Language Based Security (LBT)

Secure compilation

Chiara Bodei, **Gian-Luigi Ferrari**

Lecture May, 15 2024

Outline

Compiler Correctness

Secure compilation

Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation

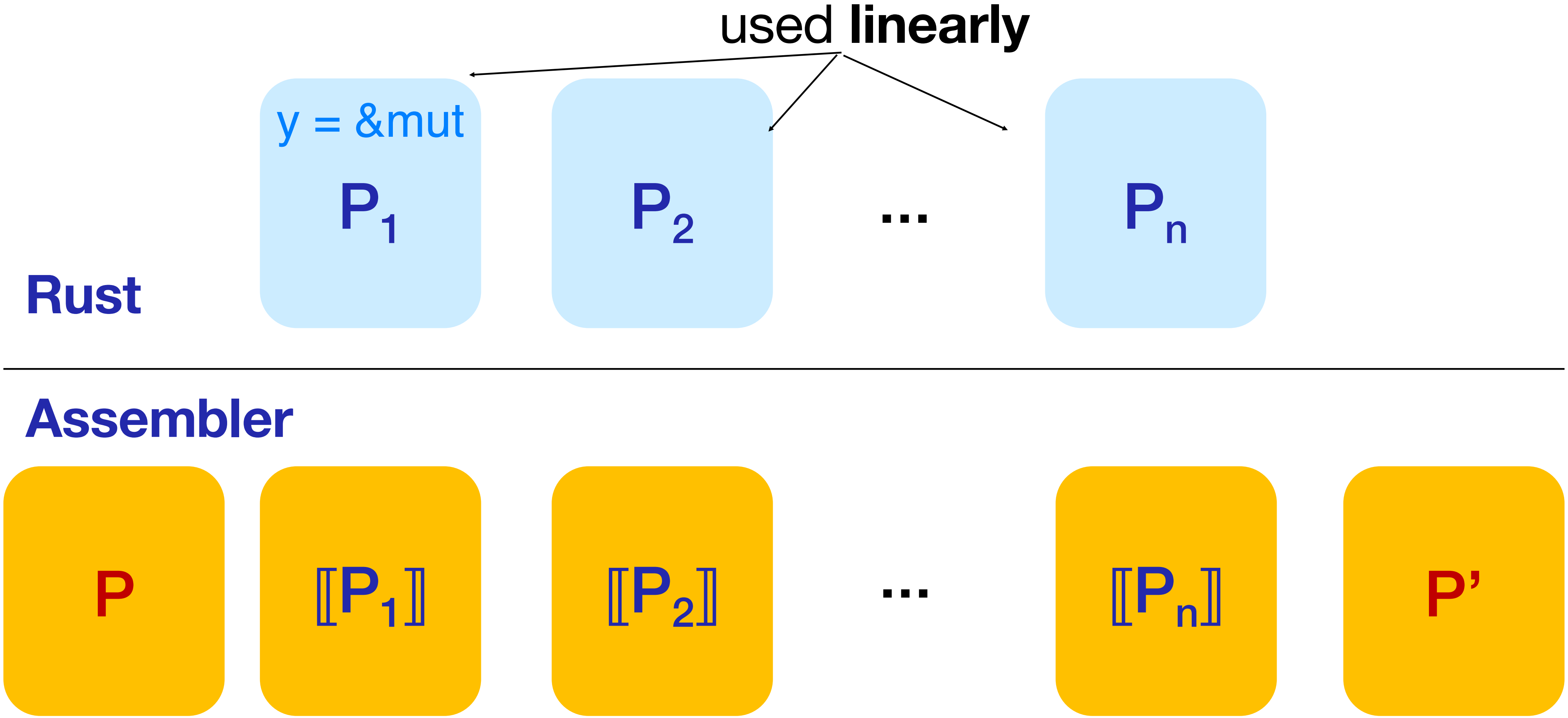
Secure compilation

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

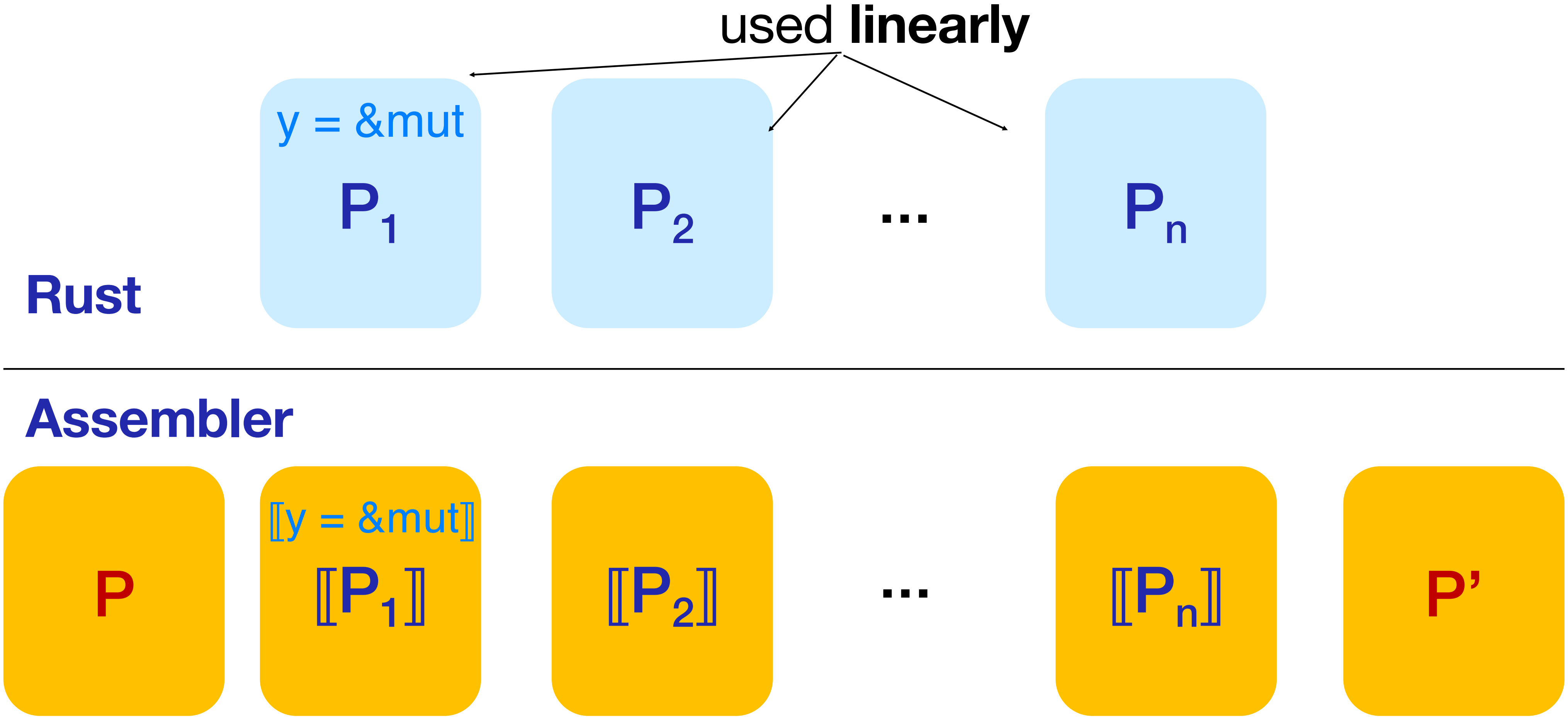
Example

Mutability

