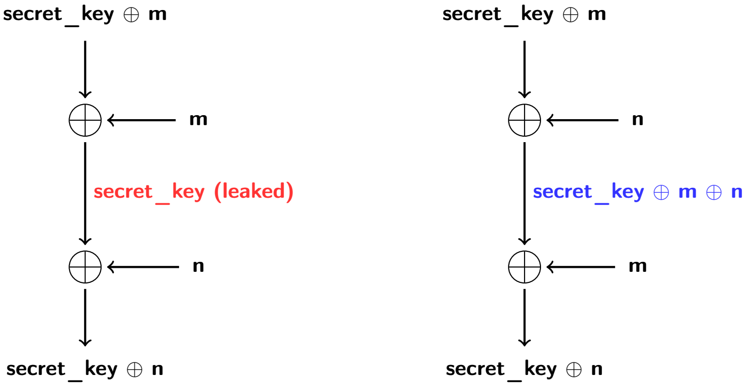
Secure Delivery of Program Properties through Optimizing Compilation

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

24 September 2020

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)

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ n) ^ 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)

secret_key* $\oplus$ *m

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ n) ^ m;
  ...
}
```

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)

Security property:
Re-masking before De-masking

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ n) ^ m;
  ...
}
```

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)

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ n) ^ m;
  ...
}
```

Evaluation reordering

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ m) ^ 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)

Property not respected

Evaluation reordering

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ n) ^ m;
  ...
}
```

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = (*mk ^ m) ^ 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)

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  int tmp = *mk ^ n;
  *mk = tmp ^ m;
  ...
}
```

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
+
Evaluation reordering

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  int tmp = *mk ^ n;
  *mk = tmp ^ m;
  ...
}
```

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = *mk ^ m ^ 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)

Property not respected

Temporary variable optimized out
+
Evaluation reordering

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  int tmp = *mk ^ n;
  *mk = tmp ^ m;
  ...
}
```

```
void compute(int *mk, int m) {
  ...
  int n = rand();

  *mk = *mk ^ m ^ 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)

Coding trick: *volatile* + *asm*

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  int tmp = *mk ^ n;
  *mk = tmp ^ m;
  ...
}
```

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  volatile int tmp = *mk ^ n;
  __asm__ __volatile__
        ("":::"memory");
  *mk = tmp ^ 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)

Coding trick: *volatile* + *asm*

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

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  int tmp = *mk ^ n;
  *mk = tmp ^ m;
  ...
}
```

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  volatile int tmp = *mk ^ n;
  __asm__ __volatile__
        ("":::"memory");
  *mk = tmp ^ 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)

```
void compute(int *mk, int m) {
  ...
  int n = rand();
  int tmp = *mk ^ n;
  *mk = tmp ^ m;
  ...
}
```

How to reliably prevent the compiler from optimizing out *tmp* thus respect the evaluation order?

Background and Motivation: WYSINWYX phenomenon

- Needs for analysis and verification of binary programs [Balakrishnan and Reps, 2010] [Bréjon et al., 2019]

- Needs for program properties in the executable binaries (e.g. countermeasure oracles, ...) [Bréjon et al., 2019]

Background and Motivation: WYSINWYX phenomenon

- Needs for analysis and verification of binary programs [Balakrishnan and Reps, 2010] [Bréjon et al., 2019]

- Needs for program properties in the executable binaries (e.g. countermeasure oracles, ...) [Bréjon et al., 2019]

⇒ Needs for preserving program properties throughout the optimizing compilation flow

Property Preservation Through Compilation: Outline

1. Definition of *property preservation through compilation*
2. Our approach to preserve program properties
3. Implementation of our approach in LLVM
4. Validation of our approach and implementation on security applications

Definitions

Functional Property

A functional property (*Prop*, *ObsPt*) is

- *Prop* a propositional logic formula expressing a program behavioral property
- *ObsPt* an observation point at which *Prop* is expected to hold

Definitions

Functional Property

A functional property (*Prop*, *ObsPt*) is

- *Prop* a propositional logic formula expressing a program behavioral property
- *ObsPt* an observation point at which *Prop* is expected to hold

```
void compute(int *mk, int m) {
  ...
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
}
```

Implicitly equivalent to
"Re-masking before De-masking"

Definitions

Functional Property

A functional property (*Prop*, *ObsPt*) is

- *Prop* a propositional logic formula expressing a program behavioral property
- *ObsPt* an observation point at which *Prop* is expected to hold

ObsPt ← here:

```
void compute(int *mk, int m) {
  ...
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
}
```

Definitions

Functional Property

A functional property (*Prop*, *ObsPt*) is

- *Prop* a propositional logic formula expressing a program behavioral property
- *ObsPt* an observation point at which *Prop* is expected to hold

```
void compute(int *mk, int m) {
  ...
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
}
```

ObsPt ***Prop***

Definitions

Functional Property and Partial State

A functional property (*Prop*, *ObsPt*) defines a partial state (*ObsPt*, *ObsVar*, *ObsMem*):

- *ObsPt* the observation point defined by the property

ObsPt ← here:

```
void compute(int *mk, int m) {
  ...
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
}
```

Definitions

Functional Property and Partial State

A functional property (*Prop*, *ObsPt*) defines a partial state (*ObsPt*, *ObsVar*, *ObsMem*):

- *ObsPt* the observation point defined by the property
- *ObsVar* $= \{(var, val) \mid var$ observed variable occurring in *Prop*$\}$

```
void compute(int *mk, int m) {
  ...
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
}
```

ObsPt

ObsVar:
{(tmp, 4860); (n, 5678)}

Functional Property and Partial State

A functional property (*Prop*, *ObsPt*) defines a partial state (*ObsPt*, *ObsVar*, *ObsMem*):

- *ObsPt* the observation point defined by the property
- $ObsVar = \{(var, val) \mid var$ observed variable occurring in $Prop\}$
- $ObsMem = \{(mem, val) \mid mem$ observed memory location occurring in $Prop\}$

```
void compute(int *mk, int m) {
  ...
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
}
```

ObsPt

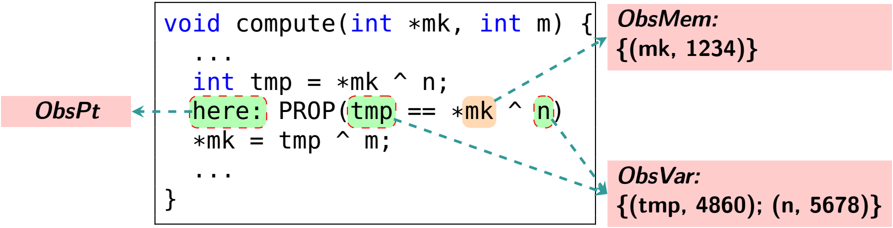
ObsMem:
{(mk, 1234)}

ObsVar:
{(tmp, 4860); (n, 5678)}

Functional Property and Partial State

A functional property (*Prop*, *ObsPt*) defines a partial state (*ObsPt*, *ObsVar*, *ObsMem*):

- *ObsPt* the observation point defined by the property
- *ObsVar* = {(*var*, *val*) | *var* observed variable occurring in *Prop*}
- *ObsMem* = {(*mem*, *val*) | *mem* observed memory location occurring in *Prop*}



Partial State:
(*ObsPt*, *ObsVar*, *ObsMem*)

Definitions

Observation trace

An observation trace is

- the sequence of partial states defined by functional properties
- encountered during a given execution of the program

```c
int main() {
  ...
  compute(mk1, m1);
  compute(mk2, m2);
  compute(mk3, m3);
  ...
}
```

Observation trace:
...
@here: (tmp, 4860); (mk, 5678); (n, 1234)
@here: (tmp, 5171); (mk, 1234); (n, 4321)
@here: (tmp, 1029); (mk, 2187); (n, 3214)
...

