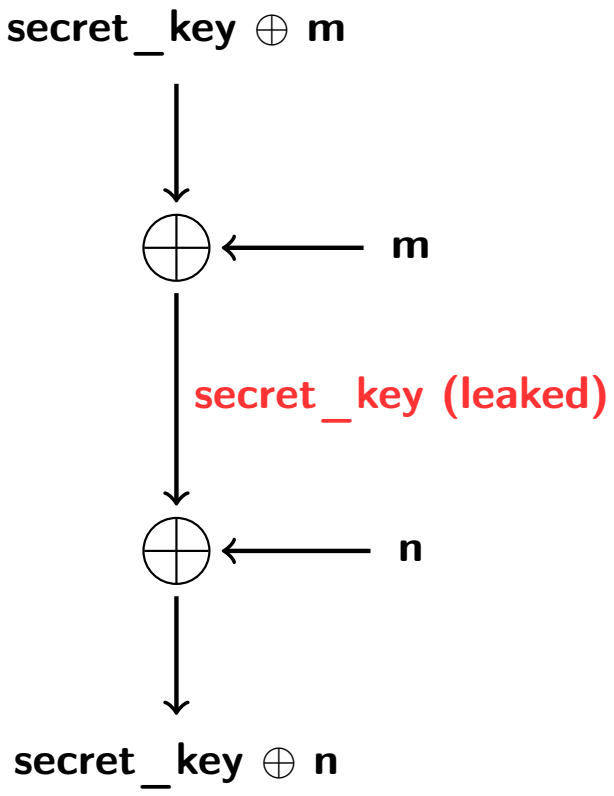
Secure Optimization Through Opaque Observations

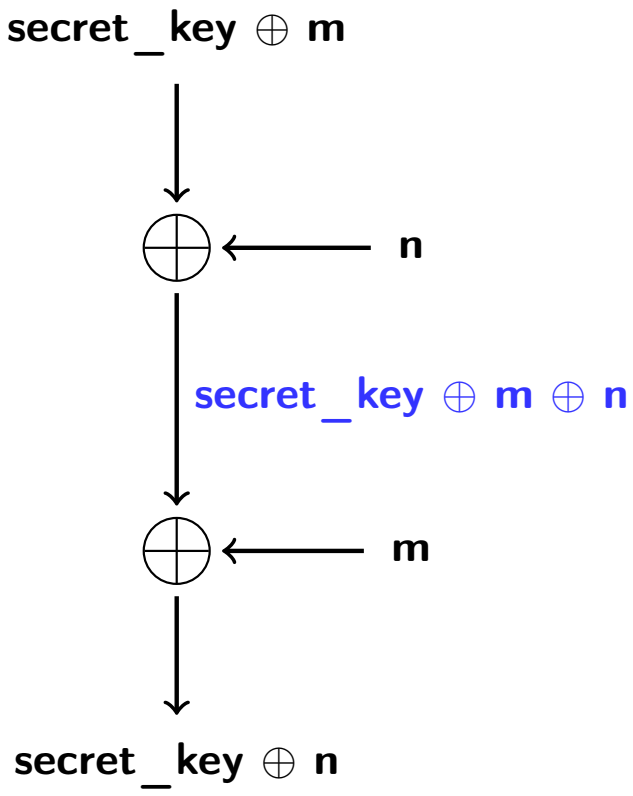
Son Tuan Vu **Karine Heydemann**
Sorbonne Université
Laboratoire d'Informatique de Paris 6
Arnaud de Grandmaison
Arm
Albert Cohen
Google AI

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)





