Language Based Security (LBT)

Secure compilation

Chiara Bodei, Gian-Luigi Ferrari

Lecture April, 28 2025

Outline

Compiler Correctness

Secure compilation

General overview

Ken Thompson
What code can we trust?
…
Can we trust the compiler?
Solution: inspect the compiler

How can we trust a compiler is correct?

- Testing can help discover bugs, but it is not enough, especially for safety-critical, high-assurance software
- Disable any optimization: too restrictive
- One possibility is to formally verify the compiler itself *?
  - This is the aim of CompCert, for instance, programmed and verified using the Coq proof assistant

*Xavier Leroy. *Proving the correctness of a compiler*. EUTypes Summer School 2019

Compiler correctness

- By applying program proof to the compiler itself, we can obtain mathematically strong guarantees that the generated executable code is faithful to the semantics of the source program

Semantic preservation
For all source code s, if the compiler generates machine target code t from source s without reporting any compilation error, then t **behaves like** s

Compiler correctness

- Is this compilation correct?

```
for(i=0; i < 10; i++)
    printf("%d\n", i);
```

```
printf("42\n");
```

Compiler correctness

- Is this compilation correct? **No**

```
for(i=0; i < 10; i++)
    printf("%d\n", i);
```

```
printf("42\n");
```

Compiler correctness

- Is this compilation correct? **No**

```
for(i=0; i < 10; i++)
    printf("%d\n", i);
```

```
printf("42\n");
```

Compiler correctness

- Is this compilation correct?

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

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

Compiler correctness

- Is this compilation correct? **Not really**, as we will see

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

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

Correctness, formally

Formally, a compiler is a function $[\![\cdot]\!]_T^S$ from S to T

- It takes programs **s** written in a source language S and
- ouputs programs **t** written in a target language T

What does it mean that a compiler is **correct**? It means that the compiled code behaves as the source code

Correctness

Behaviours can be different: a program may:

- Terminate and return a result value
- Diverge (= indefinitely run, e.g., infinite loop)
- Go wrong (= crash, e.g.. division by zero, or invalid memory access)

x = 1;

While true do skip;

Ibranch(-1)

x = 1/0;

Ihalt

Observable behaviors

What should be preserved?

Answer: observable behaviors $\mathcal{B}(s)$

- Hence, observable behaviors are normal termination or abnormal termination and divergence The three ones we have said before of course

- Inputs and outputs can be observed as well

Idea is you should decide what is observable and define things from there

Observable behaviors

We define the behaviour of a source code in terms of the set of observable actions, e.g., I/O actions, memory operations, …

For example, in terms of traces, composed by strings of observable actions

For an imperative language with I/O: add a trace of input-output operations performed during execution.

```
x := 1;  x := 2;
```
(trace: ϵ)

$\approx$

```
x := 2;
```
(trace: ϵ)

Those two traces are equivalent

```
print(1); print(2);
```
(trace: out(1).out(2))

$\not\approx$

```
print(2);
```
(trace: out(2))

Observable actions are not the same

Preservation

A correct compilation is semantics preserving if the generated code behaves as prescribed by the semantics of the source

How to characterize preservation?

Answer: we need to compare the behaviors of two programs, by using simulations and program equivalences

Two programs are **equivalent** when they behave the same, even if they are different (same semantics and possibly different syntax)

Digression on equivalences

Suppose to have two tea/coffee machines:

- **M_1** allows you to:
  - Put a coin
  - Choose to press the tea or the coffee button
  - After providing you with the required beverage, the machine is again waiting for requests
- **M_2** allows you to:
  - Put a coin
  - When a tea or coffee button is pressed, non-deterministically delivers tea or coffee
  - …
- Do **M_1** and **M_2** behave the same?

Digression on equivalences

Suppose to have two tea/coffee machines:

- **M_1** allows you to:
  - Put a coin
  - Choose to press the tea or the coffee button
  - After providing you with the required beverage, the machine is again waiting for requests
- **M_2** allows you to:
  - Put a coin
  - When a tea or coffee button is pressed, non-deterministically delivers tea or coffee
  - …
- Do **M_1** and **M_2** behave the same? It depends on what can be observed

If you see the beverage in the cup, you can tell the difference

Digression on equivalences

What does it mean that two machines “are the same” for an observer?

Intuitively, if you do something with one machine, you must be able to do the same with the other, and on the two states which the machines evolve to, the same is again true

Bisimulation (observation equivalence)

How to characterize preservation?

Answer: simulations and program equivalences

Definition (Bisimulation) We talk about bisimulation when

The source program **s** and the compiled program **t** have exactly the same observable behaviors (they are equivalent).

Every possible behavior of **s** is a possible behavior of **t**

Every possible behavior of **t** is a possible behavior of **s**

$$\mathcal{B}([[s]]) = \mathcal{B}(s)$$

↑ Compiled version of [s], and $\mathcal{B}$ is the behavior

Complex. Sometimes we only do one way proof

$\Rightarrow$ or $\Leftarrow$

Why is this important for secure compilation?

Because if you can show that the source program and the compiled program are related by a bisimulation, it means:

- **Every action** that the source can perform has a corresponding action the compiled program can perform,
- **And vice versa**,
- **And after each action**, the relation between their new states still holds.

Thus, no attacker interacting with the compiled code could ever distinguish it from interacting with the source — *security properties are preserved* across compilation.

Back to dead store elimination

Optimizations automatically transform the programmer-supplied code into equivalent code that should be more efficient

DSE is a form of **compilation** from initial programs to programs where dead store have been removed

This is a form of compilation

Back to dead store elimination

Semantic preservation for dead store elimination relates executions of c and c in absence of dead stores : This is what we'd like to prove

Assume initial states agree on live variables “before” c: (they have the same live variables)
Then, the two executions (c and dse c) terminate on final states that agree on live variables “after” c

DSE correctness can be formally proved

Back to dead store elimination

DSE correctness can be formally proved

For practical reason you do this

Suppose that dead store x := e is removed by replacing it with x := ⊥ and note that the CFG is unaltered in the transformation

The following relation is indeed a **bisimulation**

is built in a way that:

The relation connects states (m, s) of the source and (n, t) of the target if

1. m = n (i.e., same CFG nodes);
2. s(y) = t(y) for all y other than x; and
3. s(x) = t(x) if t(x) ≠ ⊥

CFG nodes

function mapping each variable to a value from its type

Corresponding traces have identical control-flow and, at corresponding points, have identical values for all variables other than x, and identical values for x if the last assignment to x is not removed

This captures idea of bisimulation and the idea of semantics preservation

Bibliography

- Xavier Leroy. *Proving the correctness of a compiler*. EUTypes Summer School 2019
- Sandrine Blazy. *Verified compilation An introduction to CompCert*. OPLSS, Eugene, 2023-07-05

Outline

Compiler Correctness

Secure compilation

Outline

Motivating example
Overview
Security via program equivalences

Secure compilation

- What does it mean for a compiler to be secure?
- It is a long-standing question
- Many anwers have been given, we focus on the formal ones

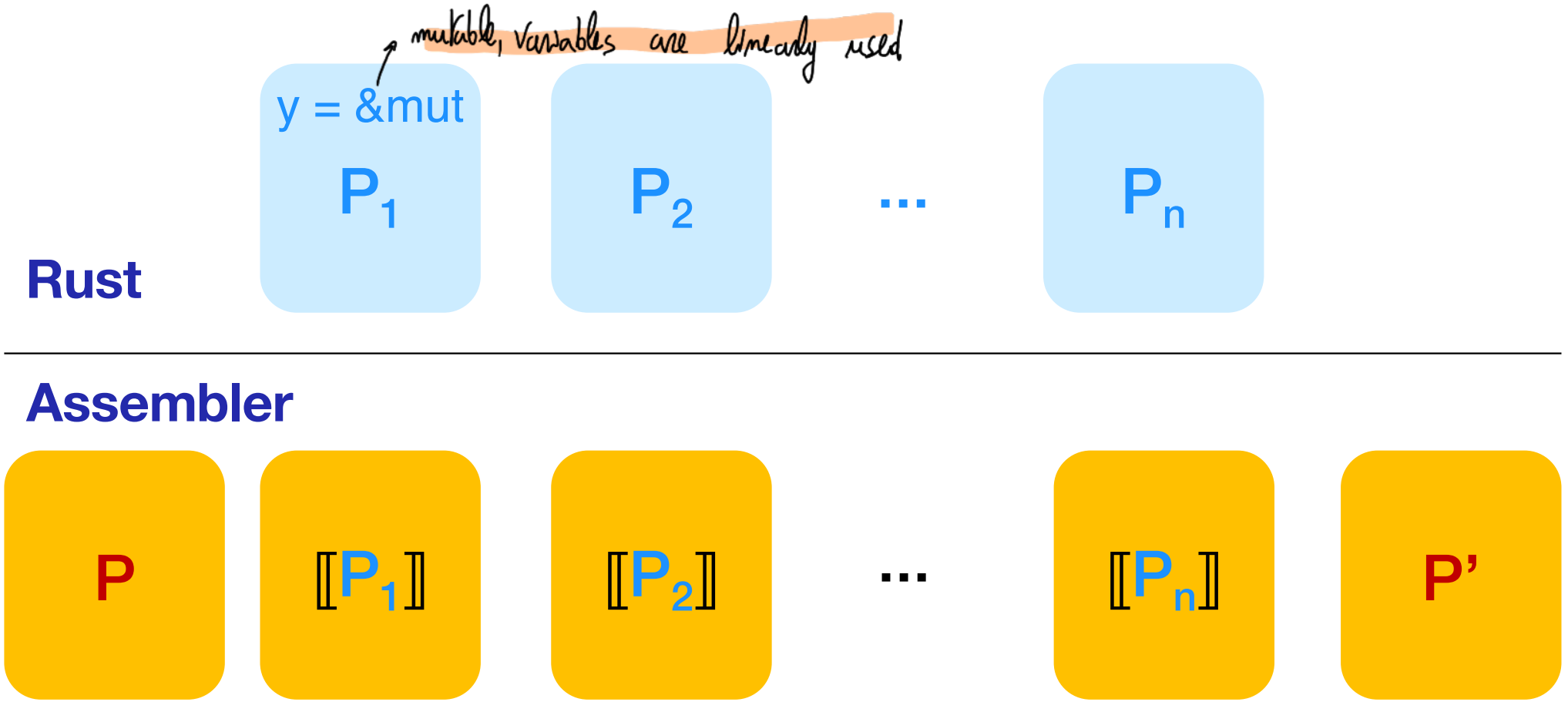
Conceptually:

