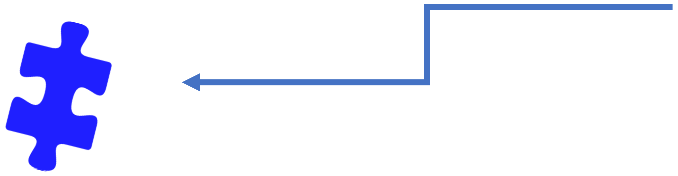
CapablePtrs

Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle

Akram El-Korashy (MPI-SWS), Stelios Tsampas (KU Leuven),
Marco Patrignani (CISPA), Dominique Devriese (VUB),
Deepak Garg (MPI-SWS), Frank Piessens (KU Leuven)

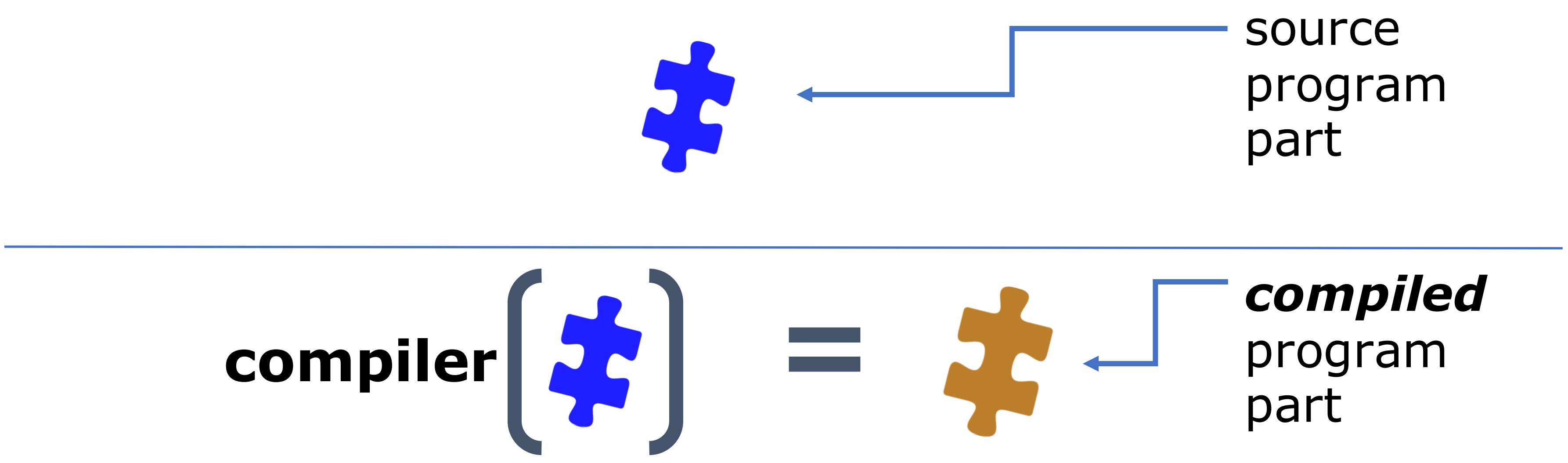
elkorashy@mpi-sws.org

Why Securely Compiling Partial Programs ?



source program part

Why Securely Compiling Partial Programs ?

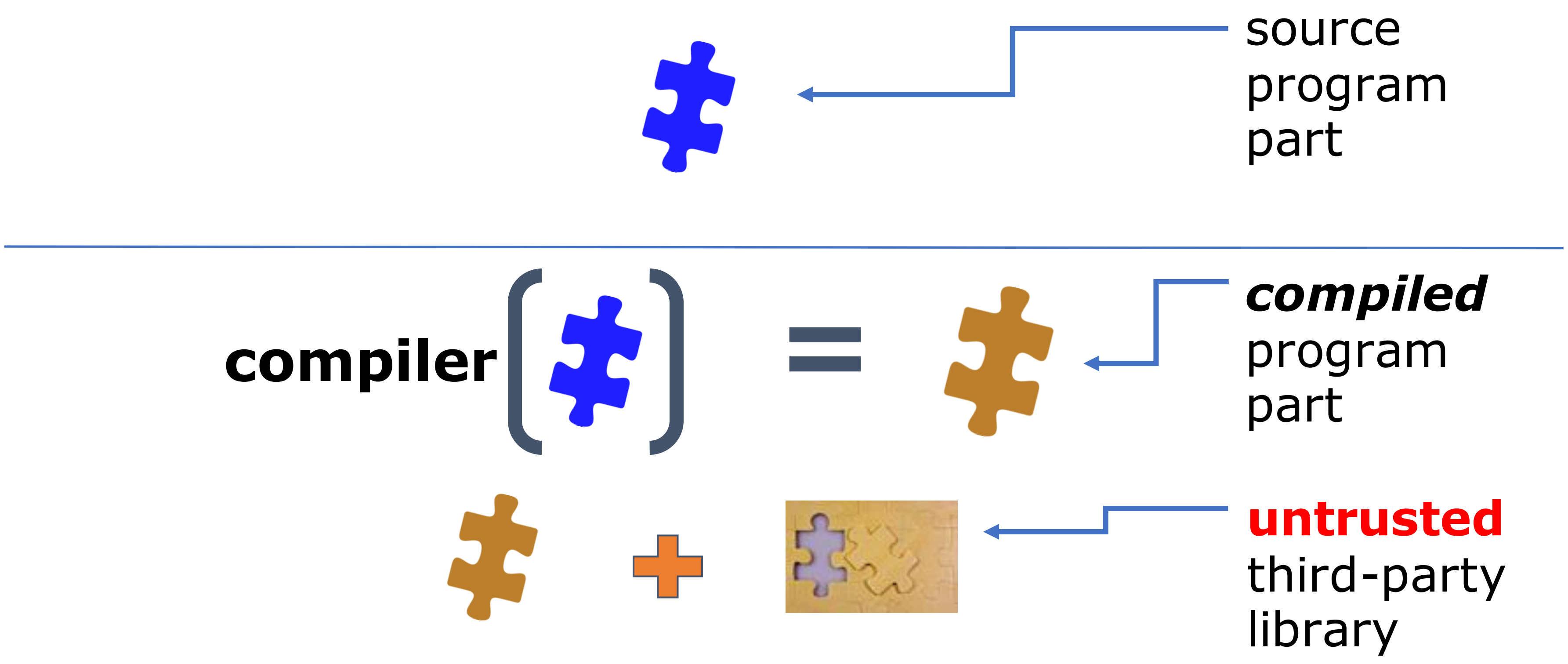


source program part

compiler =

compiled program part

Why Securely Compiling Partial Programs ?



source program part

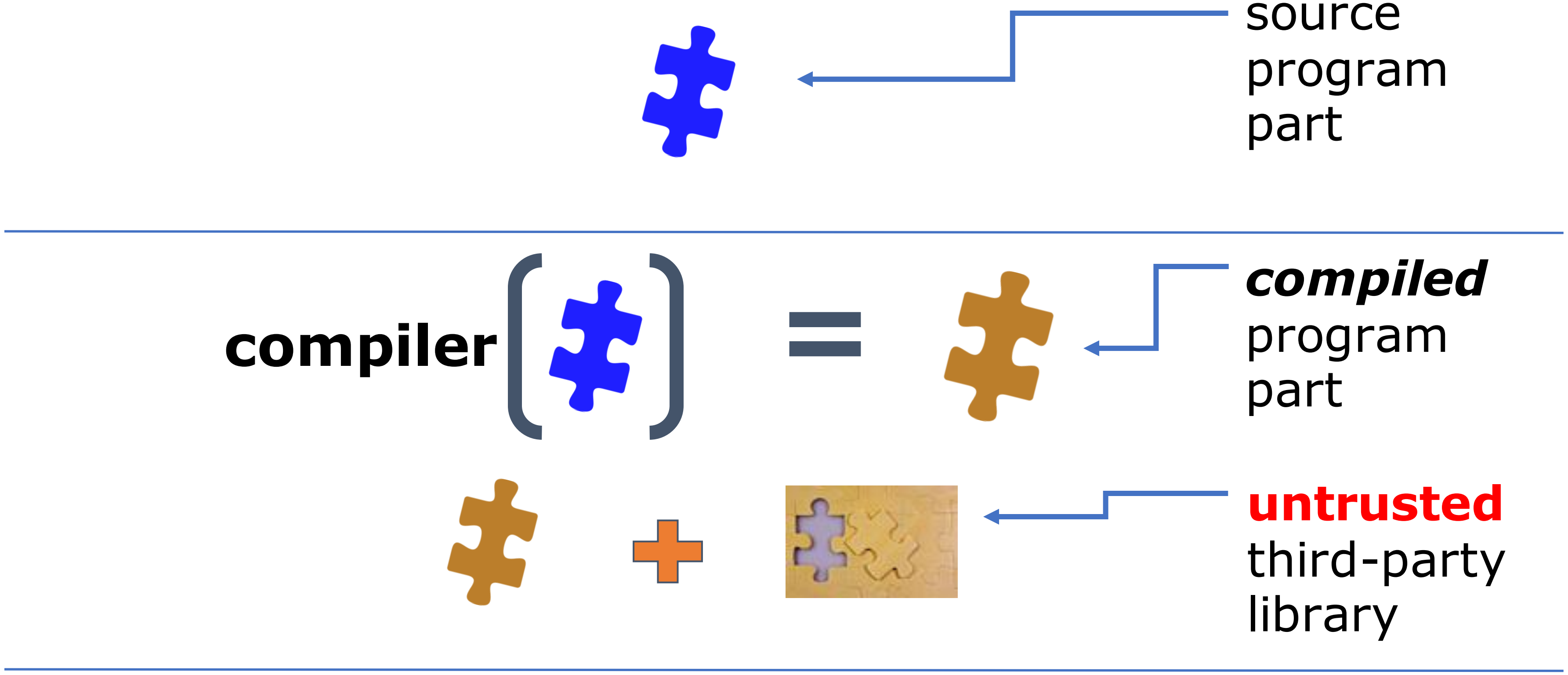
compiler

=

compiled program part

untrusted third-party library

Why Securely Compiling Partial Programs ?



source program part

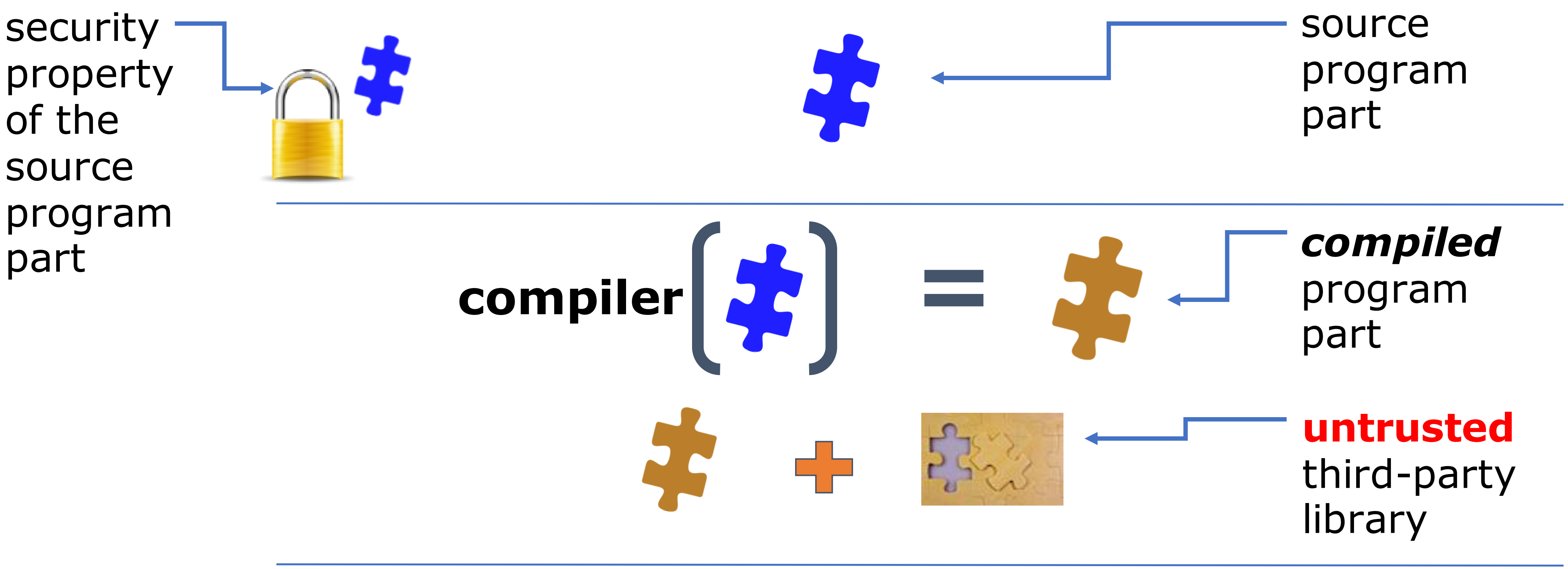
compiler

compiled program part

untrusted third-party library

could be **buggy** or **malicious**

Why Securely Compiling Partial Programs ?



security property of the source program part

source program part

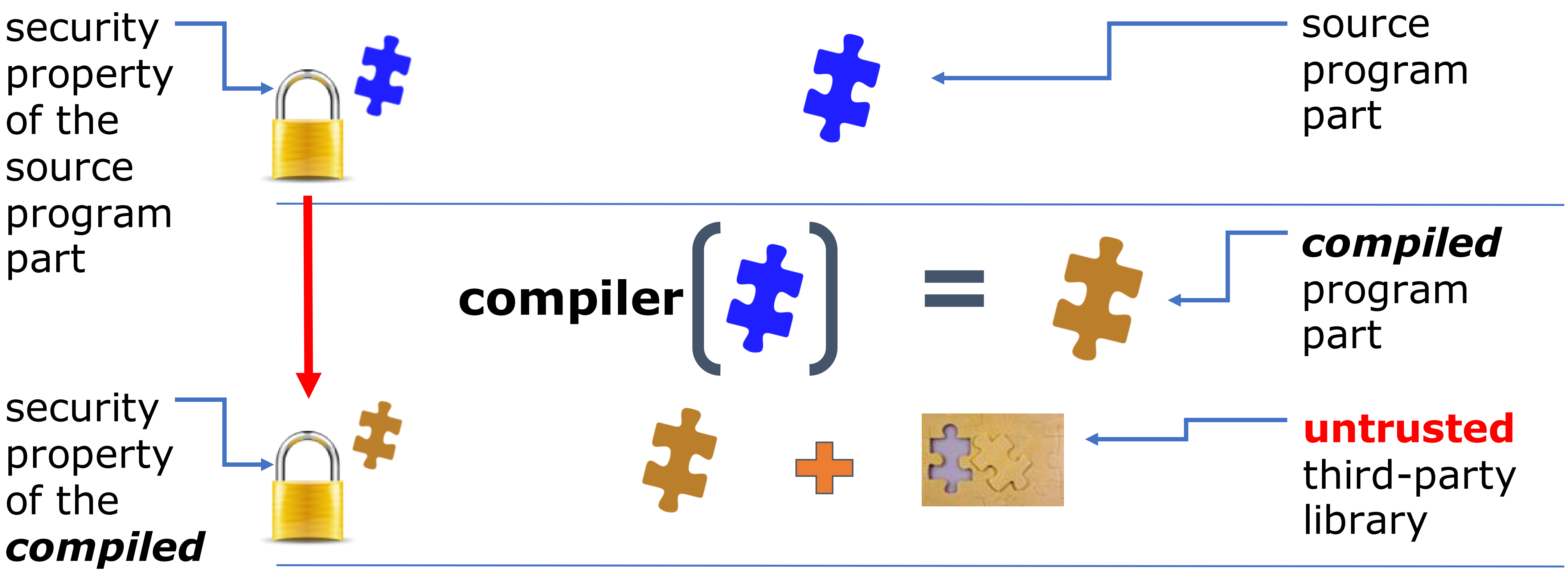
compiler

compiled program part

untrusted third-party library

could be **buggy** or **malicious**

Why Securely Compiling Partial Programs ?