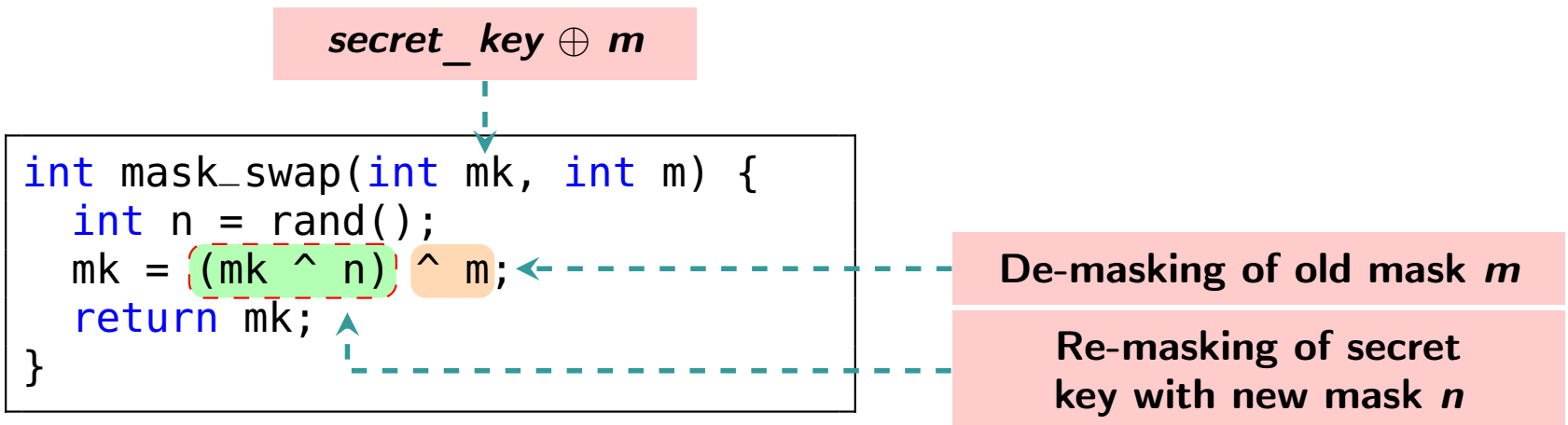
Background and Motivation: WYSINWYX phenomenon

- 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)

```
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)



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)

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

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

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)

Expression reordering

```
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;
}
```

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)

Property not respected

Expression reordering

```
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;
}
```

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)

```
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)

Temporary variable optimized out
+
Expression reordering

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

```
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)

Property not respected

Temporary variable optimized out
+
Expression reordering

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

```
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)

Coding trick: *volatile* + *asm*

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

```
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

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

```
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)

```
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

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

Property Preservation: Intuition and Challenges

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

Capture the wanted value

Property Preservation: Intuition and Challenges

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

Capture the wanted value

Make sure it is used next

Property Preservation: Intuition and Challenges

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

Capture the wanted value

Make sure it is used next

- 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 $\sigma = \big(\{\mathit{SSAValues}, \mathit{References}, \mathit{Memory}\}, \mathit{ProgramCounter}\big)$
- Event $e = \sigma \overset{\mathrm{i}}{\rightsquigarrow} \sigma'$, $i = \mathit{Inst}(e)$
- Program semantics $\mathcal{C}[\![P]\!]() =$ 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 $\mathcal{E}[\![P]\!](I) = \sigma_0 e_0 \sigma_1 e_1 \sigma_2 \ldots$

  $\Rightarrow$ induces a partial ordering relation $\overset{\mathrm{io}}{\rightarrow}$ on I/O events

Observation Semantics

- Observation is event associated with the execution of instruction `snapshot(v1, v2, ..., vn)`

  $\rightarrow$ captures the *observed values* `v1`, `v2`, ..., `vn` into a *partial observation state*

  $\rightarrow$ can be traced down to machine code for verification, debugging, monitoring, etc.

Observation Semantics

- Observation is event associated with the execution of instruction `snapshot(v1, v2, ..., vn)`

  $\rightarrow$ captures the *observed values* `v1`, `v2`, ..., `vn` into a *partial observation state*

  $\rightarrow$ 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)`

  $\rightarrow$ captures the *observed values* `v1`, `v2`, ..., `vn` into a *partial observation state*

  $\rightarrow$ can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
  - observe-from $\overset{\text{of}}{\rightarrow}$: data dependences over events defining observed values and the observation of these values

Observation Semantics

- Observation is event associated with the execution of instruction `snapshot(v1, v2, ..., vn)`

  $\rightarrow$ captures the *observed values* `v1`, `v2`, ..., `vn` into a *partial observation state*

  $\rightarrow$ can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
  - observe-from $\overset{\text{of}}{\rightarrow}$: data dependences over events defining observed values and the observation of these values
  - observation ordering $\overset{\text{oo}}{\rightarrow}$: data or control dependences over observations