“take what was secure in the source and preserve it in the target”

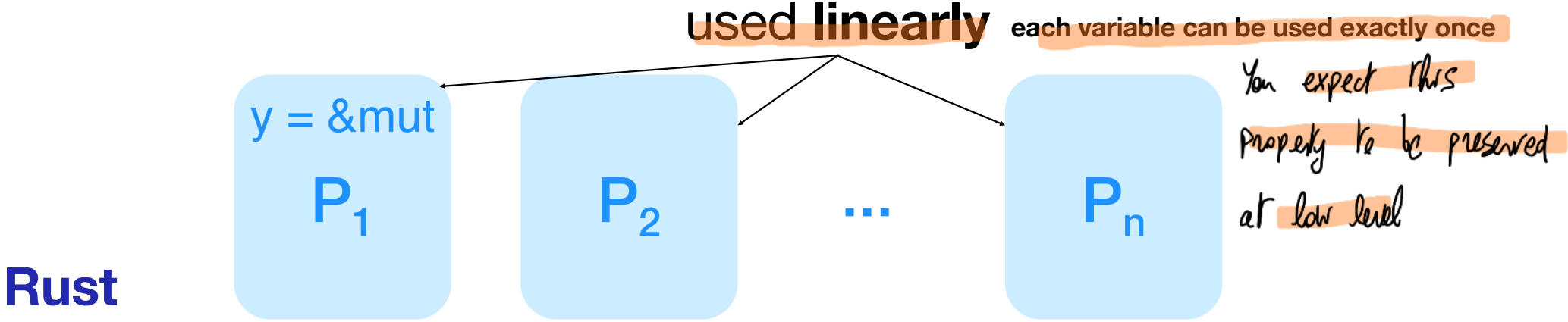
Example

Suppose you have a rust list of programs and we want to compile them into corresponding assembly code

Example


mutable variables are linearly used
y = &mut
P1
P2
...
Pn
Rust
Assembler
P
[[P1]]
[[P2]]
...
[[Pn]]
P'

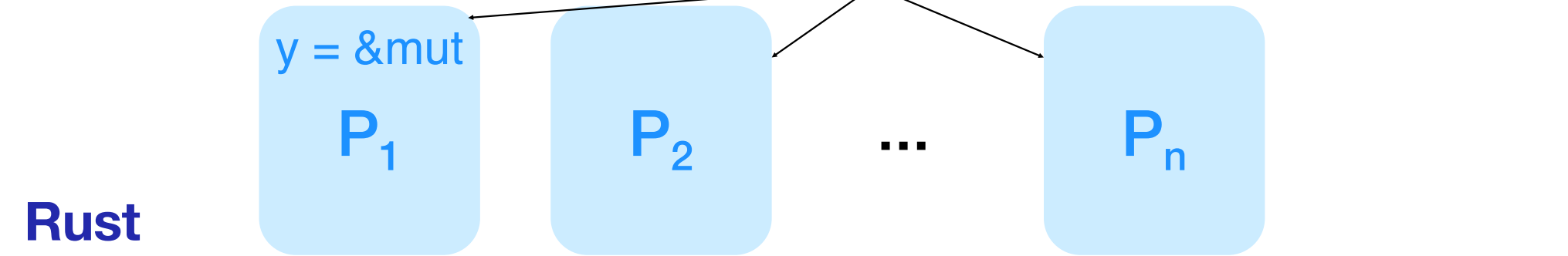
Example

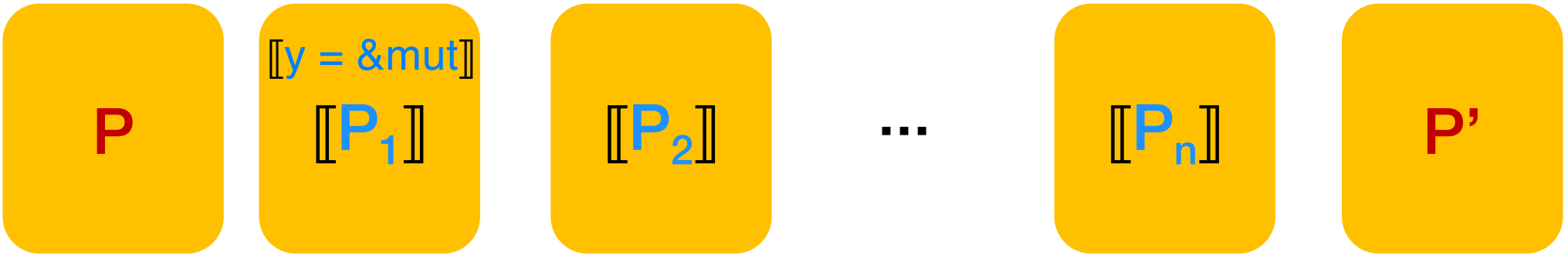

used linearly
each variable can be used exactly once
You expect this property to be preserved at low level
y = &mut
P1
P2
...
Pn
Rust


Assembler
P
[[y = &mut]]
[[P1]]
[[P2]]
...
[[Pn]]
P'