Definitions

Functional Property Preservation

A transformation $\tau()$ preserves functional properties of program P if

- P and $\tau(P)$ produce equal observation traces given the same input
- for any input vector

Source Program

Definitions

Functional Property Preservation

A transformation $\tau()$ preserves functional properties of program P if

- P and $\tau(P)$ produce equal observation traces given the same input
- for any input vector

Source Program →

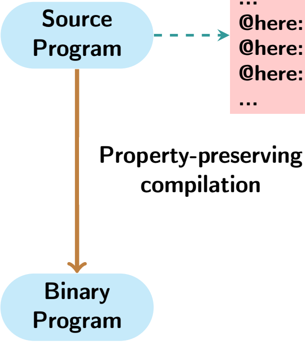
Observation trace:
...
@here: (tmp, 4860); (mk, 5678); (n, 1234)
@here: (tmp, 5171); (mk, 1234); (n, 4321)
@here: (tmp, 1029); (mk, 2187); (n, 3214)
...

Definitions

Functional Property Preservation

A transformation $\tau()$ preserves functional properties of program P if

- P and $\tau(P)$ produce equal observation traces given the same input
- for any input vector



Observation trace:

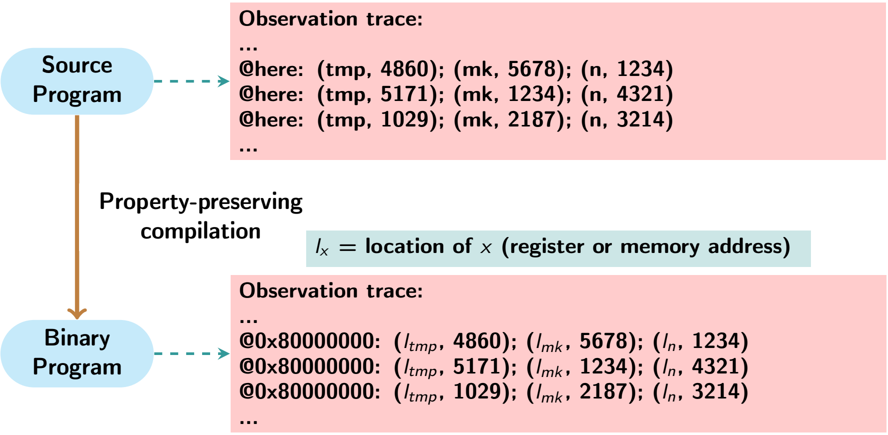
...
@here: (tmp, 4860); (mk, 5678); (n, 1234)
@here: (tmp, 5171); (mk, 1234); (n, 4321)
@here: (tmp, 1029); (mk, 2187); (n, 3214)
...

Definitions

Functional Property Preservation

A transformation $\tau()$ preserves functional properties of program P if

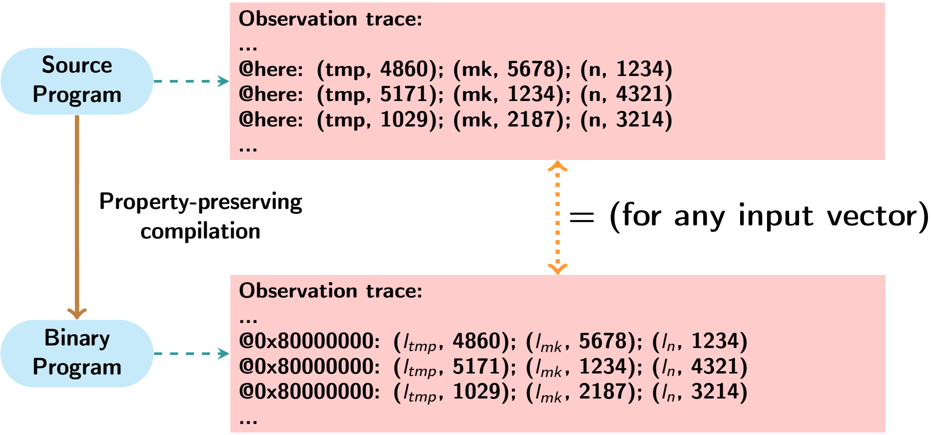
- P and $\tau(P)$ produce equal observation traces given the same input
- for any input vector



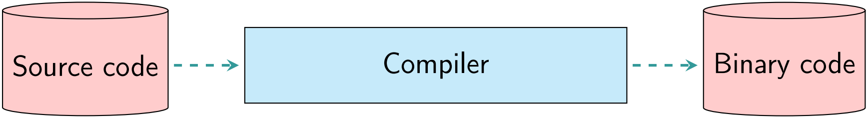
Functional Property Preservation

A transformation $\tau()$ preserves functional properties of program P if

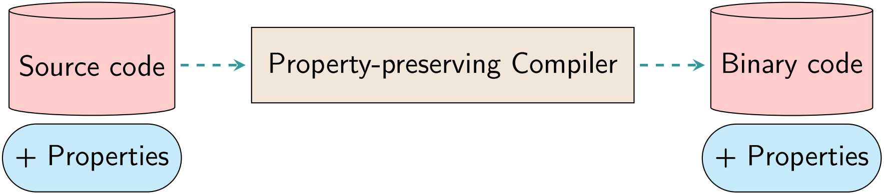
- P and $\tau(P)$ produce equal observation traces given the same input
- for any input vector



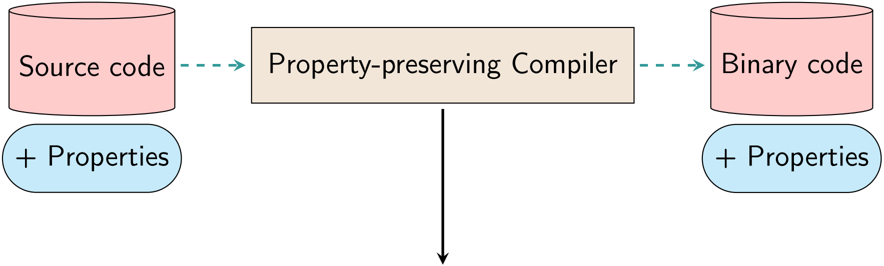
Preserving Properties Through Compilation: Overview



Preserving Properties Through Compilation: Overview

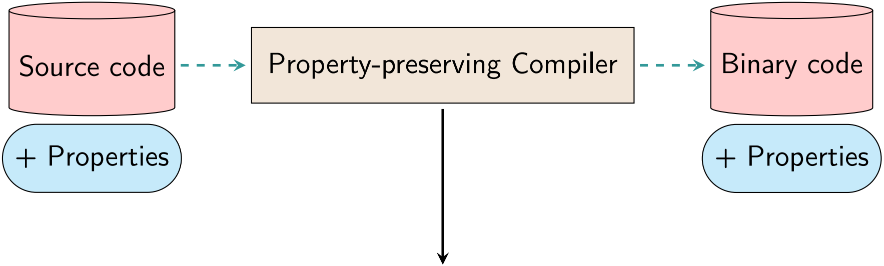


Preserving Properties Through Compilation: Overview



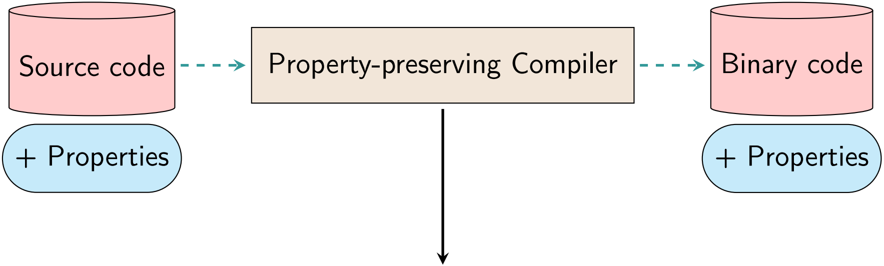
- Existing work: tuning optimization passes one-by-one to teach them about properties [Zarzani, 2013] [Namjoshi and Zuck, 2013] [Namjoshi, Tagliabue, and Zuck, 2013]

Preserving Properties Through Compilation: Overview



- Existing work: tuning optimization passes one-by-one to teach them about properties [Zarzani, 2013] [Namjoshi and Zuck, 2013] [Namjoshi, Tagliabue, and Zuck, 2013]
- Our approach: more generic solution which does not require modifying existing optimizations

Preserving Properties Through Compilation: Overview



- Existing work: tuning optimization passes one-by-one to teach them about properties [Zarzani, 2013] [Namjoshi and Zuck, 2013] [Namjoshi, Tagliabue, and Zuck, 2013]
- Our approach: more generic solution which does not require modifying existing optimizations

⇒ can be implemented in a production compiler (LLVM)

