LANGUAGE BASED SECURITY (LBT)

SECURE COMPILATION

Chiara Bodei, Gian-Luigi Ferrari

Lecture May, 15 2024 [reloaded]
Lecture May, 17 2024

Outline

Compiler Correctness

Secure compilation

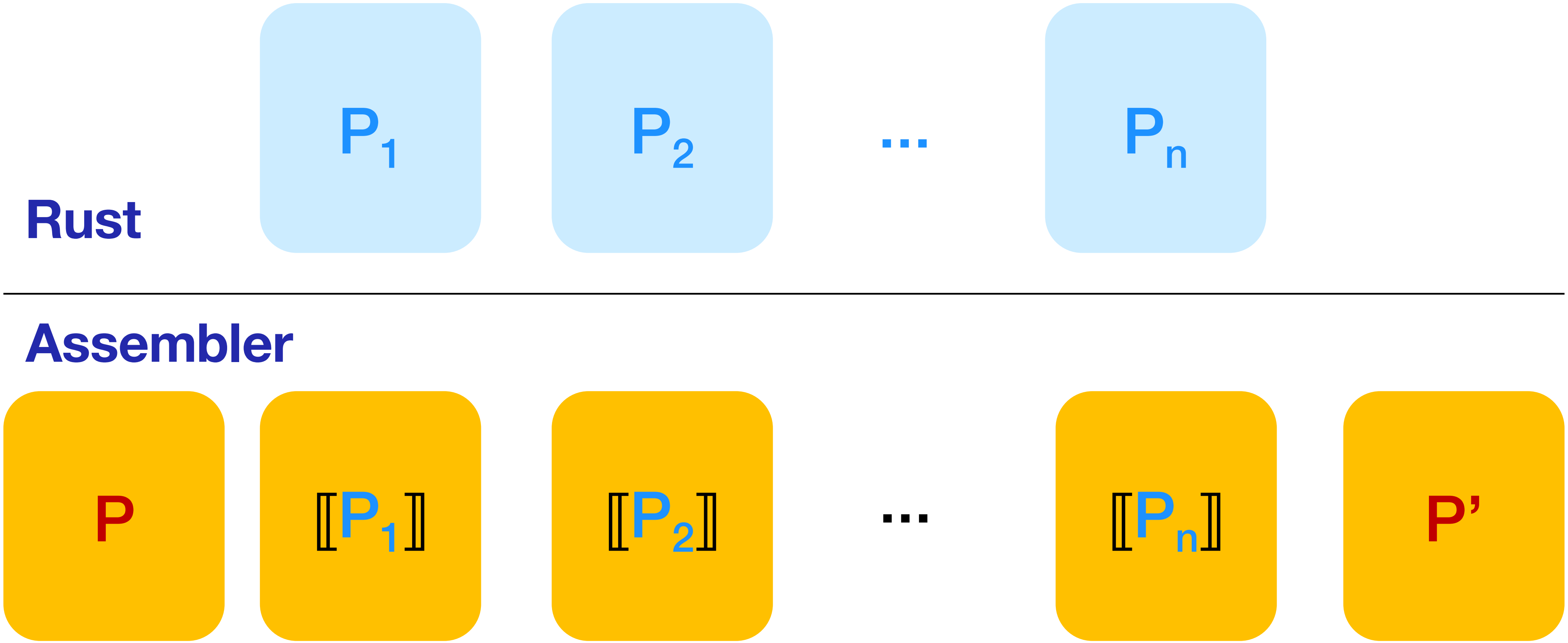
Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation
Fully abstract compilation in practice

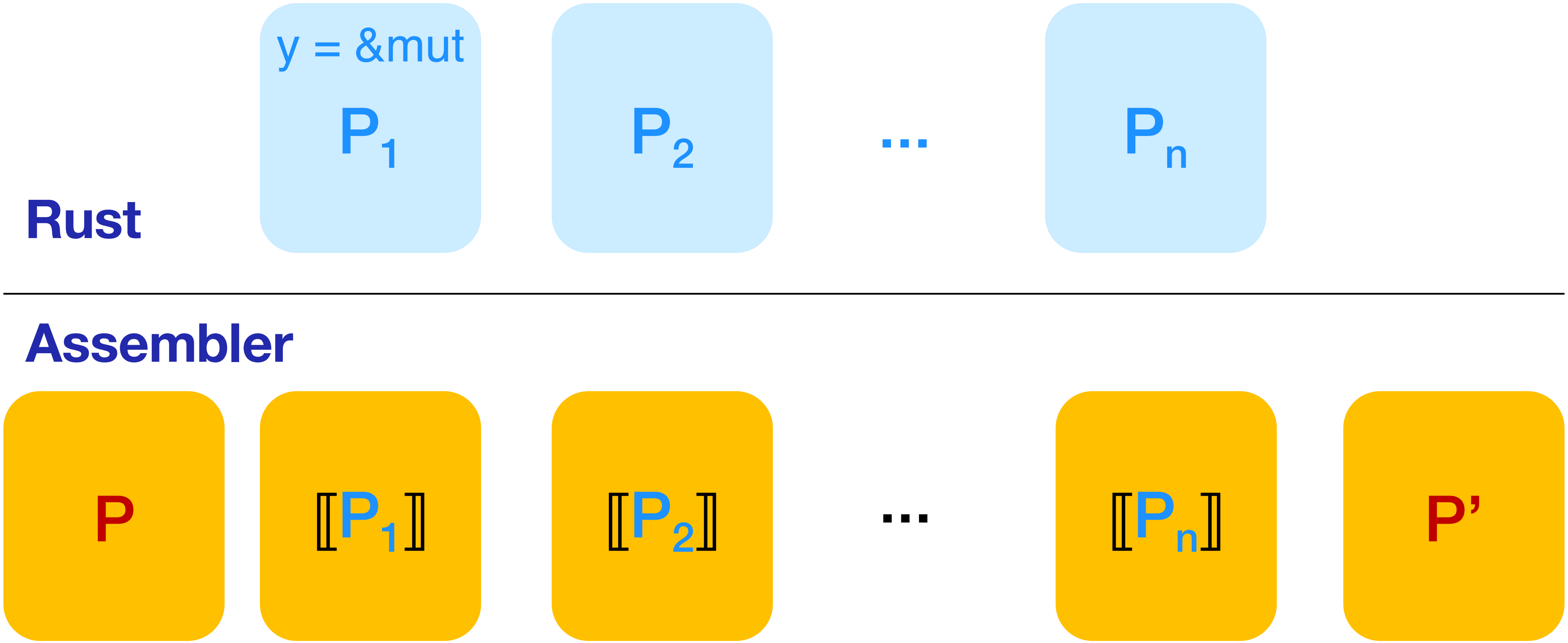
Secure compilation

- What does it mean for a compiler to be secure?