```
(1) a = b ^ c;
(2) snapshot(a);
(3) a = a + 42;
(4) snapshot(a);
```

observe-from

observation ordering

Observation Semantics

- Observation is event associated with the execution of instruction `snapshot(v1, v2, ..., vn)`

  $\rightarrow$ captures the *observed values* `v1`, `v2`, ..., `vn` into a *partial observation state*

  $\rightarrow$ can be traced down to machine code for verification, debugging, monitoring, etc.

- Additional relations involving observations:
  - observe-from $\overset{\text{of}}{\rightarrow}$: data dependences over events defining observed values and the observation of these values
  - observation ordering $\overset{\text{oo}}{\rightarrow}$: data or control dependences over observations

- Observation preservation = preserving partial states, $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$

  $\rightarrow$ preserving observations induces additional constraints on program transformations

Program Transformations

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

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

Program Transformations

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

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

Assuming the compiler implements valid transformations, how to make them observation-preserving (i.e. preserving partial states, $\xrightarrow{\text{of}}$ and $\xrightarrow{\text{oo}}$)?

Observation Preservation: Opacification

- Opacification is event associated with the execution of instruction `v1'= opacify(v1, v2, ..., vn)`

  $\rightarrow$ captures the *observed values* `v1`, `v2`, ..., `vn` into a *partial observation state*

  $\rightarrow$ returns a value `v1'= v1`, but the compiler does not know about it

- `v1'` 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 `v1`

Transformed Opacification

Given a program P, an input I, an opacification $e_{op} \in \mathcal{E}[\![P]\!](I)$, $\mathit{Inst}(e_{op}) = (\texttt{v1' = opacify(v1, ..., vn)})$, and a valid transformation τ.

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

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

Transformed Opacification

Given a program P, an input I, an opacification $e_{op} \in \mathcal{E}[\![P]\!](I)$, $Inst(e_{op}) = (\texttt{v1' = opacify(v1, ..., vn)})$, and a valid transformation τ.

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

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

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

preservation of e dependent on eop implies preservation of eop

Transformed Opacification

Given a program P, an input I, an opacification $e_{op} \in \mathcal{E}[\![P]\!](I)$, $Inst(e_{op}) = (\texttt{v1' = opacify(v1, ..., vn)})$, and a valid transformation τ.

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

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

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

preservation of e dependent on eop implies preservation of eop

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

Transformed Opacification

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

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

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

preservation of e dependent on eop implies preservation of eop

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 $e_{op} \in \mathcal{E}[\![P]\!](I)$, $Inst(e_{op}) = (\texttt{v1' = opacify(v1, ..., vn)})$, and a valid transformation τ.
Let $\overset{\text{dep}}{\rightarrow}$ denote a data or control dependence relation between two events.

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

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

preservation of e dependent on eop implies preservation of eop

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

$\Rightarrow$ properties directly induced by the definition of "opacity"

Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$ relations

Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$ relations

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

Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$ relations

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

```
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:

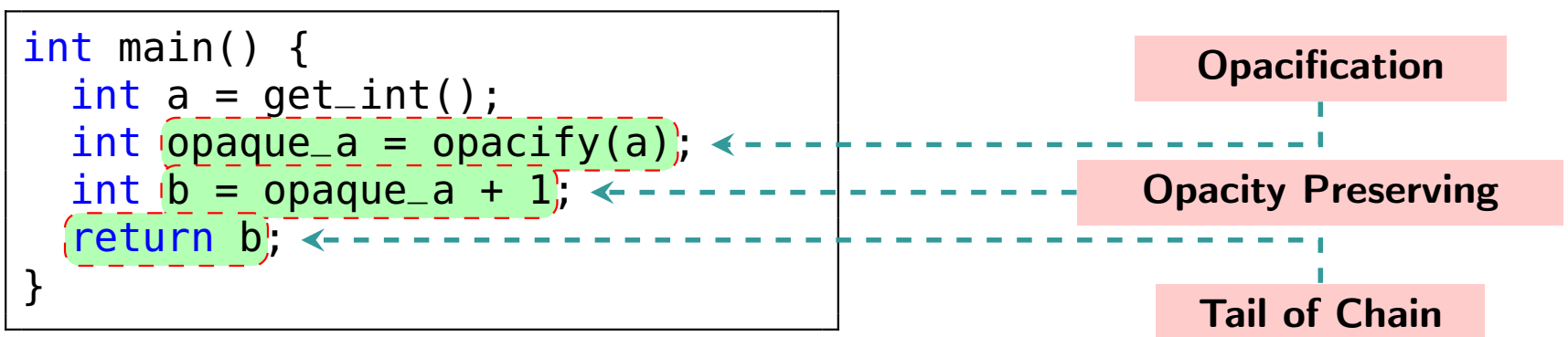
- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$ relations