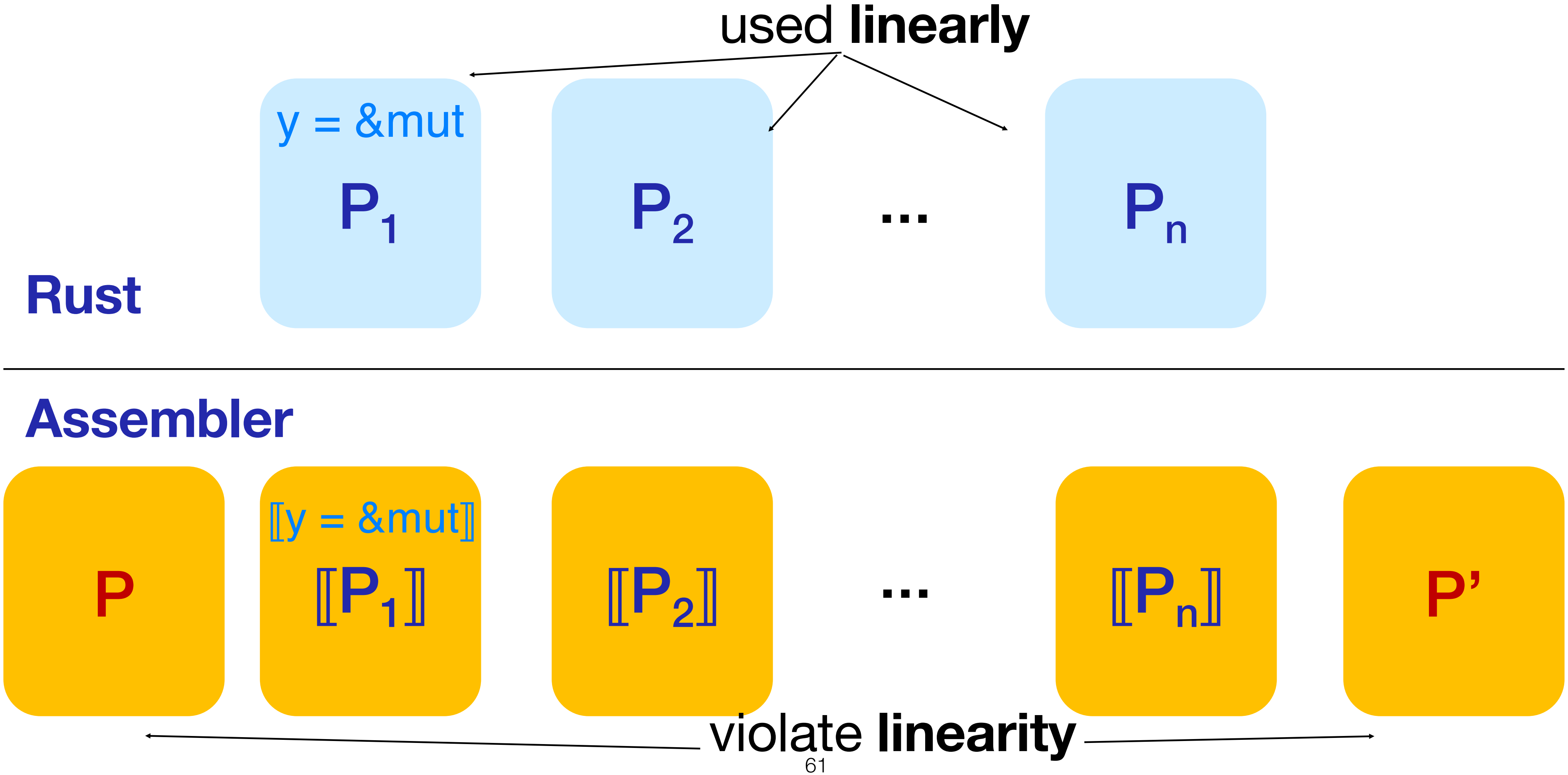
Linearity



Linearity



Possible violations of linearity



Possible violations of linearity

Preserve the security properties of

Rust

$y = \&mut$

P_1

P_2

...