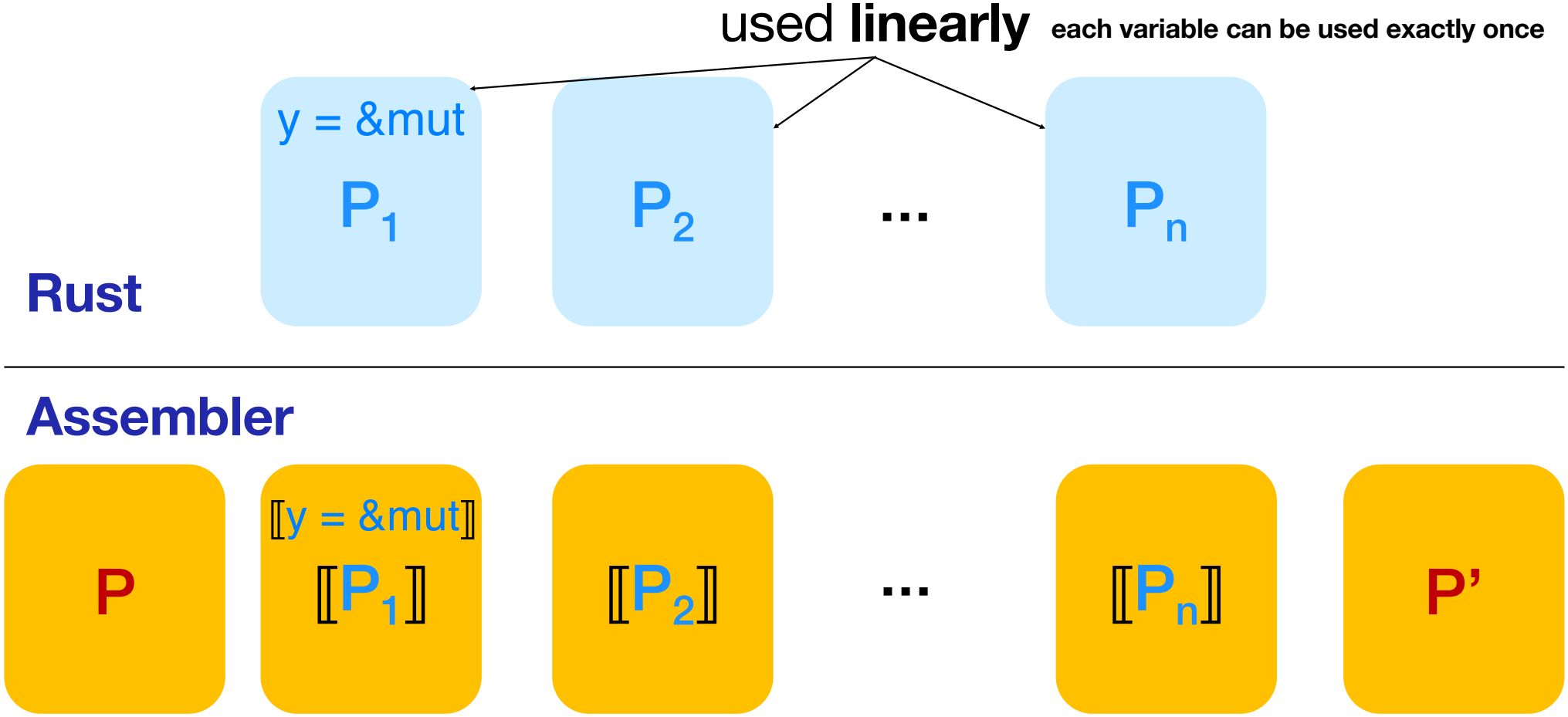
Linearity

used **linearly** each variable can be used exactly once



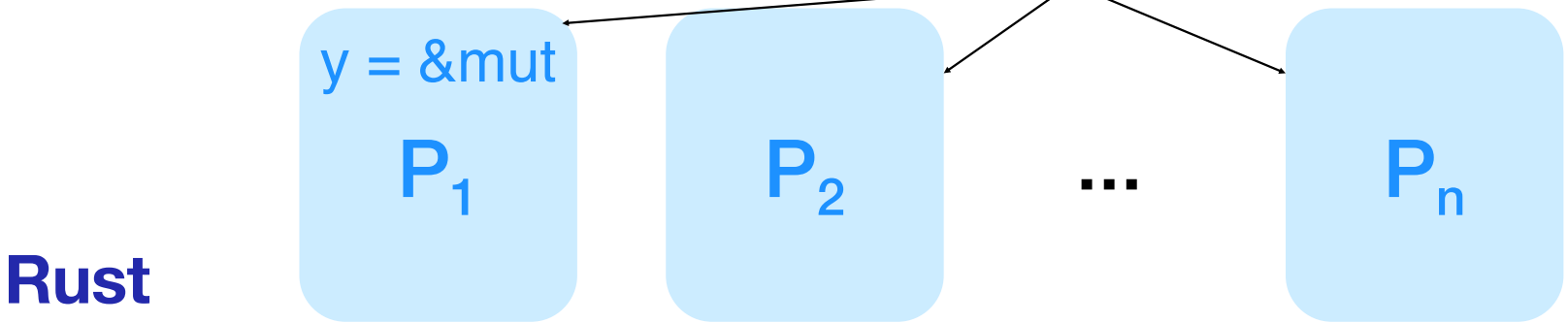


Linearity


used **linearly** each variable can be used exactly once
y = &mut
P_1
P_2
...
P_n
Rust
Assembler
P
⟦y = &mut⟧
⟦P_1⟧
⟦P_2⟧
...
⟦P_n⟧
P'

Possible violations of linearity

used **linearly** each variable can be used exactly once



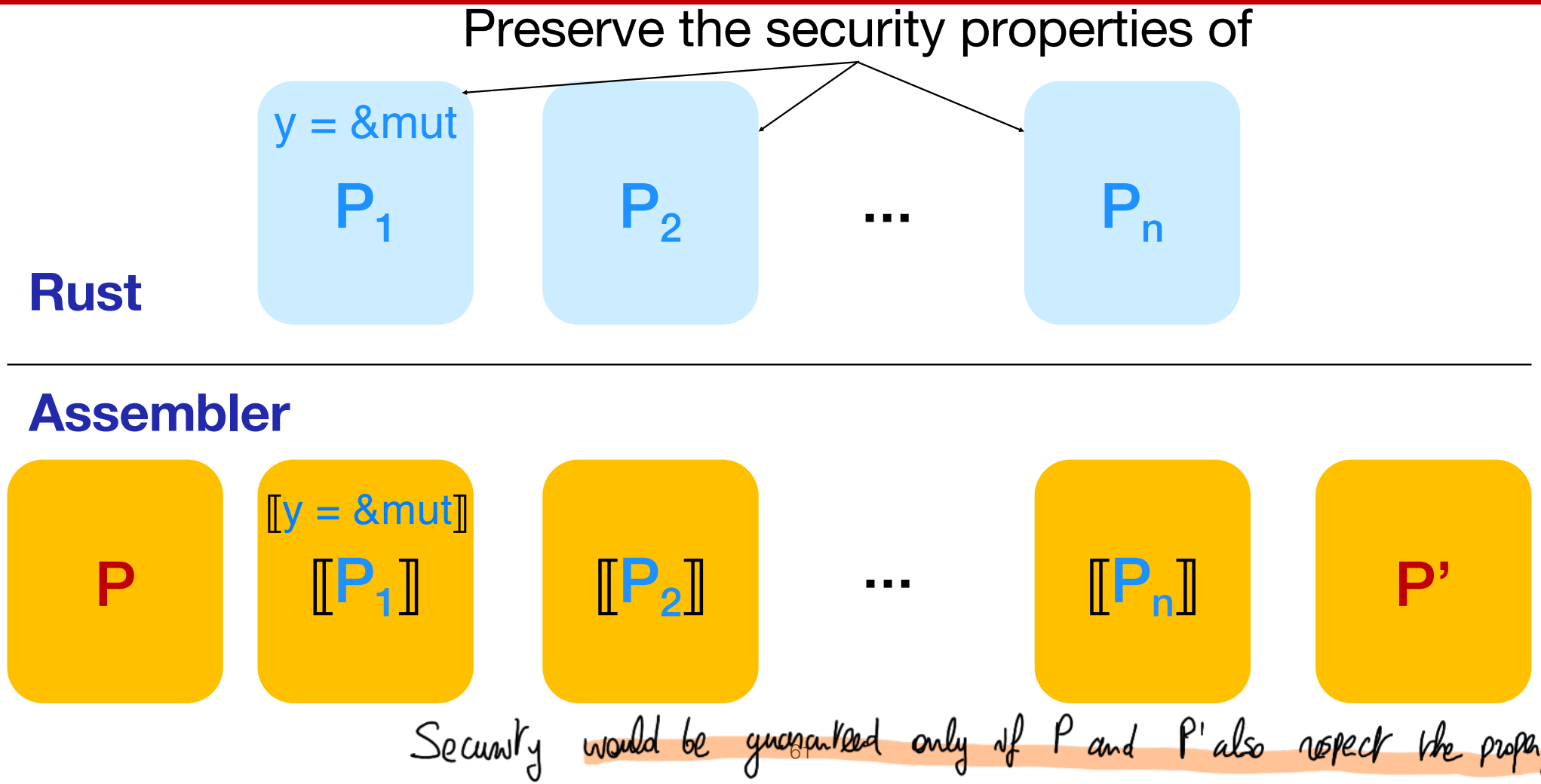
Rust

Assembler

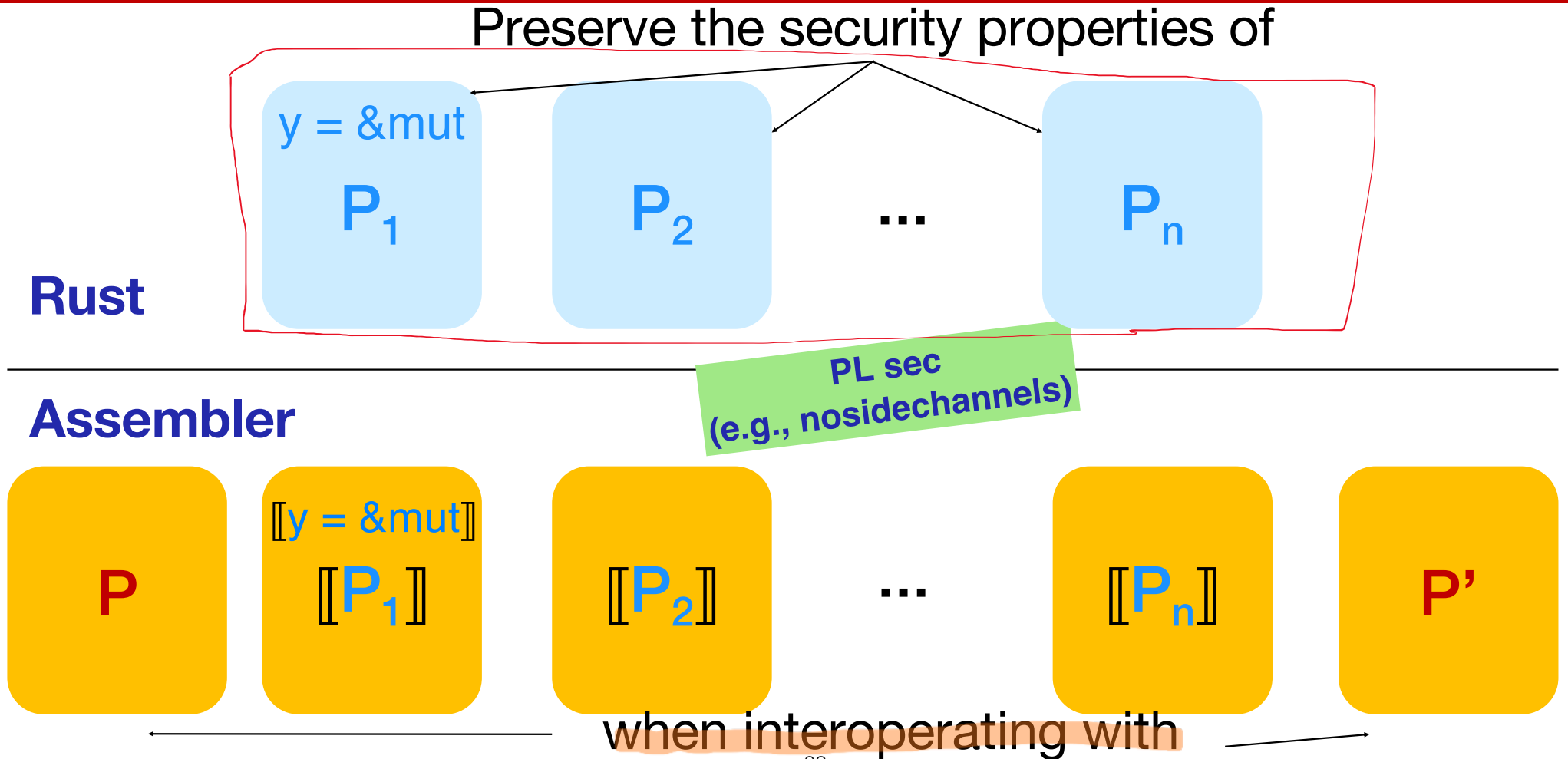
violate **linearity**

There is no guarantee that if you have programs P and P' that interact with the other programs (in the memory for example) that linearity will be respected. At assembler there is no similar linearity guarantee