Property Preservation Through Compilation: Outline

1. Definition of *property preservation through compilation*
2. Our approach to preserve program properties
3. Implementation of our approach in LLVM
4. Validation of our approach and implementation on security applications

Preserving Properties Through Compilation: Our Approach

- Preserving Property = Preserving Partial State

```
void compute(int *mk, int m) {
  int n = 0;  // def 1
  ...
  n = rand(); // def 2
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
  n = 42;  // def 3
  ...
}
```

Preserving Properties Through Compilation: Our Approach

- Preserving Property = Preserving Partial State
- Preserving Partial State = Preserving

```
void compute(int *mk, int m) {
  int n = 0;  // def 1
  ...
  n = rand(); // def 2
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
  n = 42;  // def 3
  ...
}
```

Preserving Properties Through Compilation: Our Approach

- Preserving Property = Preserving Partial State
- Preserving Partial State = Preserving

```
void compute(int *mk, int m) {
  int n = 0;  // def 1
  ...
  n = rand(); // def 2
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
  n = 42;  // def 3
  ...
}
```

locations + values of observed variables

locations + values of observed memory locations

Preserving Properties Through Compilation: Our Approach

- Preserving Property = Preserving Partial State
- Preserving Partial State = Preserving

```
void compute(int *mk, int m) {
  int n = 0;  // def 1
  ...
  n = rand(); // def 2
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
  n = 42;  // def 3
  ...
}
```

locations + values of observed variables

locations + values of observed memory locations

an equivalent observation point (w.r.t. the observed entities)

Preserving Properties Through Compilation: Our Approach

```
void compute(int *mk, int m) {
  int n = 0;  // def 1
  ...
  n = rand(); // def 2
  int tmp = *mk ^ n;
  here: PROP(tmp == *mk ^ n)
  *mk = tmp ^ m;
  ...
  n = 42;  // def 3
  ...
}
```


```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2
  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n2

  %mk2 = xor %tmp1, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

Preserving Properties Through Compilation: Our Approach

Preserve observed memory locations

memory-barrier, side-effecting: cannot be removed

```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2

  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n2

  call obs.pt(              ) ;tmp == *mk^n
  %mk2 = xor %tmp1, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

Preserving Properties Through Compilation: Our Approach

SSA variables to be preserved

```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2

  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n2

  call obs.pt(%n2 , %tmp1 ) ;tmp == *mk^n
  %mk2 = xor %tmp1, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

Preserving Properties Through Compilation: Our Approach

```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2
  %n20 = call artificial.def(%n2)
  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n20

  call obs.pt(%n20, %tmp1 ) ;tmp == *mk^n
  %mk2 = xor %tmp1, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

Preserving Properties Through Compilation: Our Approach

```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2
  %n20 = call artificial.def(%n2)
  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n20
  %tmp10 = call artificial.def(%tmp1)
  call obs.pt(%n20, %tmp10) ;tmp == *mk^n
  %mk2 = xor %tmp10, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

Preserving Properties Through Compilation: Our Approach

opaque, side-effecting: cannot be analyzed or removed

```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2
  %n20 = call artificial.def(%n2)
  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n20
  %tmp10 = call artificial.def(%tmp1)
  call obs.pt(%n20, %tmp10) ;tmp == *mk^n
  %mk2 = xor %tmp10, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

Preserving Properties Through Compilation: Our Approach

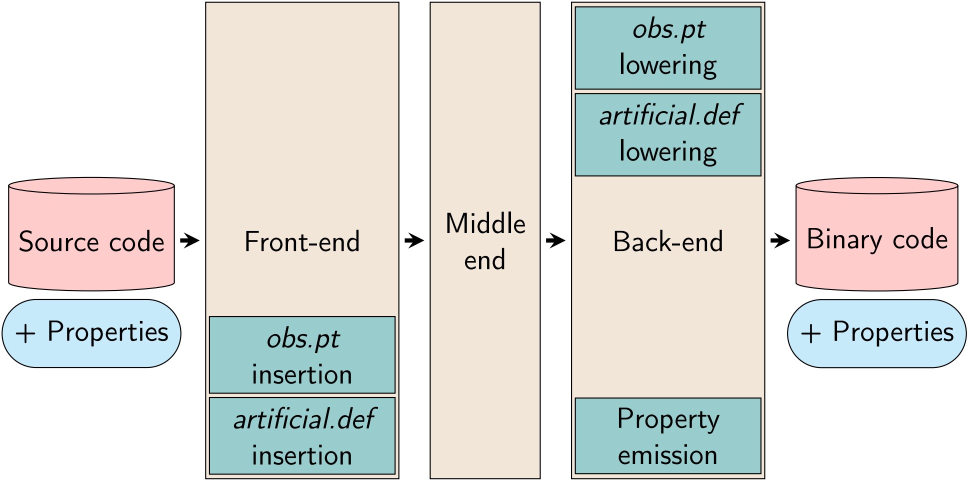
must be kept through the whole compilation flow, removed during code emission: no interference with original program

```
entry:
  %n1 = 0 ;SSA def 1
  ...
  %n2 = call rand() ;SSA def 2
  %n20 = call artificial.def(%n2)
  %mk1 = load %mk.addr
  %tmp1 = xor %mk1, %n20
  %tmp10 = call artificial.def(%tmp1)
  call obs.pt(%n20, %tmp10) ;tmp == *mk^n
  %mk2 = xor %tmp10, %m1
  ...
  %n3 = 42 ;SSA def 3
  ...
```

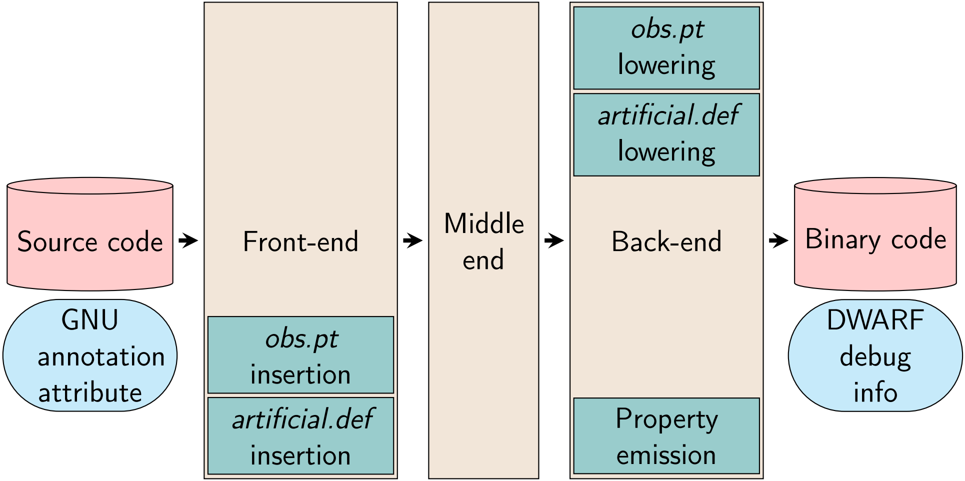
Property Preservation Through Compilation: Outline

1. Definition of *property preservation through compilation*
2. Our approach to preserve program properties
3. Implementation of our approach in LLVM
4. Validation of our approach and implementation on security applications

Preserving Properties Through Compilation: LLVM flow



Preserving Properties Through Compilation: LLVM flow



Property Preservation Through Compilation: Outline

1. Definition of *property preservation through compilation*
2. Our approach to preserve program properties
3. Implementation of our approach in LLVM
4. Validation of our approach and implementation on security applications

Property Preservation Validation: Outline

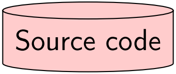
1. General Validation Methodology
2. Validation on Functional Properties
3. Validation on Security Properties
4. Performance Overhead Evaluation