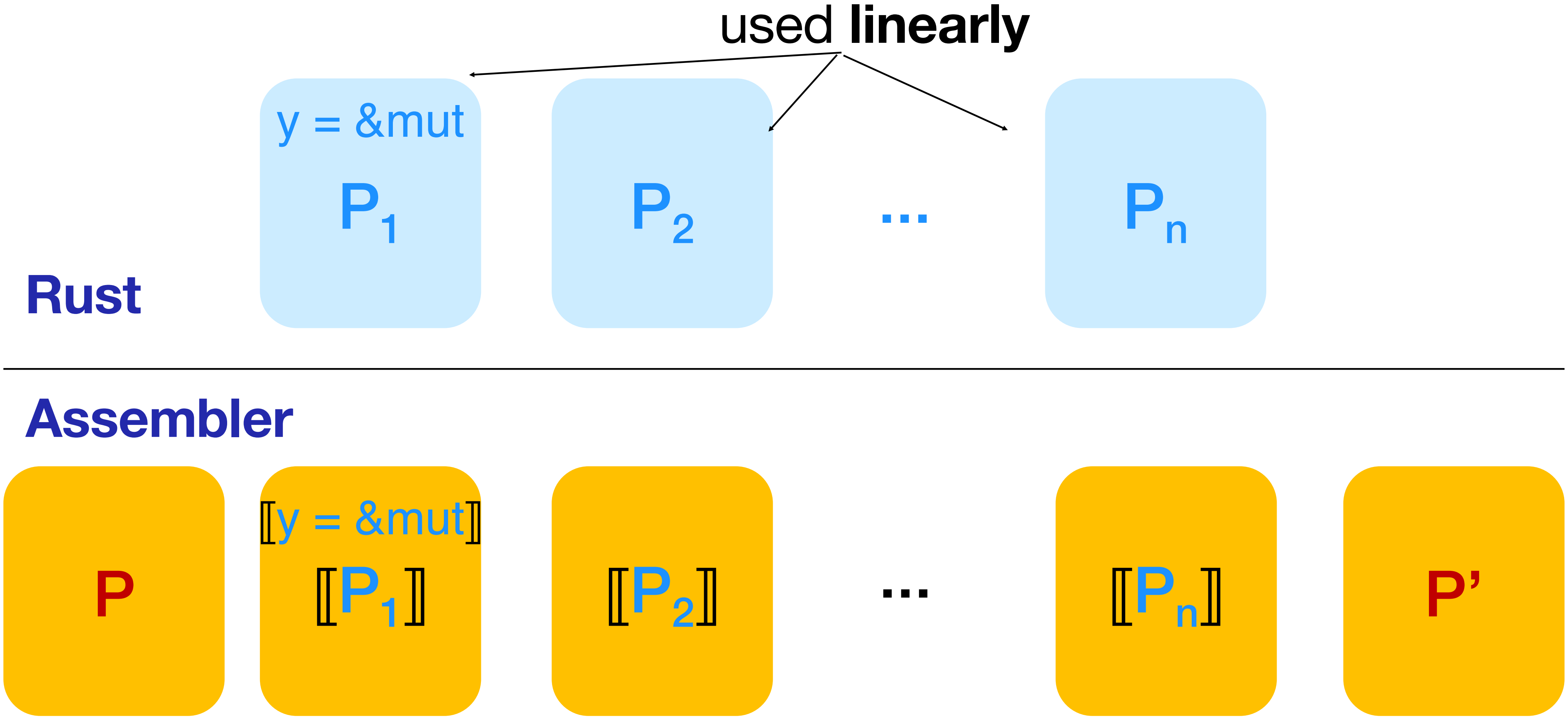
Example



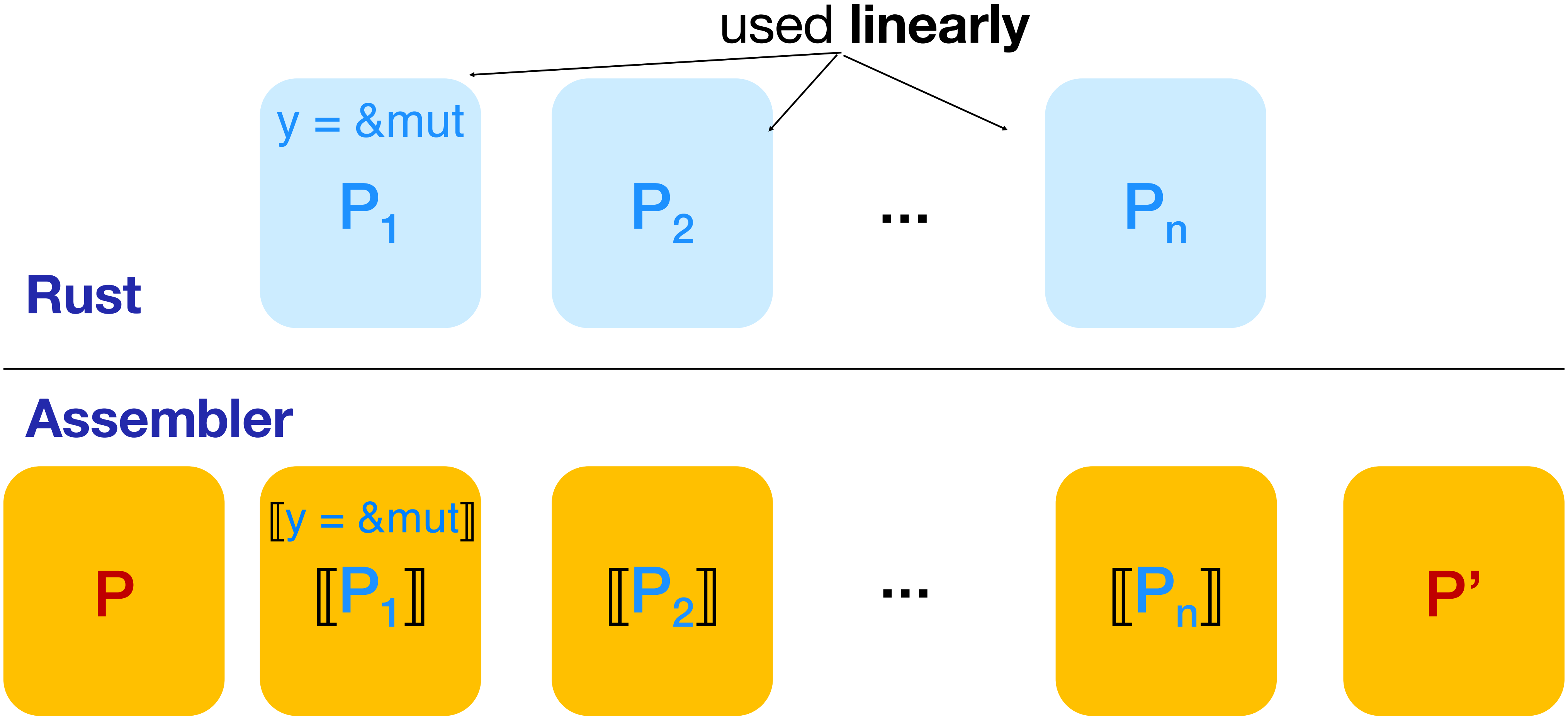
Example



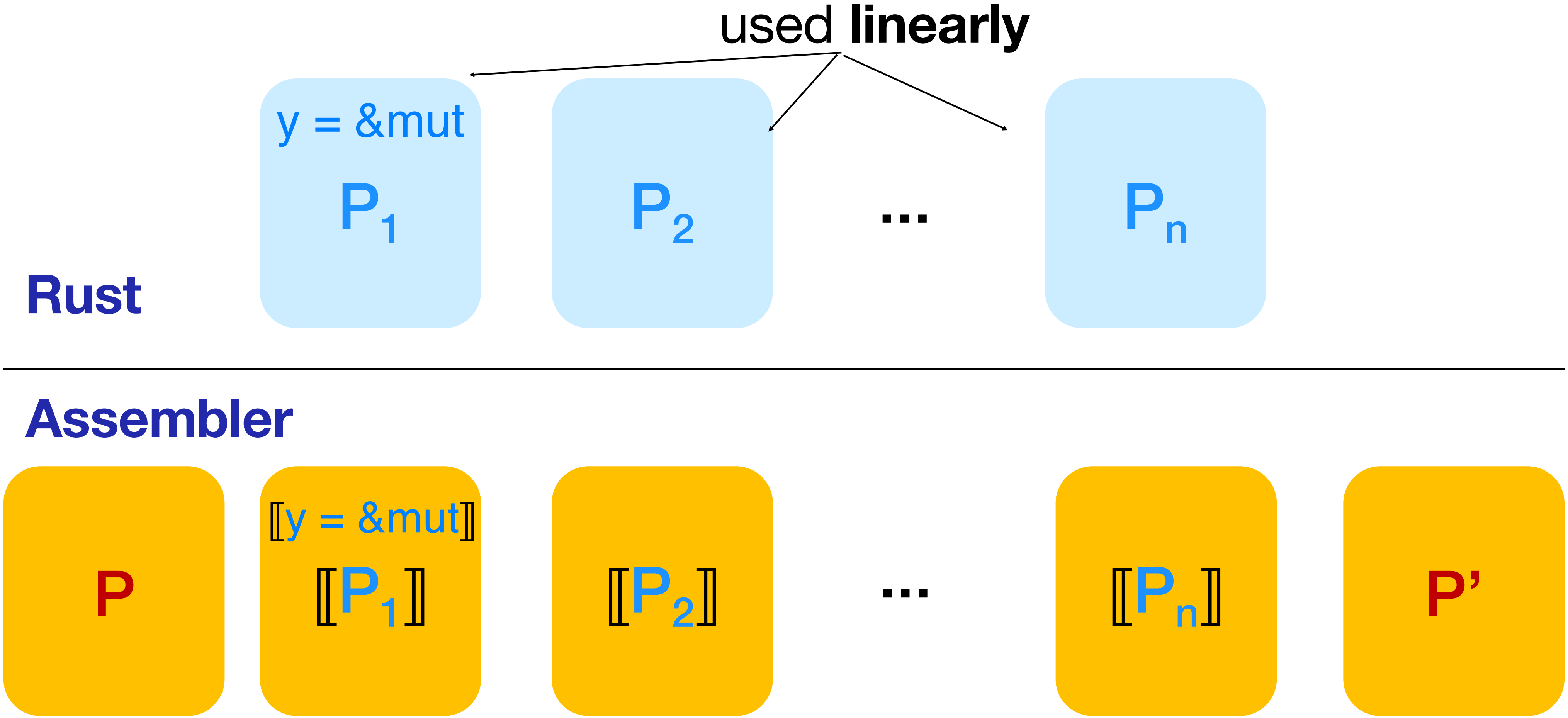
Example



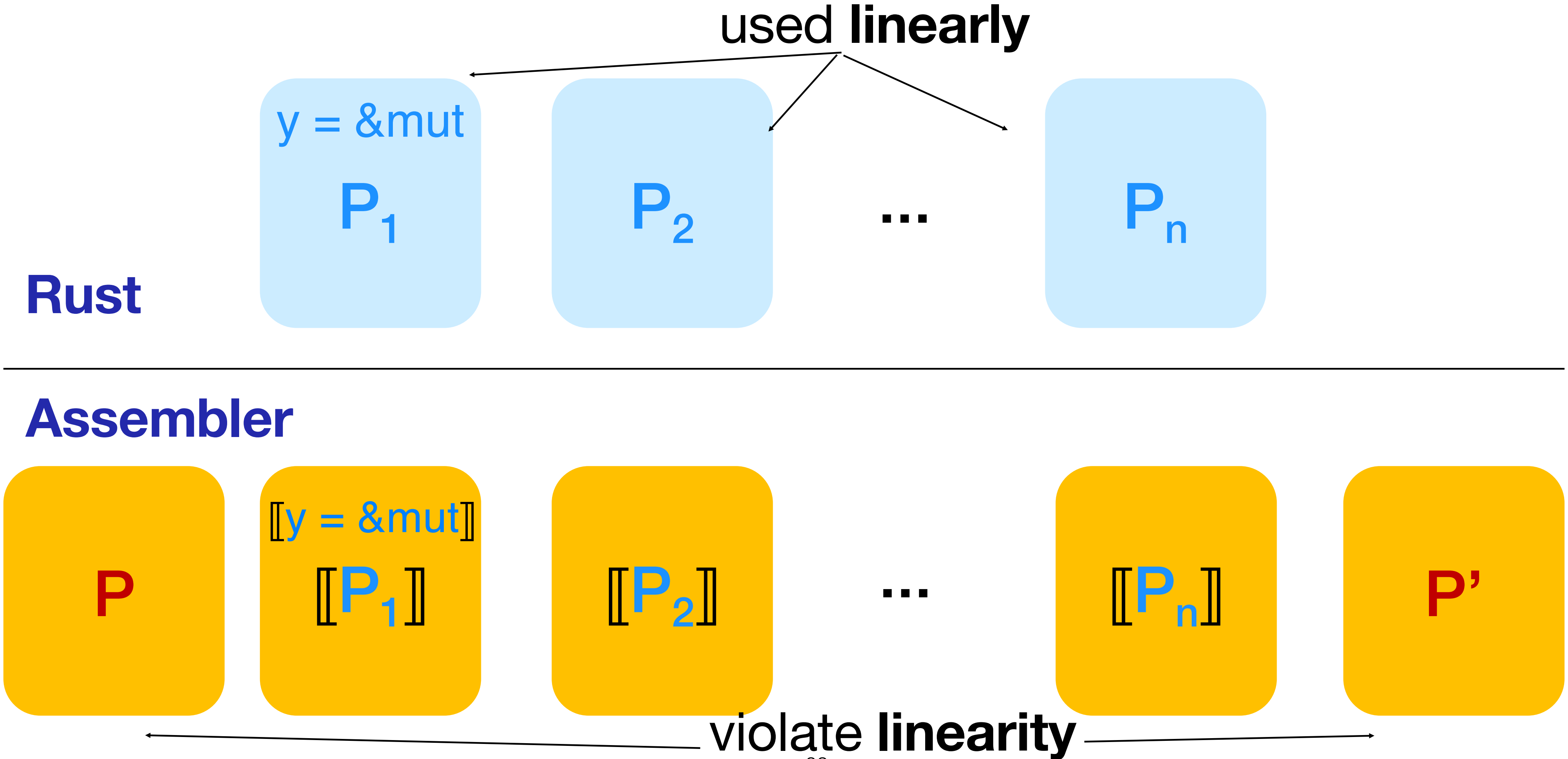
Linearity



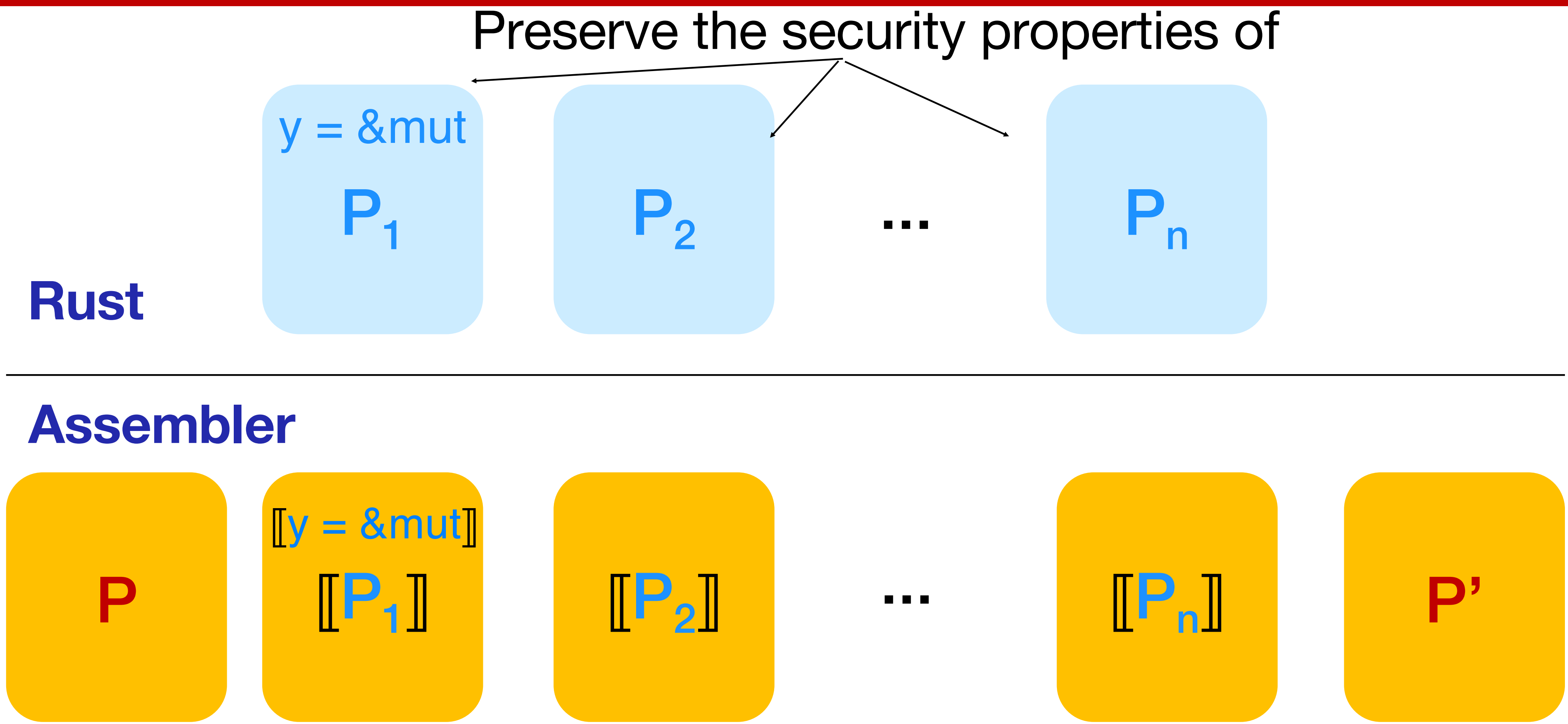
Linearity



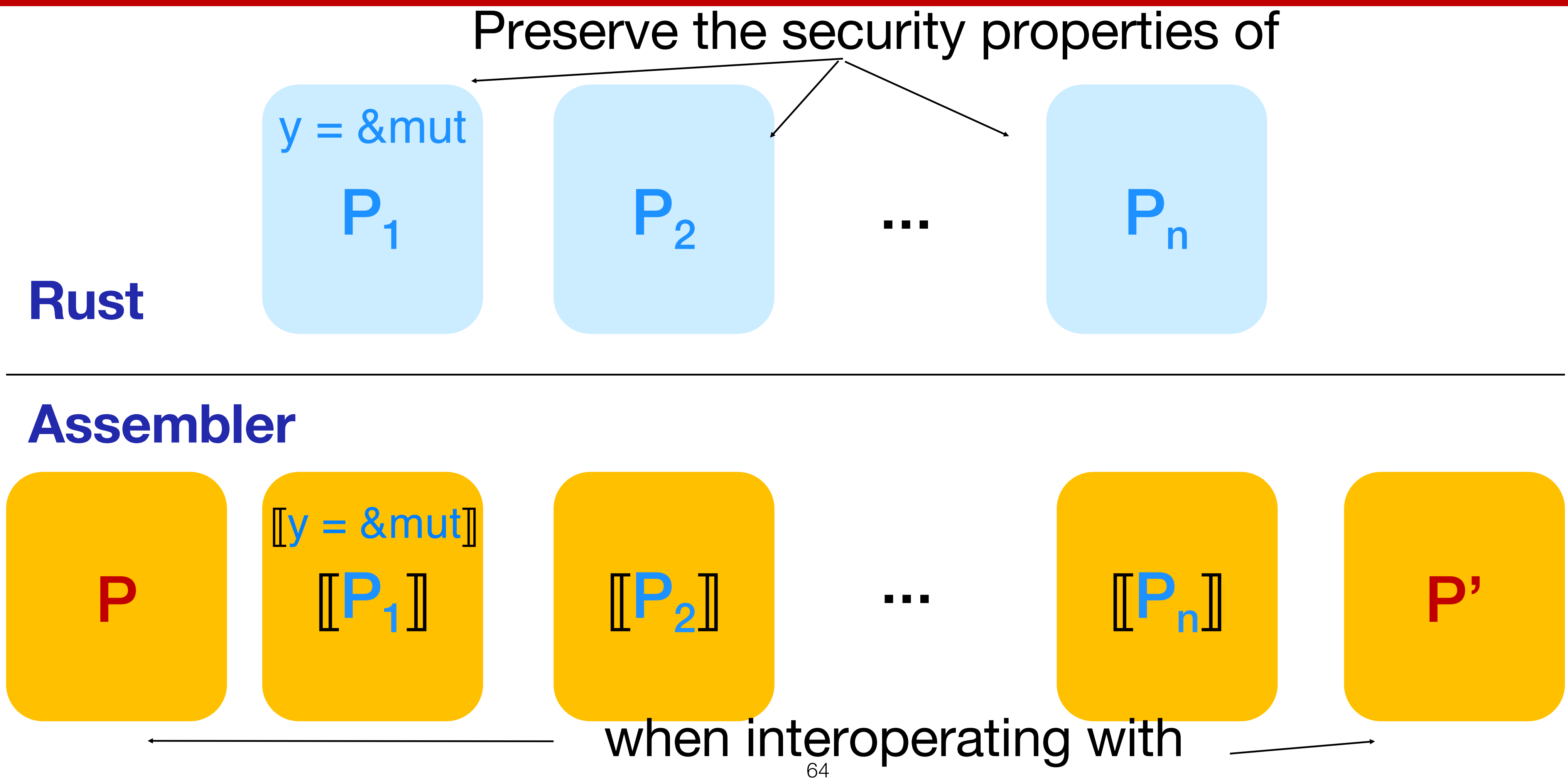
Possible violations of linearity



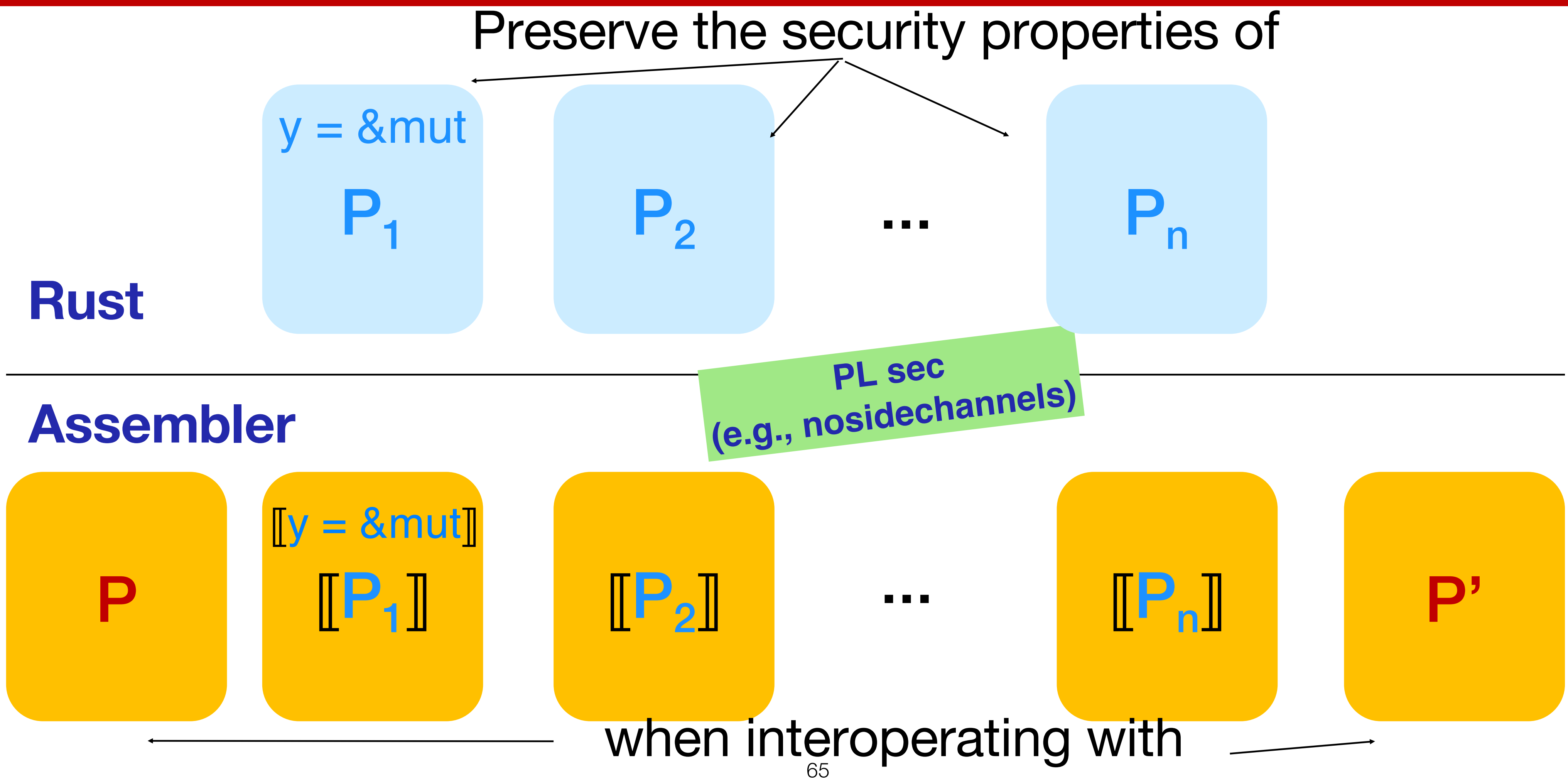
Possible violations of linearity



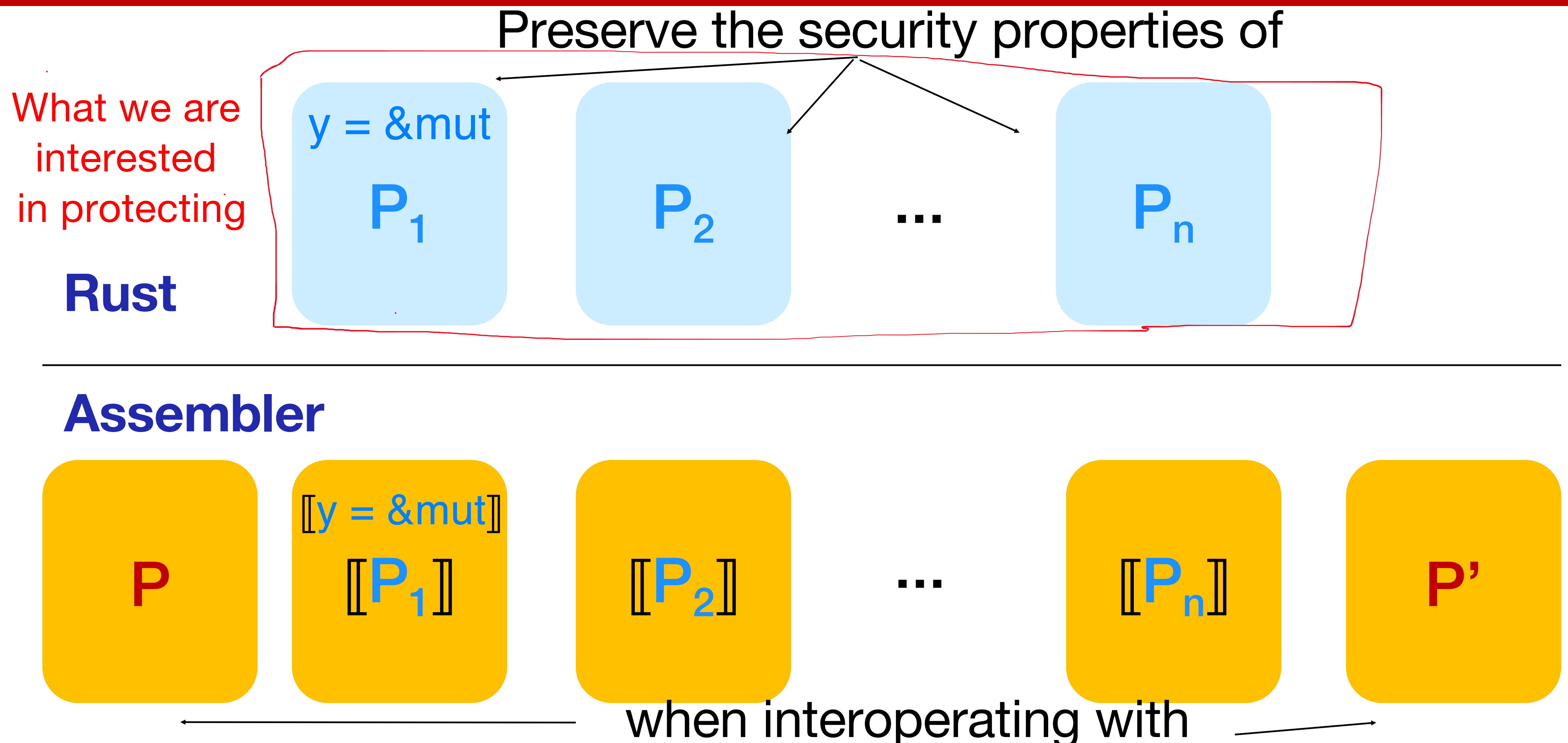
Preservation of linearity



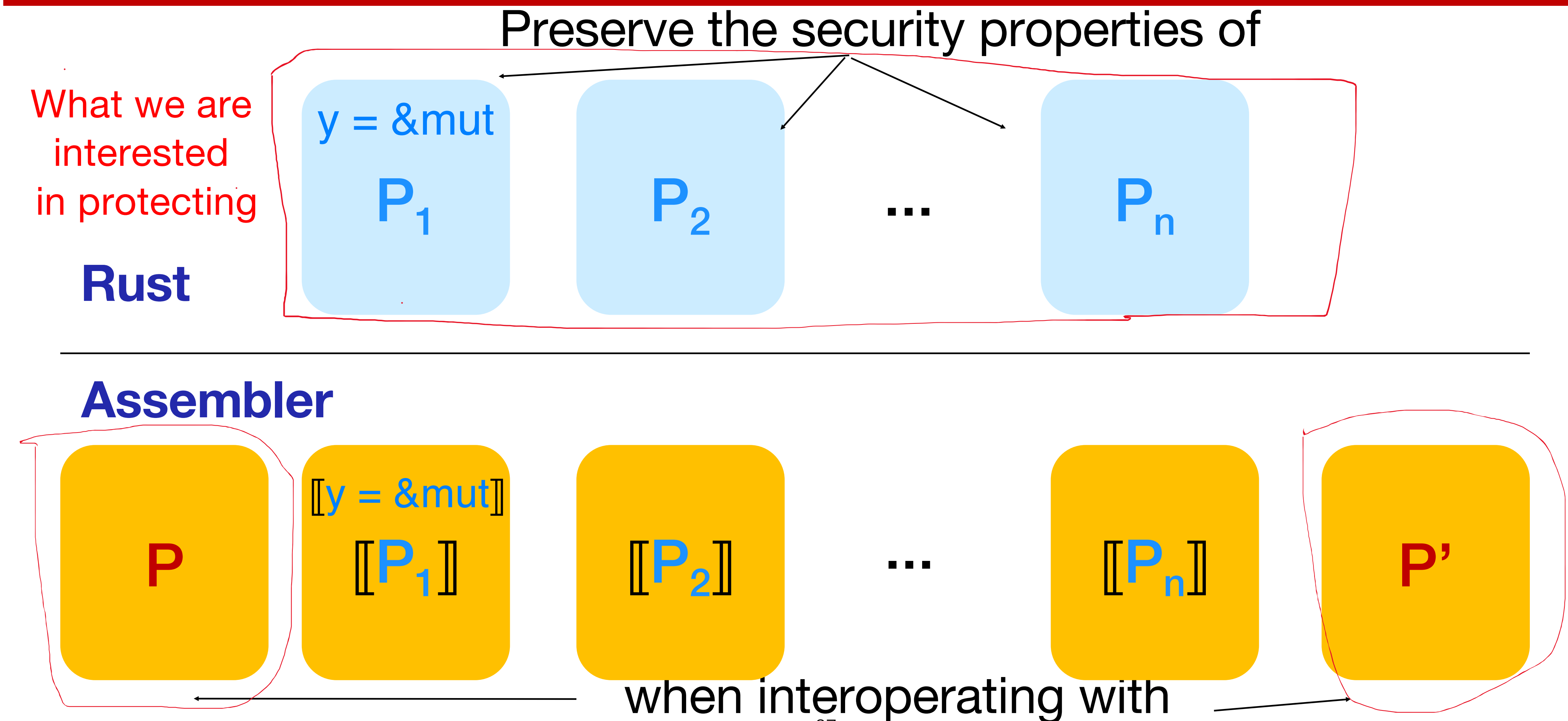
Preservation of linearity



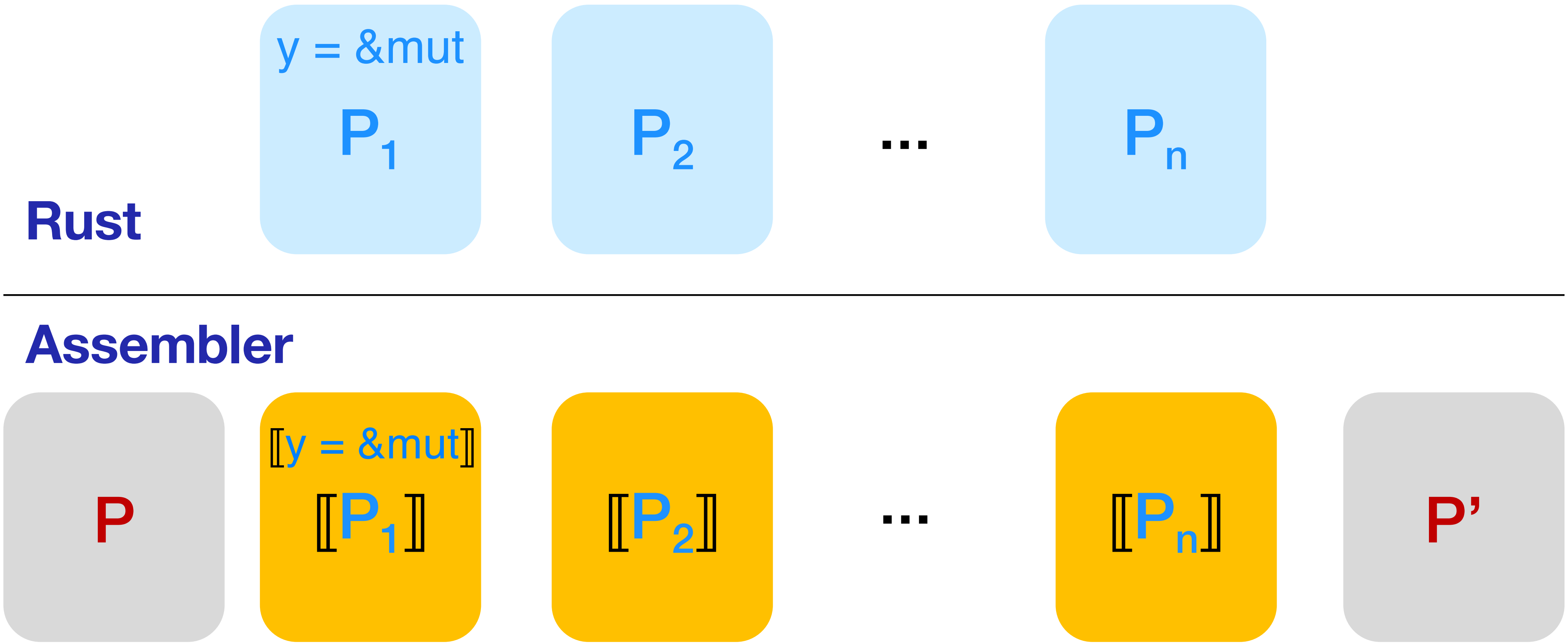
Threat model hint



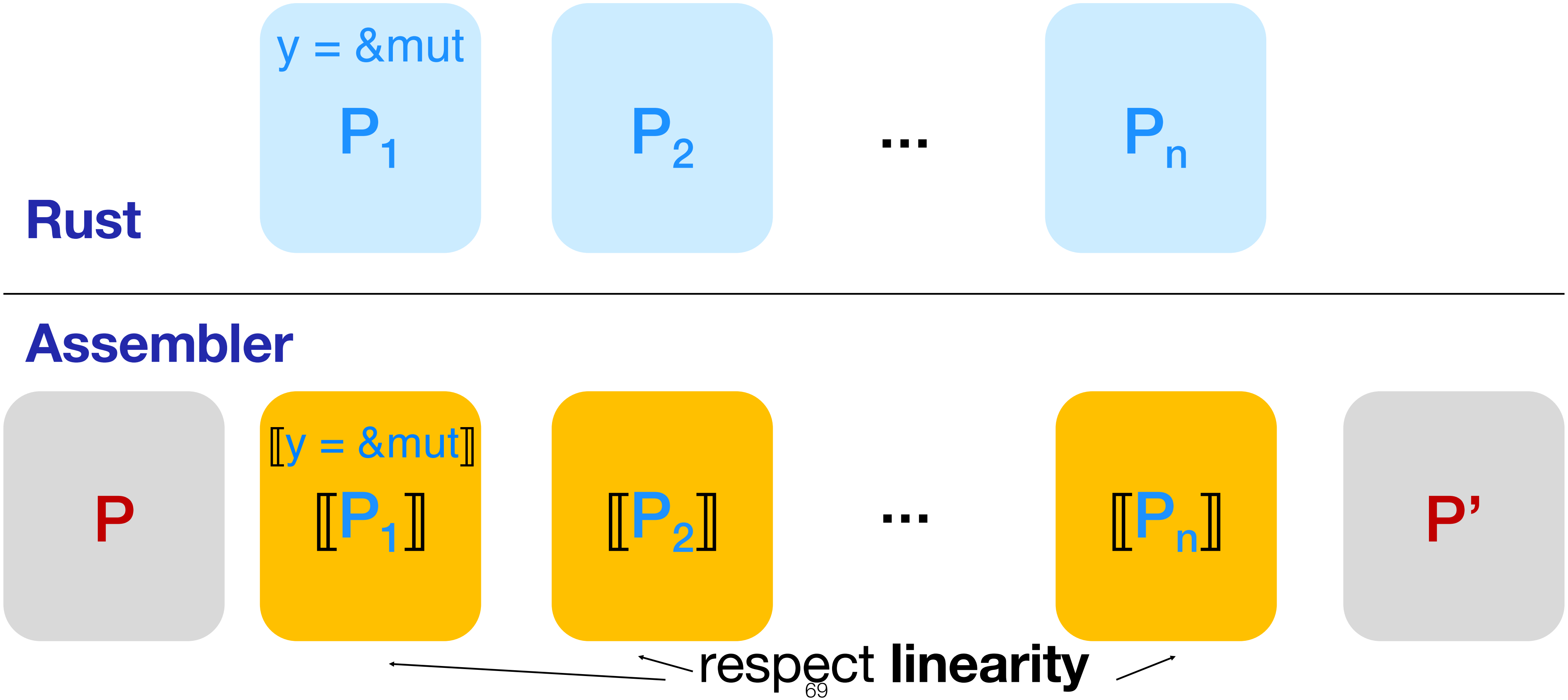
Threat model hint



Correct compilation

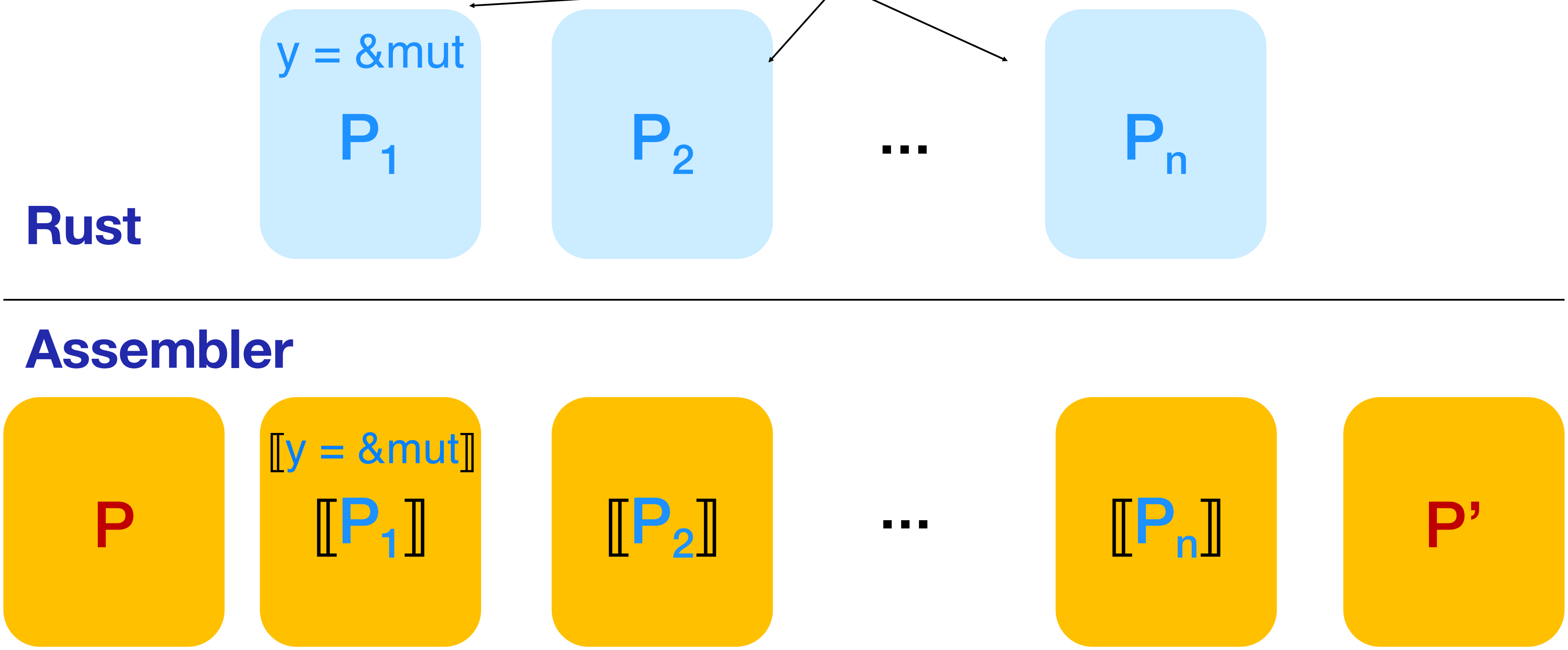


Correct compilation



Secure compilation

Enable source-level security reasoning



Secure compilation

- What does it mean for a compiler $[\![\cdot]\!]^S_T$ to be secure?
- Does a given compiler* preserve the security properties of the source programs?
- Is important this issue?
- What does it mean to preserve security properties? We focus on formal answers
- Intuitively, what is secure in the source must be as secure in the target

Correctness vs security

```
int n = some_pt->n;
if (some_pt == NULL)
    // Some code
use (n)
```

```
int n = some_pt->n;
use (n)
```

```
pin := read_secret();
if (check(pin))
   // OK!

pin := 0; // overwrite the pin
```

```
pin := read_secret();
if (check(pin))
   // OK!
```

Abstraction issues

- A high-level language provides a variety of **abstractions** and **mechanisms** (e.g., types, modules, automatic memory management) that enforce good programming
- Unfortunately, most target languages cannot preserve the abstractions of their source-level counterparts

Abstraction issues (cont.)

- The source-level abstractions can be used to enforce security properties
- when compiled (and linked with adversarial target code) these abstractions are NOT enforced
- Unfortunately, discrepancy between what abstractions the source language offers and what abstraction the target language has, make target language vulnerable to attacks

Abstractions*

- Programming abstractions are not preserved by compilers (linkers, etc) (security is an abstraction)
- What does preserving abstractions mean?
- **Secure compilation** is an emerging research field concerned with the design and the implementation of compilers that preserve source-level security properties at the object level.

*Marco Patrignani, Amal Ahmed, Dave Clarke. *Formal Approaches to Secure Compilation. A Survey of Fully Abstract Compilation and Related Work*. ACM Computing surveys, 2019.

Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation
Fully abstract compilation in practice

Example: compilation

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

Listing 1. Example of Java source code.

Vulnerability addressed
at the beginning of our course

Example: compilation

```
package Bank;

public class Account{
 private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

No access to balance from outside

Listing 1. Example of Java source code.