P_n

Assembler

P

$\llbracket y = \&mut \rrbracket$

$\llbracket P_1 \rrbracket$

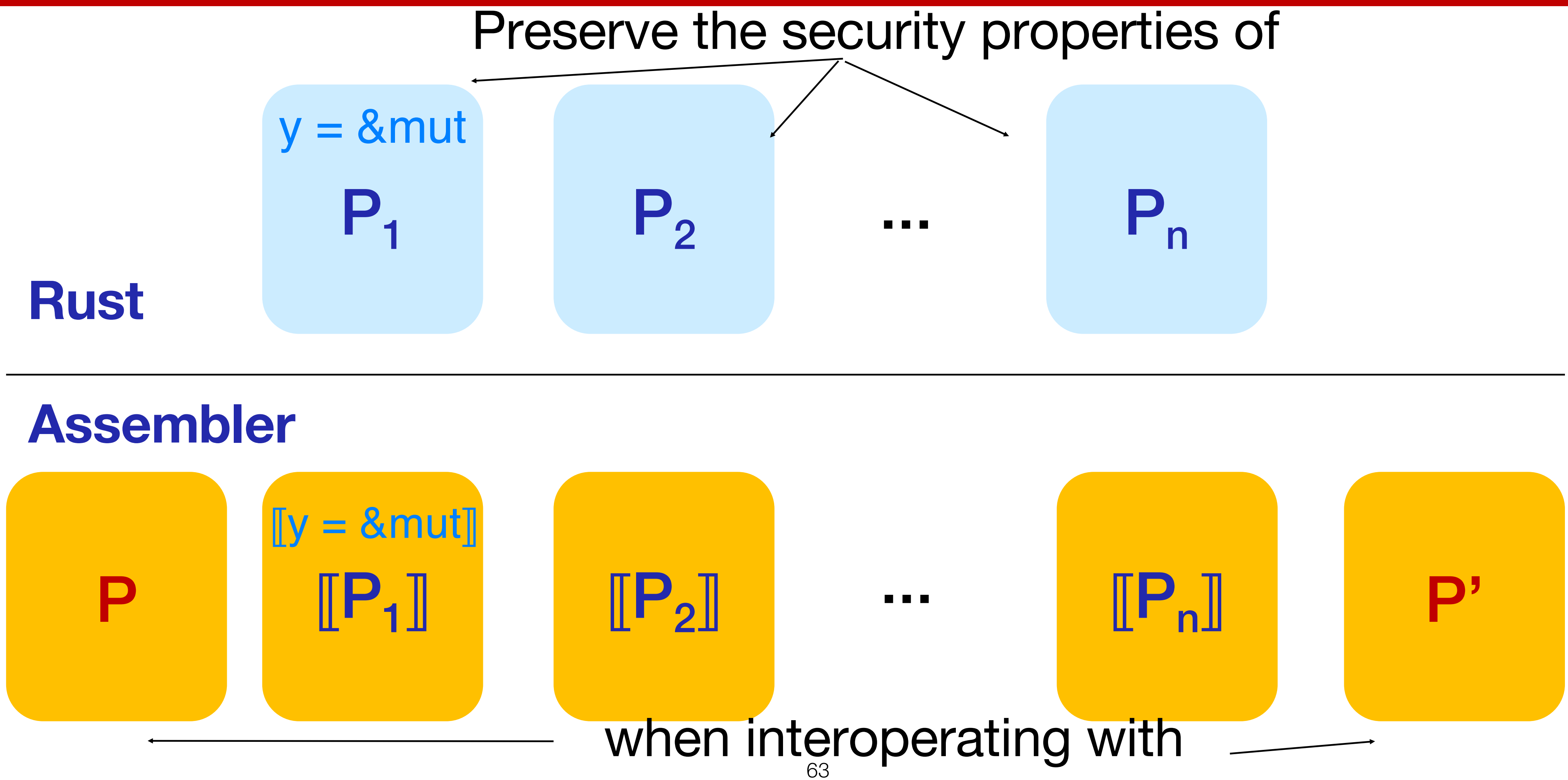
$\llbracket P_2 \rrbracket$

...