Experimental Validation Methodology

Property preservation = Equality of observation traces

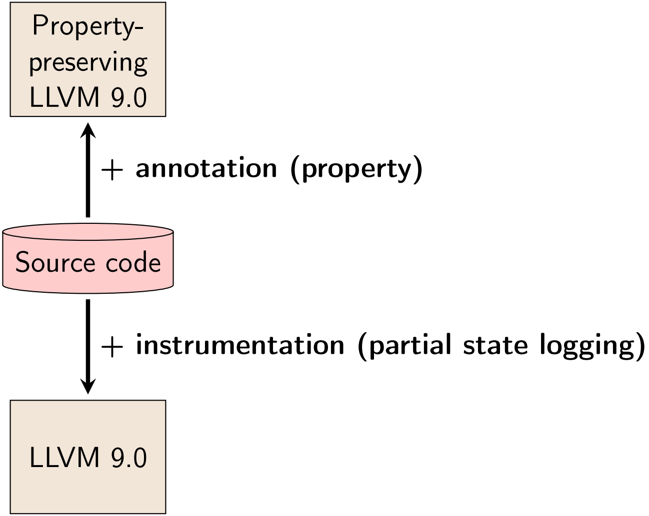
Experimental Validation Methodology

Property preservation = Equality of observation traces



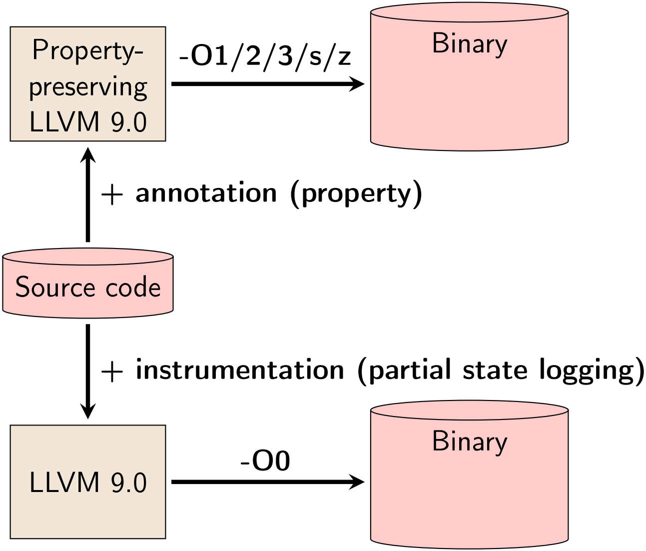
Experimental Validation Methodology

Property preservation = Equality of observation traces



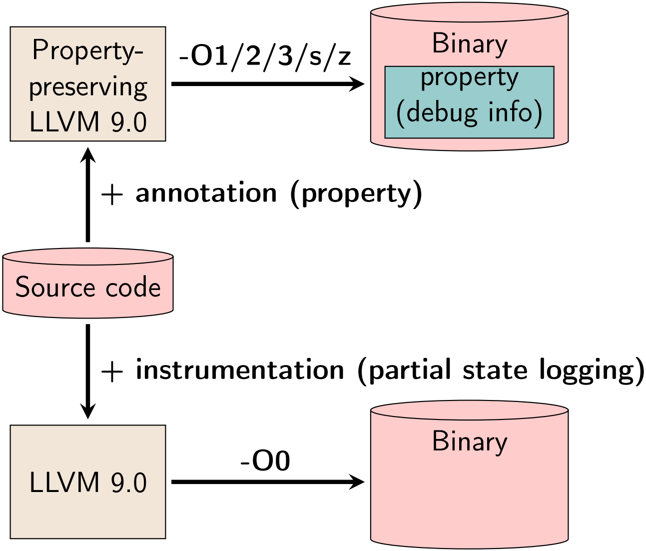
Experimental Validation Methodology

Property preservation = Equality of observation traces



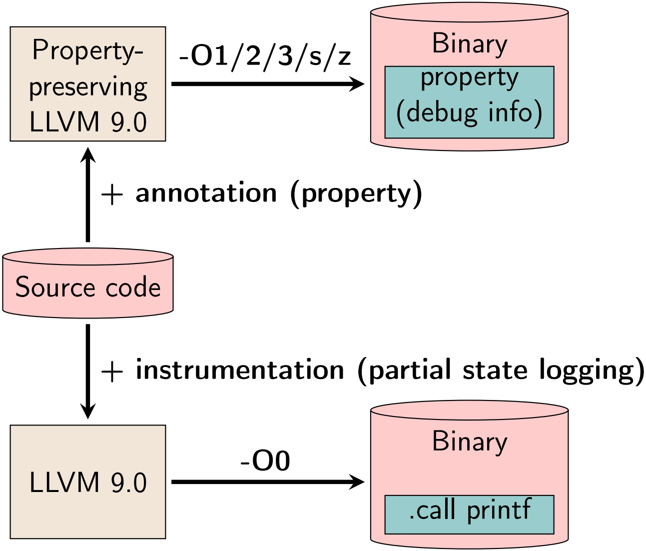
Experimental Validation Methodology

Property preservation = Equality of observation traces



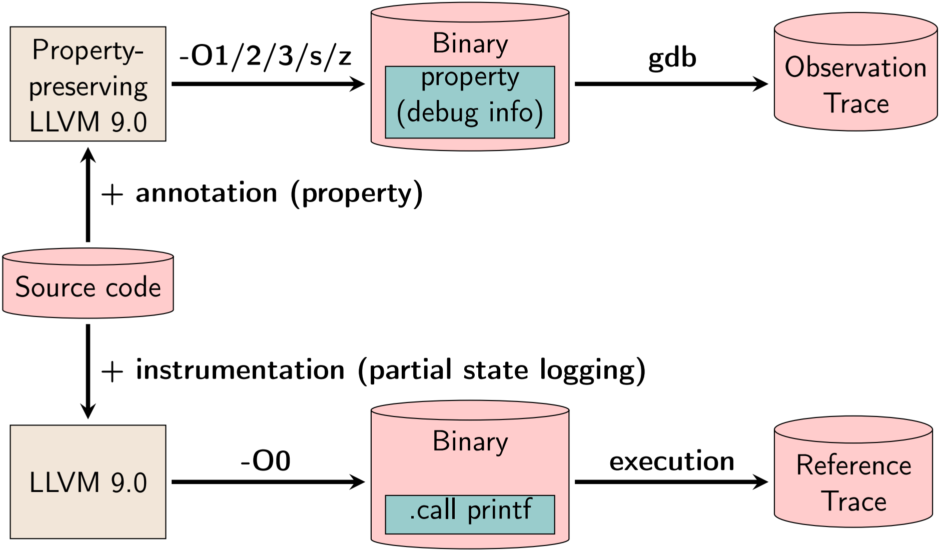
Experimental Validation Methodology

Property preservation = Equality of observation traces



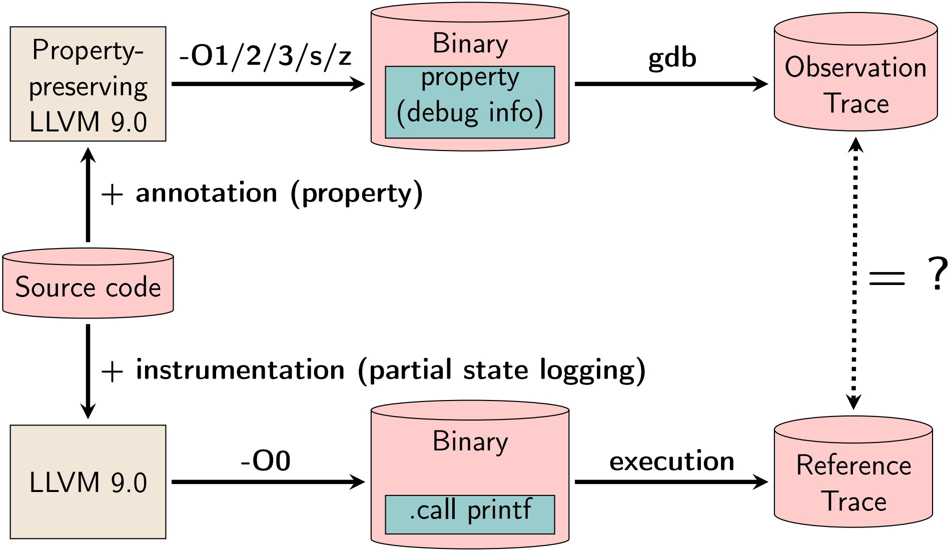
Experimental Validation Methodology

Property preservation = Equality of observation traces



Experimental Validation Methodology

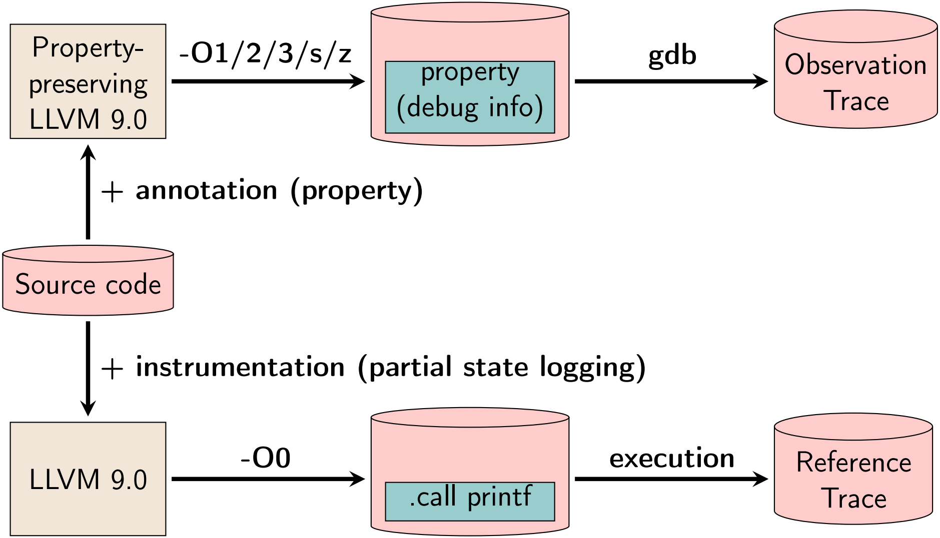
Property preservation = Equality of observation traces

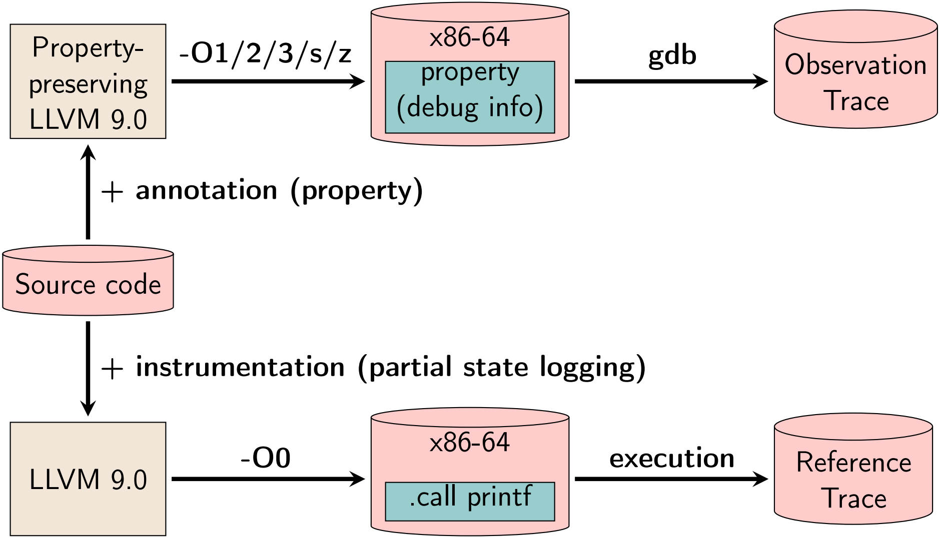


Functional Validation

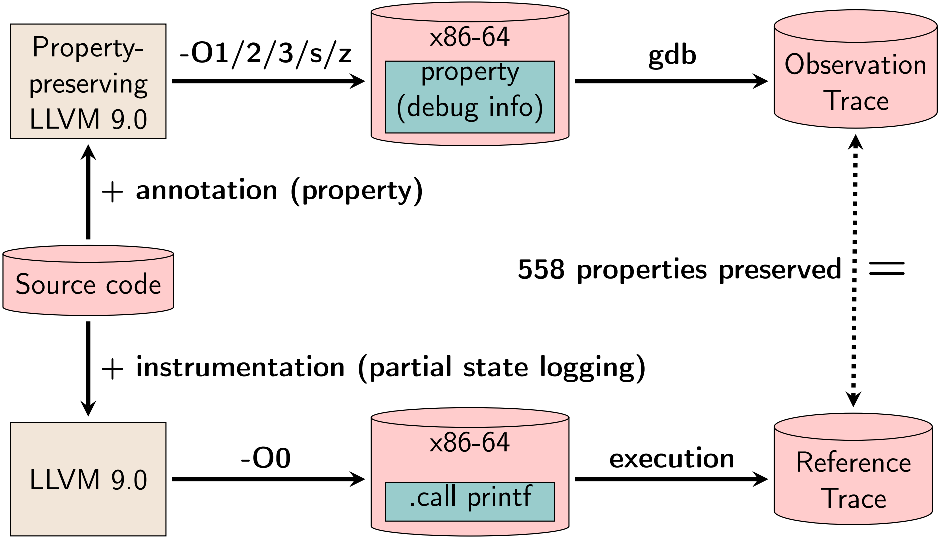
- Goal: propagating functional properties used for program static analysis from source to binary level
- Programs from *Framework for Modular Analysis of C programs* (Frama-C) test suite [Cuoq et al., 2012]
- 558 functional properties (C boolean expressions), verifying expected values of variables at a given program point

Functional Validation




Property-preserving LLVM 9.0
-O1/2/3/s/z
x86-64
property (debug info)
gdb
Observation Trace
+ annotation (property)
Source code
+ instrumentation (partial state logging)
LLVM 9.0
-O0
x86-64
.call printf
execution
Reference Trace

Functional Validation



Application to Security Properties

Considered properties:

Attack			
Protection			
Property			

Application to Security Properties

Considered properties:

Attack	Side-channel		
Protection	Masking of secret data		
Property	Instruction ordering in masking operations		

Application to Security Properties

Considered properties:

Attack	Side-channel	Data remanence	
Protection	Masking of secret data	Inserting code to erase secret data	
Property	Instruction ordering in masking operations	Presence of secret memory data erasure	

Application to Security Properties

Considered properties:

Attack	Side-channel	Data remanence	Fault injection	
Protection	Masking of secret data	Inserting code to erase secret data	Inserting redundant data and/or protection code	