Example: compilation

```java
package Bank;

public class Account{
 private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

No access to balance from outside
Enforced by the language

Listing 1. Example of Java source code.

Example: compilation

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

No access to balance from outside
Enforced by the language

Listing 1. Example of Java source code.

```
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a->balance += amount;
  return;
}
```

Listing 2. C code obtained from compiling the Java code of Listing 1.

Example: compilation

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

No access to balance from outside
Enforced by the language

Listing 1. Example of Java source code.

```c
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount )
  a→balance += amount;
  return;
}
```

Pointer arithmetic in C can lead to security violation: undesired access to balance

Listing 2. C code obtained from compiling the Java code of Listing 1.

Example: compilation

- When the Java code interacts with other Java code, the latter cannot access the contents of balance, since it is a private field

- However, when the Java code is compiled into the C code and then interacts with arbitrary C code, the latter can access the contents of balance by doing simple pointer arithmetic

- Given a pointer to a C Account `struct`, an attacker can add the size (in words) of an int to it and read the contents of balance, effectively violating a confidentiality property that the source program had

Why?

This violation occurs because the source-level abstractions used to enforce security properties is not preserved by the target languages

Source-Level Abstractions and Target-Level Attacks

- There are several examples of source-level security properties that can be violated by target-level attackers that show the security relevance of compilation

- The capabilities of an attacker vary depending on the target language considered (typed/untyped,…)

- We will present some examples of the relevant threats that a secure compiler needs to mitigate

Threats: confidentiality

```
private secret : Int = 0;

public setSecret() : Int {
  secret = 1;
  return 0;
}
```

Threats: confidentiality

```
private secret : Int = 0;

public setSecret() : Int {
  secret = 1;
  return 0;
}
```

No access to balance from outside

BUT if in the target language locations are identified by nat numbers then the address of secret can be read, by dereferencing the number

Threats: integrity

```
public proxy( callback : Unit → Unit ) : Int {
  var secret = 1;
  callback();
  return 0;
}
```

Threats: integrity

```
public proxy( callback : Unit → Unit ) : Int {
  var secret = 1;
  callback();
  return 0;
}
```

The variable secret is inaccessible to the code in the callback function at the source level

If the target language can manipulate the call stack, it can access the secret variable and change its value

Threats: memory size

```
public kernel( n : Int, callback : Unit →Unit ) : Int {
  for (i = 0 to n){