security property of the source program part

source program part

compiler [ ] =

compiled program part

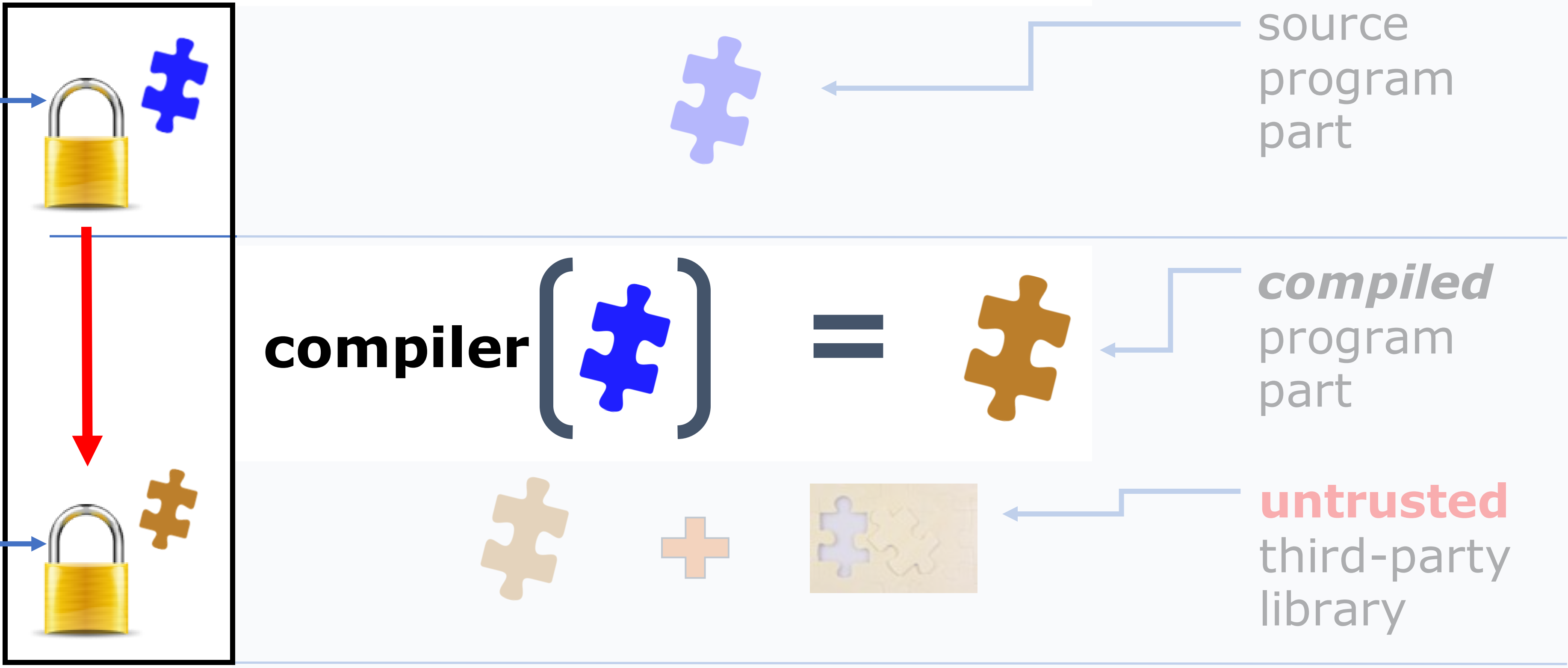
+

<span></span>**untrusted** third-party library

security property of the ***compiled*** program part

could be **buggy** or **malicious**

Why Securely Compiling Partial Programs ?



security property of the source program part

security property of the ***compiled*** program part

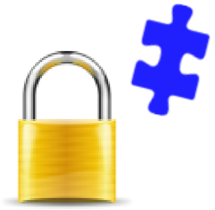
compiler [ ] =

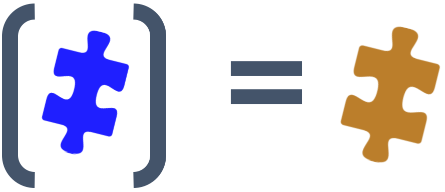
source program part

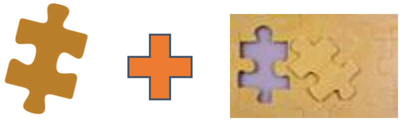
compiled program part

untrusted third-party library

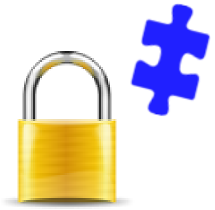
could be **buggy** or **malicious**

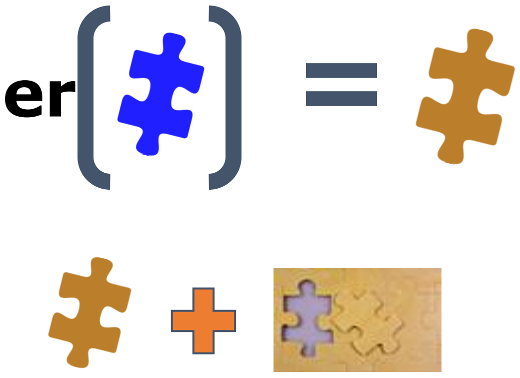
Let  be **confidentiality**

compiler 



could be **buggy** or **malicious**

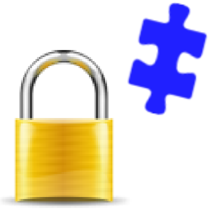
Let  be **confidentiality**



compiler

could be **buggy** or **malicious**

```
#include "networking.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

Let  be **confidentiality**

could be **buggy** or **malicious**

```
#include "ne        g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer        ;

int main(voi

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

Let 🔒🧩 be **confidentiality**

🧩 owns in memory

could be **buggy** or **malicious**

```
#include "ne
```

```
g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

   init_secret(secret);
   receive(iobuffer);
   process(iobuffer, secret);

   return 0;
}
```

Let 🔒🧩 be **confidentiality**

🧩 owns ✉ in memory

🧩 could read ✉ from the memory of 🧩

🧩 could be **buggy** or **malicious**

```
#include "ne        g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];      ✉

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

Let be **confidentiality**

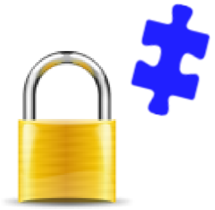
could read from the memory of

could be **buggy** or **malicious**

malicious

```
char *secret_ptr =
      (char*)4210756;

leak(*secret_ptr);
```

```
#include "ne          g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

Let  be **confidentiality**

could read from the memory of

could be **buggy** or **malicious**

buggy

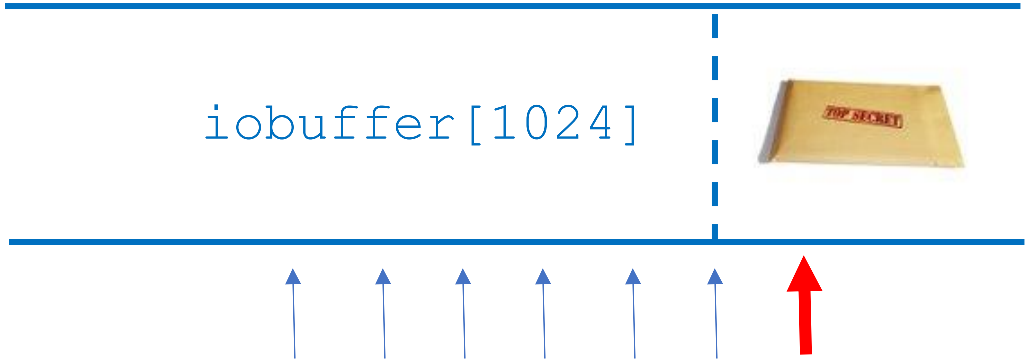
```
#include "ne        g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

Let 🔒🧩 be **confidentiality**

🧩 could read TOP SECRET from the memory of 🧩

🧩 could be **buggy** or **malicious**

buggy 🧩



```
#include "ne🧩g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

TOP SECRET

Let 🔒🧩 be **confidentiality**

🧩 could read TOP SECRET from the memory of 🧩

🧩 could be **buggy** or **malicious**

buggy 🧩

```
int receive (char* buffer) {
  ...
  int checksum = 0;
  for (int i=0; i<=1024; i++)
    checksum += buffer[i];
  send_checksum(checksum);
}
```

```
#include "ne🧩g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

TOP SECRET

Let 🔒🧩 be **confidentiality**

could read

from the memory of 🧩

could be **buggy** or **malicious**

```
#include "ne          g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];
```

```
char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

buggy

```
int receive (char* buffer) {
  ...
  int checksum = 0;
  for (int i=0; i<=1024; i++)
    checksum += buffer[i];
  send_checksum(checksum);
}
```

Let 🔒🧩 be **confidentiality**

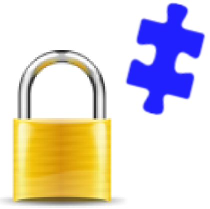
🧩 could read ✉ TOP SECRET from the memory of 🧩

🧩 could be **buggy** or **malicious**

buggy 🧩

```
int receive (char* buffer) {
  ...
  int checksum = 0;
  for (int i=0; i<=1024; i++)
    checksum += buffer[i];
  send_checksum(checksum);
}
```

```
#include "ne🧩g.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];          ✉ TOP SECRET

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

Let  be **confidentiality**

could read from the memory of

could be **buggy** or **malicious**

buggy

```
int receive (char* buffer) {
  ...
  int checksum = 0;
  for (int i=0; i<=1024; i++)
    checksum += buffer[i];
  send_checksum(checksum);
}
```

`#include "ne` `g.h"`

```
void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];
```

```
char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
  process(iobuffer, secret);

  return 0;
}
```

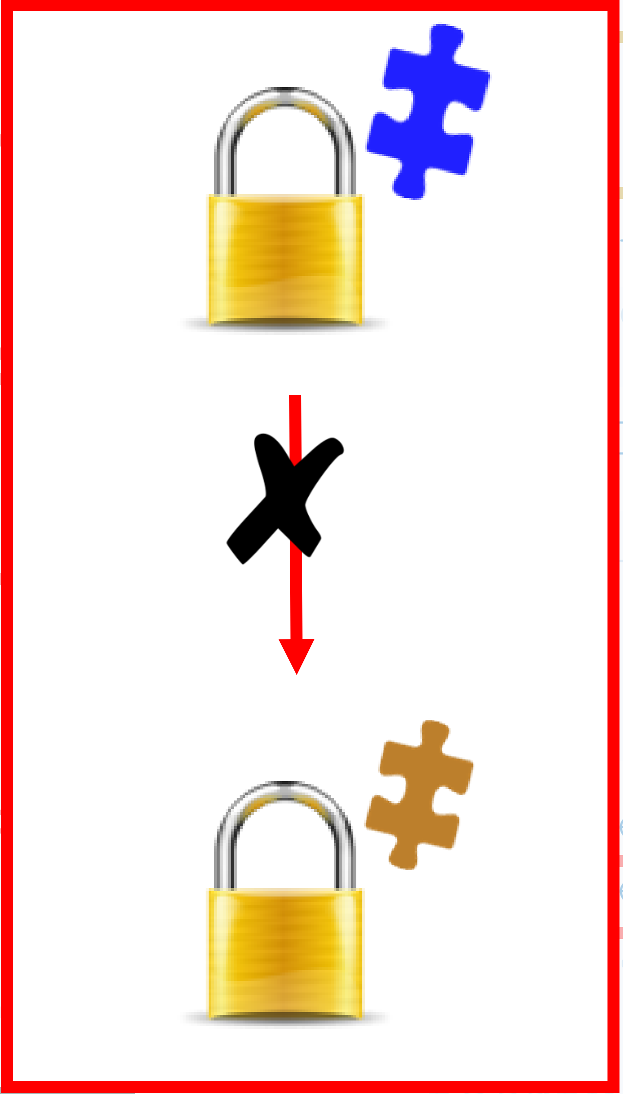
Let be confidentiality

could read from the memory of

could be buggy or mal

buggy

```
int receive (char* buf
  ...
  int checksum = 0;
  for (int i=0; i<=102
    checksum += buffer
  send_checksum(checksum);
}
```



```
"ne        g.h"
t_secret(char* s);
cess(char* b, char* s);

har secret[256];

uffer[1024];

(void) {

ecret(secret);
e(iobuffer);
s(iobuffer, secret);

0;
}
```

Let be **confidentiality**

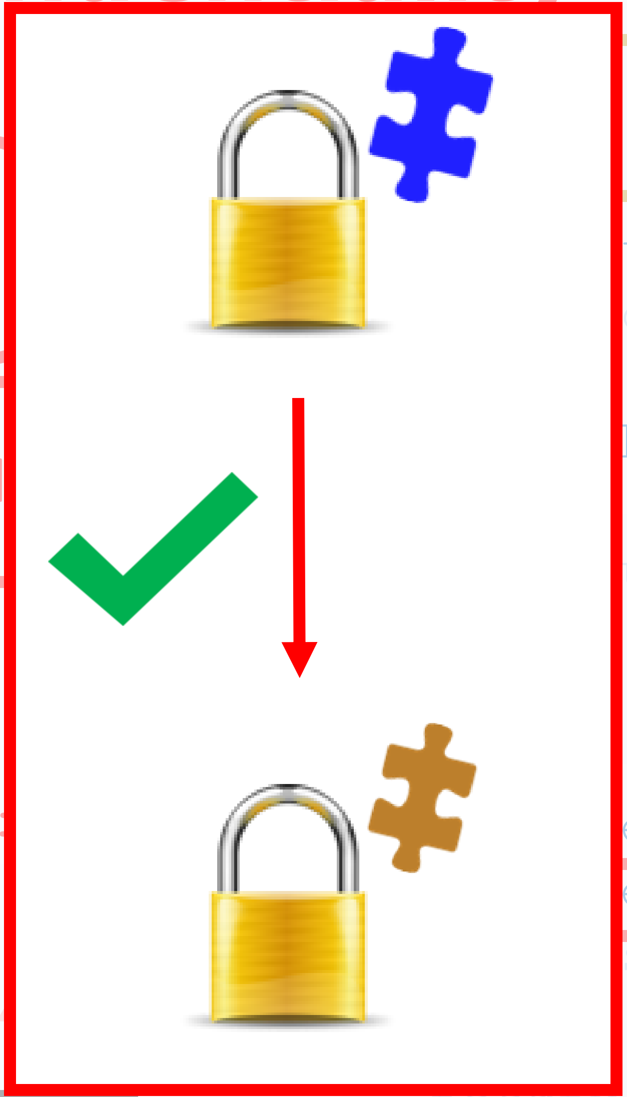
could read from the memory of

could be **buggy** or **mal**

buggy

```
int receive (char* buf
  ...
  int checksum = 0;
  for (int i=0; i<=102
    checksum += buffer
  send_checksum(checksum);
}
```

```
"ne        g.h"

t_secret(char* s);
cess(char* b, char* s);

har secret[256];

uffer[1024];

 (void) {

ecret(secret);
e(iobuffer);
s(iobuffer, secret);

       0;
}
```



Let be confidentiality

could read TOP SECRET fr m

could be bug

buggy

```
int receive (c
  ...
  int checksum
  for (int i=0; i<=1024; i++)
    checksum += buffer[i];
  send_checksum(checksum);
}
```

```
g.h"
ar* s);
b, char* s);
256];
;
t);
;
, secret);
  return 0;
}
```

☑ Can use **process-based isolation**.

Let be confidentiality

could read TOP SECRET fr… m…

could be bug…

buggy

```
int receive (c…
  ...
  int checksum
  for (int i=0; i<=1024; i++)
    checksum += buffer[i];
  send_checksum(checksum);
}
```

```
…g.h"
…ar* s);
…b, char* s);
…256];
…;
…t);
…;
…, secret);
  return 0;
}
```

☑ Can use **process-based isolation**.

? Shared memory needs be set up ahead of time. **No pointer passing at run-time**.

We have **two requirements** for **compiler security**

Isolate the memory of the different parts of the program from each other (with low performance overhead) **while allowing pointer passing**.

We have **two requirements** for **compiler security**

Isolate the memory of th
of the program from each o
performance overhead) **wh**
pointer passing.