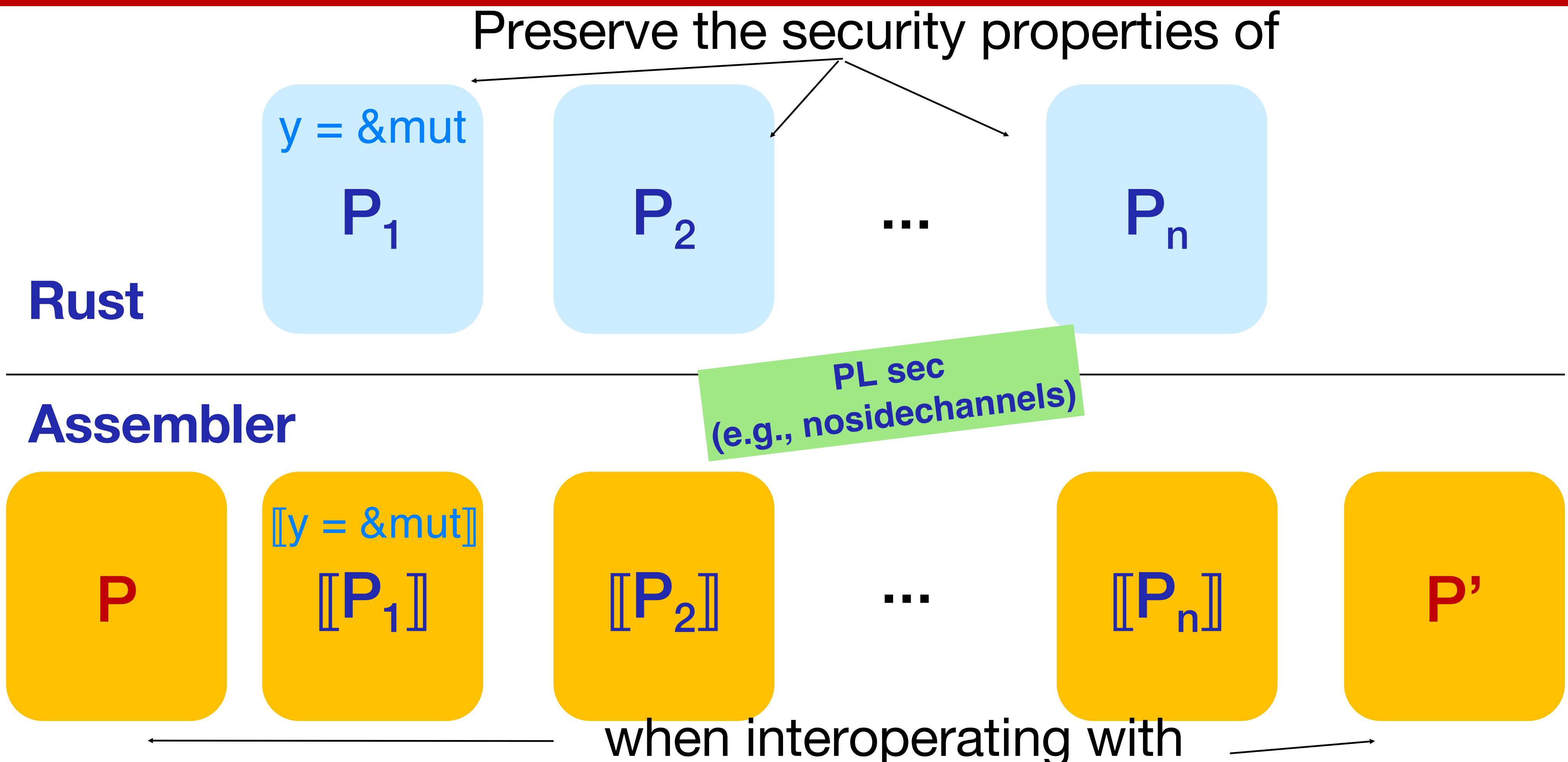
$\llbracket P_n \rrbracket$

P’