Possible violations of linearity

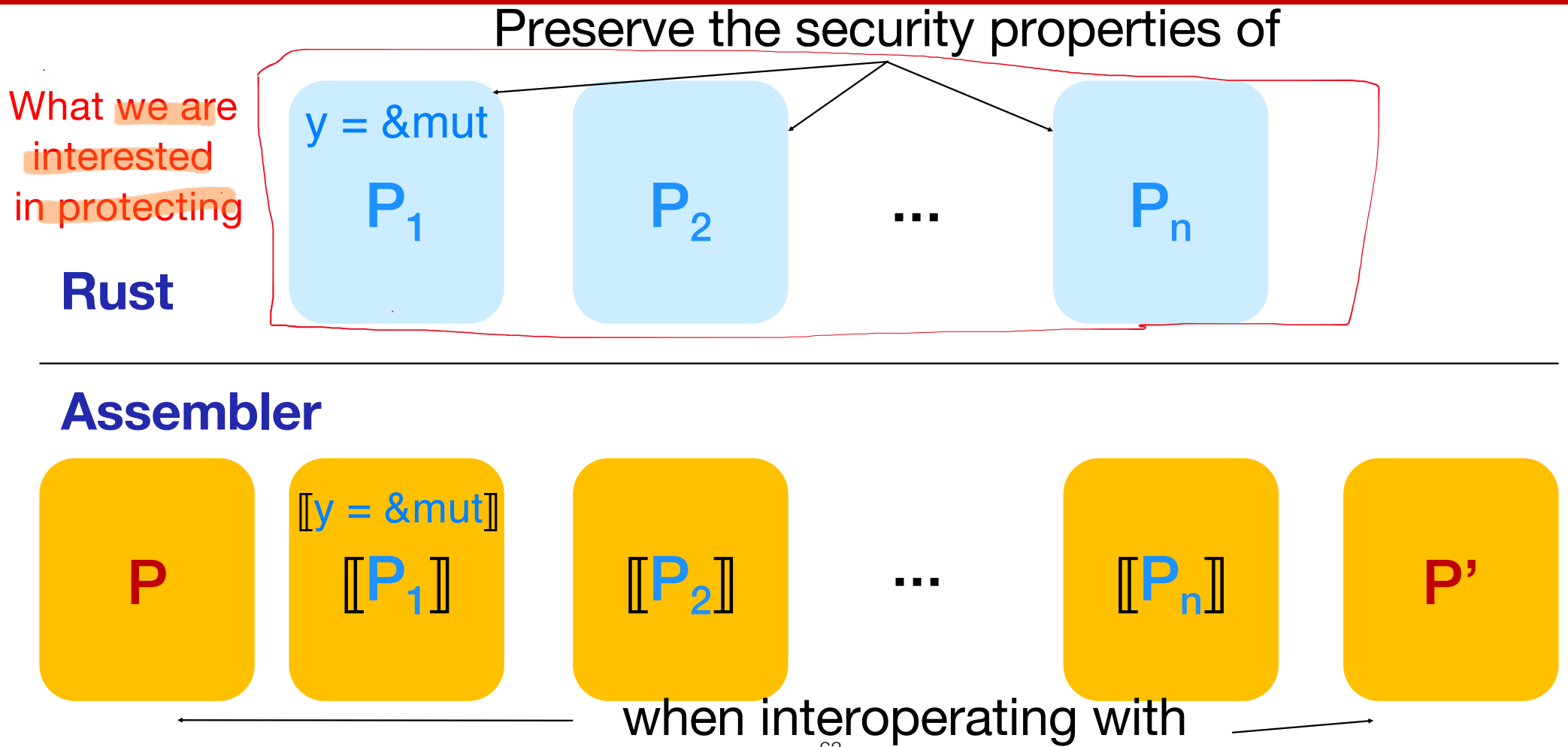


Preservation of linearity

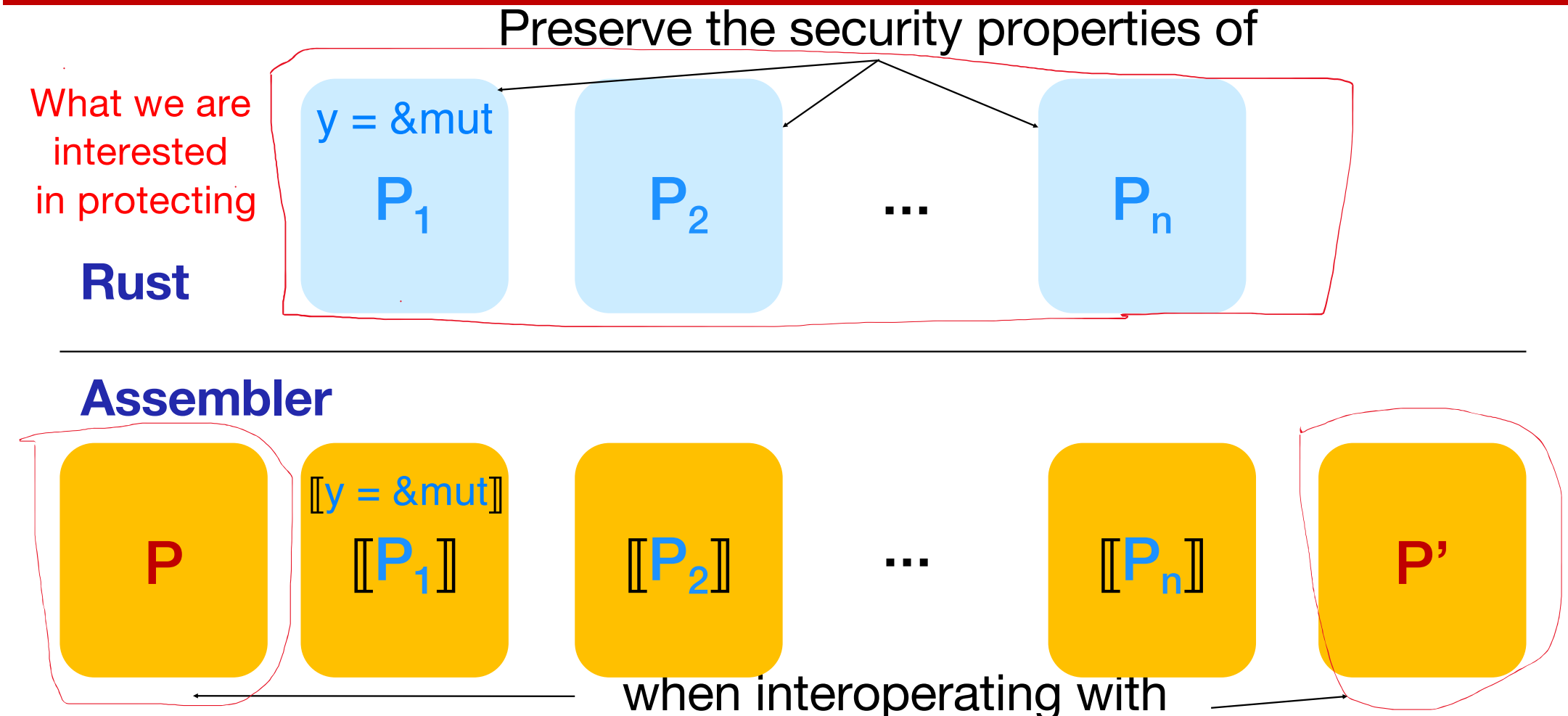


P and P' are not necessarily designed securely. For security we also care about the rest of the environment, unlike correctness