Property	Instruction ordering in masking operations	Presence of secret memory data erasure	Interleaving of functional and protection code	Presence of redundant data detecting fault injections

Application to Security Properties

Considered properties:

Attack	Side-channel	Data remanence	Fault injection	
Protection	Masking of secret data	Inserting code to erase secret data	Inserting redundant data and/or protection code	
Property	Instruction ordering in masking operations	Presence of secret memory data erasure	Interleaving of functional and protection code	Presence of redundant data detecting fault injections

⇒ these security properties are non-functional (refer to notions not clearly defined in the source program semantics)

Application to Security Properties

Considered properties:

Attack	Side-channel	Data remanence	Fault injection	
Protection	Masking of secret data	Inserting code to erase secret data	Inserting redundant data and/or protection code	
Property	Instruction ordering in masking operations	Presence of secret memory data erasure	Interleaving of functional and protection code	Presence of redundant data detecting fault injections

⇒ these security properties are non-functional (refer to notions not clearly defined in the source program semantics)

⇒ preserving source-level protections by forcibly observing its variables at specific program points

Application to Security Properties

- Defining new predicate *observe(v)* which includes *v* into the partial state to be preserved

```
void compute(int *mk, int m) {
  int n = 0;  // def 1
  ...
  n = rand(); // def 2
  int tmp = *mk ^ n;
  here: PROP(observe(tmp))
  *mk = tmp ^ m;
  ...
  n = 42;  // def 3
  ...
}
```

Proper Interleaving of Functional code and Protection

A source-level countermeasure against fault attacks altering the program control flow [Lalande, Heydemann, and Berthomé, 2014]

```
if (cond) {
  stmt1

  stmt2

}
```

Proper Interleaving of Functional code and Protection

A source-level countermeasure against fault attacks altering the program control flow [Lalande, Heydemann, and Berthomé, 2014]

```
int cnt_if = 0;
if (cond) {
  stmt1

  stmt2

}
```

1. Defining step counter at each control construct

Proper Interleaving of Functional code and Protection

A source-level countermeasure against fault attacks altering the program control flow [Lalande, Heydemann, and Berthomé, 2014]

```
int cnt_if = 0;
if (cond) {
  stmt1
  cnt_if++;
  stmt2
  cnt_if++;
}
```