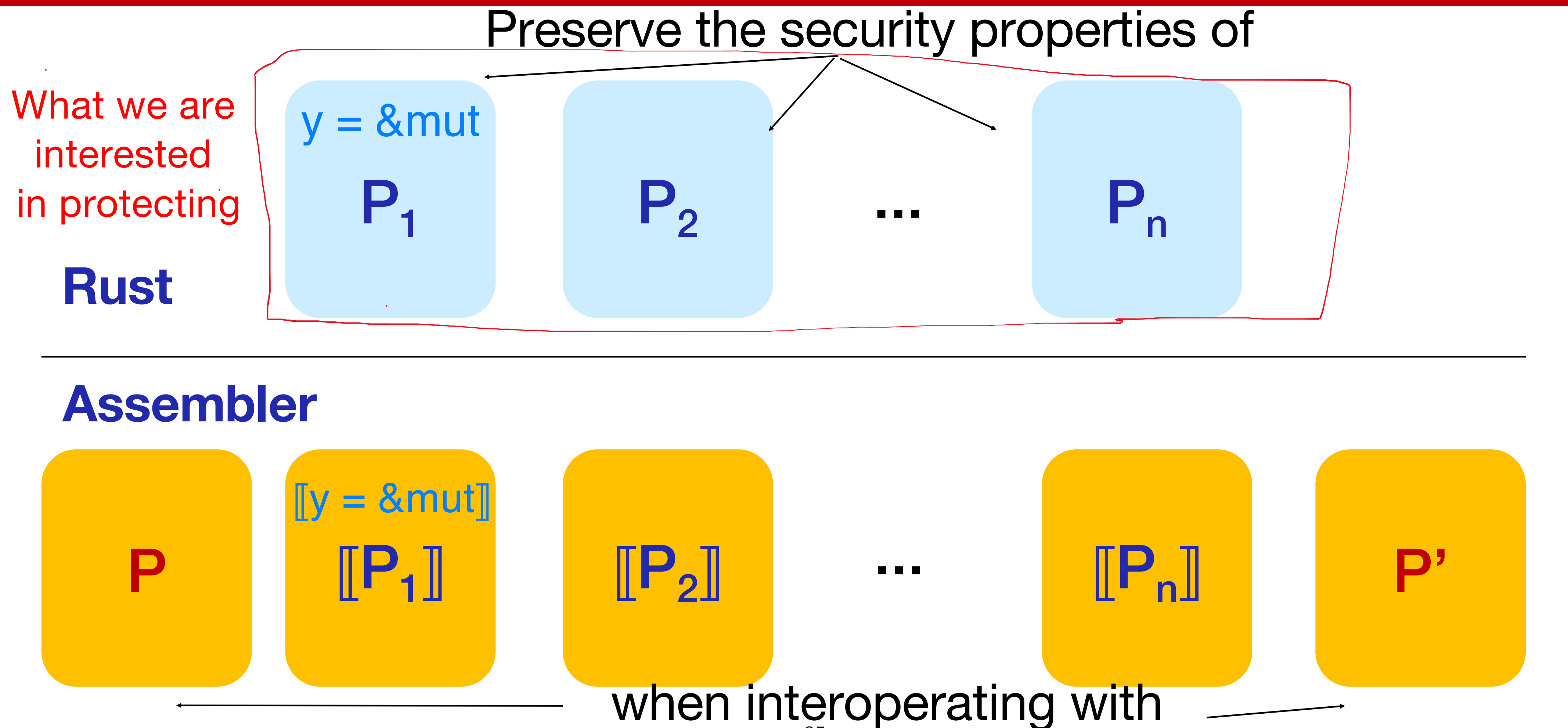
Preservation of linearity



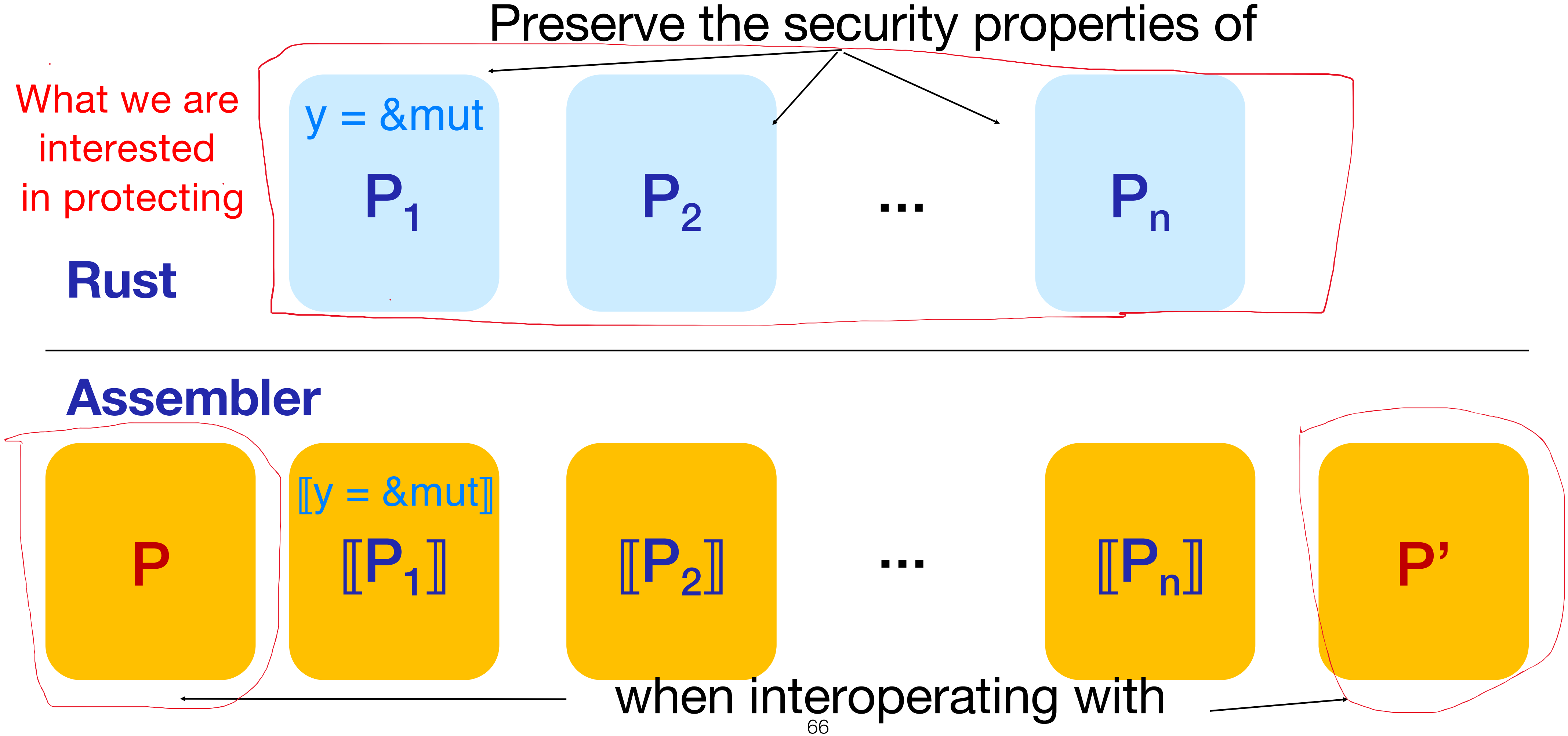
Preservation of linearity



Threat model hint

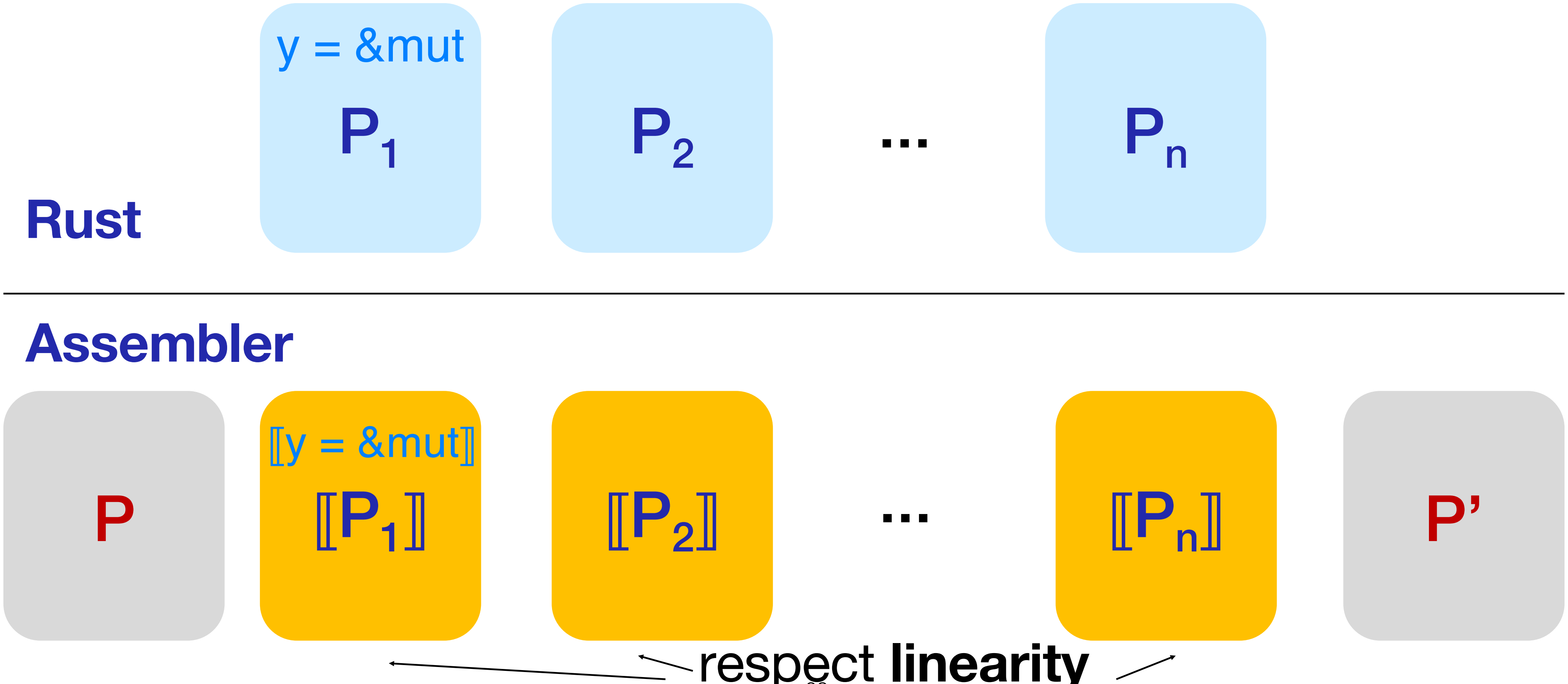


Threat model hint



Correct compilation

Correct compilation



Secure compilation

Enable source-level security reasoning

Secure compilation

- What does it mean for a compiler $[\![\cdot]\!]_T^S$ 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

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

```java
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

```
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;
  }
}
```

Listing 1. Example of Java source code.

No access to balance from outside
Enforced by the language

```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;
}
```

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

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

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
```

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

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

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

- in particular, we are interested in contextual or observational equivalence

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( )
  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;
}
```

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

  return 0;


}
```

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

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 external observer interacts with them that observer cannot distinguish the programs