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



Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$ relations

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

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

Observation Ordering Preservation: Opaque Chains

Opaque Chain:

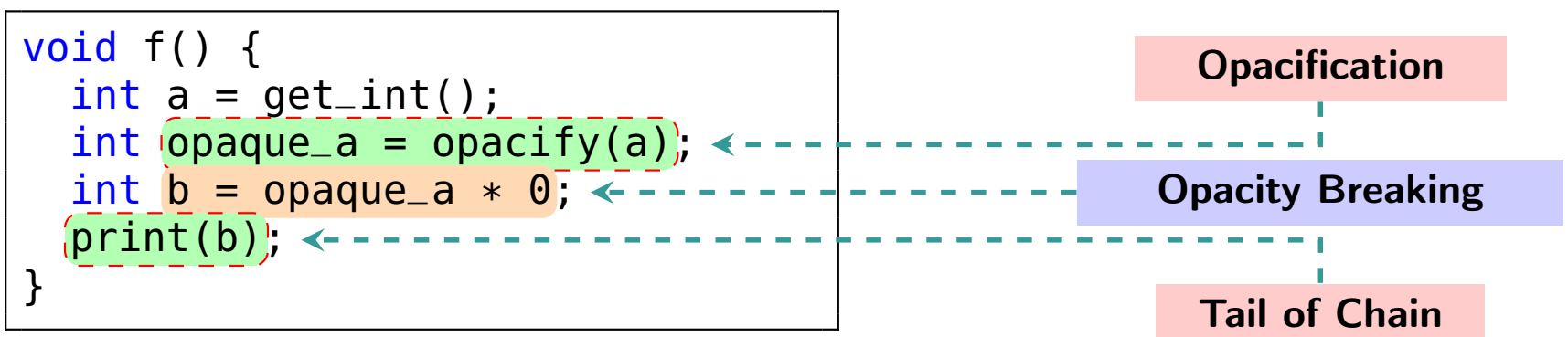
- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\xrightarrow{\text{of}}$ and $\xrightarrow{\text{oo}}$ relations

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



Observation Ordering Preservation: Opaque Chains

Opaque Chain:

- Used to enforce opacification preservation

  $\Rightarrow$ preserving observations and partial states

- Used to enforce opacification ordering preservation

  $\Rightarrow$ preserving $\xrightarrow{\text{of}}$ and $\xrightarrow{\text{oo}}$ relations