Threat model hint



Threat model hint



Correct compilation

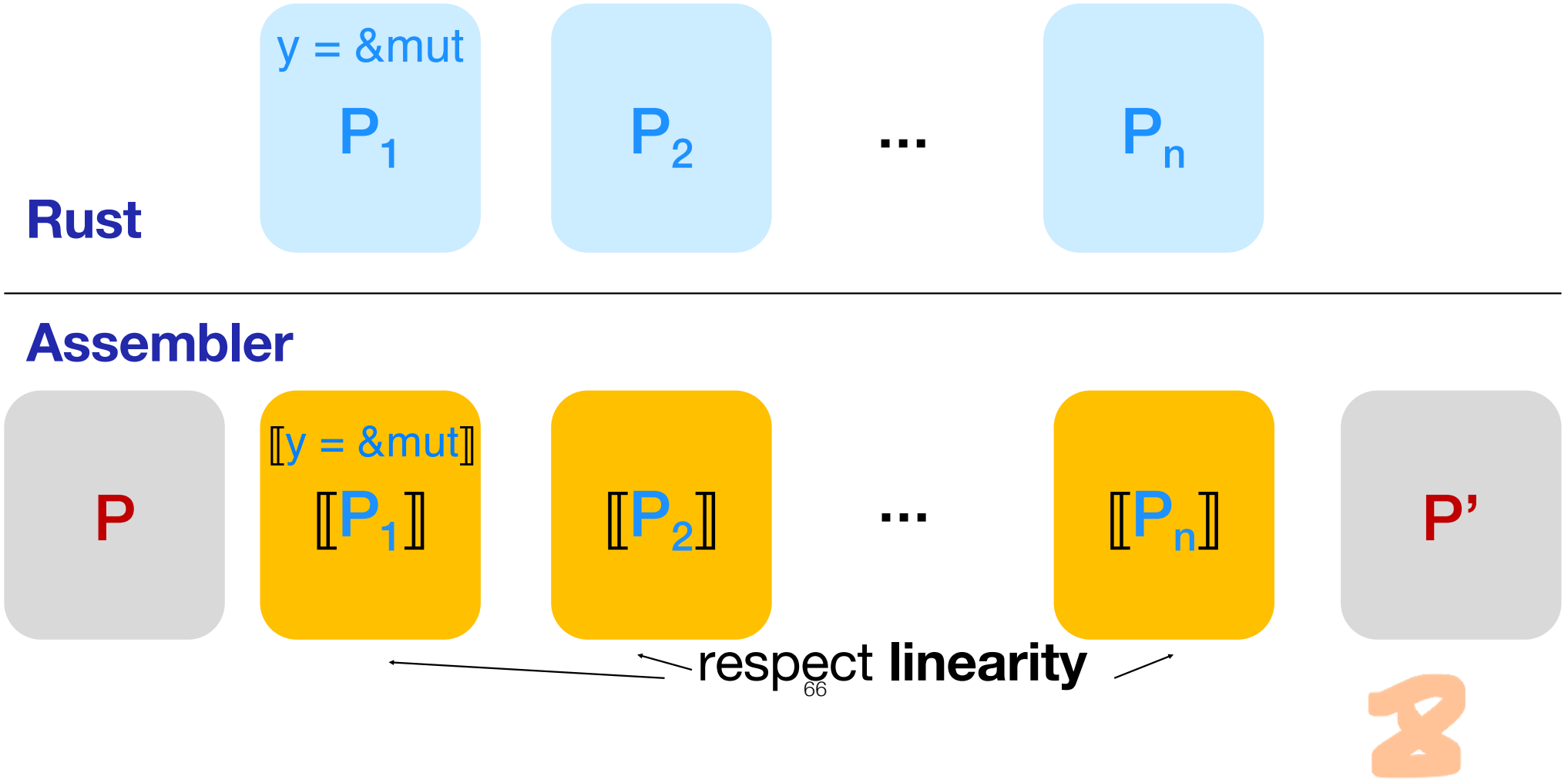
Rust

y = &mut P_1 | P_2 | ... | P_n

Assembler

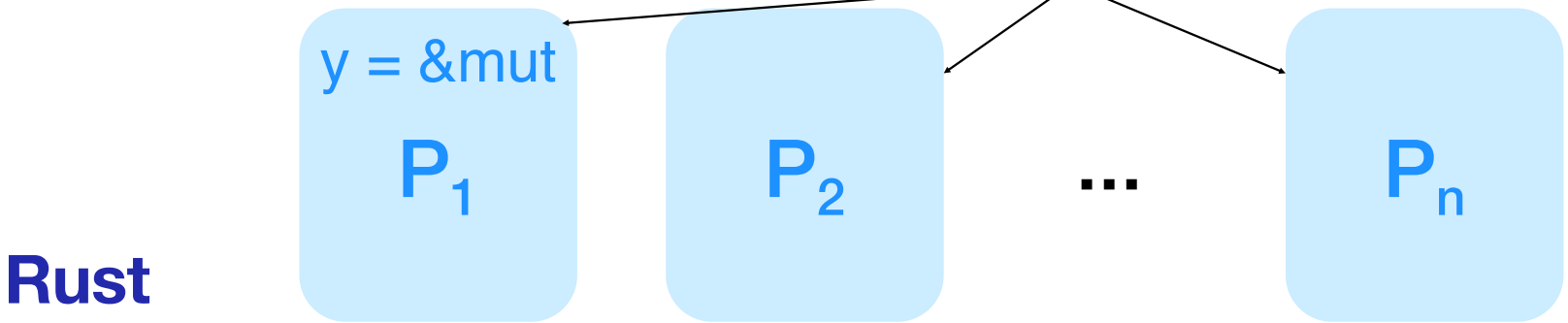
P | ⟦y = &mut⟧ ⟦P_1⟧ | ⟦P_2⟧ | ... | ⟦P_n⟧ | P’

Correct compilation



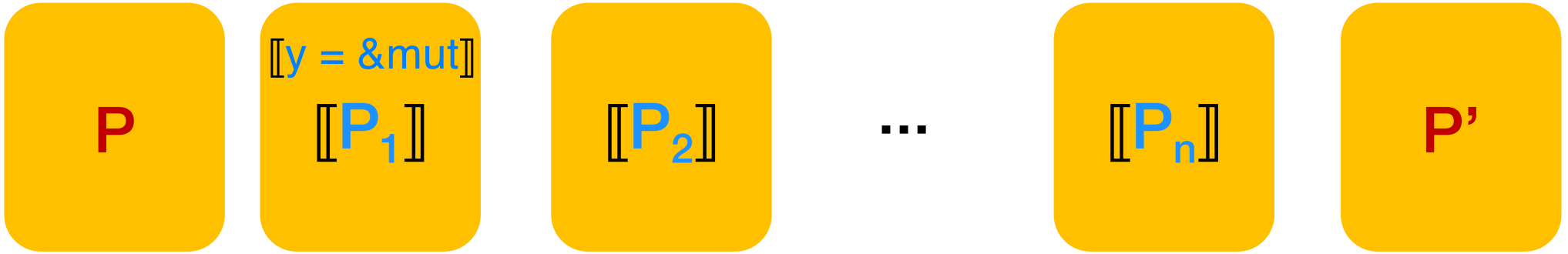
Secure compilation

Enable source-level security reasoning



Rust

Assembler



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? There are "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

Very often, this discrepancy is where security vulnerability arise

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 Compilation and Related Work*. ACM Computing surveys, 2019.

In this context, *object level* comes from a more old-school meaning of "object": think **object files**, like the .o files you get when you compile C code but before linking everything together. It's about the machine's view of your program — instructions, memory addresses, raw compiled pieces — not "objects" as in programming entities.

So if you had a C program (not even remotely object-oriented), once you compile it, you get object files. That's the level they are talking about — the gritty, compiled reality, not the fancy classes and methods.

Outline

Motivating example
Overview
Security via program equivalences
Fully-abstract compilation

Example: compilation

```
1 package Bank;
2
3 public class Account{
4   private int balance = 0;
5
6   public void deposit( int amount ) {
7     this.balance += amount;
8   }
9 }
```

Listing 1. Example of Java source code.

Vulnerability addressed at the beginning of our course

Example: compilation

```
1 package Bank;
2
3 public class Account{
4   private int balance = 0;
5
6   public void deposit( int amount ) {
7     this.balance += amount;
8   }
9 }
```

No access to balance from outside

Listing 1. Example of Java source code.

Example: compilation

```
1 package Bank;
2
3 public class Account{
4   private int balance = 0;
5
6   public void deposit( int amount ) {
7     this.balance += amount;
8   }
9 }
```

No access to balance
from outside
Enforced by the language

Listing 1. Example of Java source code.

Example: compilation

```
1  package Bank;
2
3  public class Account{
4    private int balance = 0;
5
6    public void deposit( int amount ) {
7      this.balance += amount;
8    }
9  }
```

No access to balance from outside
Enforced by the language

Listing 1. Example of Java source code.

```
1  typedef struct account_t {
2    int balance = 0;
3    void ( *deposit ) ( struct Account*, int ) = deposit_f;
4  } Account;
5
6  void deposit_f( Account* a, int amount ) {
7    a→balance += amount;
8    return;
9  }
```

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

Example: compilation

```
1  package Bank;
2
3  public class Account{
4    private int balance = 0;
5
6    public void deposit( int amount ) {
7      this.balance += amount;
8    }
9  }
```

Listing 1. Example of Java source code.