1. Defining step counter at each control construct

2. Incrementing step counter after *every* C statement of the construct

Proper Interleaving of Functional code and Protection

A source-level countermeasure against fault attacks altering the program control flow [Lalande, Heydemann, and Berthomé, 2014]

```
int cnt_if = 0;
if (cond) {
  stmt1
  cnt_if++;
  stmt2
  cnt_if++;
}
if (cond && cnt_if != 2)
  exception_handler();
```

1. Defining step counter at each control construct

2. Incrementing step counter after *every* C statement of the construct

3. Checking counters against their expected values at the end of the construct, calling exception handler when it fails

Proper Interleaving of Functional code and Protection

A source-level countermeasure against fault attacks altering the program control flow [Lalande, Heydemann, and Berthomé, 2014]

```
int cnt_if = 0;
if (cond) {
  stmt1
  cnt_if++;
  stmt2
  cnt_if++;
}
if (cond && cnt_if != 2)
  exception_handler();
```

```
int cnt_if = 0;
if (cond) {
  stmt1
  stmt2
  cnt_if += 2;
}
```

Optimizations will remove counter checks and group counter incrementations

Proper Interleaving of Functional code and Protection

A source-level countermeasure against fault attacks altering the program control flow [Lalande, Heydemann, and Berthomé, 2014]

```
int cnt_if = 0;
if (cond) {
  stmt1
  cnt_if++;
  stmt2
  cnt_if++;
}
if (cond && cnt_if != 2)
  exception_handler();
```

```
int cnt_if = 0;
if (cond) {
  stmt1
  stmt2
  cnt_if += 2;
}
```

Optimizations will remove counter checks and group counter incrementations

Traditional secure approach: compiling at `-O0` (disabling optimizations)

Proper Interleaving of Functional code and Protection

Our approach based on property preservation:

```
int cnt_if = 0;
if (cond) {
  stmt1

  cnt_if++;
  stmt2

  cnt_if++;
}
if (cond && cnt_if != 2)
  exception_handler();
```

Proper Interleaving of Functional code and Protection

Our approach based on property preservation:

```
int cnt_if = 0;
if (cond) {
  stmt1
  here1: PROP(observe(cnt_if))
  cnt_if++;
  stmt2
  here2: PROP(observe(cnt_if))
  cnt_if++;
}
if (cond && cnt_if != 2)
  exception_handler();
```

1. Observe counter before incrementation to prevent optimizations from removing it

Proper Interleaving of Functional code and Protection

Our approach based on property preservation:

```
int cnt_if = 0;
if (cond) {
  stmt1
  here1: PROP(observe(cnt_if, cond, ...))
  cnt_if++;
  stmt2
  here2: PROP(observe(cnt_if, cond, ...))
  cnt_if++;
}
if (cond && cnt_if != 2)
  exception_handler();
```

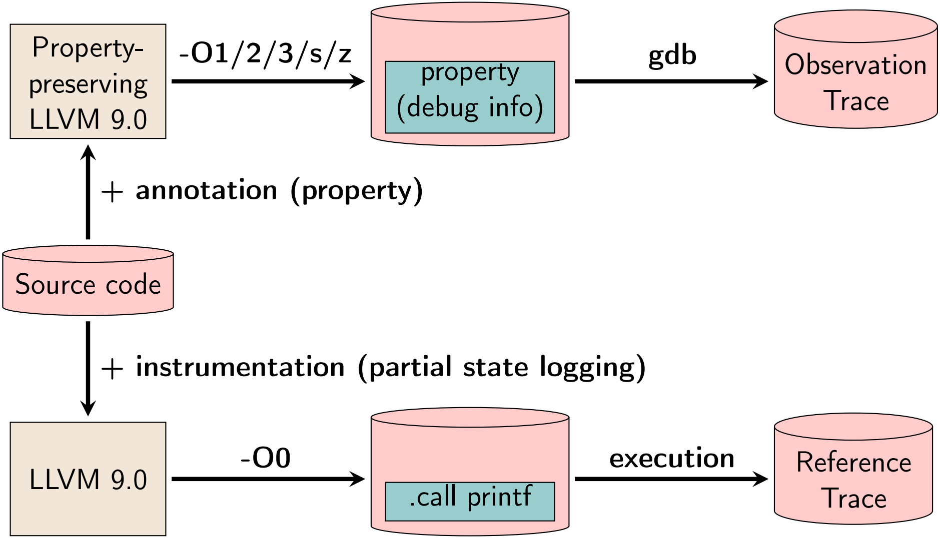
1. Observe counter before incrementation to prevent optimizations from removing it

2. Observe all variables + memory locations to guarantee the proper interleaving of functional code and incrementation

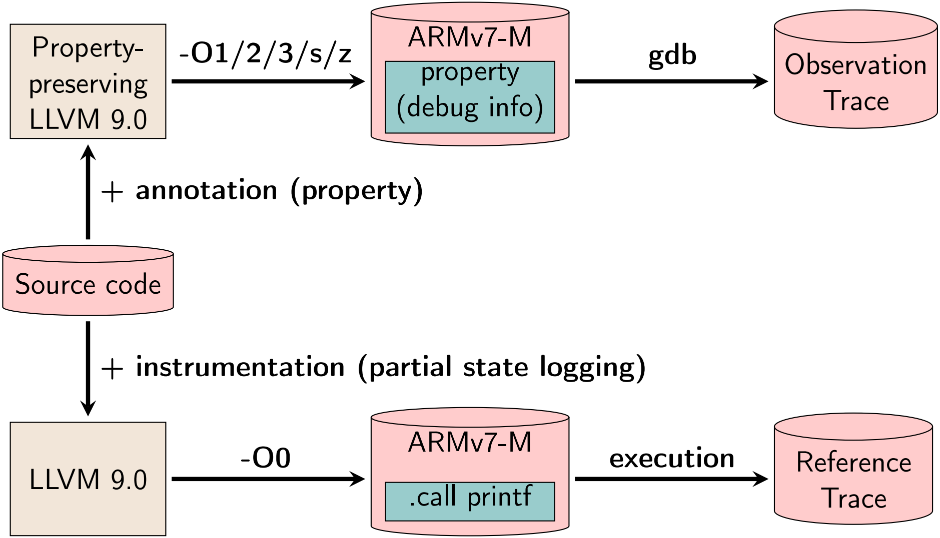
Security Property Preservation Validation

Attack	Side-channel	Data remanence	Fault injection	
Protection	Masking of secret data	Inserting code to erase secret data	Inserting redundant data and/or protection code	
Property	Instruction ordering in masking operations	Presence of sensitive memory data erasure	Interleaving of functional and protection code	Presence of redundant data detecting fault injections
Application	aes-herbst	rsa-encrypt	pin-sci	loop-redundant
		rsa-decrypt	aes-sci	

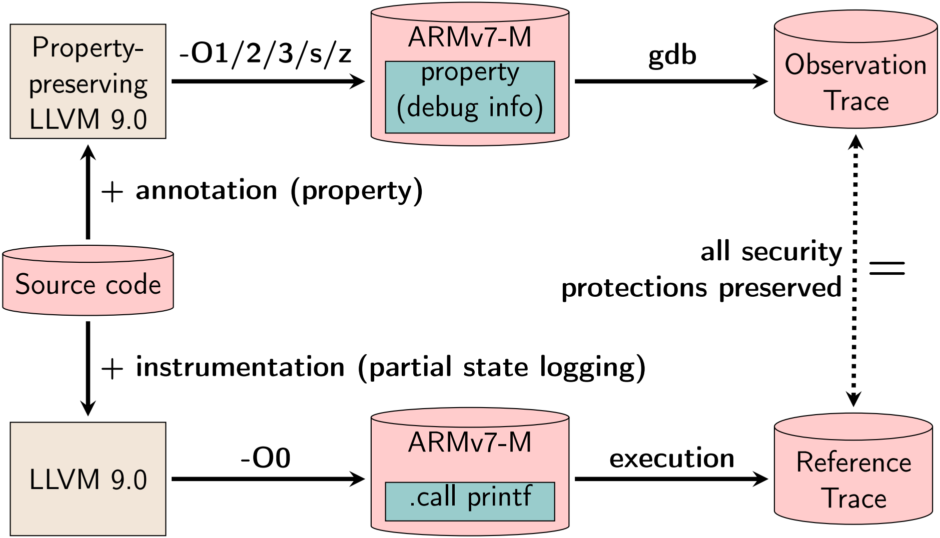
Security Property Preservation Validation