```
#include "ne        ing.h"

void init_secret(char* s);
void process(char* b, char* s);

static char secret[256];

char iobuffer[1024];

int main(void) {

  init_secret(secret);
  receive(iobuffer);
```

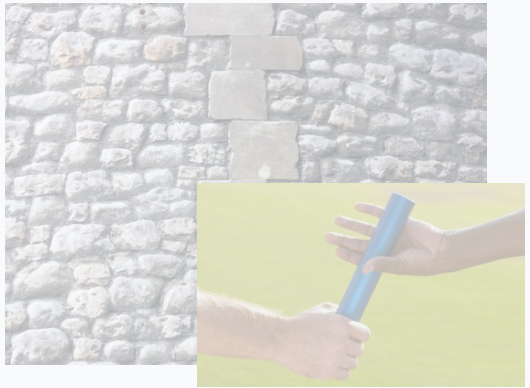
In the program:

+

we will need to isolate the memory of from the memory of

We have **two requirements** for **compiler security**

Isolate the memory of the different parts of the program from each other (with low performance overhead) **while allowing pointer passing**.



Want a proof technique that allows us to **reuse a whole-program compiler correctness theorem**.

We have **two requirements** for **compiler security**

Isolate the men
of the program fro
performance over
pointer passing

Compiler correctness is a more standard verification criterion.

Goal: avoid repeating **years-worth of proof effort**.

Want a proof technique that allows us to **reuse a whole-program compiler correctness theorem**.

We have **two requirements** for **compiler security**

Isolate the memory of the different parts
of the pr
perform
pointer

Hardware capabilities

Want a proof technique that allows us to **reuse a whole-program compiler correctness theorem**.

We have **two requirements** for **compiler security**

Isolate the memory of the different parts of the pr

performa

pointer

Hardware capabilities

Want a proof technique that allows us to

reuse a

correct

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

Prior work on Compiler security

✗ Reproved correctness implicitly as part of the security proof.

Prior work on Compiler security

✗ Reproved correctness implicitly as part of the security proof.

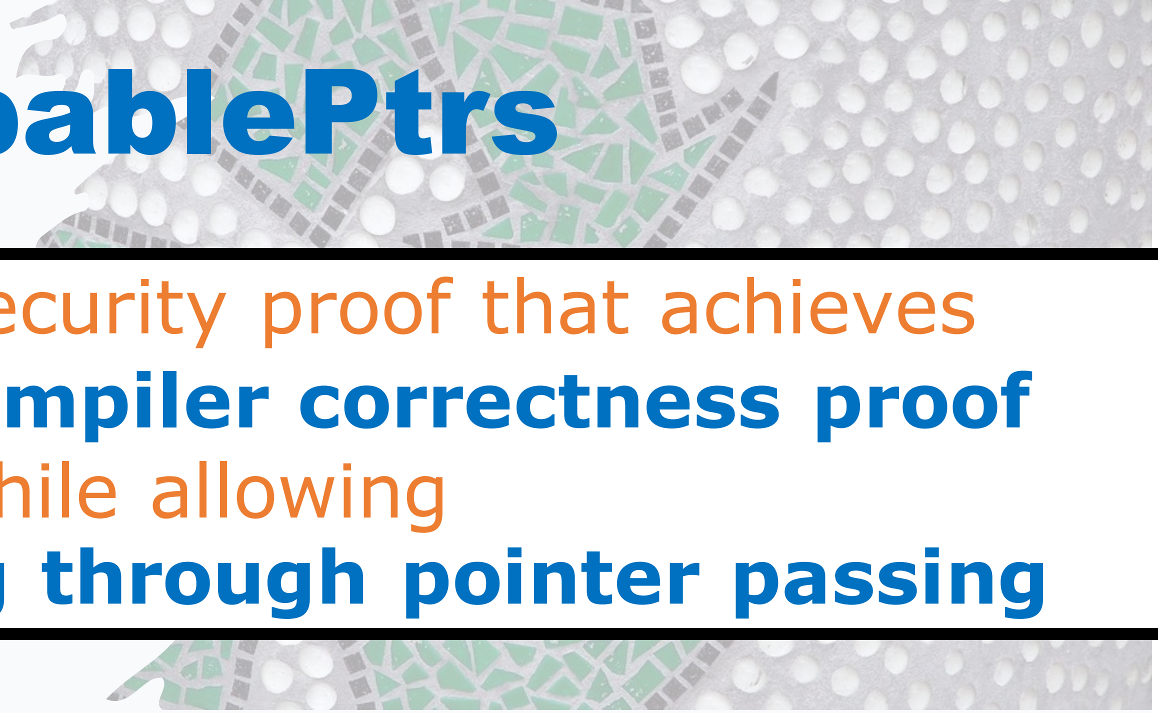
Prior work on Compiler security

✗ Achieved isolation by preventing memory sharing altogether.

CapablePtrs

✗ Reproved correctness

First compiler security proof that achieves
reuse of the compiler correctness proof
while allowing
memory sharing through pointer passing



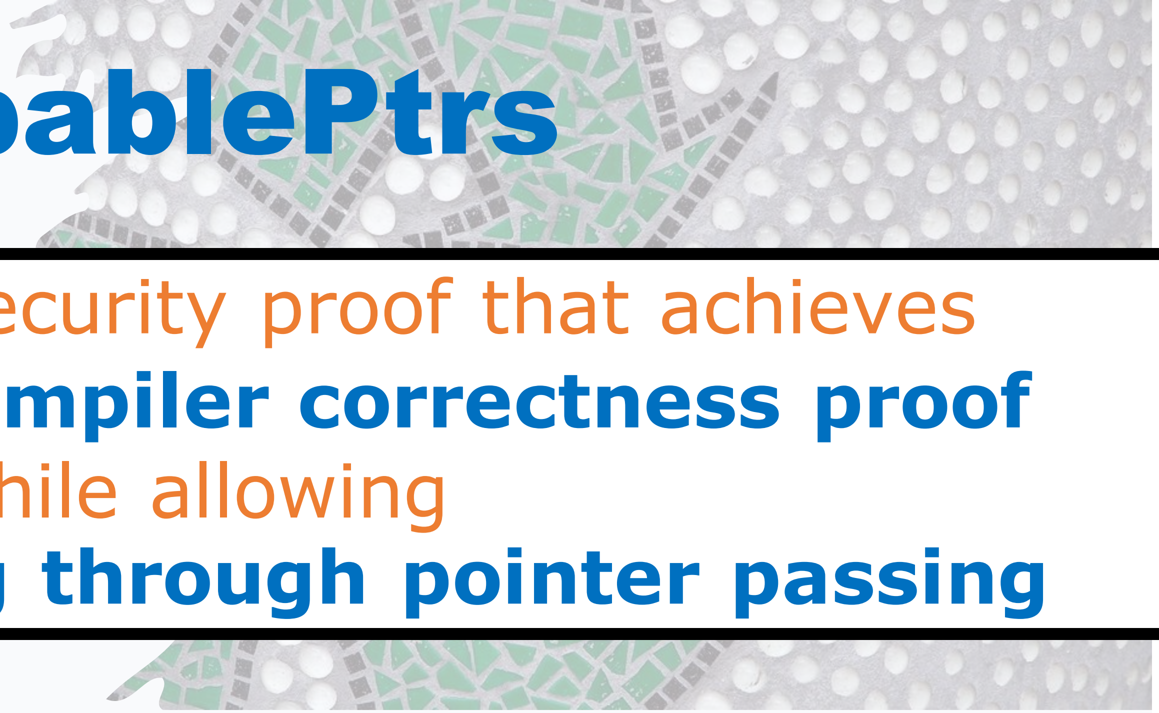
Prior work on Compiler security

✗ Achieved isolation by preventing memory sharing altogether.

CapablePtrs

❌ Reproved correctness

First compiler security proof that achieves
reuse of the compiler correctness proof
while allowing
memory sharing through pointer passing



Prior work on

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

security

❌ Achieved isolation by preventing memory sharing altogether.

CapablePtrs

✗ Reproved correctness

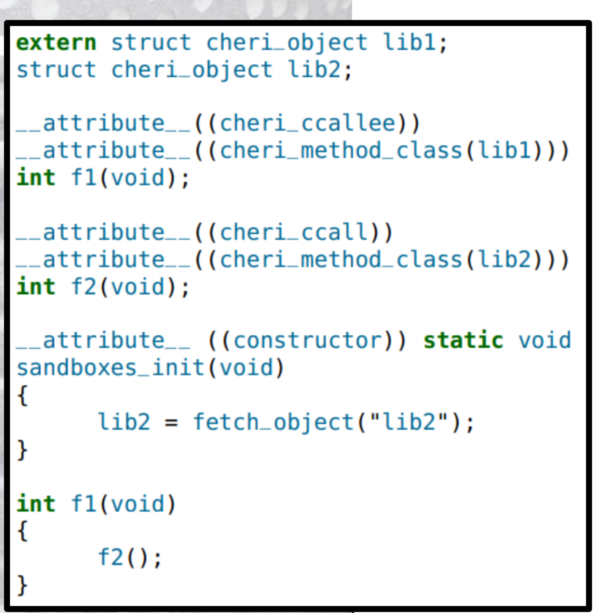
First compiler security proof that achieves **reuse of the compiler correctness proof** while allowing **memory sharing through pointer passing**

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

security

✗ Achieved isolation by preventing memory sharing altogether.

C-to-C source transform that adds CHERI annotations automatically



```
extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
}

int f1(void)
{
      f2();
}
```

CapablePtrs

Reproved correctness

First compiler security proof that achieves
reuse of the compiler correctness proof
while allowing
memory sharing through pointer passing

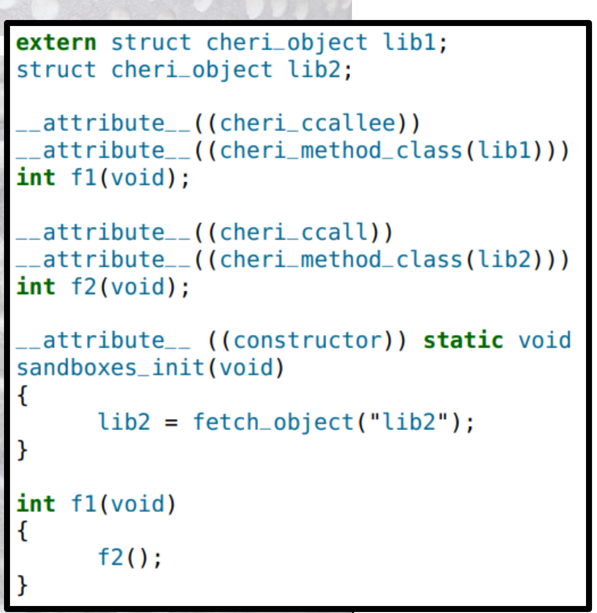
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

security

Achieved isolation by preventing memory sharing altogether.

libpng
LibYAML
zlib
GNU-barcode

C-to-C source transform that adds CHERI annotations automatically



```
extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
}

int f1(void)
{
      f2();
}
```

CapablePtrs

Reproved correctness

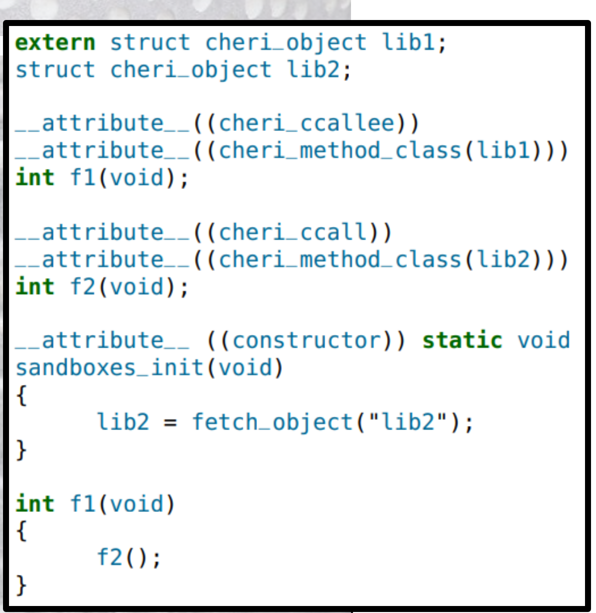
First compiler security proof that achieves
reuse of the compiler correctness proof
while allowing
memory sharing through pointer passing

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

security

libpng	0.15%
LibYAML	0.89%
zlib	1.15%
GNU-barcode	3.5%

C-to-C source transform that adds CHERI annotations automatically



```
extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
}

int f1(void)
{
      f2();
}
```

Achieved isolation by preventing memory sharing altogether.

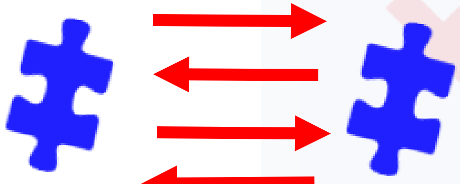
CapablePtrs

Reproved correctness

First compiler security proof that achieves **reuse of the compiler correctness proof** while allowing **memory sharing through pointer passing**

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

security



Achieved isolation by preventing memory sharing altogether.

libpng	0.15%
LibYAML	0.89%
zlib	1.15%
GNU-barcode	3.5%

C-to-C source transform that adds CHERI annotations automatically

```
extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
}

int f1(void)
{
      f2();
}
```

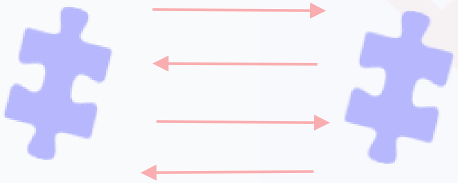
CapablePtrs

Reproved correctness

First compiler security proof that achieves
reuse of the compiler correctness proof
while allowing
memory sharing through pointer passing

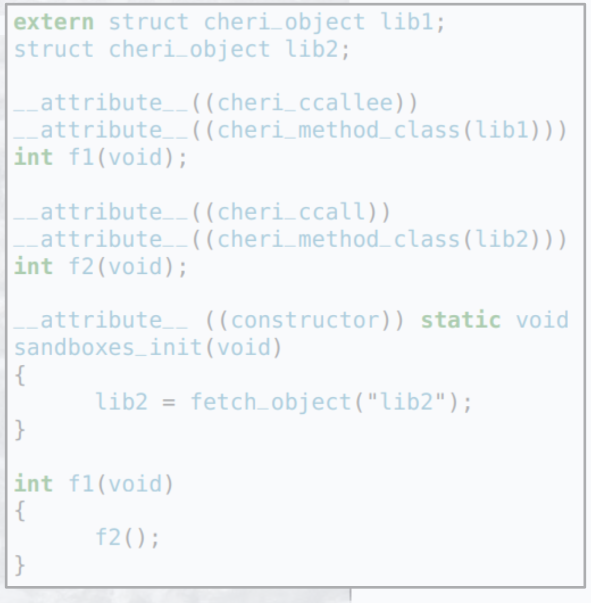
Novel proof technique
(called TrICL "/ˈtrɪk(ə)l/")

security



libpng	0.15%
LibYAML	0.89%
zlib	1.15%
GNU-barcode	3.5%

C-to-C source transform that adds CHERI annotations automatically