   new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

Threats: memory size

```
public kernel( n : Int, callback : Unit →Unit ) : Int {
  for (i = 0 to n){
    new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

If the target language can allocate only n objects and callback allocates another object, then the security relevant code will not be executed

Threats: deterministic memory allocation

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

Threats: deterministic memory allocation

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

At source level, `Object y` is inaccessible.

A target level attacker that knows the memory allocation order and can guess where an object will be allocated and influence its memory contents

Threats: well-typedness

```
class Pair {
  private first, second : Obj = null;
  public getFirst(): Obj {
    return this.first;
  }
}
class Secret {
  private secret : Int = 0;
}
object o : Secret
```

Threats: well-typedness

```
class Pair {
  private first, second : Obj = null;
  public getFirst(): Obj {
    return this.first;
  }
}
class Secret {
  private secret : Int = 0;
}
object o : Secret
```

- The Pair class provides a way to store a pair of objects, but it is incomplete as there are no methods to set its values. It only has a method to retrieve the value of first
- The Secret class does not expose any methods to access or modify its secret value
- The o object of type Secret is declared but not utilized

Threats: well-typedness

```
class Pair {
  private first, second : Obj = null;
  public getFirst(): Obj {
    return this.first;
  }
}
class Secret {
  private secret : Int = 0;
}
object o : Secret
```

At target level, an attacker can call `getFirst()` with current object o; this will return the secret field, since fields are accessed by offset in untyped assembly

- An attacker, aware of the memory layout of the Secret class, could determine the offset of the secret field within an instance of the Secret class in memory.
- By performing a method call on the o object of type Secret, the attacker could potentially access the memory location corresponding to the secret field using the offset obtained earlier.
- This would effectively bypass the access control mechanisms provided by Java and allow the attacker to read the value of the secret field, despite it being declared as private.

Threats: information flow

```
public isZero( value : Int_h ) : Int_l {
  if ( value == 0 ) {
    return 1
  }
  return 0
}
```

Listing 4. Example code with indirect information flow.

Threats: information flow

```
public isZero( value : Int_h ) : Int_l {
  if ( value == 0 ) {
    return 1
  }
  return 0
}
```

The attacker can detect whether value is 0 or not by observing the output. The target language that doesn't prevent information flow cannot withstand these leaks

Secure compilation

- **[Formally] secure compilation** studies compilers that preserve the security properties of source languages in their compiled, target level counterparts

- What does it mean to preserve security properties across compilation?

