
  - automated checking of observation integrity and ordering
  - manual inspection of security countermeasure integrity

- **Performance Evaluation:** comparison with other solutions:
  - unoptimized code → speedup with harmonic mean of 2.8
## Validation and Performance Evaluation

<table>
<thead>
<tr>
<th>Attack</th>
<th>Side-channel</th>
<th>Data remanence</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection</td>
<td>Masking of secret data</td>
<td>Constant-time selection</td>
<td>Inserting code to erase secret data</td>
</tr>
<tr>
<td>Property</td>
<td>Instruction ordering in masking operations</td>
<td>No jump conditioned by secret value</td>
<td>Presence of sensitive memory data erasure</td>
</tr>
</tbody>
</table>

**Targets:** x86-64 + ARMv7-M/Thumb-2, compiled at `-01/2/3/s/z`

- **Validation:**
  - automated checking of observation integrity and ordering
  - manual inspection of security countermeasure integrity

- **Performance Evaluation:** comparison with other solutions:
  - unoptimized code $\rightarrow$ speedup with harmonic mean of 2.8
  - embedding I/O effects into observation intrinsics to guarantee their preservation $\rightarrow$ speedup with harmonic mean of 1.3
Conclusion

- Transformation-independent and future-proof mechanism to preserve security countermeasures through optimizing compilation

- Formal model of opaque observations and their preservation

- Stronger guarantees and higher performance than current practice

- Perspective: contribute this work to the community and build a compilation framework upon