Security Property Preservation Validation



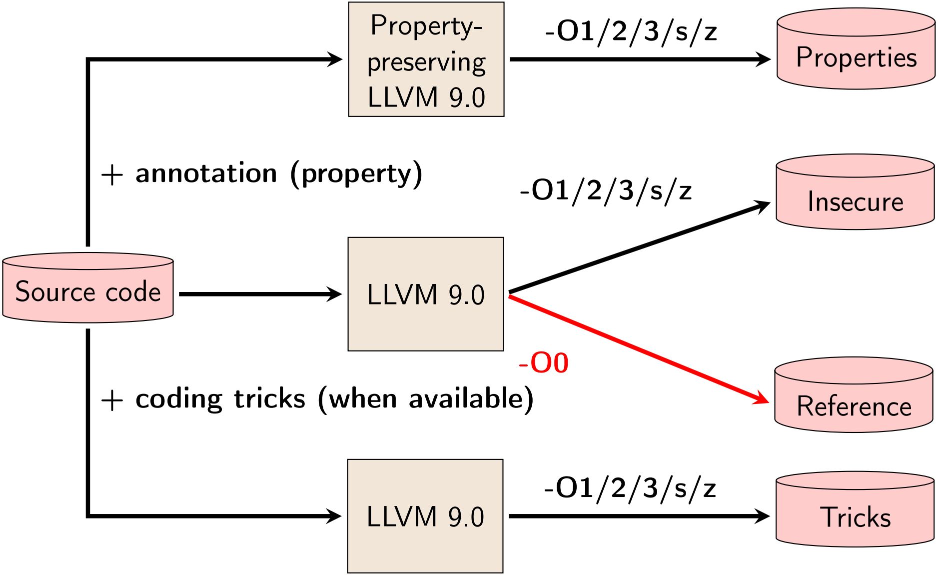
Security Property Preservation Validation



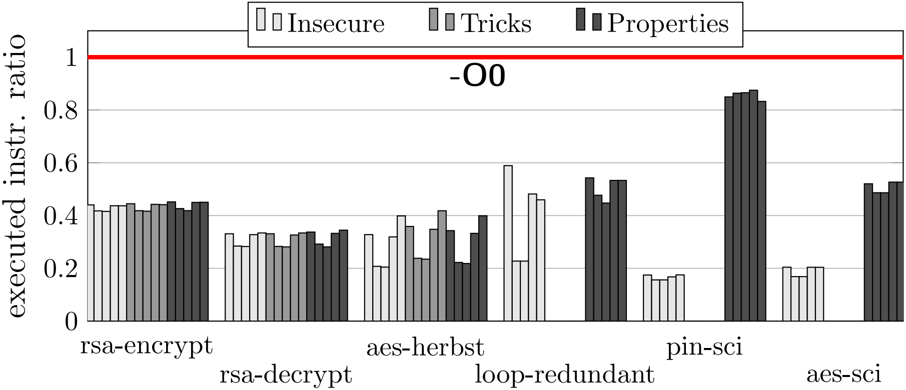
Performance Evaluation

Is the performance penalty due to blocking some optimizations acceptable?

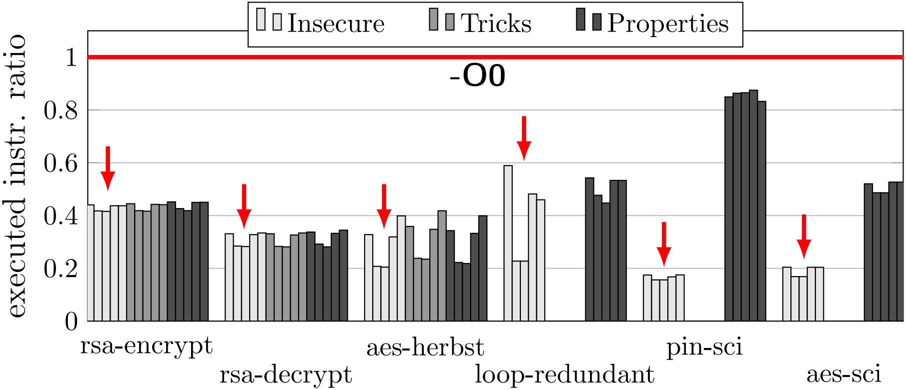
Attack	Side-channel	Data remanence	Fault injection	
Protection	Masking of secret data	Inserting code to erase secret data	Inserting redundant data and/or protection code	
Property	Instruction ordering in masking operations	Presence of sensitive memory data erasure	Interleaving of functional and protection code	Presence of redundant data detecting fault injections
Application	aes-herbst	rsa-encrypt	pin-sci	loop-redundant
		rsa-decrypt	aes-sci	


Property-
preserving
LLVM 9.0
-O1/2/3/s/z
Properties
+ annotation (property)
-O1/2/3/s/z
Insecure
Source code
LLVM 9.0
-O0
Reference
+ coding tricks (when available)
LLVM 9.0
-O1/2/3/s/z
Tricks

Performance Evaluation

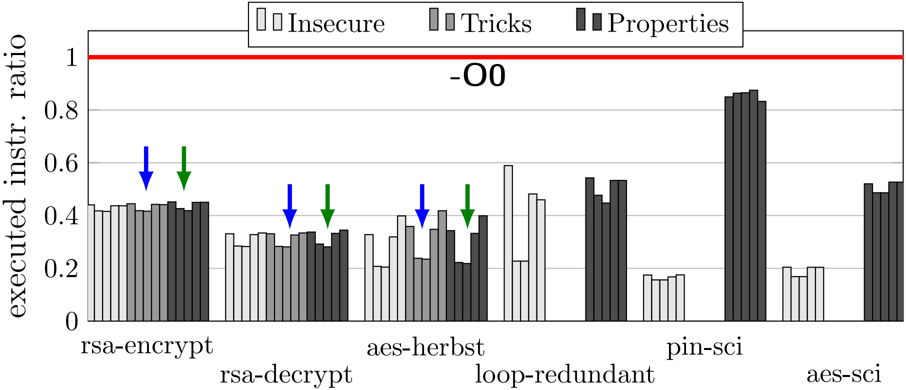


Performance Evaluation



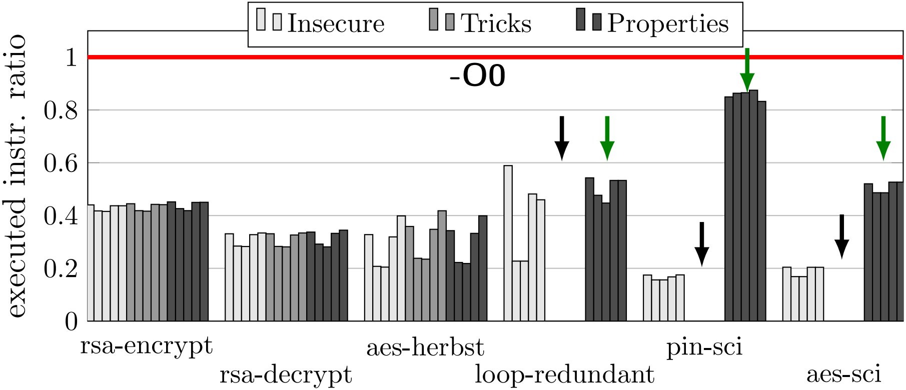
- Insecure: fastest executables but protections are modified or removed when optimizations enabled

Performance Evaluation



- Insecure: fastest executables but protections are modified or removed when optimizations enabled
- Properties **preserve** source-level protections
  - with similar performance compared to fragile tricks