- Roughly, having that something secure in the source is still secure in the target

Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation
Fully abstract compilation in practice

Program equivalences

- A possible way to know what is secure in a program is by exploiting **program equivalences**

Are these programs equivalent?

```
public Bool getTrue( x : Bool )
  return true;
```

- P1

```
public Bool getTrue( x : Bool )
  return x or true;
```

- P2

```
public Bool getTrue( x : Bool )
  return x and false;
```

- P3

```
public Bool getTrue( x : Bool )
  return false;
```

- P4

```
public Bool getFalse( x : Bool )
  return x and true;
```

- P5

Are these programs equivalent?

```
public Bool getTrue( x : Bool )
  return true;
```

• P1

=

```
public Bool getTrue( x : Bool )
  return x or true;
```

• P2

```
public Bool getTrue( x : Bool )
  return x and false;
```

• P3

=

```
public Bool getTrue( x : Bool )
  return false;
```

• P4

```
public Bool getFalse( x : Bool )
  return x and true;
```

• P5

Program equivalences

A possible way to know what is secure in a program is by exploiting **program equivalences**

Roughly, two programs are equivalent when they behave the same even if they are different (same semantics and possibly different syntax)… in a way that respects the security property

- **contextual equivalence**
- observational equivalence (e.g., for non-interference property)
- timing/resource-sensitive equivalence (e.g., security of constant-time code)

Confidentiality as equivalence

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

If the two snippets are equivalent
then secret is confidential

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 1;
  return 0;
}
```

Confidentiality as equivalence

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

If the two snippets are equivalent then secret is confidential

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 1;
  return 0;
}
```

With a Java-like semantics, secret is never accessed from outside
With a C-like semantics, secret can be accessed from outside

Integrity as equivalence

```
public proxy( callback : Unit → Unit )
        : Int {
  var secret = 0;
  callback();
  if ( secret == 0 ) {
    return 0;
  }
  return 1;
}
```

If the two snippets are equivalent then secret cannot be modified during the callback

```
public proxy( callback : Unit → Unit )
        : Int {
  var secret = 0;
  callback();

  return 0;


}
```

Integrity as equivalence

```
public proxy( callback : Unit → Unit )
       : Int {
  var secret = 0;
  callback();
  if ( secret == 0 ) {
    return 0;
  }
  return 1;
}
```

If the two snippets are equivalent then secret cannot be modified during the callback

```
public proxy( callback : Unit → Unit )
       : Int {
  var secret = 0;
  callback();

  return 0;


}
```

When `callback()` is invoked, `secret` is on the stack

Unbounded memory size as equivalence

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  for (Int i = 0; i < n; i++){
    new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {



  callback();
  // security-relevant code
  return 0;
}
```

Unbounded memory size as equivalence

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  for (Int i = 0; i < n; i++){
    new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

If the target language can
allocate only n objects
and callback allocates another object,
then the security
relevant code will not be executed

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {



  callback();
  // security-relevant code
  return 0;
}
```

Memory allocation as equivalence

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return y;
}
```

Memory allocation as equivalence

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return y;
}
```

If the two snippets are equivalent then the memory order is invisible to an attacker

Question

Does compilation transform equivalent source-level components into equivalent target-level ones?

Hence

- The assumption is that program equivalence capture security properties of source code

Question

How to express program equivalence?

Which program equivalence?

How to express program equivalence?

Contextual equivalence

Two programs are contextually equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs

Contextual equivalence

How to express program equivalence?

Contextual equivalence

Two programs are equivalent if no matter what context/external observer interacts with them that observer cannot distinguish the programs

$$\boxed{P_1 \simeq_{ctx} P_2 = \forall\, \mathscr{C}.\ \mathscr{C}(P_1)\downarrow \Leftrightarrow \mathscr{C}(P_2)\downarrow}$$

$\downarrow$ refers to termination

Contextual equivalence

The external observer C is generally called context

- it is a program, written in the same language as P_1 and P_2
- it is the same program C interacting with both P_1 and P_2 in two different runs
- so, it cannot express out of language attacks (e.g., side channels)
- interaction means link and run together (like a library)

Contextual equivalence

Distinguishing means: terminate with different values

- the observer basically asks the question: is this program P_1?
- if the observer can find a way to distinguish P_1 from P_2, it will return true, otherwise false
- often, we use divergence and termination as opposed to this boolean termination

Contexts: an example

Partial programs: sequences of assignments of expressions to locations.

- Expressions: combination of arithmetic operators and variables a0, a1,…
- Locations: $X_0, X_1, \ldots$
- Contexts: non-empty lists of natural numbers
- Linking a context C and a partial program P gives a whole program C[P] in which the variables in expressions of P are initialized with the information provided by C

P	If **C** = [2,3,7], then **C[P]** is
X_0 := (a0 x a1) x a2	X_0 := (2 x 3) x 7
X_1 := a0	X_1 := 2

Context

- A **context C** is a program with a **hole** (denoted by **[.]**), which can be filled by a component **P**, generating the whole program **C[P]** to be executed

- Plugging a component in a context makes the program whole, so its behaviour can be observed via its operational semantics

Contextual equivalence

In a simple functional language

$$\lambda x.\ x * 2 \approx \lambda x.\ x + x$$

Contextual equivalence

In a simple functional language

$$\lambda x.\ x * 2 \approx \lambda x.\ x + x$$

in Javascript

function(x){ return x * 2; } ≈ function(x){ return x + x; }?

Contextual equivalence

In a simple functional language

$$\lambda x.\ x * 2 \approx \lambda x.\ x + x$$

In Javascript

function(x){ return x * 2; } ≈ function(x){ return x + x; }?

Is there a program context that can distinguish them?

Contextual equivalence

In a simple functional language

$$\lambda x.\ x * 2 \approx \lambda x.\ x + x$$

In Javascript

function(x){ return x * 2; } ≈ function(x){ return x + x; }?

Is there a program context that can distinguish them?

Yes: ([·]).toString()

Troublesome for optimizations

toString() returns the source code of the function

Confidentiality as equivalence

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 1;
  return 0;
}
```

```
// Observer P in Java
public static isItP1( ) : Bool
{
Secret.getSecret();
...
}
```

P cannot tell the difference!

Confidentiality as equivalence

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 1;
  return 0;
}
```

```
// Observer P in Java
public static isItP1( ) : Bool {
Secret.getSecret();
...
}
```

P cannot tell the difference!
There is no `getSecret()` method

Confidentiality as equivalence

```
typedef struct secret { // P1
int secret = 0;
void (*setSec) (struct Secret*) = setSec;
} Secret;
void setSec(Secret* s) {s→secret = 0; return;}
```

```
typedef struct secret { // P2
int secret = 0;
void (*setSec) (struct Secret*) = setSec;
} Secret;
void setSec(Secret* s) {s→secret = 1; return;}
```

Confidentiality as equivalence

```
typedef struct secret { // P1
int secret = 0;
void (*setSec) (struct Secret*) = setSec;
} Secret;
void setSec(Secret* s) {s→
```

```
typedef struct secret { //
int secret = 0;
void (*setSec) (struct Secr
} Secret;
void setSec(Secret* s) {s→
```

```
// Observer P in C
int isItP1( ){
struct Secret x;
sec = &x + sizeof(int);
if *sec == 0 then return true else return false
}
```

P can see the difference!
The two programs are inequivalent

Security violations

Inequivalences as security violations

if the target programs are not equivalent then the intended security property is violated

Security preservation

What does it mean to preserve security properties across compilation?

Security preservation

What does it mean to preserve security properties across compilation?

Given source equivalent programs (which have a security property), compile them into equivalent target programs

Security preservation

What does it mean to preserve security properties across compilation?

Given source equivalent programs (which have a security property), compile them into equivalent target programs, provided that:

- the security property is captured in the source by program equivalence

Being equivalent in the target means contextual equivalence w.r.t. target observers (i.e., target programs), i.e., the attackers in this setting

Secure compilation

Recall that

- Attackers are modelled as the environment programs to be checked interact with (link and run together)
- Attackers can act and not only observe the program behaviour

Partial programs

Note that

- For **correct compilation**, we considered whole programs
- **Secure compilation** is instead concerned with the security of partial programs (or components) that are linked together with an environment (or context)

Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation
Fully-abstract compilation in practice

Formal (secure compilation): full abstraction

A compiler $[\![.]\!]$ is **fully abstract** when it translates equivalent source-level components into equivalent target-level ones

$\forall\ P_1, P_2$

$$P_1 \simeq_{\mathrm{ctx}} P_2 \Rightarrow [\![P_1]\!] \simeq_{\mathrm{ctx}} [\![P_2]\!]$$

If the target programs are not equivalent, then the intended security property is violated

Is it enough?

Formal (secure compilation): full abstraction

A compiler $[\![.]\!]$ is **fully abstract** when it translates equivalent source-level components into equivalent target-level ones

$\forall\ P_1, P_2$

$$P_1 \simeq_{\text{ctx}} P_2 \Rightarrow [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]$$

If the target programs are not equivalent, then the intended security property is violated

Is it enough? No, it is not

An empty translation would fit

Correctness is needed

We need the compiler also to be correct, i.e.,

$\forall\ P_1, P_2$

$$P_1 \simeq_{\text{ctx}} P_2 \Leftarrow [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]$$

Fully Abstract compilation

Then we have

$\forall\ P_1, P_2$

$$\boxed{P_1 \simeq_{\text{ctx}} P_2 \Leftrightarrow [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]}$$

Correctness

Reflection of $\simeq$

$P_1 \simeq P_2 \Leftarrow [\![P_1]\!] \simeq [\![P_2]\!]$

The compiler outputs behave as their source-level counterparts

Security

Preservation of $\simeq$

$P_1 \simeq P_2 \Rightarrow [\![P_1]\!] \simeq [\![P_2]\!]$

The source-level abstractions are not violated in the target-level output

Fully Abstract compilation

Then we have

$\forall\, P_1, P_2$

$$\boxed{P_1 \simeq_{\text{ctx}} P_2 \Leftrightarrow [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]}$$

$\forall\, P_1, P_2.$

$$\boxed{\forall\, \mathscr{C}..\ \mathscr{C}(P_1) \approx_{\text{ctx}} \mathscr{C}(P_2) \Rightarrow \forall\, \mathscr{C}..\ \mathscr{C}([\![P_1]\!]) \approx_{\text{ctx}} \mathscr{C}([\![P_2]\!])}$$

Correctness is needed

We need the compiler to be correct

$$\forall\, P_1, P_2 .\, P_1 \simeq_{\text{ctx}} P_2 \Leftarrow [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]$$

- The contrapositive* is actually easier to understand: if the source components were not equivalent then the target components would have to be different
- If this did not hold, then the compiler could take different source components and compile them to the same target component

* P ⇒ Q corresponds to ¬ Q ⇒ ¬ P (If it is raining, then I open my umbrella" — "If I do not open my umbrella, then it is not raining.")

Correctness is needed

Note that equivalence is related to compiler correctness, but the two notions do not coincide

$$\forall P_1, P_2 . P_1 \simeq_{ctx} P_2 \Leftarrow [\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$$

As long as your compiler produced different target programs for different source programs, all would be fine: e.g., hash the source program and produce target programs that just printed the hash

Different target programs not only for source programs that were observationally different, but even syntactically different!

A pretty bad compiler, and certainly not correct, but it would be equivalence reflecting

Equivalence preservation

Equivalence preservation is the hallmark of fully abstract compilers

$$\forall\, P_1, P_2.\; P_1 \simeq_{ctx} P_2 \Rightarrow [\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$$

Observers in the target cannot make observations that are not possible to distinguish in the source

If a programming language includes features for information hiding, such as private fields, and two source components are identical except for having different values stored in those private fields, problems can arise if the compiler does not properly preserve the privacy of those fields

Compiler full abstraction

If two programs are equivalent in the source language (i.e., no source context can distinguish them), the two programs obtained by compiling them are equivalent in the target language (i.e., no target context can distinguish them)

A fully abstract compilation chain protects source-level abstractions all the way down, ensuring that linked adversarial target code cannot observe more about the compiled program than what some linked source code could about the source program

A fully abstract compiler does not eliminate source-level security flaws, it only introduces no more vulnerabilities at the target-level

Compiler full abstraction

- Equivalence-preserving compilation considers all target-level contexts when establishing indistinguishability, so it captures the power of an attacker operating at the level of the target language

- Full abstraction allows for source-level reasoning: the programmer need not be concerned with the behaviour of target-level code (attackers) and can focus only potential source-level behaviors when reasoning about safety and security properties of their code

Compiler full abstraction

- FA only preserves security property expressed as program equivalence

- FA is not the silver bullet: there are shortcomings of fully abstract compilation
  - Difficult to (dis)-prove a compiler (not) to be FA
  - FA compilers may produce inefficient code
  - Mainstream compilers are not usually FA

Proving full abstraction

A compiler $[\![.]\!]$ is fully abstract if it reflects and preserves the contextual equivalence

$$\mathit{Preservation} = \forall \mathrm{P}_1, \mathrm{P}_2 \in \mathrm{S}.\ \mathrm{P}_1 \simeq_{\mathrm{ctx}} \mathrm{P}_2 \Rightarrow [\![\mathrm{P}_1]\!]^{\mathrm{S}}_{\mathrm{T}} \simeq_{\mathrm{ctx}} [\![\mathrm{P}_2]\!]^{\mathrm{S}}_{\mathrm{T}},$$

$$\mathit{Reflection} = \forall \mathrm{P}_1, \mathrm{P}_2 \in \mathrm{S}.\ [\![\mathrm{P}_1]\!]^{\mathrm{S}}_{\mathrm{T}} \simeq_{\mathrm{ctx}} [\![\mathrm{P}_2]\!]^{\mathrm{S}}_{\mathrm{T}} \Rightarrow \mathrm{P}_1 \simeq_{\mathrm{ctx}} \mathrm{P}_2.$$

Recall that $P_1 \simeq_{ctx} P_2$ means that $\forall\ \mathscr{C}.\ \mathscr{C}(P_1)\downarrow \Leftrightarrow \mathscr{C}(P_2)\downarrow$

- Both parts are difficult to prove
- Preservation is particularly tricky because of $\simeq$ and $\forall$ contexts

Proving preservation

Preservation

Unfolding context equivalence at the target level:

$\forall\ P_1, P_2.$

$P_1 \simeq_{ctx} P_2 \Rightarrow \forall\ \mathscr{C}..\ \mathscr{C}(\llbracket P_1 \rrbracket)\downarrow\ \Leftrightarrow\ \mathscr{C}(\llbracket P_2 \rrbracket)\downarrow$

Contrapositive*: $\forall\ P_1, P_2.$

$\exists\ \mathscr{C}..\ \mathscr{C}(\llbracket P_1 \rrbracket)\downarrow\ \not\Leftrightarrow\ \mathscr{C}(\llbracket P_2 \rrbracket)\downarrow\ \Rightarrow\ P_1 \not\simeq_{ctx}\ P_2$

Unfolding context equivalence at the source level:

$$\boxed{\exists\ \mathscr{C}..\ \mathscr{C}(\llbracket P_1 \rrbracket)\downarrow\ \not\Leftrightarrow\ \mathscr{C}(\llbracket P_2 \rrbracket)\downarrow\ \Rightarrow \exists\ \mathscr{C}..\ \mathscr{C}(P_1)\downarrow\ \not\Leftrightarrow\ \mathscr{C}(P_2)\downarrow}$$

* P ⇒ Q corresponds to ¬ Q ⇒ ¬ P

Proving preservation

Preservation

Unfolding context equivalence at the target level:

$\forall\ P_1, P_2.$

$P_1 \simeq_{ctx} P_2 \Rightarrow \forall\ \mathscr{C}..\ \mathscr{C}(\llbracket P_1 \rrbracket)\downarrow \Leftrightarrow \mathscr{C}(\llbracket P_2 \rrbracket)\downarrow$