```
extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
}

int f1(void)
{
      f2();
}
```

Achieved isolation by preventing memory sharing altogether.

We have **two requirements** for **compiler security**

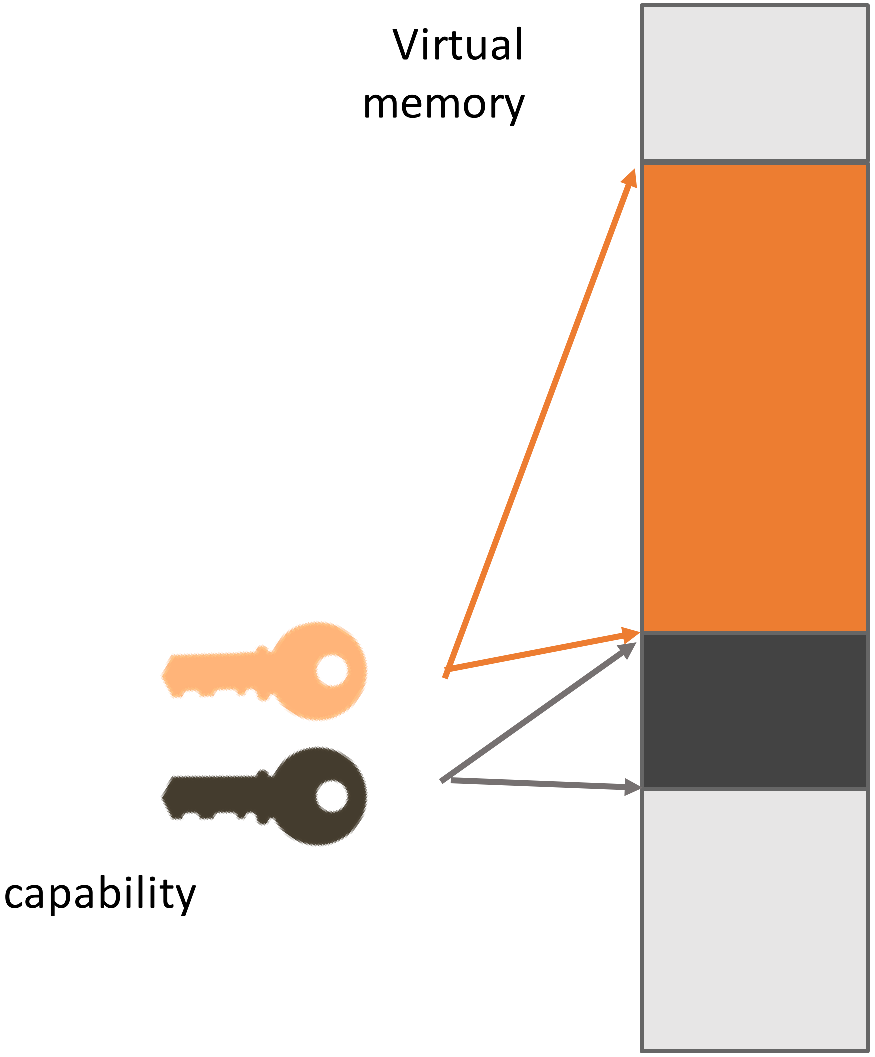
Isolate the memory of the different parts of the pr
perform
pointer

Hardware capabilities

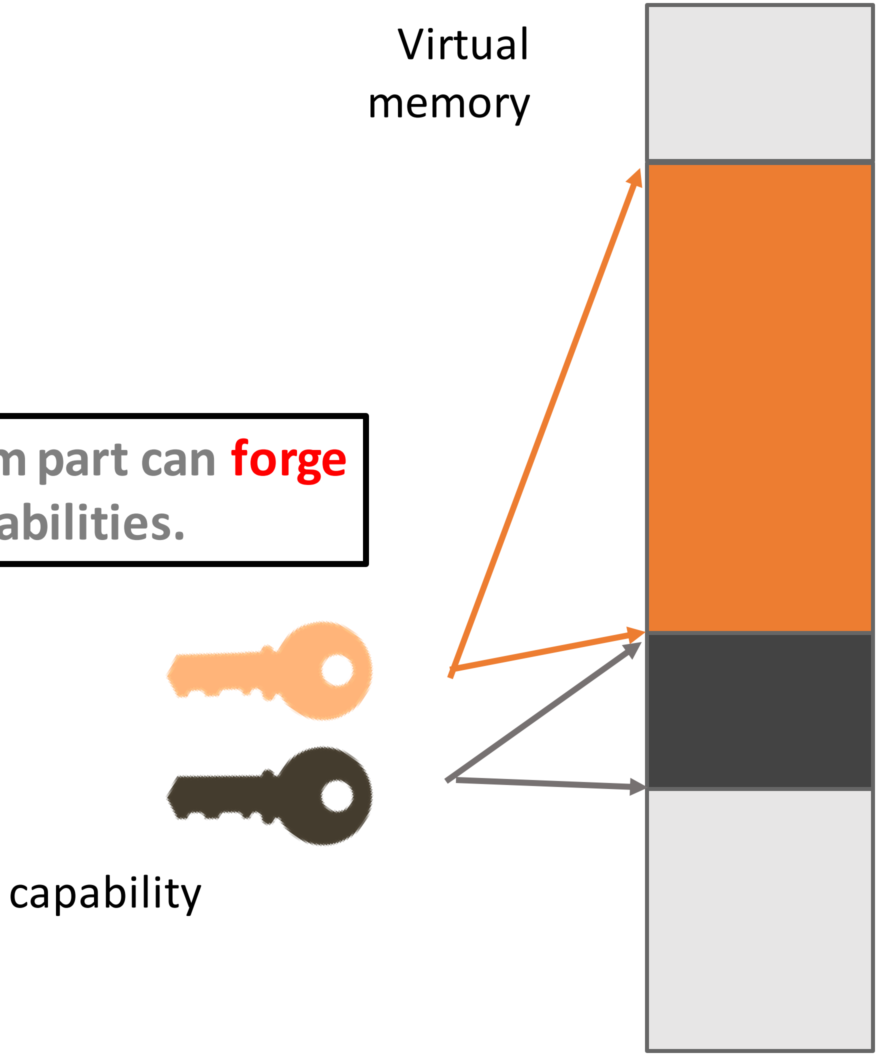
Want a proof technique that allows us to
reuse a
correct
re-prove

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

Hardware capabilities



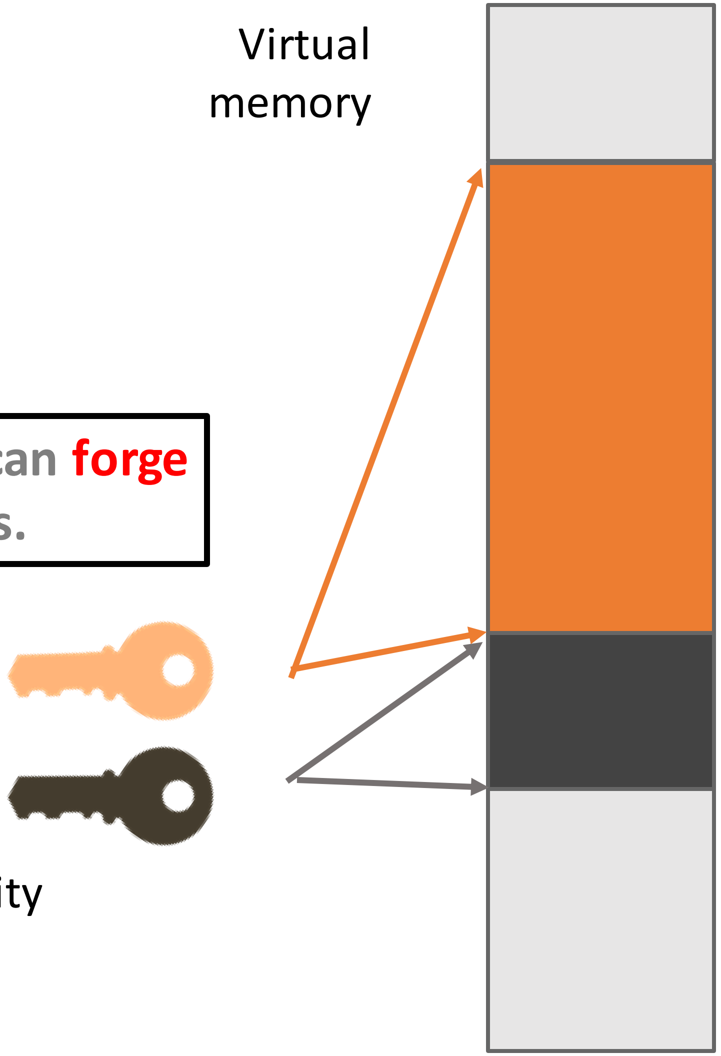
Hardware capabilities



Every memory access instruction expects a **capability as an argument.**

No program part can **forge** capabilities.

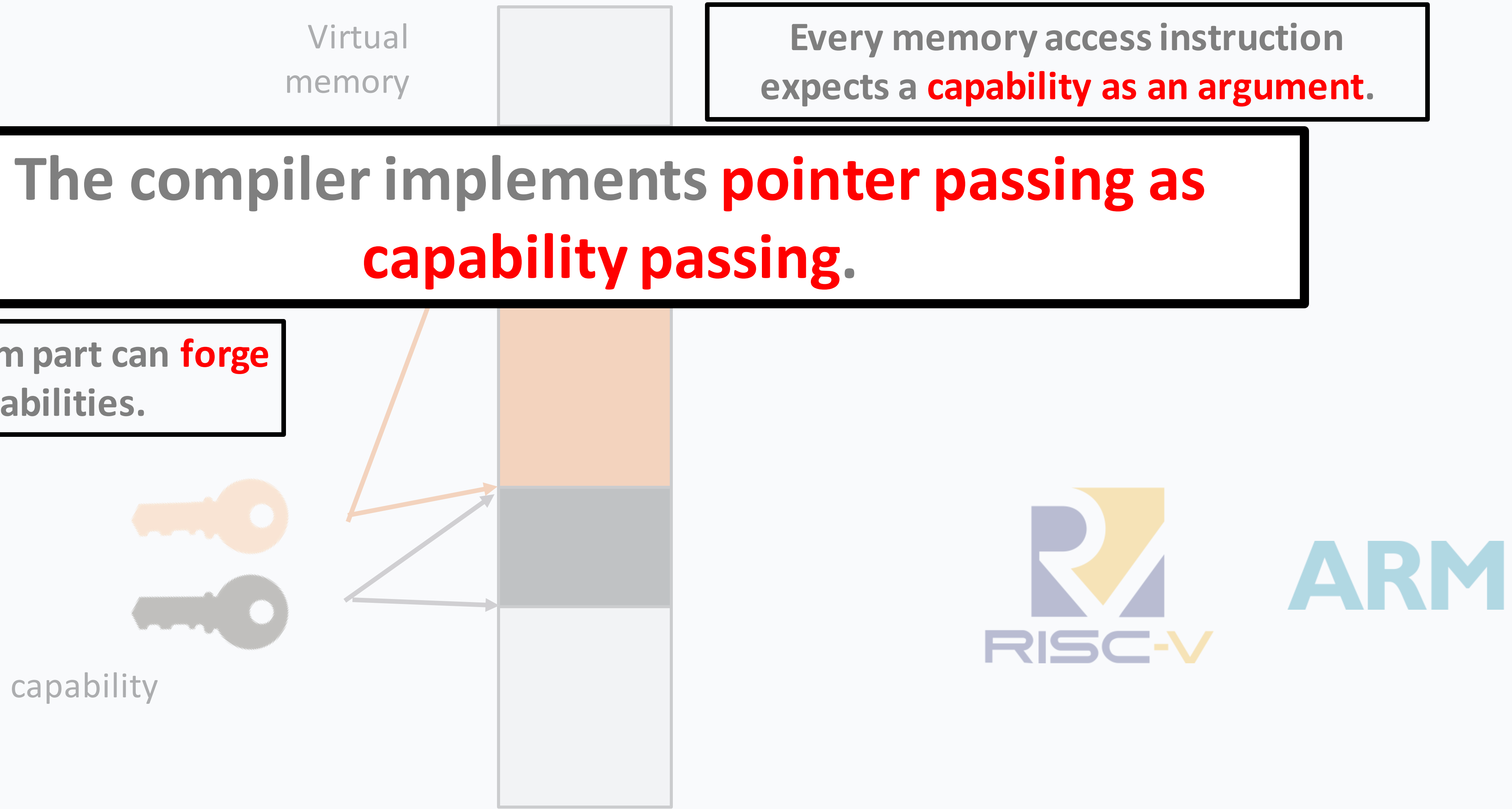
Hardware capabilities



Every memory access instruction expects a **capability as an argument.**

No program part can **forge** capabilities.

Hardware capabilities



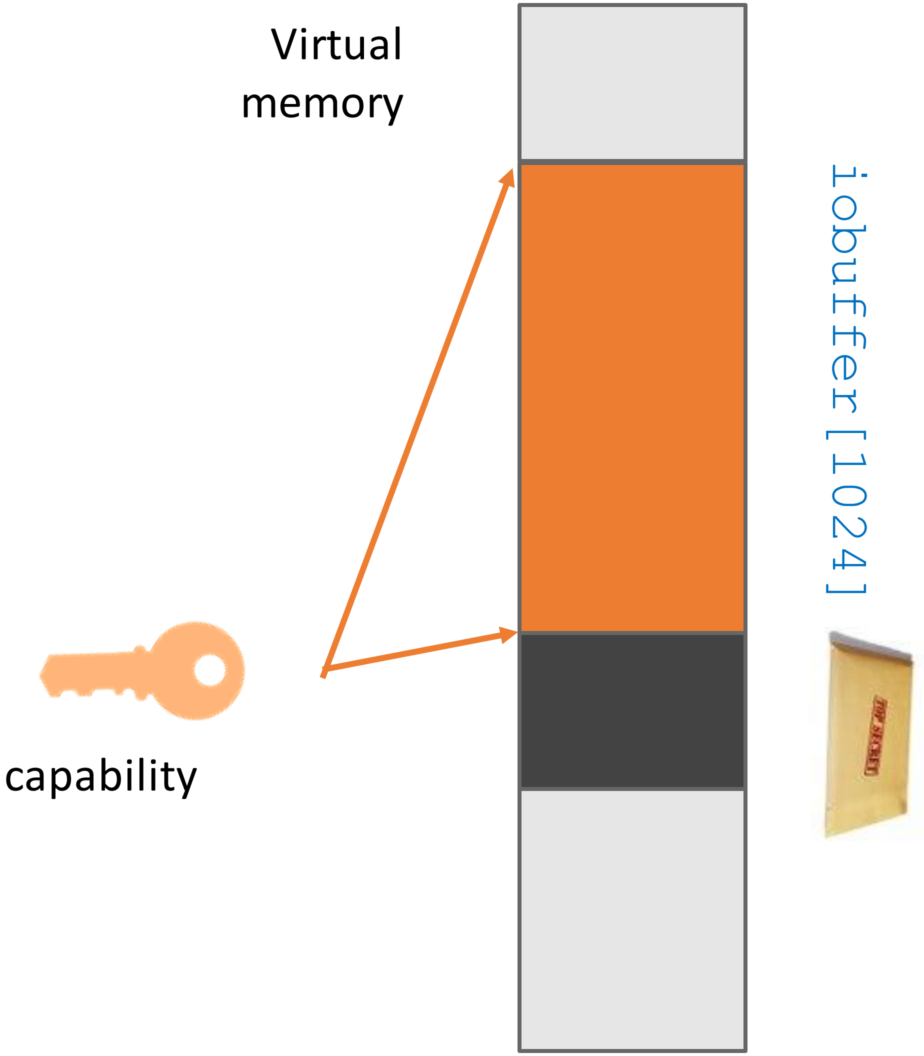
Every memory access instruction expects a **capability as an argument.**

The compiler implements **pointer passing as capability passing.**

No program part can **forge** capabilities.

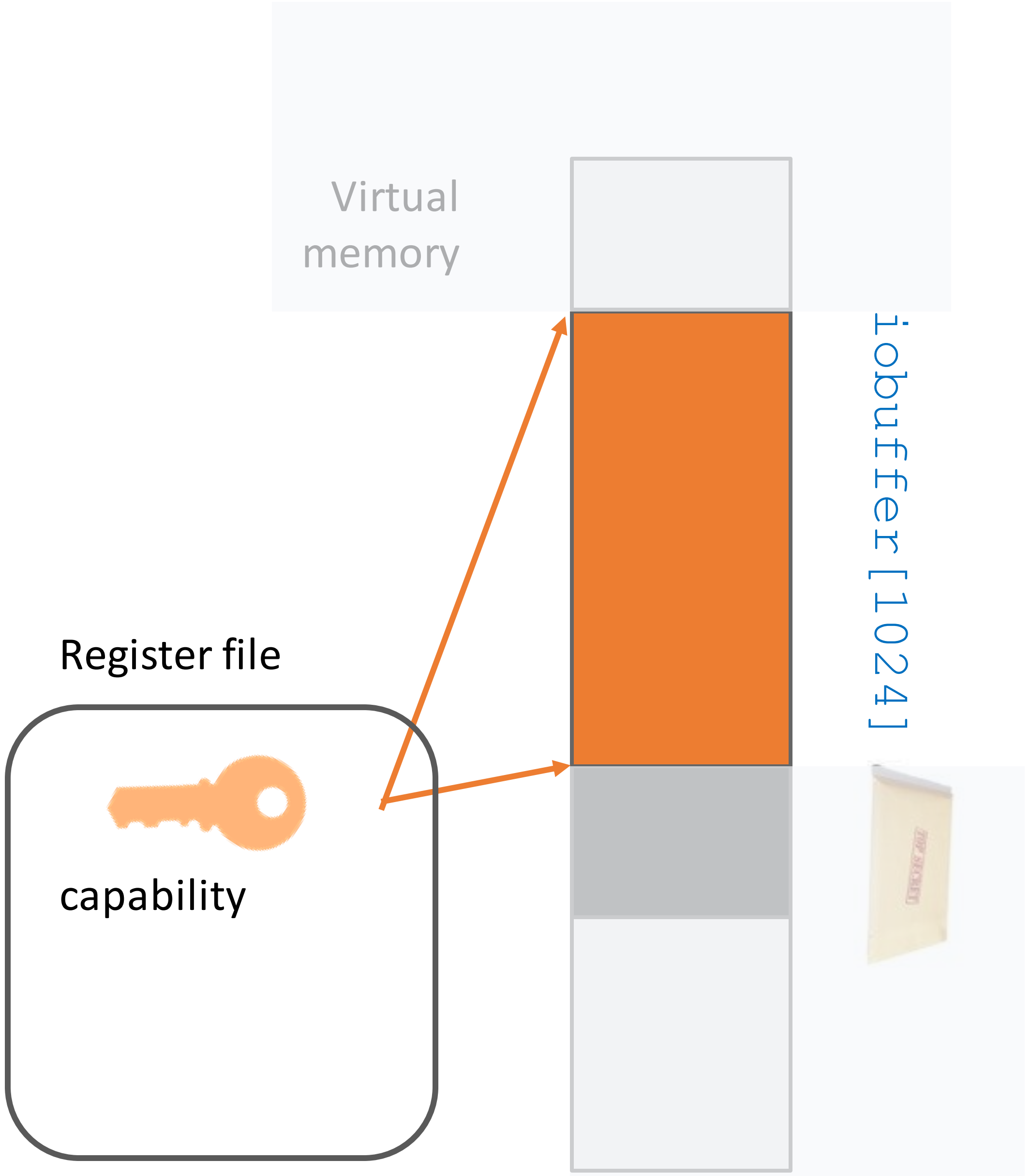
Hardware capabilities

could be **buggy** or **malicious**



Hardware capabilities

could be **buggy** or **malicious**

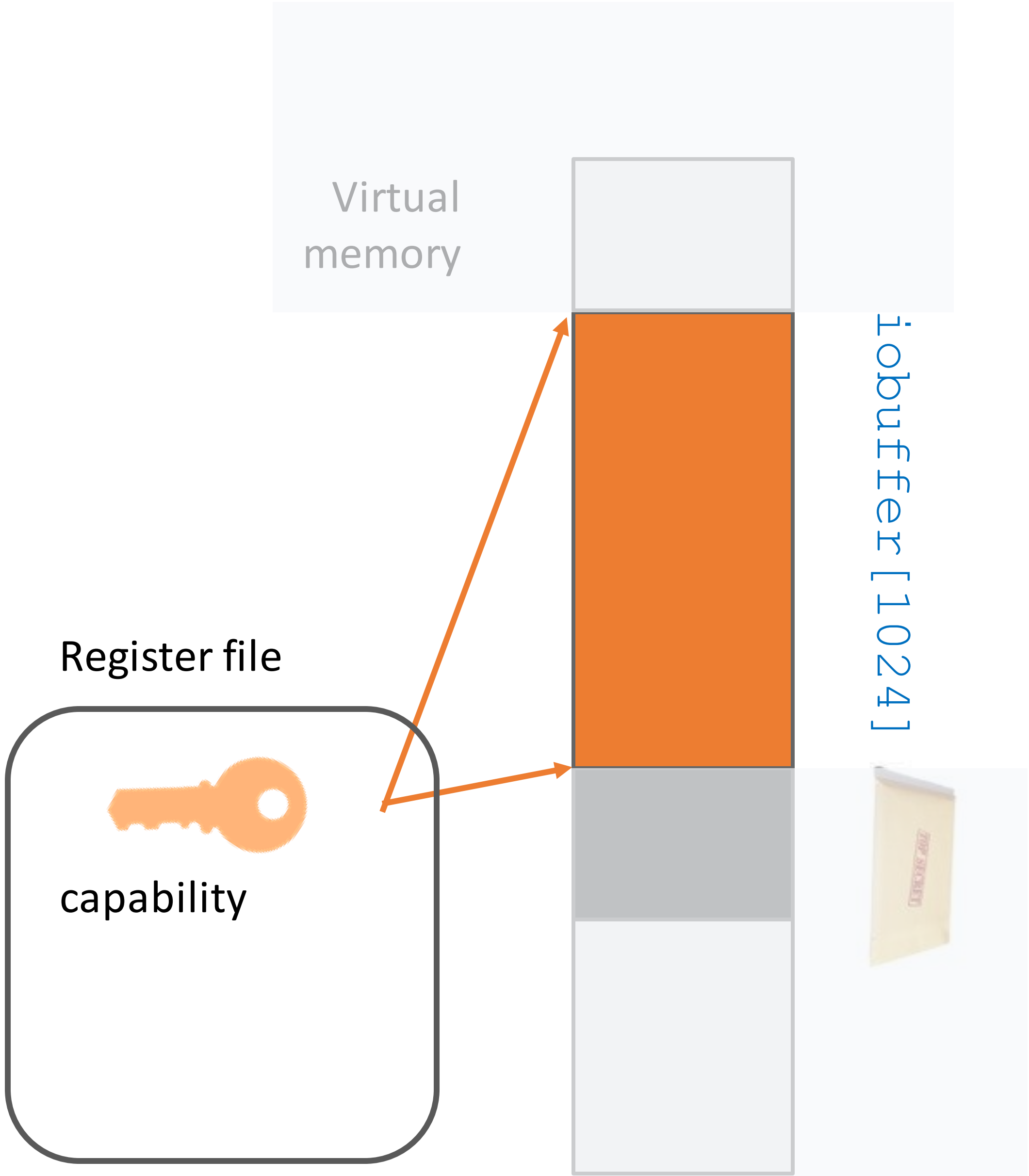


Hardware capabilities

could be **buggy** or **malicious**

malicious