No access to balance from outside
Enforced by the language

```
1  typedef struct account_t {
2    int balance = 0;
3    void ( *deposit ) ( struct Account*, int ) = deposit_f;
4  } Account;
5
6  void deposit_f( Account* a, int amount )
7    a→balance += amount;
8    return;
9  }
```

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 need to know what the attacker can do

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

Threats: confidentiality

```
1 private secret : Int = 0;
2
3 public setSecret() : Int {
4   secret = 1;
5   return 0;
6 }
```

Threats: confidentiality

```
1 private secret : Int = 0;
2
3 public setSecret() : Int {
4   secret = 1;
5   return 0;
6 }
```

No access to secret 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

```
1 public proxy( callback : Unit → Unit ) : Int {
2   var secret = 1;
3   callback();
4   return 0;
5 }
```

Threats: integrity

```
1 public proxy( callback : Unit → Unit ) : Int {
2   var secret = 1;
3   callback();
4   return 0;
5 }
```

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

This could be violated at low level but attacker can manipulate the stack to access and change secret value

Threats: memory size

```
1 public kernel( n : Int, callback : Unit →Unit ) : Int {
2   for (i = 0 to n){
3     new Object();
4   }
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

Threats: memory size

```
1 public kernel( n : Int, callback : Unit →Unit ) : Int {
2   for (i = 0 to n){
3     new Object();
4   }
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

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

Something may go wrong at low level: the limit of what can be produced is n for the memory, then the security relevant code will be unreachable

Threats: deterministic memory allocation

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return x;
5 }
```

Threats: deterministic memory allocation

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return x;
5 }
```

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

```
1  class Pair {
2    private first, second : Obj = null;
3    public getFirst(): Obj {
4      return this.first;
5    }
6  }
7  class Secret {
8    private secret : Int = 0;
9  }
10 object o : Secret
```

You can only get 1st obj.

Threats: well-typedness

```
1  class Pair {
2    private first, second : Obj = null;
3    public getFirst(): Obj {
4      return this.first;
5    }
6  }
7  class Secret {
8    private secret : Int = 0;
9  }
10 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

```
1  class Pair {
2    private first, second : Obj = null;
3    public getFirst(): Obj {
4      return this.first;
5    }
6  }
7  class Secret {
8    private secret : Int = 0;
9  }
10 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

offset 0,

•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.

getFirst would allow to know location in memory, and with evaluated offset we can get the secret field

Threats: information flow

```
1 public isZero( value : Int_h ) : Int_l {
2   if ( value == 0 ) {
3     return 1
4   }
5   return 0
6 }
```

Listing 4. Example code with indirect information flow.

Threats: information flow

```
1 public isZero( value : Int_h ) : Int_l {
2   if ( value == 0 ) {
3     return 1
4   }
5   return 0
6 }
```

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

At the *source level* — when you, the programmer, are writing — you might think you're doing things safely. Maybe in your source language, there are rules or tools that are supposed to prevent secret information from leaking into outputs.
But **if your compiler doesn't carefully preserve those protections** when turning your code into object code, suddenly your "safe" program could start leaking secrets at runtime, without you even realizing it!

In the quote you gave me, it's talking about how **some target languages** — meaning the languages that the source compiles into (like machine code, assembly, etc.) — **don't automatically prevent these kinds of leaks**.
If the target environment is "loose," if it doesn't help enforce secrecy, then even well-written source code can become vulnerable once it's compiled.

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

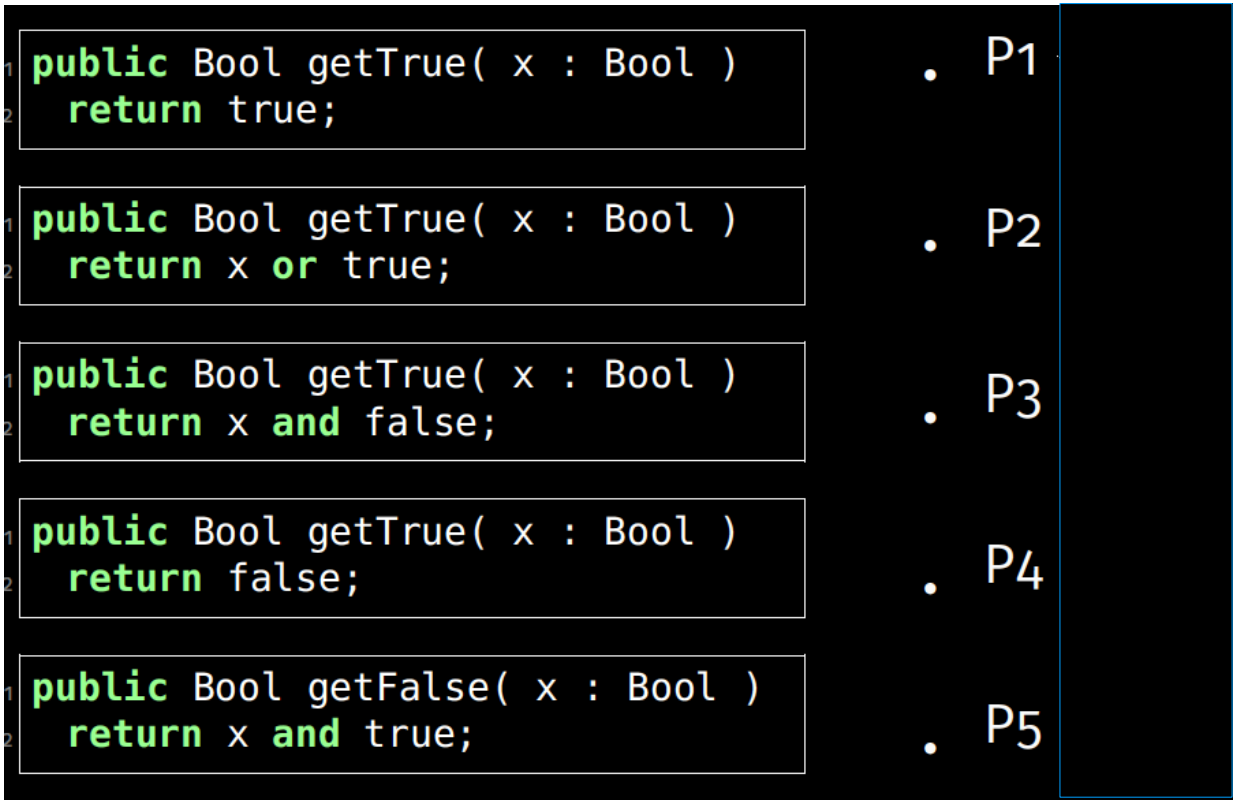
Program equivalences

- A possible way to know what is secure in a program is by exploiting **program equivalences**, under the assumption is that program equivalence capture security properties of source code

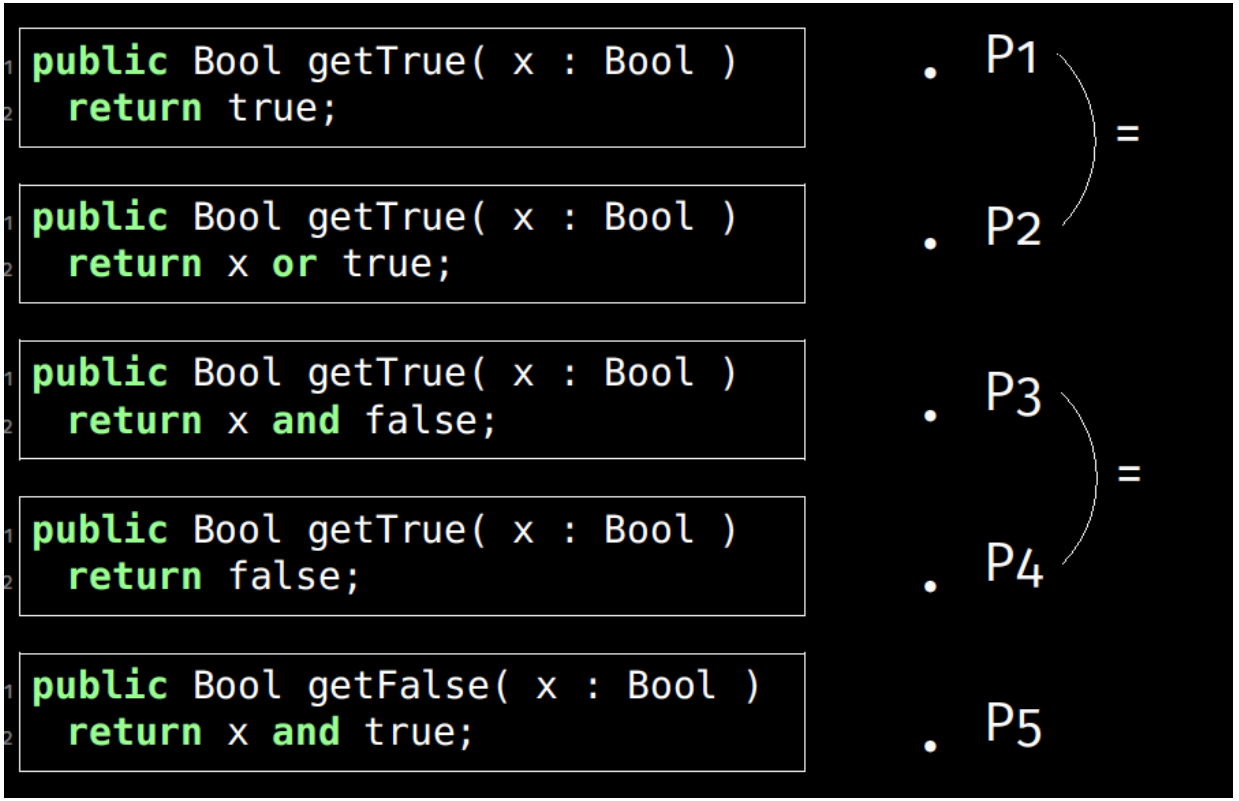
Program equivalences are:

- reflexive
- transitive $x=y, y=z \Rightarrow x=z$
- symmetric $x=y \Rightarrow y=x$

Are these programs equivalent?