Contrapositive*: $\forall\ P_1, P_2.$

$\exists\ \mathscr{C}..\ \mathscr{C}(\llbracket P_1 \rrbracket)\downarrow \not\Leftrightarrow \mathscr{C}(\llbracket P_2 \rrbracket)\downarrow\ \Rightarrow\ P_1 \not\simeq_{ctx}\ P_2$

Preservation amounts to proving that no new vulnerabilities are introduced at the target-level

Unfolding context equivalence at the source level:

$$\boxed{\exists\ \mathscr{C}..\ \mathscr{C}(\llbracket P_1 \rrbracket)\downarrow \not\Leftrightarrow \mathscr{C}(\llbracket P_2 \rrbracket)\downarrow \Rightarrow \exists\ \mathscr{C}..\ \mathscr{C}(P_1)\downarrow \not\Leftrightarrow \mathscr{C}(P_2)\downarrow}$$

* P ⇒ Q corresponds to ¬ Q ⇒ ¬ P

Proving preservation

Preservation:

$\forall\, P_1, P_2.\ \forall\, \mathscr{C}..\ \mathscr{C}(P_1) \approx_{\text{ctx}} \mathscr{C}(P_2) \Rightarrow \forall\, \mathscr{C}..\ \mathscr{C}([\![P_1]\!]) \approx_{\text{ctx}} \mathscr{C}([\![P_2]\!])$

$$\begin{array}{ccc} \mathscr{C}(P_1) & \approx_{\text{ctx}} & \mathscr{C}(P_2) \\ \approx & & \approx \\ \mathscr{C}([\![P_1]\!]) & \overset{?}{\approx_{\text{ctx}}} & \mathscr{C}([\![P_2]\!]) \end{array}$$

Security proof schema

$$\forall\ P_1,\ P_2 .\ P_1 \simeq_{ctx} P_2 \Rightarrow [\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$$

Given an arbitrary target-level context $\mathscr{C}$, we need to prove that

$$\mathscr{C}([\![P_1]\!]) \simeq_{ctx} \mathscr{C}([\![P_2]\!])$$

If the source language is strong enough, we can construct a back-translation «$\mathscr{C}$» for any target-level context $\mathscr{C}$, obtaining a corresponding valid source-level context $\mathscr{C}'$ “observational equivalent to” $\mathscr{C}$

Back-translations

Back-translation of target-level program contexts to behaviourally-equivalent source-level contexts, a sort of compiler in reverse

$[\![.]\!]$: $S \rightarrow T$
«.»: $T \rightarrow S$

For any target-level context $\mathfrak{C}$ returns a corresponding valid source-level context $\mathfrak{C}$

Constructing such a back-translation can be difficult when the source language is not strong enough to embed an encoding of the target language

Security proof schema

$$\forall\, P_1, P_2 .\ P_1 \simeq_{ctx} P_2 \Rightarrow [\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$$

Given an arbitrary target-level context $\mathscr{C}$, we need to prove that

$$\mathscr{C}([\![P_1]\!]) \simeq_{ctx} \mathscr{C}([\![P_2]\!])$$

If the source language is strong enough, we can construct a back-translation «$\mathscr{C}$» for any target-level context $\mathscr{C}$, obtaining a corresponding valid source-level context $\mathscr{C}'$ “observational equivalent to” $\mathscr{C}$

If $\mathscr{C}'(P_1) \simeq \mathscr{C}([\![P_1]\!])$ and $\mathscr{C}'(P_2) \simeq \mathscr{C}([\![P_2]\!])$ we can prove that

$$\mathscr{C}([\![P_1]\!]) \simeq_{ctx} \mathscr{C}([\![P_2]\!])$$

Proving preservation

Preservation:

$\forall\, P_1, P_2.\ \forall\, \mathscr{C}_S.\ \mathscr{C}_S(P_1) \approx_{\text{ctx}} \mathscr{C}_S(P_2) \Rightarrow \forall\, \mathscr{C}_T.\ \mathscr{C}_T(\llbracket P_1 \rrbracket) \approx_{\text{ctx}} \mathscr{C}_T(\llbracket P_2 \rrbracket)$

$$\begin{array}{ccc} \mathscr{C}(P_1) & \approx_{\text{ctx}} & \mathscr{C}(P_2) \\ \approx & ? & \approx \\ \mathscr{C}(\llbracket P_1 \rrbracket) & \approx_{\text{ctx}} & \mathscr{C}(\llbracket P_2 \rrbracket) \end{array}$$

Security proof schema

$$
\begin{array}{ccc}
P_1 & \simeq_{ctx} & P_2 \\
\simeq & & \simeq \\
[\![P_1]\!] & \overset{?}{\simeq}_{ctx} & [\![P_2]\!]
\end{array}
$$

Security proof schema

$$P_1 \simeq_{ctx} P_2$$

$$\simeq \qquad\qquad \simeq$$

$$[\![P_1]\!] \overset{?}{\simeq}_{ctx} [\![P_2]\!]$$

$$\mathcal{C}([\![P_1]\!])\downarrow \overset{?}{\Rightarrow} \mathcal{C}([\![P_2]\!])\downarrow$$

Given an arbitrary context $\mathcal{C}$, we need to prove that $\mathcal{C}([\![P_1]\!]) \simeq_{ctx} \mathcal{C}([\![P_2]\!])$

If «$\mathcal{C}$» = $\mathcal{C}'$ such that «$\mathcal{C}$» = $\mathcal{C}' \simeq \mathcal{C}$ and $P_i \simeq [\![P_i]\!]$ and hence $\mathcal{C}'(P_1) \simeq \mathcal{C}([\![P_1]\!])$, we have done

Security proof schema

$$P_1 \simeq_{ctx} P_2$$

$$«\mathcal{C}» \simeq \mathcal{C} \qquad\qquad «\mathcal{C}» \simeq \mathcal{C}$$

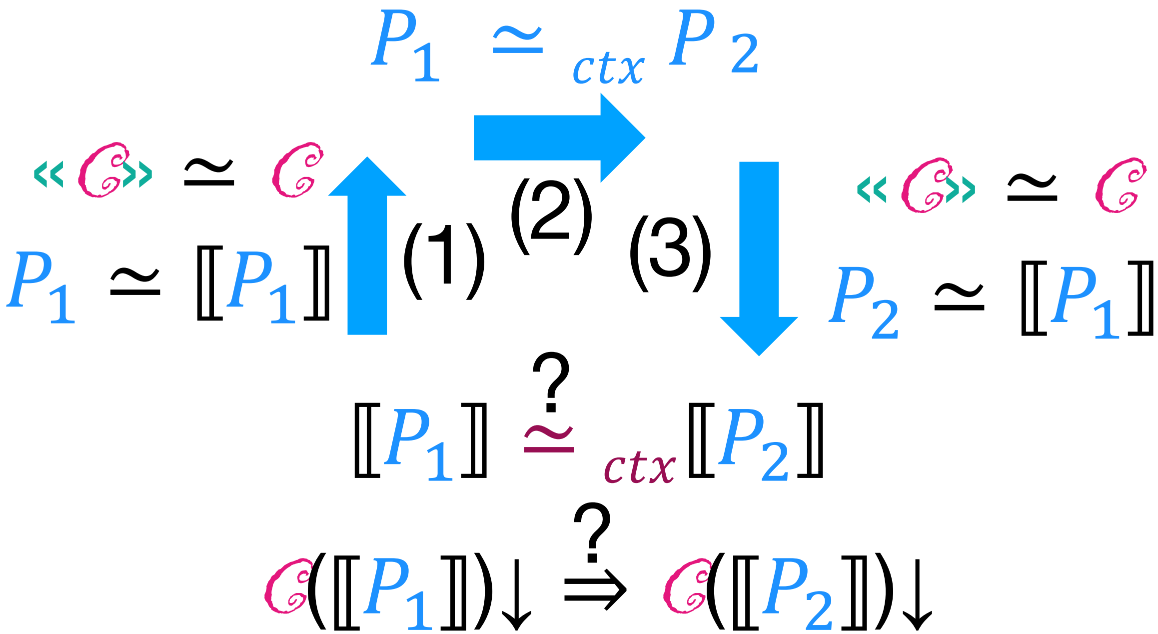
$$P_1 \simeq [\![P_1]\!] \qquad\qquad P_2 \simeq [\![P_1]\!]$$

$$[\![P_1]\!] \overset{?}{\simeq}_{ctx} [\![P_2]\!]$$

$$\mathcal{C}([\![P_1]\!])\downarrow \overset{?}{\Rightarrow} \mathcal{C}([\![P_2]\!])\downarrow$$

Given an arbitrary context $\mathcal{C}$, we need to prove that $\mathcal{C}([\![P_1]\!]) \simeq_{ctx} \mathcal{C}([\![P_2]\!])$

Security proof schema



To prove $\mathcal{C}([\![P_1]\!]) \simeq_{ctx} \mathcal{C}([\![P_2]\!])$ we exploit the chain of equivalences (1), (2), (3), where (2) comes from the assumed (contextual) equivalence of P_1 and P_2

Note on equivalences

Some kinds of equivalences may simplify these proofs. E.g., contextual equivalence at the target level can be replaced with trace equivalence if it is proved just as precise

- **Context-based**: relies on the structure of the context

- **Trace-based**: relies on trace semantics

Note on equivalences

Some kinds of equivalences may simplify these proofs. E.g., contextual equivalence at the target level can be replaced with trace equivalence if it is proved just as precise

- **Context-based**: relies on the structure of the context: when source and target contexts are similar

- **Trace-based**: relies on trace semantics: when there is a large abstraction gap between source and target

Trace Semantics

We replace context equivalence with something equivalent

- but simpler to reason about
- a semantics that abstracts from the context (observer)
- and still describes the behaviour of a program precisely

A trace semantics

Trace Semantics

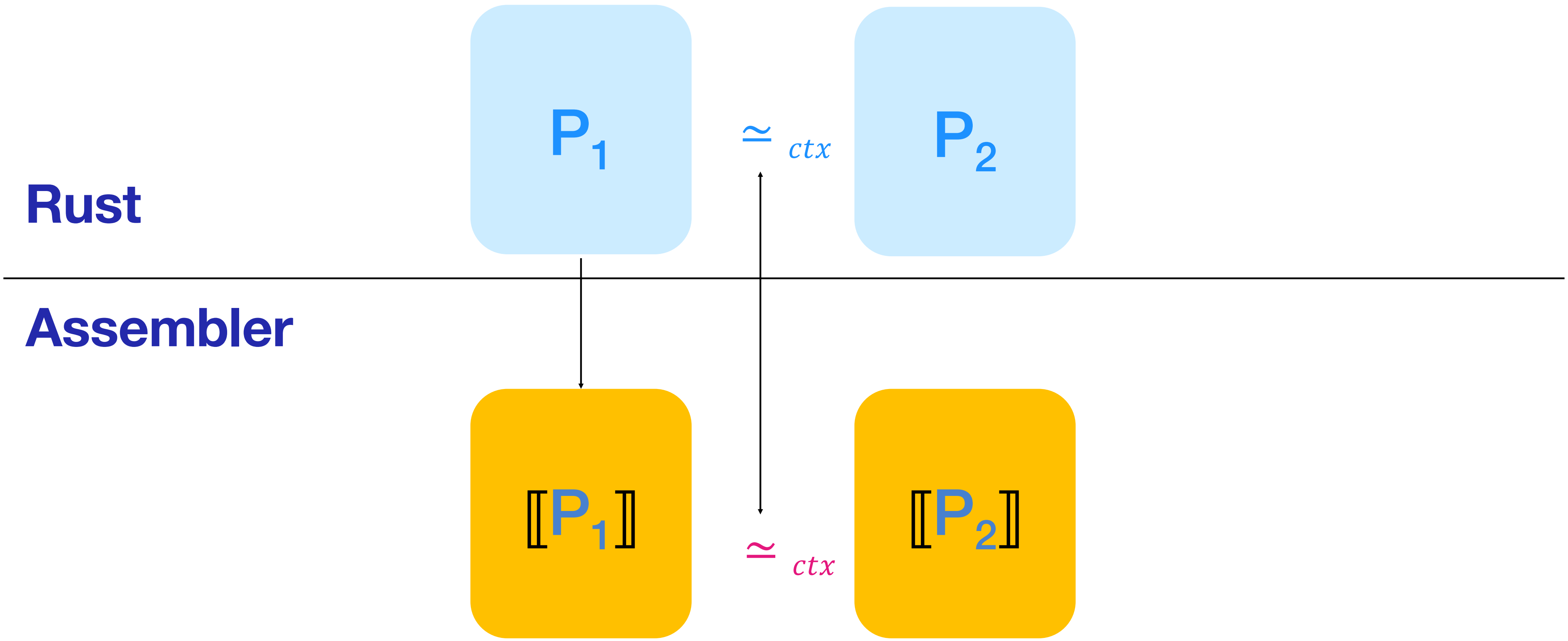
Trace semantics for partial programs (component)

- relies on the operational semantics
- denotational: describes the behaviour of a component as sets of traces
- a trace is (typically) a sequence of actions that describe how a component interacts with an observer
- without needing to specify the observer

- Trace semantics come with trace equivalence
- Two programs are trace equivalent, when the sets of traces coincide
- We need a trace equivalence that is correct and complete w.r.t. context equivalence