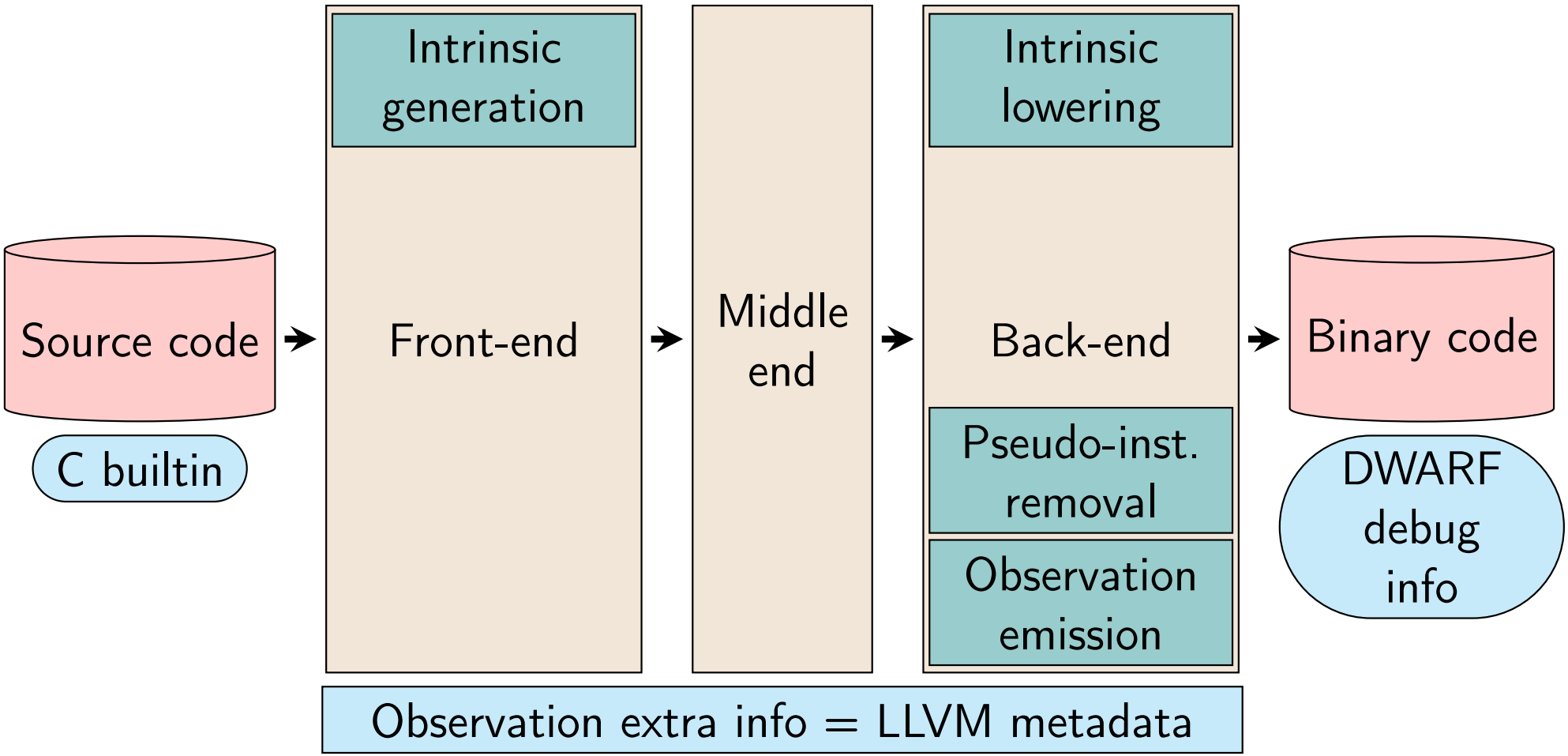
$\rightarrow$ Opaque Chain = Opacifications in Dependence Chain
+ Opacity-Preserving Instruction

$\rightarrow$ **If the tailing instruction is preserved, the opaque chain will also be preserved**

Opaque Chain preserved $\Rightarrow$ Opacifications + Ordering preserved

Putting it to Work

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



$\rightarrow$ no additional instructions generated in machine code

Applications

- Enforcing countermeasures requiring value preservation

Applications

- Enforcing countermeasures requiring value preservation

```
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

```
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

```
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

```
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

Applications

- Enforcing countermeasures requiring value preservation

```
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

```
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

```
int ct_sel(bool b, int x, int y) {

  return b ? x : y;
}
```

Selecting between two values without jump conditioned by secret value

Applications

- Enforcing countermeasures requiring value preservation

```
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

```
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

Applications

- Enforcing countermeasures requiring value preservation

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

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

Targets: `x86-64` + `ARMv7-M/Thumb-2`, compiled at `-O1/2/3/s/z`

Validation and Performance Evaluation

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

Targets: `x86-64` + `ARMv7-M/Thumb-2`, compiled at `-O1/2/3/s/z`

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

Validation and Performance Evaluation

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

Targets: `x86-64` + `ARMv7-M/Thumb-2`, compiled at `-O1/2/3/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

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

Targets: `x86-64` + `ARMv7-M/Thumb-2`, compiled at `-O1/2/3/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

Validation and Performance Evaluation

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

Targets: `x86-64` + `ARMv7-M/Thumb-2`, compiled at `-O1/2/3/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