Are these programs equivalent?



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

Confidentiality as equivalence

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 0;
5   return 0;
6 }
```

If the two snippets are equivalent then secret is confidential

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 1;
5   return 0;
6 }
```

Confidentiality as equivalence

Source level equivalent

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 0;
5   return 0;
6 }
```

If the two snippets are equivalent then secret is confidential

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int
4   secret = 1;
5   return 0;
6 }
```

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

With a C-like semantics, it does

We can rethink of the example in terms of equivalence. If these two are equivalent then you enjoy confidentiality.

Integrity as equivalence

```
1 public proxy( callback : Unit → Unit )
        : Int {
2   var secret = 0;
3   callback();
4   if ( secret == 0 ) {
5     return 0;
6   }
7   return 1;
8 }
```

```
1 public proxy( callback : Unit → Unit )
        : Int {
2   var secret = 0;
3   callback();
4
5   return 0;
6
7
8 }
```

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

Integrity as equivalence

```
1 public proxy( callback : Unit → Unit )
        : Int {
2   var secret = 0;
3   callback();
4   if ( secret == 0 ) {
5     return 0;
6   }
7   return 1;
8 }
```

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

```
1 public proxy( callback : Unit → Unit )
        : Int {
2   var secret = 0;
3   callback();
4
5   return 0;
6
7
8 }
```

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

↓ So they are not!

Unbounded memory size as equivalence

```
1 public kernel( n : Int, callback : Unit
      → Unit ) : Int {
2   for (Int i = 0; i < n; i++){
3     new Object();
4   }
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

```
1 public kernel( n : Int, callback : Unit
      → Unit ) : Int {
2
3
4
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

Unbounded memory size as equivalence

```
1 public kernel( n : Int, callback : Unit
        → Unit ) : Int {
2   for (Int i = 0; i < n; i++){
3     new Object();
4   }
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

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

```
1 public kernel( n : Int, callback : Unit
        → Unit ) : Int {
2
3
4
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

If they are equivalent they should both preserve property of memory size

Memory allocation as equivalence

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return x;
5 }
```

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return y;
5 }
```

Memory allocation as equivalence

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return x;
5 }
```

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return y;
5 }
```

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

unexploitable by attacker

Program equivalences

Defining a security property using program equivalence amounts to finding two programs that, albeit syntactically different, both behave in a way that respects the property, no matter how they are used

How to express program equivalence? formally? There are different ways, we will focus on ①

- **contextual equivalence** ①
- observational equivalence (e.g., for non-interference property)
- timing/resource-sensitive equivalence (e.g., security of constant-time code)

Contextual equivalence

Contextual equivalence

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

$$P_1 \simeq_{ctx} P_2 \Rightarrow \forall\, \mathcal{C}.\ \mathcal{C}(P_1)\downarrow \Leftrightarrow \mathcal{C}(P_2)\downarrow$$

Plugin of program in the context

This is a context

With correctness only programs themselves, here outside is considered

$\downarrow$ refers to termination

Contextual equivalence

Contextual equivalence

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

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

$\downarrow$ refers to termination

Contextual equivalence

Contextual equivalence

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

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

$\downarrow$ refers to termination

Contextual equivalence

Contextual equivalence

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

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

$\downarrow$ refers to termination

Contextual equivalence

Contextual equivalence

Two programs are equivalent if no matter what context/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

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, it can also observe termination, non termination etc.
- 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

Program is partial because it cannot evaluate anything, no values associated with variables

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

You compose context with the component

Contextual equivalence

In Javascript

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

Is there a program context that can distinguish them?

Contextual equivalence

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

Contextual equivalence

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() turns the function into a string containing its source code

console.log((function(x){ code; }).toString()); prints function(x){ code; }

Confidentiality as equivalence

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 0;
5   return 0;
6 }
```

If the two snippets are equivalent then secret is confidential

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 1;
5   return 0;
6 }
```

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

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

Java

P cannot tell the difference!

Observer program trying to do something

Confidentiality as equivalence

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

C

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

C

P can tell the difference!

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

With a C-like semantics, secret can be accessed from outside

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

↑ attackers are active and do something in this interaction (ex: access memory etc)

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)

meaning interested in their interaction with environment

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?

Formal (secure compilation): full abstraction

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

$\forall\ P_1, P_2$

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

also the target level ones are

context eq.

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.,

$\forall\ P_1, P_2$

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

Something like correct. If targets are equivalent, then also the opposite must be true

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 (not equivalent)
  ↑ can also be interpreted as
- 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")

Fully Abstract compilation

Then we have

$\forall\ P_1, P_2$

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

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

Fully Abstract compilation

Then we have

$\forall\ P_1, P_2$

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

$\forall\ P_1, P_2.$

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

Correctness is needed

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

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

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

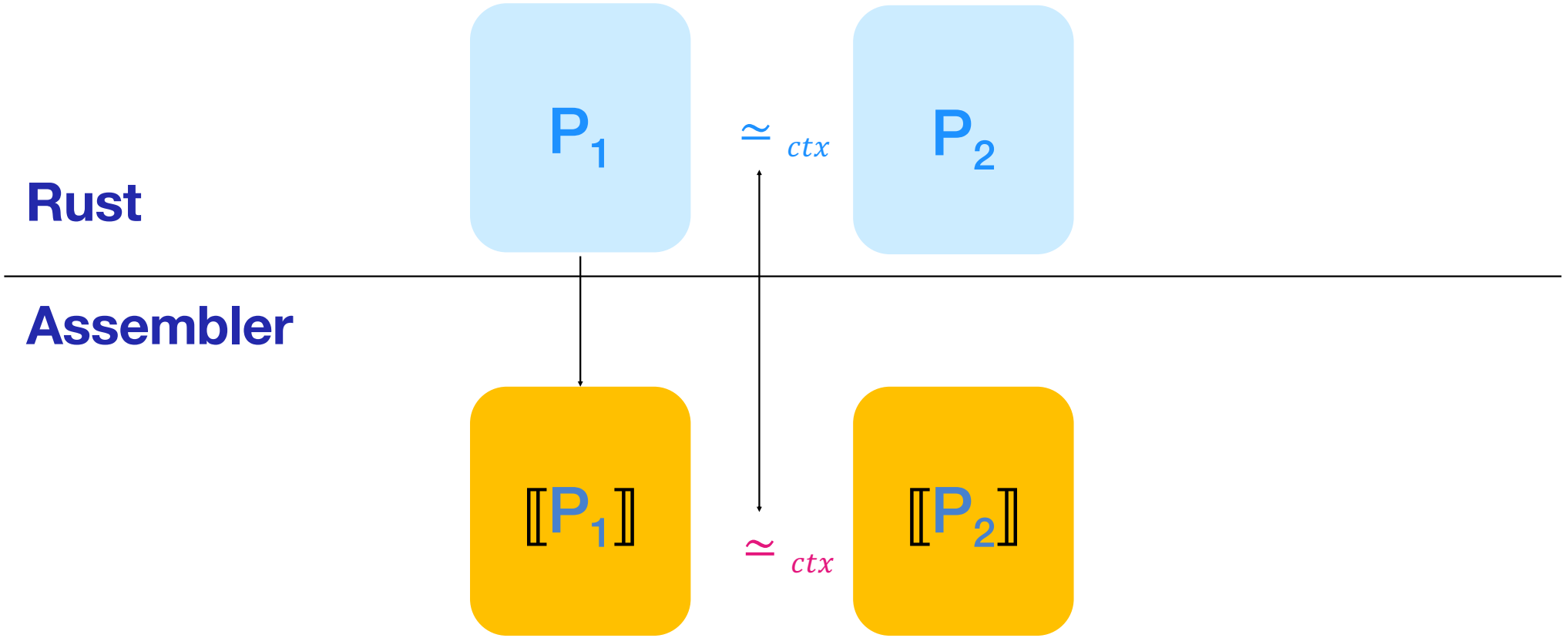
$$\forall\, P_1, P_2 .\; P_1 \simeq_{ctx} P_2 \Rightarrow [\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$$