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

$\downarrow$ refers to 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,…
- 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

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

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 = 1;

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→
```

```
// 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

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

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)

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?

Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation

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

$\forall\ P_1, P_2$

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

Correctness is needed

We need the compiler also to be correct

$\forall\ P_1, P_2$

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

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

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

Proving full abstraction

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

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

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

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

- Both parts are difficult to prove
- Preservation is particularly tricky because of $\simeq$ and $\forall$ contexts means that
- It amounts to prove that no new vulnerabilities are introduced at the target-level

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:

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

Backtranslations

This proof structure is called backtranslation from T to S

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

Backtranslation goes in the opposite direction and is responsible to gp from the target level context breaking the equivalence to the corresponding of target-level source-level context

«.» : T $\rightarrow$ S

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

Security proof

compiler security direction $\Downarrow$

$$\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$$

$$\begin{tikzcd}
\langle\!\langle \mathfrak{C} \rangle\!\rangle\left[\mathbf{t_1}\right] \Downarrow \arrow[r, Rightarrow, "(2)"'] & \langle\!\langle \mathfrak{C} \rangle\!\rangle\left[\mathbf{t_2}\right] \Downarrow \arrow[d, Rightarrow, "(3)"'] \\
\mathfrak{C}\left[[\![\mathbf{t_1}]\!]\right] \Downarrow \arrow[u, Rightarrow, "(1)"'] \arrow[r, Rightarrow, "?"] & \mathfrak{C}\left[[\![\mathbf{t_2}]\!]\right] \Downarrow
\end{tikzcd}$$

$\langle\!\langle \mathfrak{C} \rangle\!\rangle \approx \mathfrak{C}$
$\mathbf{t_1} \approx [\![\mathbf{t_1}]\!]$

$\langle\!\langle \mathfrak{C} \rangle\!\rangle \approx \mathfrak{C}$
$\mathbf{t_2} \approx [\![\mathbf{t_2}]\!]$

$$[\![\mathbf{t_1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{t_2}]\!]$$

Proving full abstraction

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

Proving full abstraction

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

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.

End