```
char *secret_ptr =
        (char*)4210756;

leak(*secret_ptr);
```



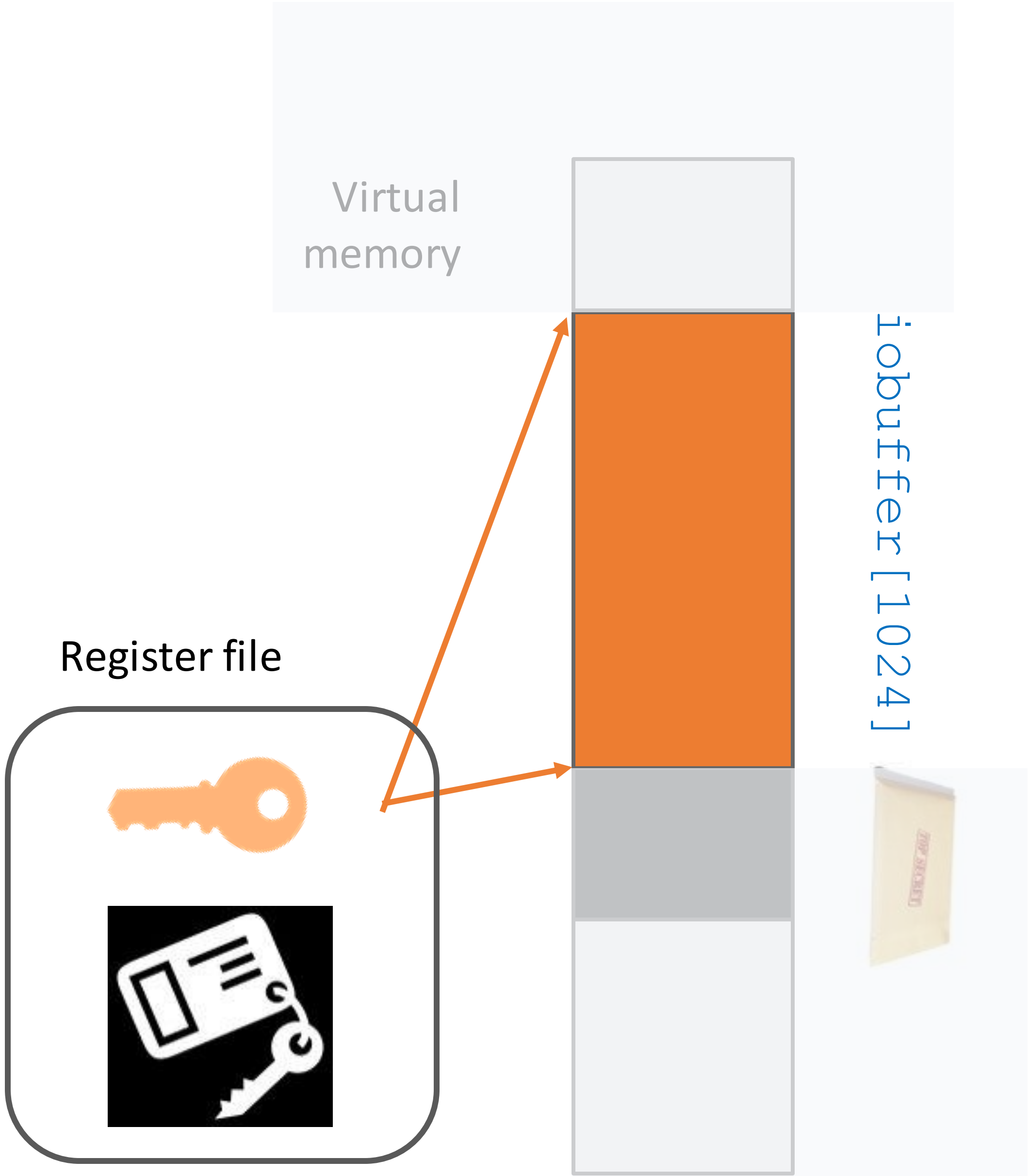
Hardware capabilities

could be **buggy** or **malicious**

malicious

```
char *secret_ptr =
        (char*)4210756;
```

```
leak(*secret_ptr);
```



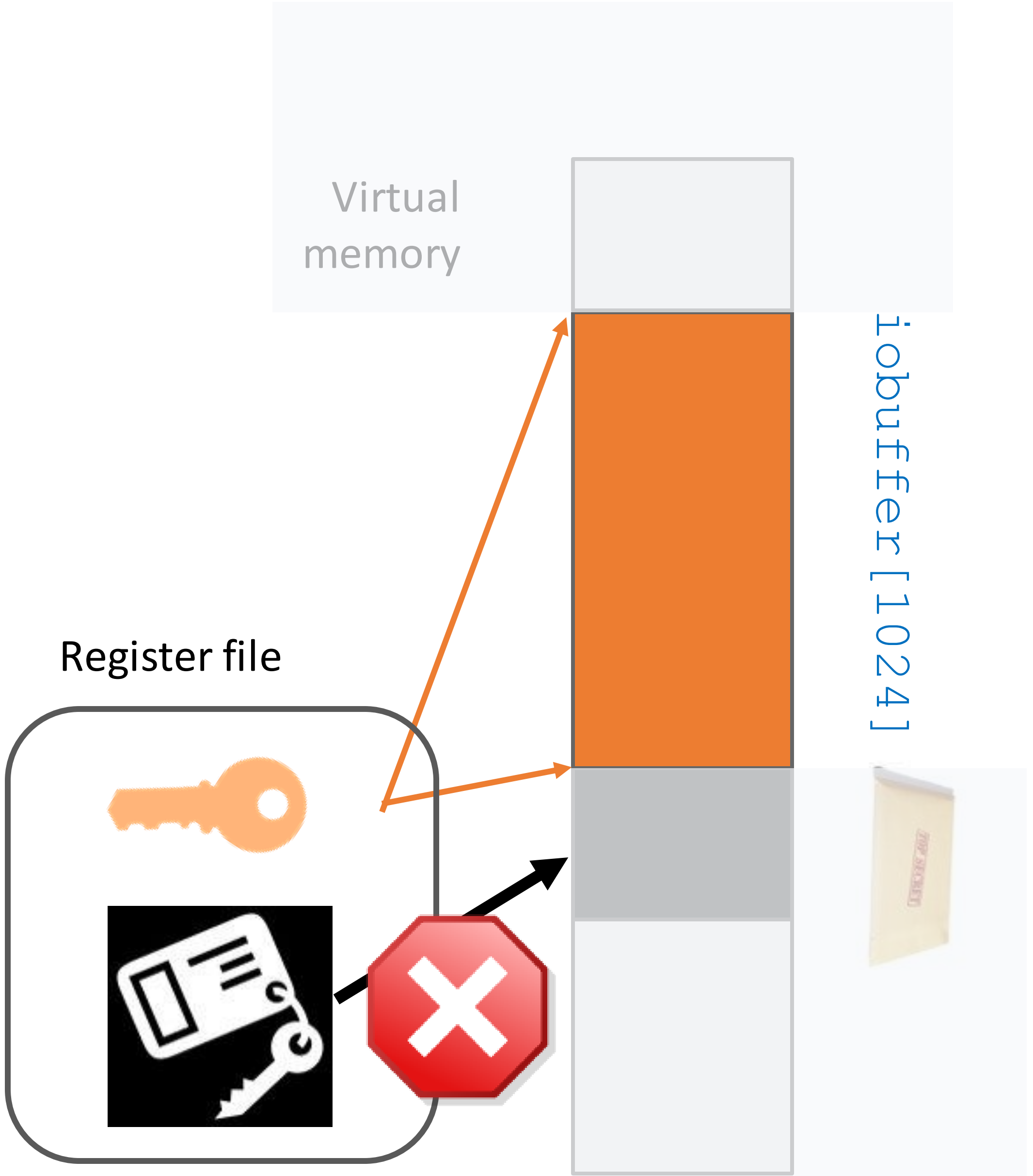
Hardware capabilities

could be **buggy** or **malicious**

malicious