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

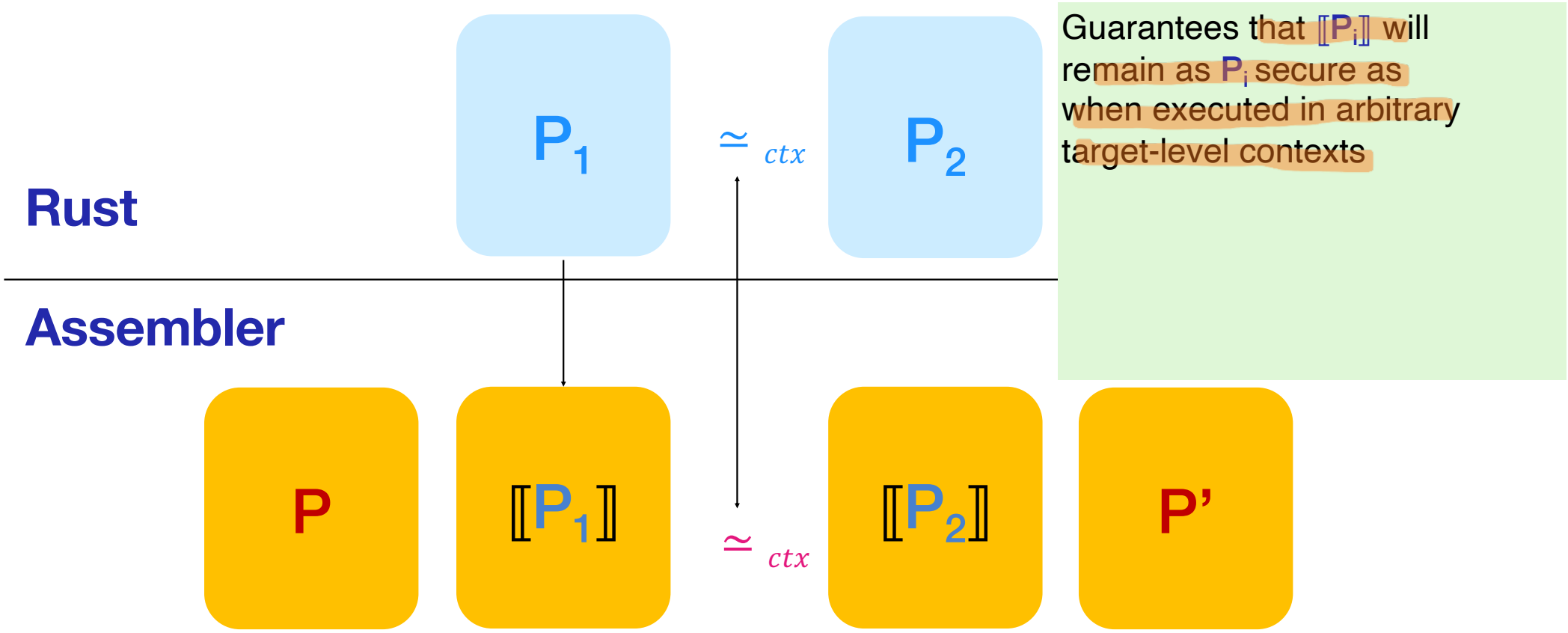
Full abstract compilation

Preserve and reflect contextual equivalence



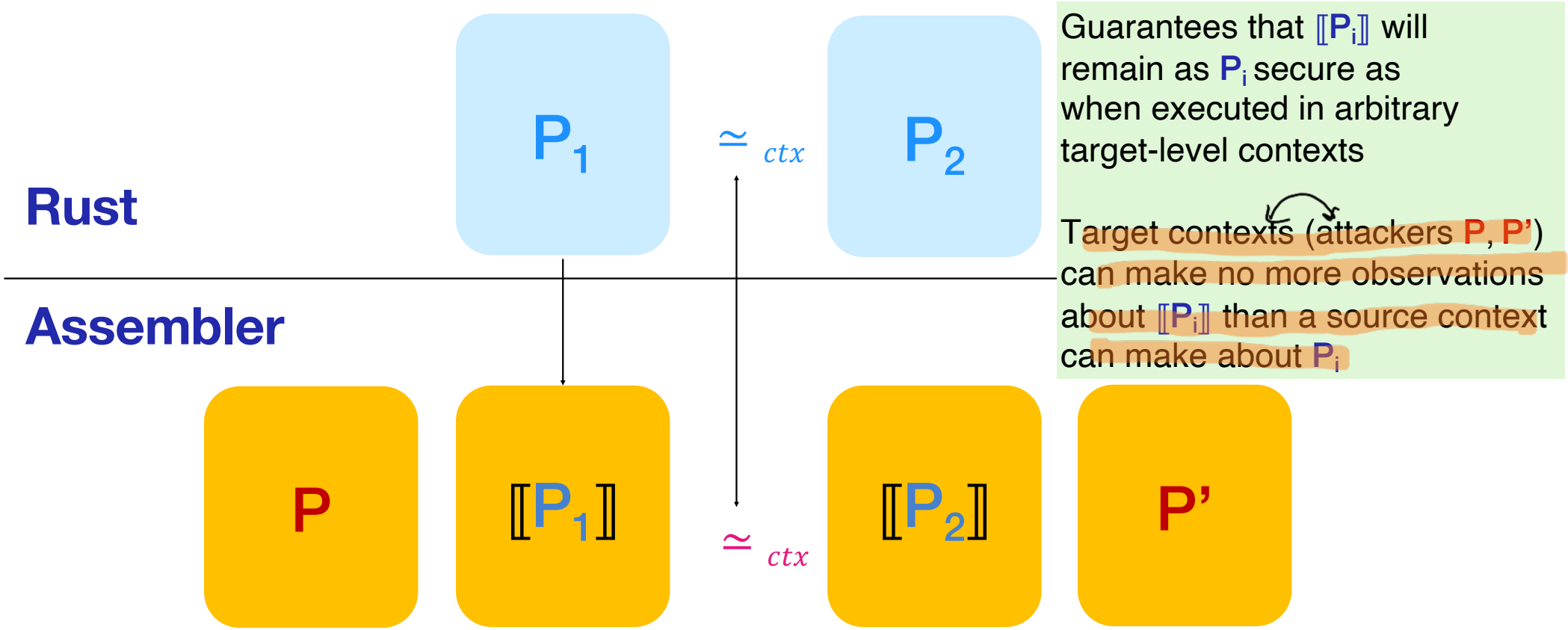
Full abstract compilation

Preserve and reflect contextual equivalence



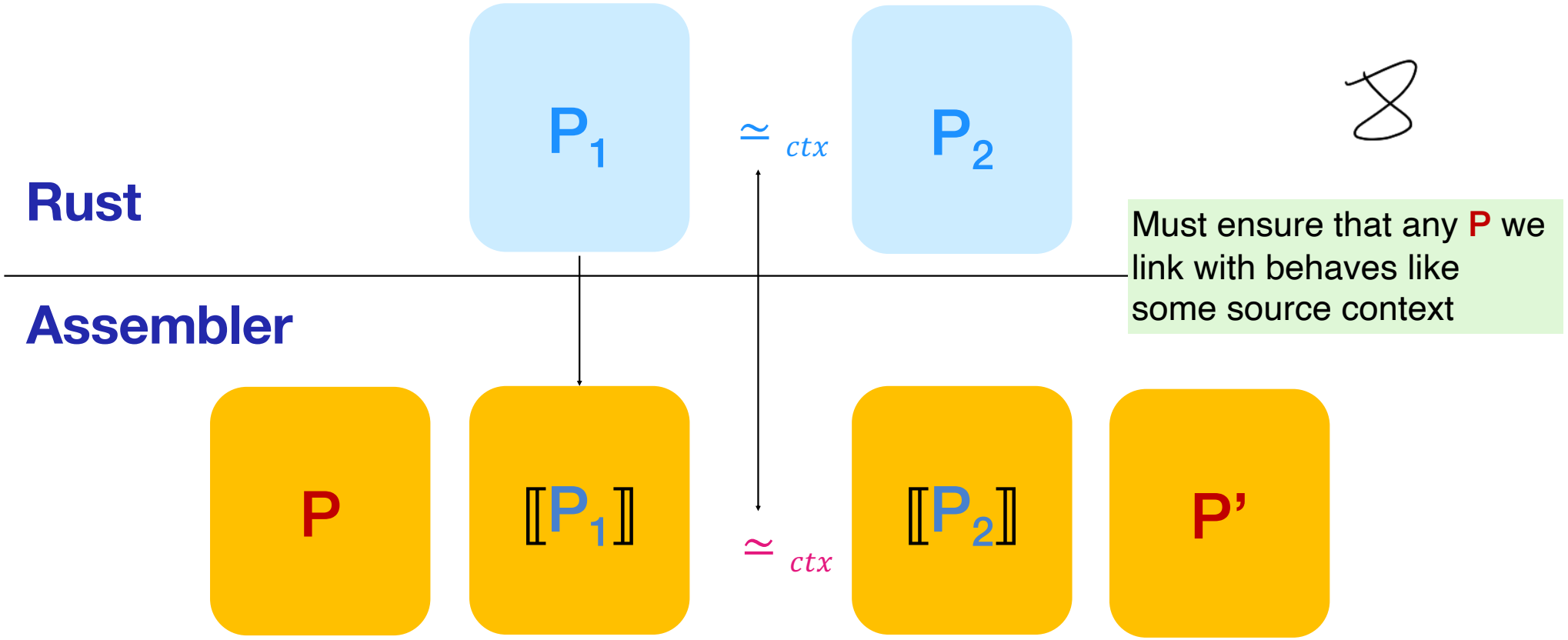
Full abstract compilation

Preserve and reflect contextual equivalence



Full abstract compilation

Preserve and reflect contextual equivalence



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 So we can just reason at high level!

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

We work with all possible contexts, so we target attackers that can do anything

- 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

Conclusion

- Program equivalences can be used to define security properties
- Preserving (and reflecting) equivalences can be used to define a secure compiler

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