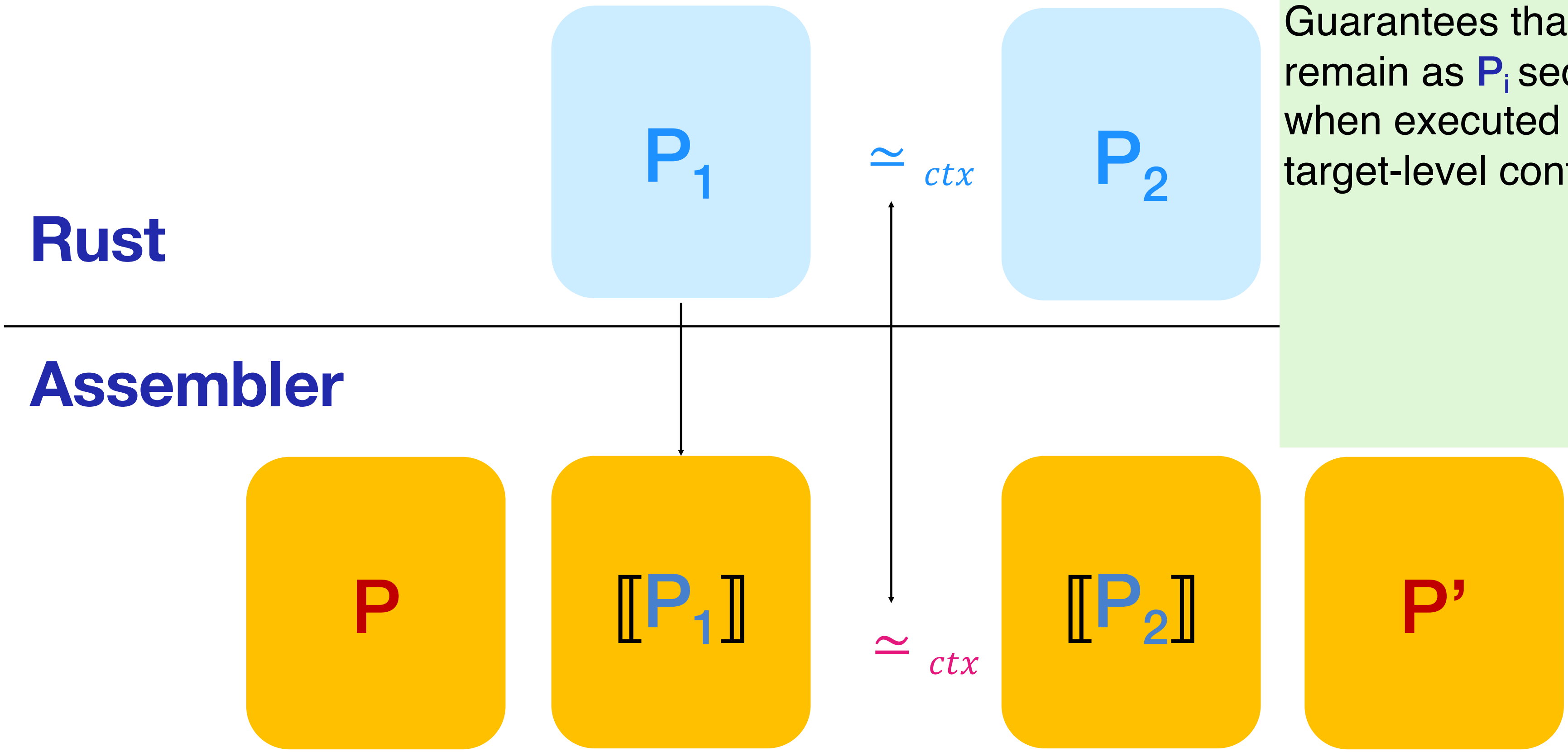
Full abstract compilation

Preserve and reflect contextual equivalence



Full abstract compilation

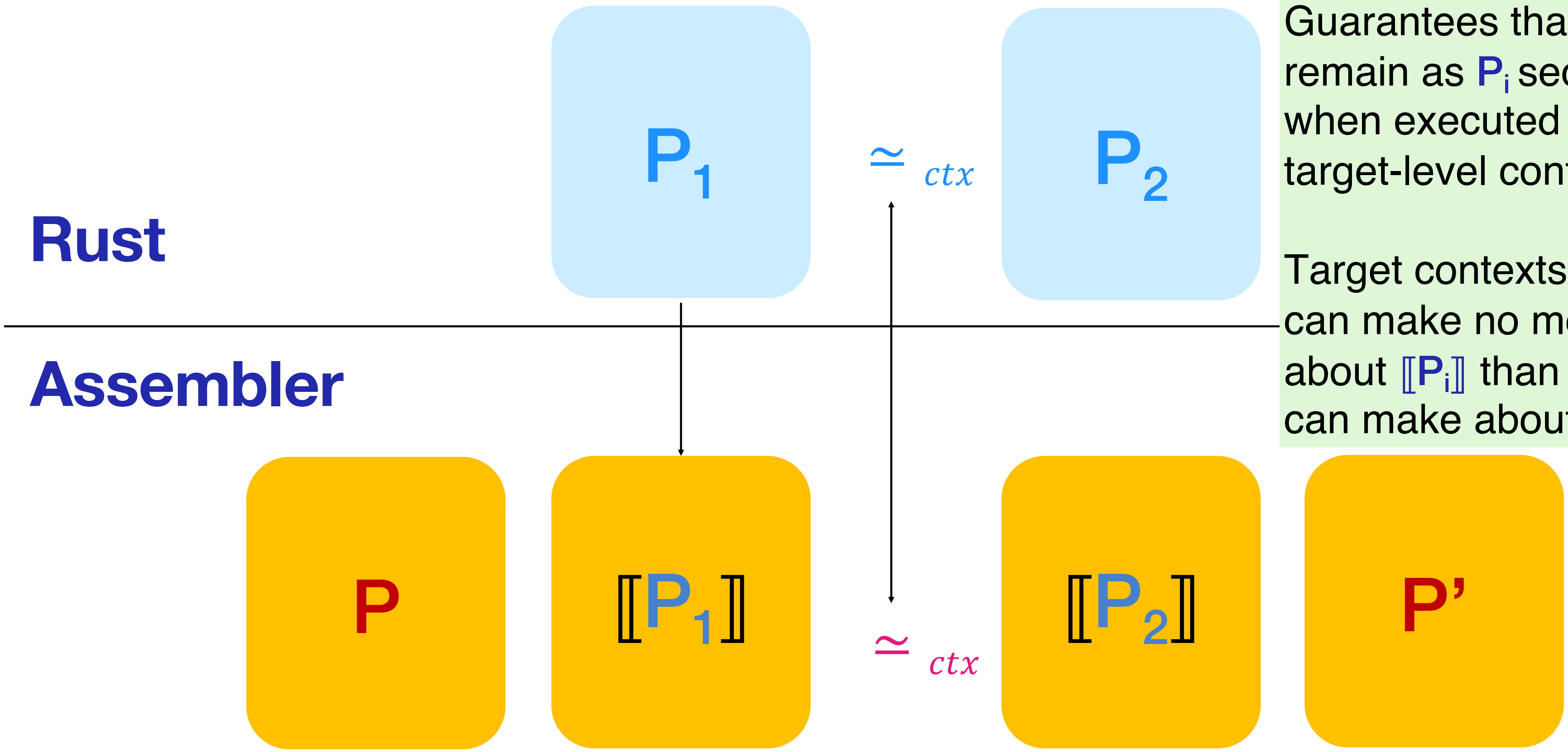
Preserve and reflect contextual equivalence



Guarantees that $[\![P_i]\!]$ will remain as P_i secure as when executed in arbitrary target-level contexts

Full abstract compilation

Preserve and reflect contextual equivalence

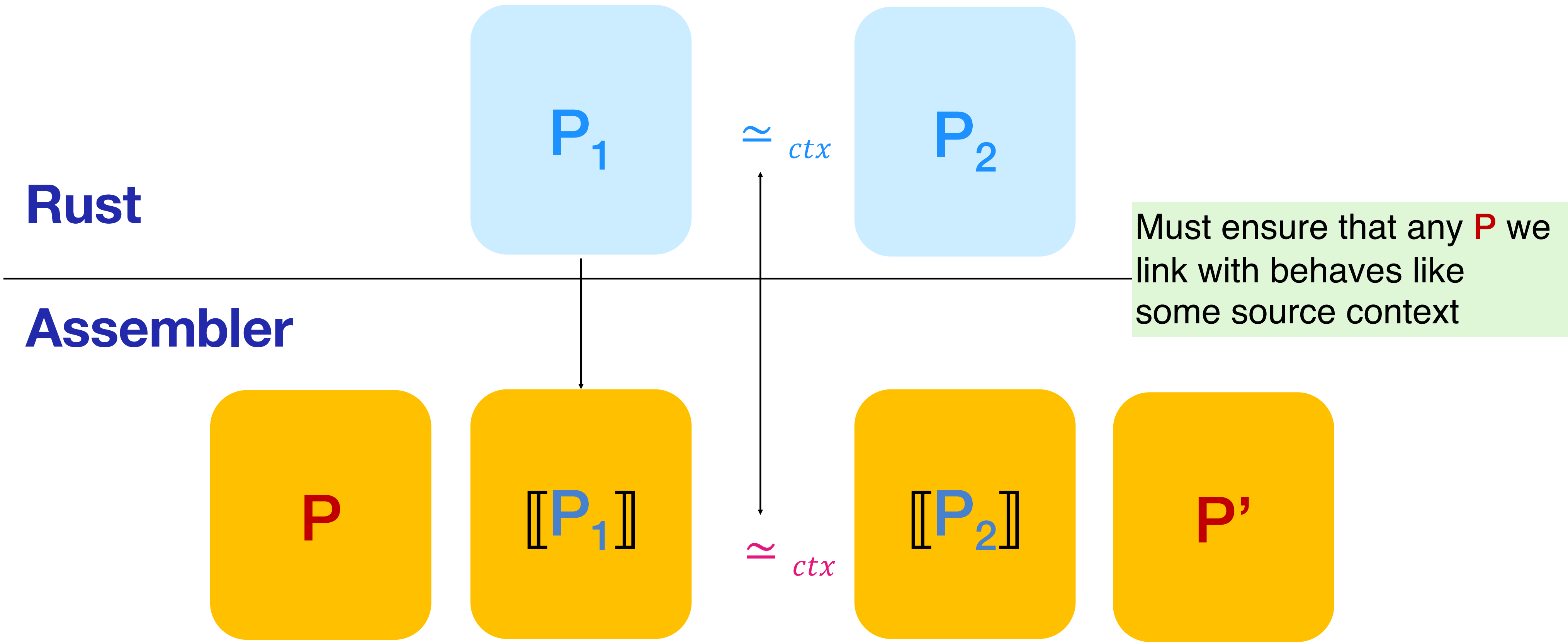


Guarantees that ⟦P_i⟧ will remain as P_i secure as when executed in arbitrary target-level contexts

Target contexts (attackers P, P’) can make no more observations about ⟦P_i⟧ than a source context can make about P_i

Full abstract compilation

Preserve and reflect contextual equivalence



Fully abstract compilation in practice*

Back to our previous example of compilation from a simple imperative language IMP to a stack-based Virtual Machine VM

We will use **Coq**, a formal proof management system

*https://dbp.io/essays/2018-04-19-how-to-prove-a-compiler-fully-abstract.html

IMP Arithmetic expressions

```
Inductive Expr : Set := | Num : Z -> Expr | Plus : Expr -> Expr -> Expr.
```

Evaluation function with 2 constructors

```
Fixpoint eval_Expr (e : Expr) : Z := match e with
| Num n => n  /*takes n and constructs an expr representing that number */
```

IMP Arithmetic expressions

Inductive Expr : Set := | Num : Z -> Expr | Plus : Expr -> Expr -> Expr.

Evaluation function with 2 constructors

Fixpoint eval_Expr (e : Expr) : Z := match e with
| Num n => n /*takes n and constructs an expr representing that number */
| Plus e1 e2 => eval_Expr e1 + eval_Expr e2 /*takes two arguments and constructs an expression representing their sum. */
end.

IMP Arithmetic expressions

Inductive Expr : Set := | Num : Z -> Expr | Plus : Expr -> Expr -> Expr.

Evaluation function with 2 constructors

Fixpoint eval_Expr (e : Expr) : Z := match e with
| Num n => n /*takes n and constructs an expr representing that number */
| Plus e1 e2 => eval_Expr e1 + eval_Expr e2 /*takes two arguments and constructs an expression representing their sum. */
end.

Example: eval_Expr(Plus (Num 1) (Num 2)) = eval_Expr(1) + eval_Expr(2) gives 3 as output

VM Operations

```
Inductive Op : Set :=
| Push : Z -> Op /* takes an integer value n and constructs an operation to push that value onto the stack */
```

VM Operations

```
Inductive Op : Set :=
| Push : Z -> Op /* takes an integer value n and constructs an operation to push that value onto the stack */
| Add : Op /*represents addition on the top two elements of the stack */
```

VM Operations

```
Inductive Op : Set :=
| Push : Z -> Op /* takes an integer value n and constructs an operation to push that value onto the stack */
| Add : Op /*represents addition on the top two elements of the stack */
| OpCount : Op. /* returns the count of remaining operations on the stack machine and puts that integer on the top of the stack */
```

VM Operations

Inductive Op : Set :=
| Push : Z -> Op /* takes an integer value n and constructs an operation to push that value onto the stack */
| Add : Op /*represents addition on the top two elements of the stack */
| OpCount : Op. /* returns the count of remaining operations on the stack machine and puts that integer on the top of the stack */

The stack must be empty at the end of execution

Evaluation of a list of VM operations

Fixpoint eval_Op (s : list Z) (ops : list Op) : option Z :=
match (ops, s) with
| ([], [n]) => Some n /* If the list is empty and there is only one element on the stack, it returns Some n, indicating successful evaluation with the result n */

Evaluation of a list of VM operations

Fixpoint eval_Op (s : list Z) (ops : list Op) : option Z :=
match (ops, s) with
| ([], [n]) => Some n /* If the list is empty and there is only one element on the stack, it returns Some n, indicating successful evaluation with the result n */
| (Push z :: rest, _) => eval_Op (z :: s) rest /* If the 1st operation is Push, it pushes z onto the stack s and continues evaluating the remaining ops*/

Evaluation of a list of VM operations

Fixpoint eval_Op (s : list Z) (ops : list Op) : option Z :=
match (ops, s) with
| ([], [n]) => Some n /* If the list is empty and there is only one element on the stack, it returns Some n, indicating successful evaluation with the result n */
| (Push z :: rest, _) => eval_Op (z :: s) rest /* If the 1st operation is Push, it pushes z onto the stack s and continues evaluating the remaining ops*/
| (Add :: rest, n2 :: n1 :: ns) => eval_Op (n1 + n2 :: ns)%Z rest
/* pops the top n1 and n2 from the stack, adds them together, and pushes the result back onto the stack*/

Evaluation of a list of VM operations

```
Fixpoint eval_Op (s : list Z) (ops : list Op) : option Z :=
match (ops, s) with
| ([], [n]) => Some n /* If the list is empty and there is only one element on the stack, it returns Some n, indicating successful evaluation with the result n */
| (Push z :: rest, _) => eval_Op (z :: s) rest /* If the 1st operation is Push, it pushes z onto the stack s and continues evaluating the remaining ops*/
| (Add :: rest, n2 :: n1 :: ns) => eval_Op (n1 + n2 :: ns)%Z rest
/* pops the top n1 and n2 from the stack, adds them together, and pushes the result back onto the stack*/
| (OpCount :: rest, _) => eval_Op (Z.of_nat (length rest) :: s) rest /* counts the remaining ops, converts it to a Z and pushes it onto the stack*/
```