```
char *secret_ptr =
        (char*)4210756;
```

```
leak(*secret_ptr);
```



Hardwa

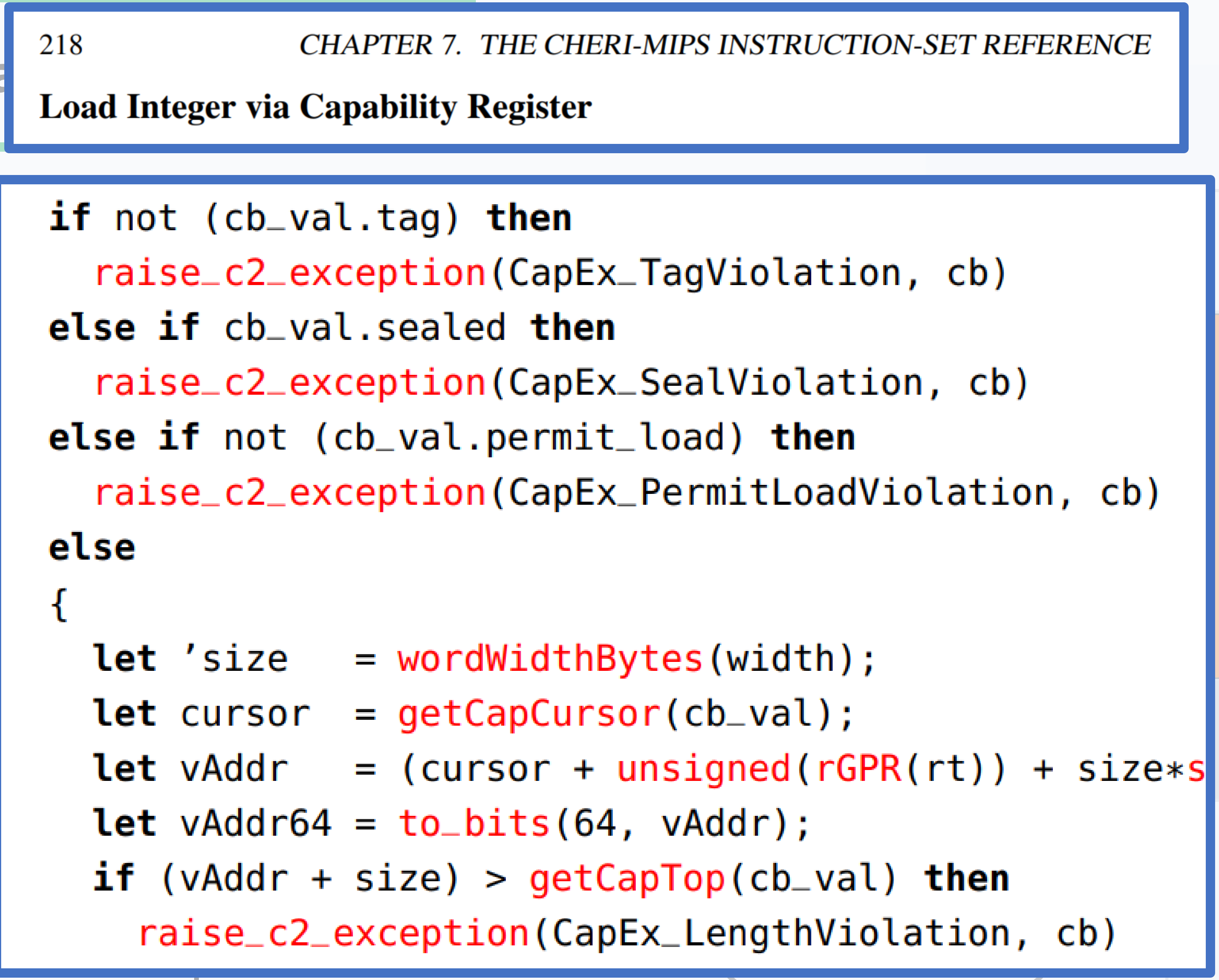
could be

malicious

```
char *secret_p
      (char*)4
```

```
leak(*secret_p
```

iobuffer[1024]

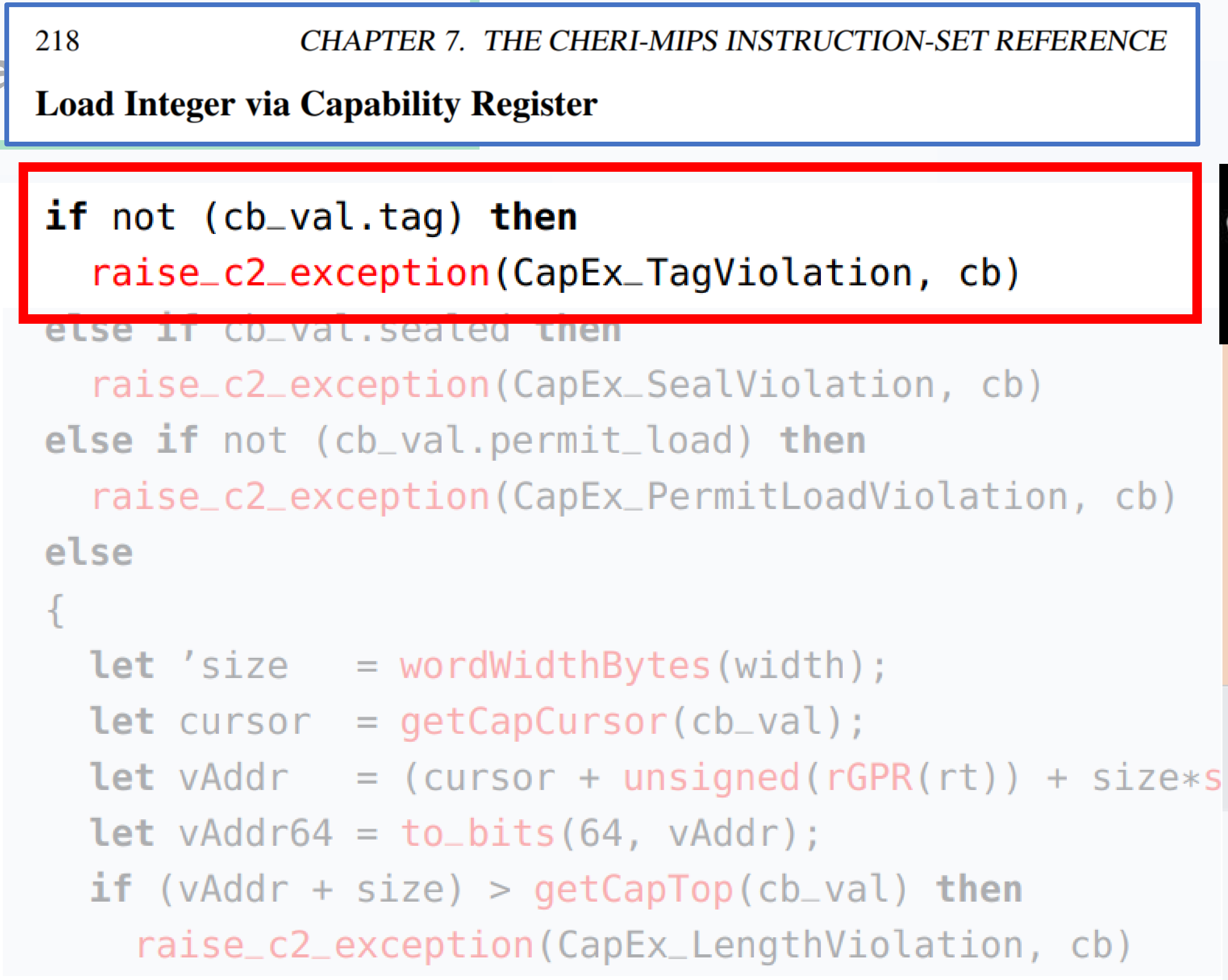


218 CHAPTER 7. THE CHERI-MIPS INSTRUCTION-SET REFERENCE

Load Integer via Capability Register

```
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let 'size   = wordWidthBytes(width);
  let cursor  = getCapCursor(cb_val);
  let vAddr   = (cursor + unsigned(rGPR(rt)) + size*s
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
```

Hardwa



218 *CHAPTER 7. THE CHERI-MIPS INSTRUCTION-SET REFERENCE*

Load Integer via Capability Register

```
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
  raise_c2_exception(CapEx_PermitLoadViolation, cb)
else
{
  let 'size    = wordWidthBytes(width);
  let cursor   = getCapCursor(cb_val);
  let vAddr    = (cursor + unsigned(rGPR(rt)) + size*s
  let vAddr64 = to_bits(64, vAddr);
  if (vAddr + size) > getCapTop(cb_val) then
    raise_c2_exception(CapEx_LengthViolation, cb)
```

could be

malicious

```
char *secret_p
      (char*)4
```

```
leak(*secret_p
```

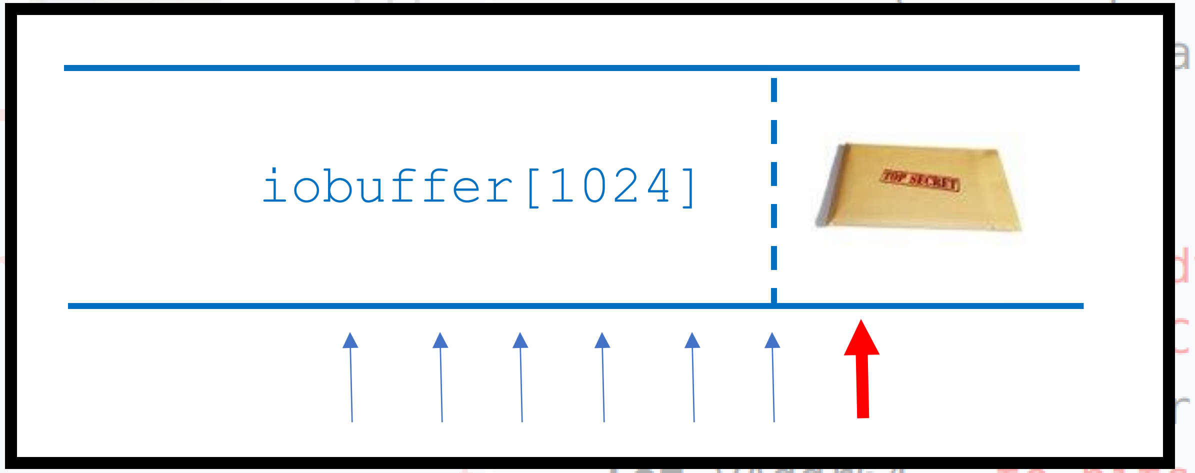
iobuffer[1024]

Hardwa

218 CHAPTER 7. THE CHERI-MIPS INSTRUCTION-SET REFERENCE

Load Integer via Capability Register

```
if not (cb_val.tag) then
  raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
  raise_c2_exception(CapEx_SealViolation, cb)
else if not (cb_val.permit_load) then
                   apEx_PermitLoadViolation, cb)

                   dthBytes(width);
                   Cursor(cb_val);
                   r + unsigned(rGPR(rt)) + size*s
  let vAddr64 = to_bits(64, vAddr);
```



(char*)4

leak(*secret_p