Performance Evaluation

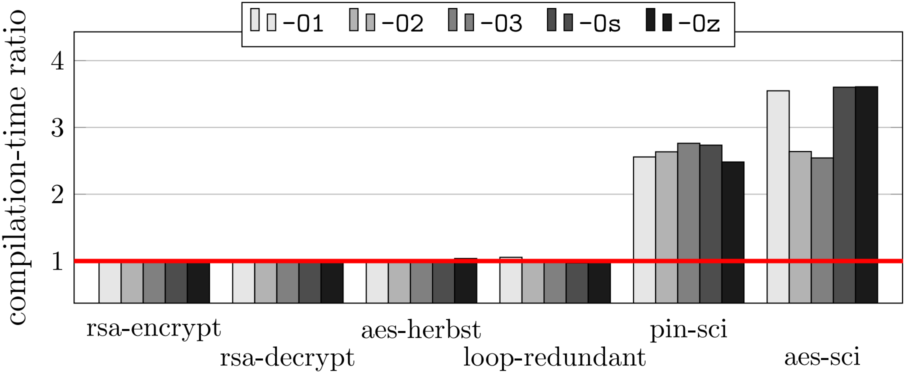


- Insecure: fastest executables but protections are modified or removed when optimizations enabled
- Properties **preserve** source-level protections
  - with similar performance compared to fragile tricks
  - with performance improvement over programs compiled at `-O0` when no trick exists

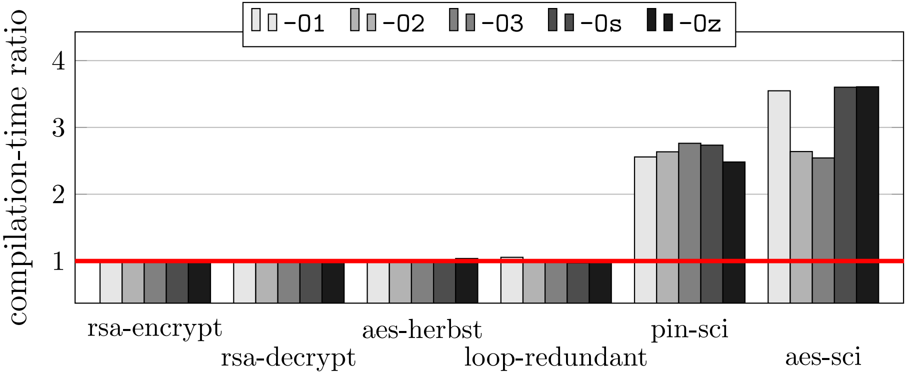
Conclusion

- Mechanism to preserve functional properties through optimizing compilation, enabling automated analyses and verifications at binary level [Bréjon et al., 2019]
- Application to preserving source-level protections
- Current work: formalization of a lightweight approach to preserve security protections, based on data-dependence.
- Perspective: contribute this work to the community, graduate and get a position!

Compilation-time Evaluation

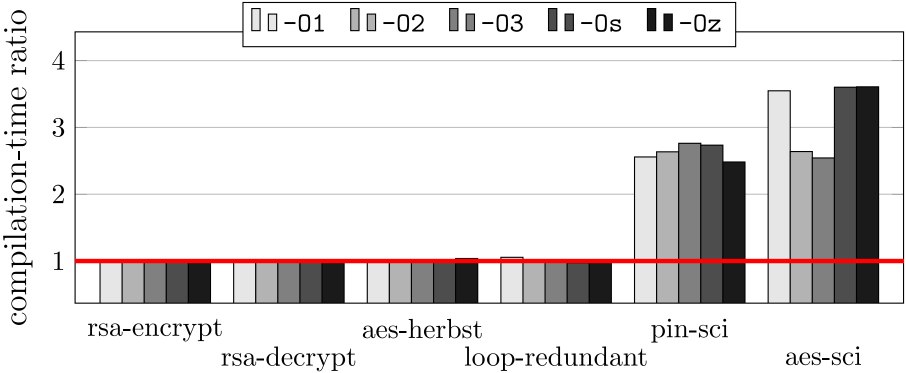


Compilation-time Evaluation



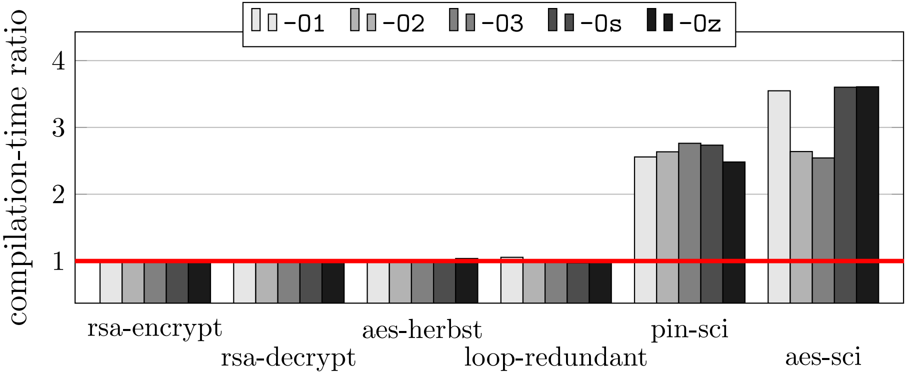
- Compilation-time overhead compared to the original program compiled with the same optimization flag
- High overhead for step counter incrementation protection

Compilation-time Evaluation



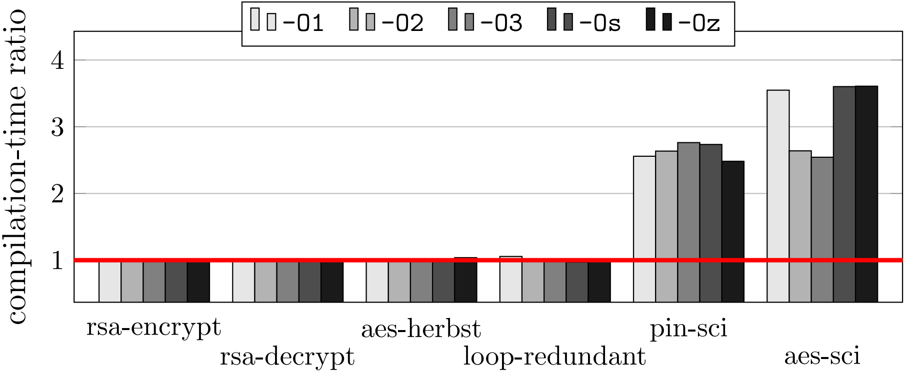
- Compilation-time overhead compared to the original program compiled with the same optimization flag
- High overhead for step counter incrementation protection
  - Complete program state is observed before each incrementation

Compilation-time Evaluation



- Compilation-time overhead compared to the original program compiled with the same optimization flag
- High overhead for step counter incrementation protection
  - Complete program state is observed before each incrementation
  - At least one property for *every* functional C statement

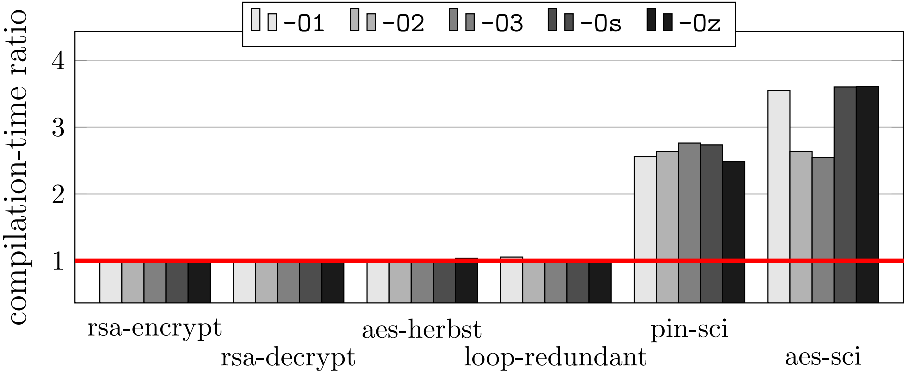
Compilation-time Evaluation



- Compilation-time overhead compared to the original program compiled with the same optimization flag
- High overhead for step counter incrementation protection
  - Complete program state is observed before each incrementation
  - At least one property for *every* functional C statement

$\Rightarrow$ worst-case scenario for our approach

Compilation-time Evaluation



- Compilation-time overhead compared to the original program compiled with the same optimization flag
- High overhead for step counter incrementation protection
  - Complete program state is observed before each incrementation
  - At least one property for *every* functional C statement

  ⇒ worst-case scenario for our approach

⇒ price worth paying for preserving source-code protections