Evaluation of a list of VM operations

```
Fixpoint eval_Op (s : list Z) (ops : list Op) : option Z :=
match (ops, s) with
| ([], [n]) => Some n /* If the list is empty and there is only one element on the
stack, it returns Some n, indicating successful evaluation with the result n */
| (Push z :: rest, _) => eval_Op (z :: s) rest /* If the 1st operation is Push, it
pushes z onto the stack s and continues evaluating the remaining ops*/
| (Add :: rest, n2 :: n1 :: ns) => eval_Op (n1 + n2 :: ns)%Z rest
/* pops the top n1 and n2 from the stack, adds them together, and pushes
the result back onto the stack*/
| (OpCount :: rest, _) => eval_Op (Z.of_nat (length rest) :: s) rest /* counts the
remaining ops, converts it to a Z and pushes it onto the stack*/
| _ => None end. /* if none matches, returns none */
```

Evaluation of a list of VM operations

Example

Starting from an empty stack and with the list of operations
[Push 1; Push 2; Add], eval_Op will return 3

The stack evolves as follows
[ ]
[1]
[2 1]
[3]

The evaluation succeeds, and returns ‘Some 3’

Compilation

We compile IMP arithmetic expressions represented by the Expr type into a list of operations (Op) for out stack-based VM

```
Fixpoint compile_Expr (e : Expr) : list Op :=
match e with
 | Num n => [Push n] /* If e is a number n, it creates a singleton list
containing Push n. This operation pushes n onto the stack.*/
```

Compilation

We compile IMP arithmetic expressions represented by the Expr type into a list of operations (Op) for out stack-based VM

```
Fixpoint compile_Expr (e : Expr) : list Op :=
match e with
 | Num n => [Push n] /* If e is a number n, it creates a singleton list
containing Push n. This operation pushes n onto the stack.*/
 | Plus e1 e2 => compile_Expr e1 ++ compile_Expr e2 ++ [Add]
/* If e is an addition, it recursively compiles e1 and e2 into lists of
operations, concatenates these lists, and appends an Add operation to the
end.*/
end.
```

Compilation

Example

compile_Expr (Plus (Num 1) (Num 2)) =>

Compilation

Example

compile_Expr (Plus (Num 1) (Num 2)) =>
compile_Expr (Num 1) ++ compile_Expr (Num 2) ++ [Add] =>

Compilation

Example

compile_Expr (Plus (Num 1) (Num 2)) =>
compile_Expr (Num 1) ++ compile_Expr (Num 2) ++ [Add] =>
[Push 1] ++ [Push 2] ++ [Add] =

Compilation

Example

compile_Expr (Plus (Num 1) (Num 2)) =>
compile_Expr (Num 1) ++ compile_Expr (Num 2) ++ [Add] =>
[Push 1] ++ [Push 2] ++ [Add] =
[Push 1; Push 2; Add]

Correctness: lemma

```
Lemma eval_step : forall a : Expr, forall s : list Z, forall xs : list Op,
        eval_Op s (compile_Expr a ++ xs) = eval_Op (eval_Expr a :: s) xs.
```

Correctness: lemma

```
Lemma eval_step : forall a : Expr, forall s : list Z, forall xs : list Op,
           eval_Op s (compile_Expr a ++ xs) = eval_Op (eval_Expr a :: s) xs.
```

Evaluating a list of operations produced by compiling an expression a and concatenating it with another list of operations xs

is equivalent to

evaluating the original expression a and pushing its result onto the stack s, and then evaluating the list of operations xs

It confirms that the compilation process preserves the semantics of the original expression

Compiler correctness

```
Theorem compiler_correctness :
forall a : Expr,
eval_Op [] (compile_Expr a) = Some (eval_Expr a).
```

If you

- compile an expression a into a list of operations and then
- evaluate these operations starting with an empty stack,

you obtain the same result as directly evaluating the original expression a

Equivalences

Equivalences follow from evaluation

- Definition equiv_Expr (e1 e2 : Expr) : Prop := eval_Expr e1 = eval_Expr e2.

e1 and e2 are equivalent if their evaluations (eval_Expr e1 and eval_Expr e2, respectively) produce the same result, i.e., $e1 \simeq e2$

- Definition equiv_Op (p1 p2 : list Op) : Prop := eval_Op [] p1 = eval_Op [] p2.

p1 and p2 are equivalent if their evaluations, when starting with an empty stack ([]), produce the same result, i.e., $p1 \simeq p2$

Source contexts

```
Inductive ExprCtxt : Set :=
| Hole : ExprCtxt /*hole in the expression where a subexpression can be inserted */
| Plus1 : ExprCtxt -> Expr -> ExprCtxt  /* context where the right operand of an addition operation is missing*/
| Plus2 : Expr -> ExprCtxt -> ExprCtxt /*context where the left operand of an addition operation is missing */
```

Expression contexts are used to represent partially completed expressions where certain subexpressions are left as "holes" to be filled in later

Target contexts

For our stack machine, partial programs are much easier

A program is just a list of Op, therefore any program can be extended by adding new Ops on either end (or inserting in the middle)

Filling the holes

```
Fixpoint link_Expr (c : ExprCtxt) (e : Expr) : Expr :=
match c with
| Hole => e /*If the context is a Hole, it returns the expression e. This effectively fills the hole with the expression e*/
| Plus1 c' e' => Plus (link_Expr c' e) e’ /* e needs to be inserted as the right operand of an addition operation, with c' representing the left operand. */
| Plus2 e' c' => Plus e' (link_Expr c' e) /* e needs to be inserted as the left operand of an addition operation, with c' representing the right operand. */
end.
```

The function link_Expr constructs complete expressions by linking an expression e into an expression context c

Context equivalences

At source level

Definition ctxtequiv_Expr (e1 e2 : Expr) : Prop :=

forall c : ExprCtxt, eval_Expr (link_Expr c e1) = eval_Expr (link_Expr c e2).

$$e1 \simeq_{ctx} e2$$

At target level

Definition ctxtequiv_Op (p1 p2 : list Op) : Prop := forall c1 c2 : list Op,
eval_Op [] (c1 ++ p1 ++ c2) = eval_Op [] (c1 ++ p2 ++ c2).

$$p1 \simeq_{ctx} p2$$

Reflection and preservations

We have now all the ingredients to prove

- Reflection and
- Preservation

Reflection

$$\forall\, e1, e2.\; e1 \simeq_{ctx} e2 \;\Leftarrow\; [\![e1]\!] \simeq_{\boldsymbol{ctx}} [\![e2]\!]$$

Lemma equivalence_reflection :
forall e1 e2 : Expr, forall p1 p2 : list Op,
forall comp1 : compile_Expr e1 = p1,
forall comp2 : compile_Expr e2 = p2,
forall eqtarget : ctxtequiv_Op p1 p2, ctxtequiv_Expr e1 e2.

If two compiled expressions are contextually equivalent at the target level, then the original expressions will also be contextually equivalent at the source level.

Preservation

$\forall\ e1, e2.\ e1 \simeq_{ctx} e2 \Rightarrow [\![e1]\!] \simeq_{ctx} [\![e2]\!]$

Lemma equivalence_preservation :
forall e1 e2 : Expr, forall p1 p2 : list Op,
forall comp1 : compile_Expr e1 = p1,
forall comp2 : compile_Expr e2 = p2,
forall eqsource : ctxtequiv_Expr e1 e2, ctxtequiv_Op p1 p2.

If two expressions are contextually equivalent at the source level, then their compiled forms will also be contextually equivalent at the target level.
Easy to write, but…

Preservation

$\forall\ e1, e2.\ e1 \simeq_{ctx} e2 \Rightarrow [\![e1]\!] \simeq_{ctx} [\![e2]\!]$

Lemma equivalence_preservation :
forall e1 e2 : Expr, forall p1 p2 : list Op,
forall comp1 : compile_Expr e1 = p1,
forall comp2 : compile_Expr e2 = p2,
forall eqsource : ctxtequiv_Expr e1 e2, ctxtequiv_Op p1 p2.

If two expressions are contextually equivalent at the source level, then their compiled forms will also be contextually equivalent at the target level.
Easy to write, but it is not provable, because it is not true

Counterexample

At source level

$$\text{Plus (Num 1) (Num 2)} \simeq_{ctx} \text{(Num 3)}$$

because they have the same evaluation, no matter which is the context

src_equiv : ctxtequiv_Expr (Plus (Num 1) (Num 2)) (Num 3).

Counterexample

At target level, the compiled counterparts are not contextually equivalent, because there exists at least a context that leads to different evaluation results

```
eval_Op [] (OpCount :: compile_Expr (Plus (Num 1) (Num 1)) ++ [Add]) <>
eval_Op [] (OpCount :: compile_Expr (Num 2) ++ [Add]).
```

Why?

Counterexample (cont.)

eval_Op [] (OpCount :: compile_Expr (Plus (Num 1) (Num 1)) ++ [Add]) <>
eval_Op [] (OpCount :: compile_Expr (Num 2) ++ [Add]).

If we put OpCount followed by Add instruction afterwards, the result will be:

- the value and
- the number of instructions it took to compute it, that is different
  - for OpCount :: compile_Expr (Plus (Num 1) (Num 1)) ++ [Add] is [6].
  - for OpCount :: compile_Expr (Num 2) ++ [Add] is [4].

- Note that OpCount context at target level does not have a corresponding context (obtainable by back-translation) at source level

Counterexample

This is an example of common situation

There could be cases where e1 and e2 are contextually equivalent as expressions but their compiled forms p1 and p2 might not be equivalent as operations due to

- the structure of the compilation process or
- how the operations interact with the stack machine

Any solution?

To ensure the correctness of this theorem, we would need:

- Strong Equivalence Guarantees:
  Prove that compile_Expr preserves not just the final results but the entire behavior of expressions in all contexts
- Correct Handling of Stack State:
  Ensure that operations resulting from compile_Expr maintain equivalent stack states across all possible contexts

Any solution?

We need to somehow protect the compiled code from having the equivalences disrupted

Here, for instance, we might put some flag on instructions that meant that they should not be counted, and OpCount would just not return anything if it saw any of those (or would count them as 0).

Alternately, we might give our target language a type system that is able to rule out linking with code that uses the OpCount instruction, or perhaps restricts how it can be used

To keep it simple

For the sake of simplicity, we opt here for an extreme solution:

- rather than a list, we allow one Op before and one Op after our compiled program, neither of which can be OpCount
- the resulting program to be well-formed (i.e., no errors, only one number on stack at end), so
  - either there should be nothing before and after,
  - or there is a Push n before and either Add or Sub after
  - no other combination of Op before or after will fulfill our requirement

To keep it simple

We highly restrict what we can link with: rather than a list, we allow one Op before and one Op after our compiled program, neither of which OpCount

```
Inductive OpCtxt : Set :=
| PushAdd : Z -> OpCtxt
| Empty : OpCtxt.
```

```
Definition link_Op (c : OpCtxt) (p : list Op) : list Op :=
match c with
| PushAdd n => Push n :: p ++ [Add]
| Empty => p end.
```

To keep it simple

We need to redefine contextual equivalence for our target language, only permitting these new contexts

```
Definition ctxtequiv_Op (p1 p2 : list Op) : Prop :=
forall c : OpCtxt, eval_Op [] (link_Op c p1) = eval_Op [] (link_Op c p2).
```

Now our compiler, when linked against these restricted contexts, can be proved fully abstract

Back-translation

To complete the proof, we rely upon a back-translation from target contexts to source context, where target contexts are restricted as described above: from OpCtxt to ExprCtxt

```
Definition backtranslate (c : OpCtxt) : ExprCtxt :=
match c with
| PushAdd n => Plus2 (Num n) Hole
/* the context involving pushing n and the add is back-translated into a
context where the expression (Num n) + Hole is formed */
```

Back-translation

To complete the proof, we rely upon a back-translation from target contexts to source context, where target contexts are restricted as described above: from OpCtxt to ExprCtxt

```
Definition backtranslate (c : OpCtxt) : ExprCtxt :=
match c with
| PushAdd n => Plus2 (Num n) Hole
/* the context involving pushing n and the add is back-translated into a context where the expression (Num n) + Hole is formed */
| Empty => Hole end. /*an empty context is back-translated in a context a context with no additional expressions or operations */
```

Back-translation lemma

Lemma back_translation_equiv : forall c : OpCtxt, forall p : list Op, forall e : Expr, forall c' : ExprCtxt, compile_Expr e = p -> backtranslate c = c' -> eval_Op [] (link_Op c p) = Some (eval_Expr (link_Expr c' e)).

The lemma aims to prove that evaluating the operational context applied to the compiled expression yields the same result as evaluating the expression context applied to the original expression

Equivalence preservation

Now, we can prove equivalence preservation directly.

Lemma equivalence_preservation :
forall e1 e2 : Expr, forall p1 p2 : list Op,
forall comp1 : compile_Expr e1 = p1,
forall comp2 : compile_Expr e2 = p2,
forall eqsource : ctxtequiv_Expr e1 e2, ctxtequiv_Op p1 p2.

The theorem is used to prove that
$\forall\ e1, e2.\ e1 \simeq_{ctx} e2 \Rightarrow [\![e1]\!] \simeq_{ctx} [\![e2]\!]$

Proving equivalence

This is obviously a very tiny language and a very restrictive linker that only allowed very restrictive contexts, but the general shape of the proof is the same as that used in more realistic languages

Secure compilation

Must ensure that any P we link with behaves like some source context

- Add target features to the source language Bad!
- Dynamics checks: catch badly behaved code in the act Expensive
- Static checks: rule our badly behaved code in the first place Verification

Secure compilation: recap

- Preserving security by preserving equivalence

- Different compilation targets and threat models
  - is the target language typed or untyped?
  - what observations can the attacker make?

- Different ways of enforcing secure compilation
  - static checking
  - dynamic checking (e.g., runtime monitoring, hardware enforcement)

- Proof techniques: "back-translating" target attackers to source

Bibliography

- Marco Patrignani, Amal Ahmed, Dave Clarke. *Formal Approaches to Secure Compilation. A Survey of Fully Abstract Compilation and Related Work*. ACM Computing surveys, 2019.

- https://dbp.io/essays/2018-04-19-how-to-prove-a-compiler-fully-abstract.html

End