```
if (vAddr + size) > getCapTop(cb_val) then
  raise_c2_exception(CapEx_LengthViolation, cb)
```

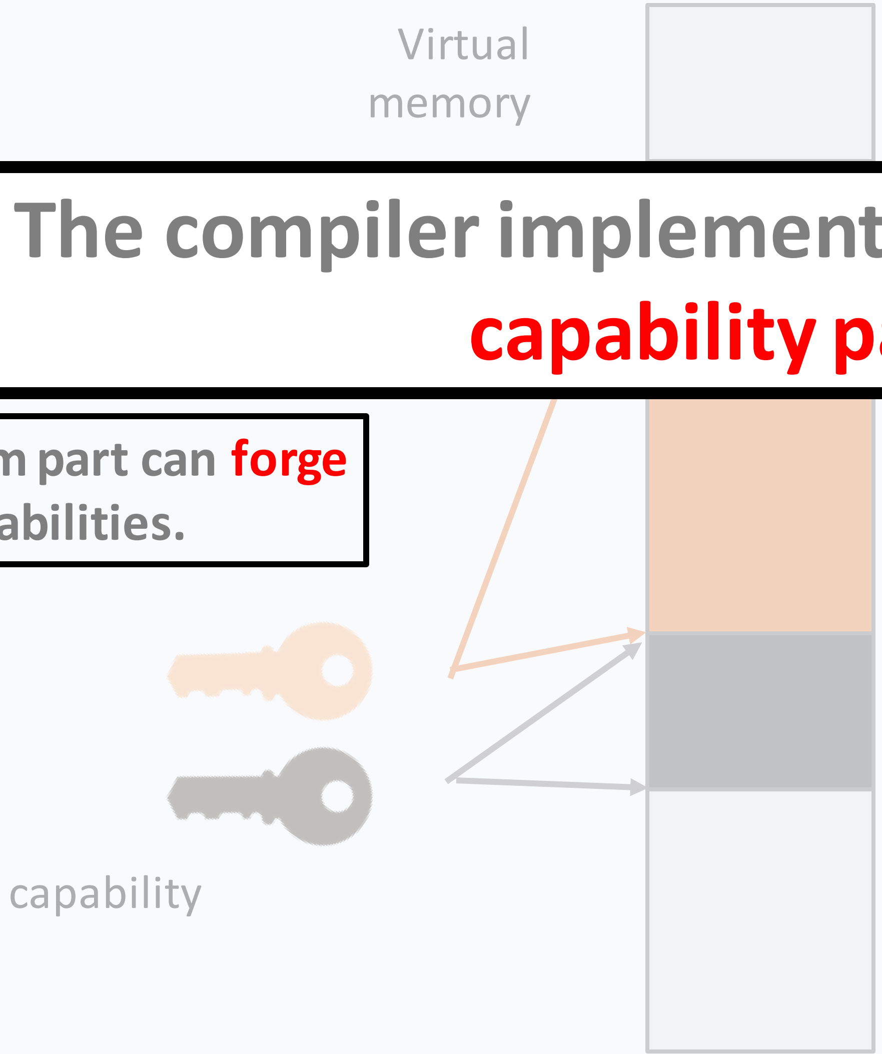
iobuffer[1024]

Hardware capabilities

Every memory access instruction expects a **capability as an argument.**

The compiler implements pointer passing as capability passing.

No program part can **forge** capabilities.



We have **two requirements** for **compiler security**

Isolate the memory of the different parts of the pr
performa
pointer

Hardware capabilities

Want a proof technique that allows us to
reuse a
correct

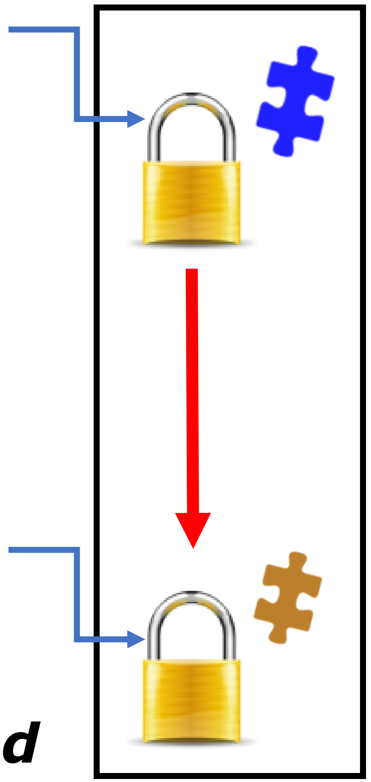
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

The definition of **compiler security** that we use:

Compiler Full Abstraction

The definition of **compiler security** that we use:

Compiler Full Abstraction



security property of the source program part

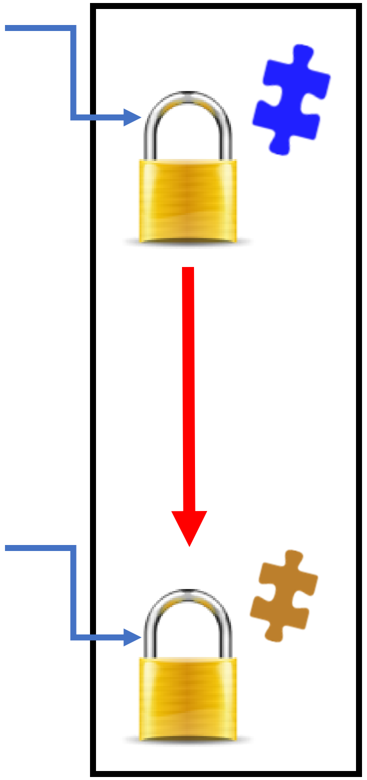
security property of the ***compiled*** program part

The definition of **compiler security** that we use:

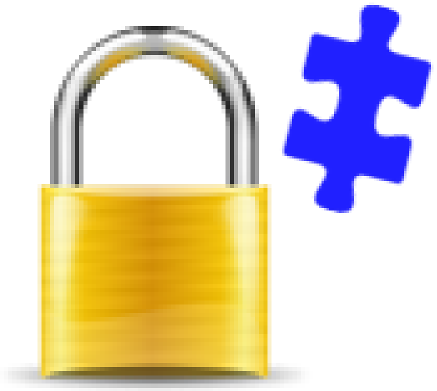
Compiler Full Abstraction

Confidentiality of the secrets of the source program part

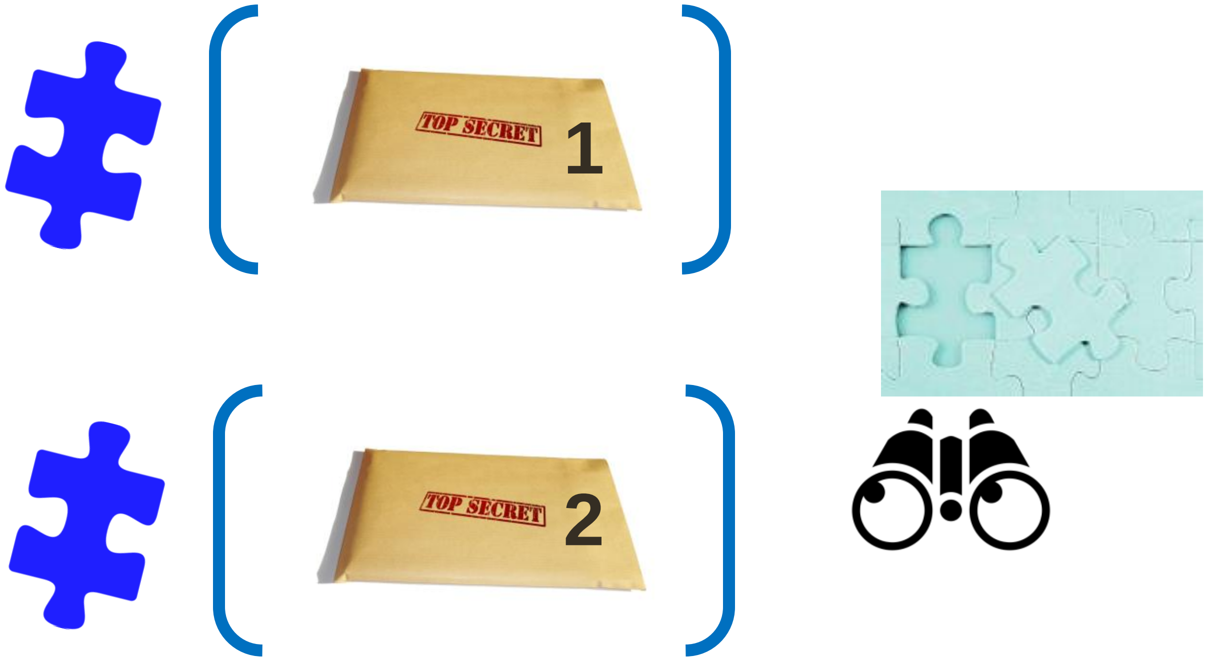
Confidentiality of the secrets of the ***compiled*** program part



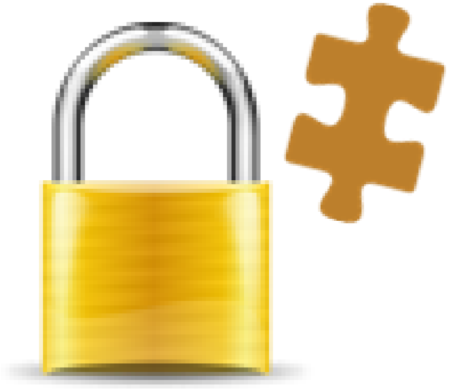
A partial program is **secure**



when **NO library** can distinguish two runs (with **two different secrets**) from each other.

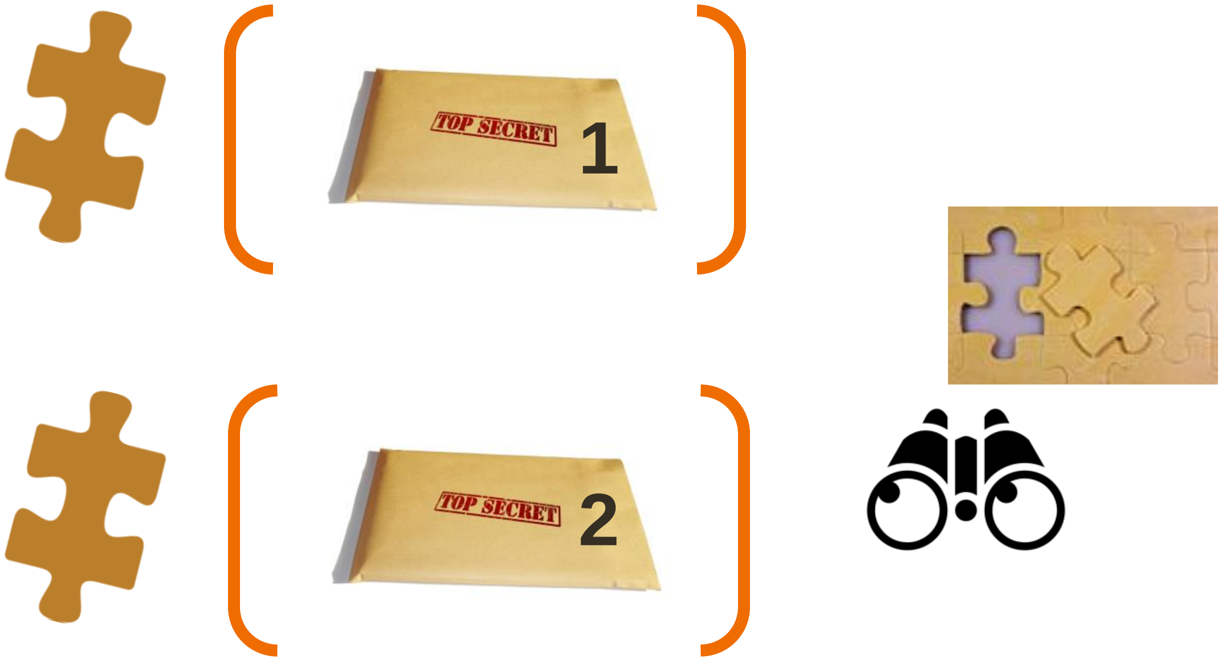


A partial program is **secure**



The same definition for the **target language** too

when **NO library** can distinguish two runs (with **two different secrets**) from each other.

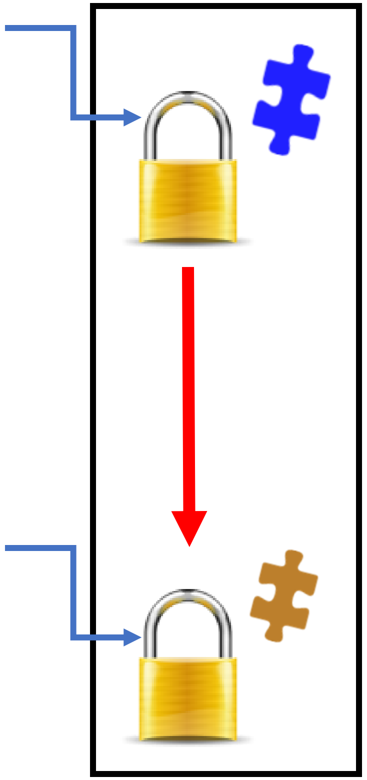


The definition of **compiler security** that we use:

Compiler Full Abstraction

Confidentiality of the secrets of the source program part

Confidentiality of the secrets of the ***compiled*** program part



We have **two requirements** for **compiler security**

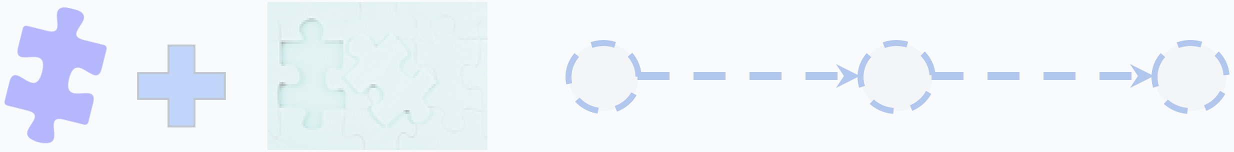
Isolate the memory of the different parts of the pr
performa
pointer

Hardware capabilities

Want a proof technique that allows us to
reuse a
correct

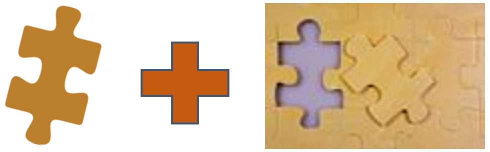
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

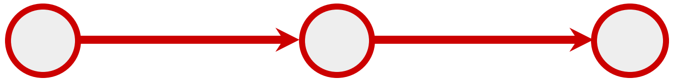
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



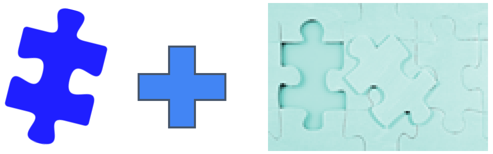
compiled program part

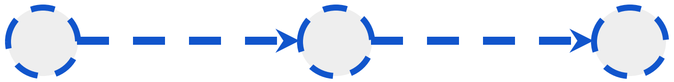
untrusted third-party library





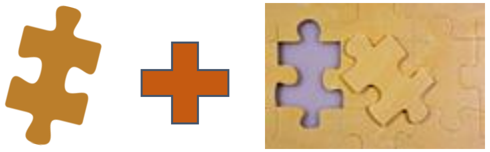
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

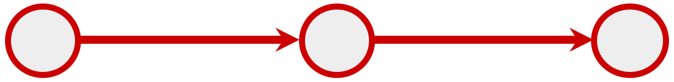




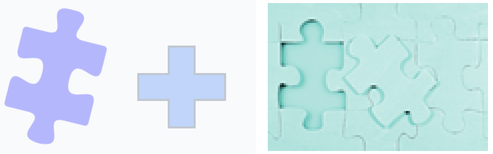
source program part ∃ mimicking source library

compiled program part **untrusted** third-party library





Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



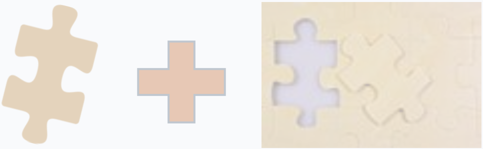
source program part

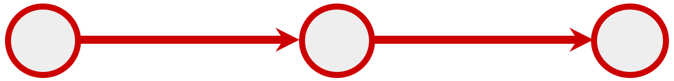
∃ mimicking source library

Trace-directed Back-translation

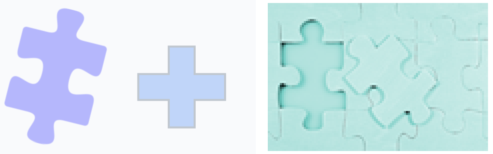
compiled program part

untrusted third-party library





Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

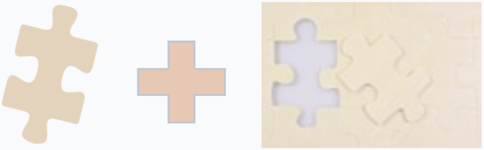


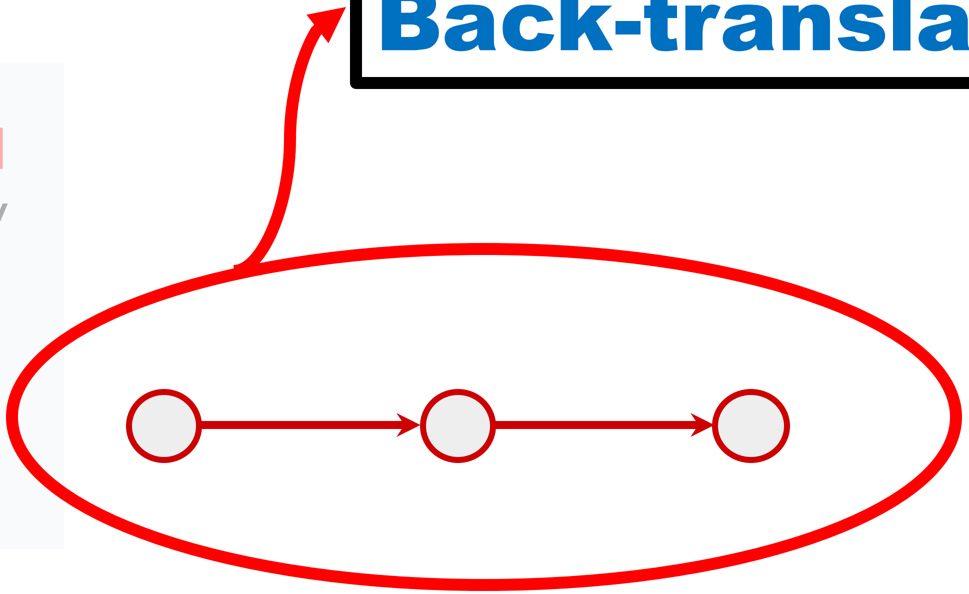
source program part

∃ mimicking source library

compiled program part

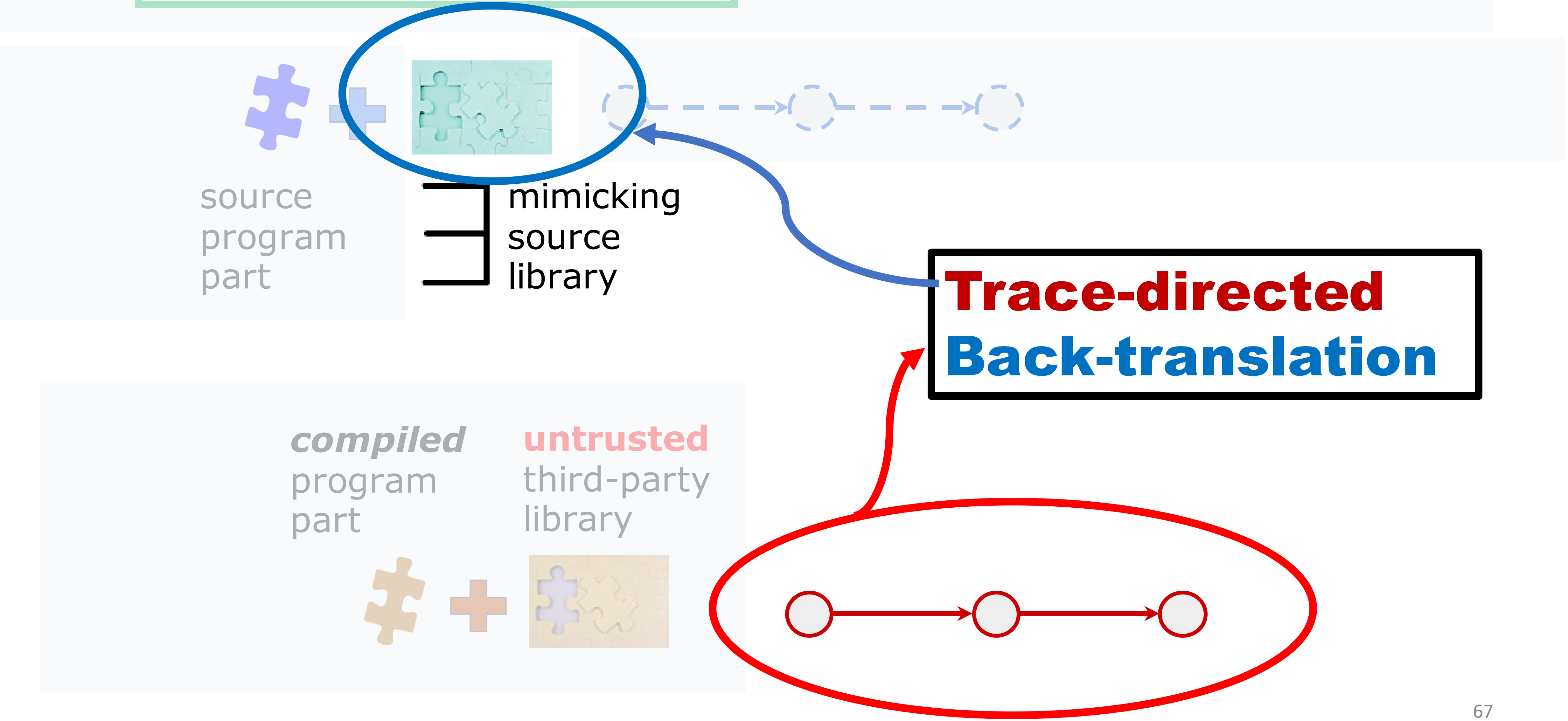
untrusted third-party library





Trace-directed Back-translation

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



source program part

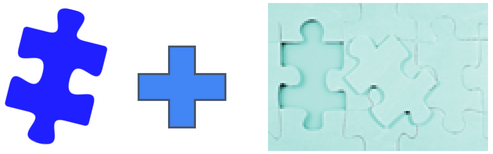
mimicking source library

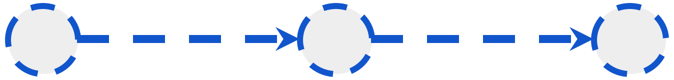
Trace-directed Back-translation

compiled program part

untrusted third-party library

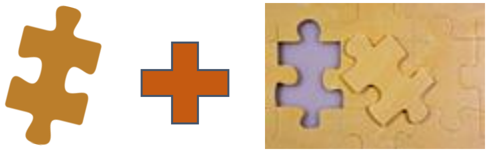
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

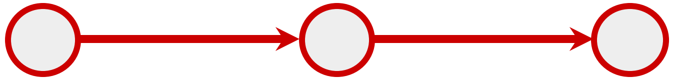




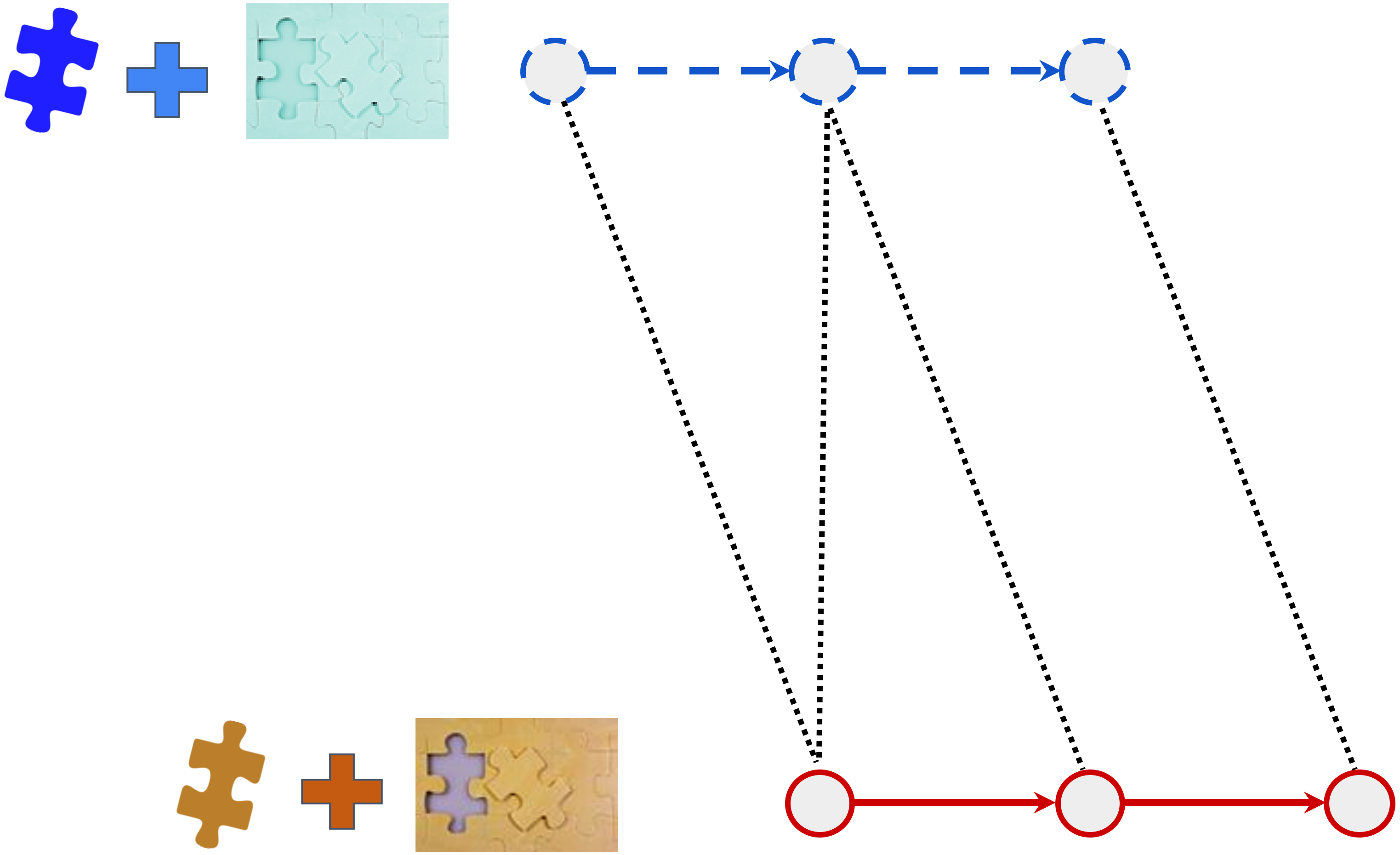
source program part ∃ mimicking source library

compiled program part **untrusted** third-party library

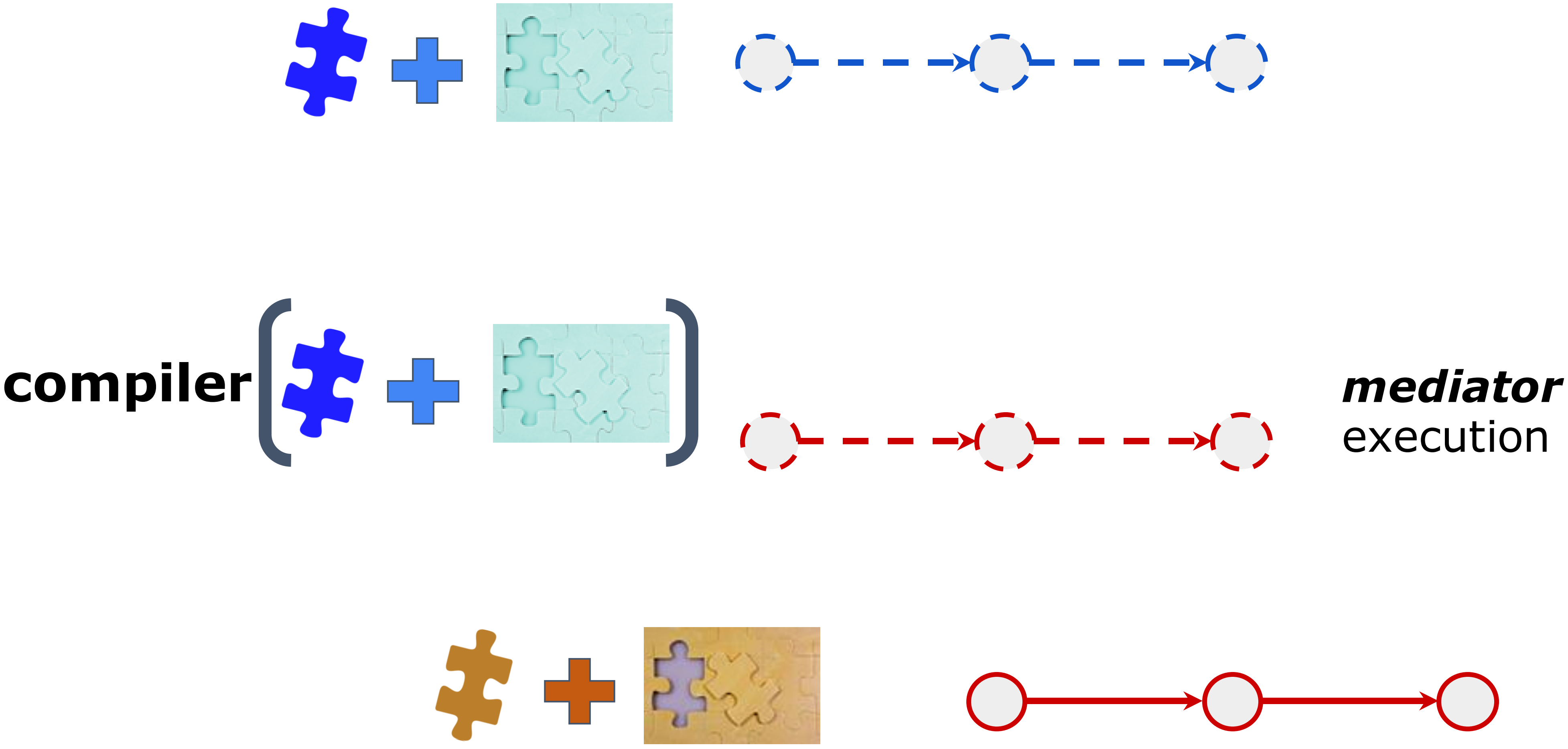




Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



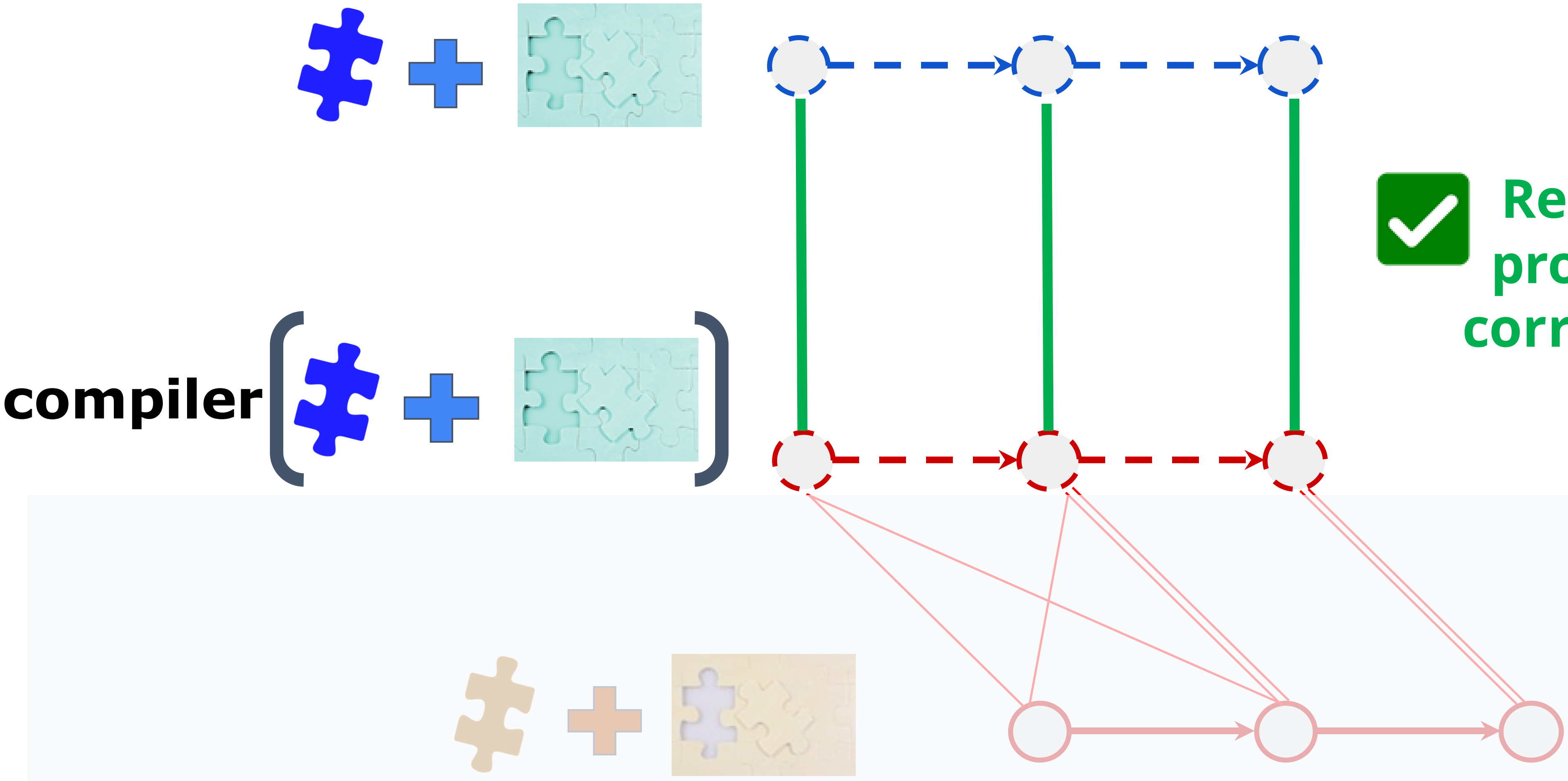
Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



compiler

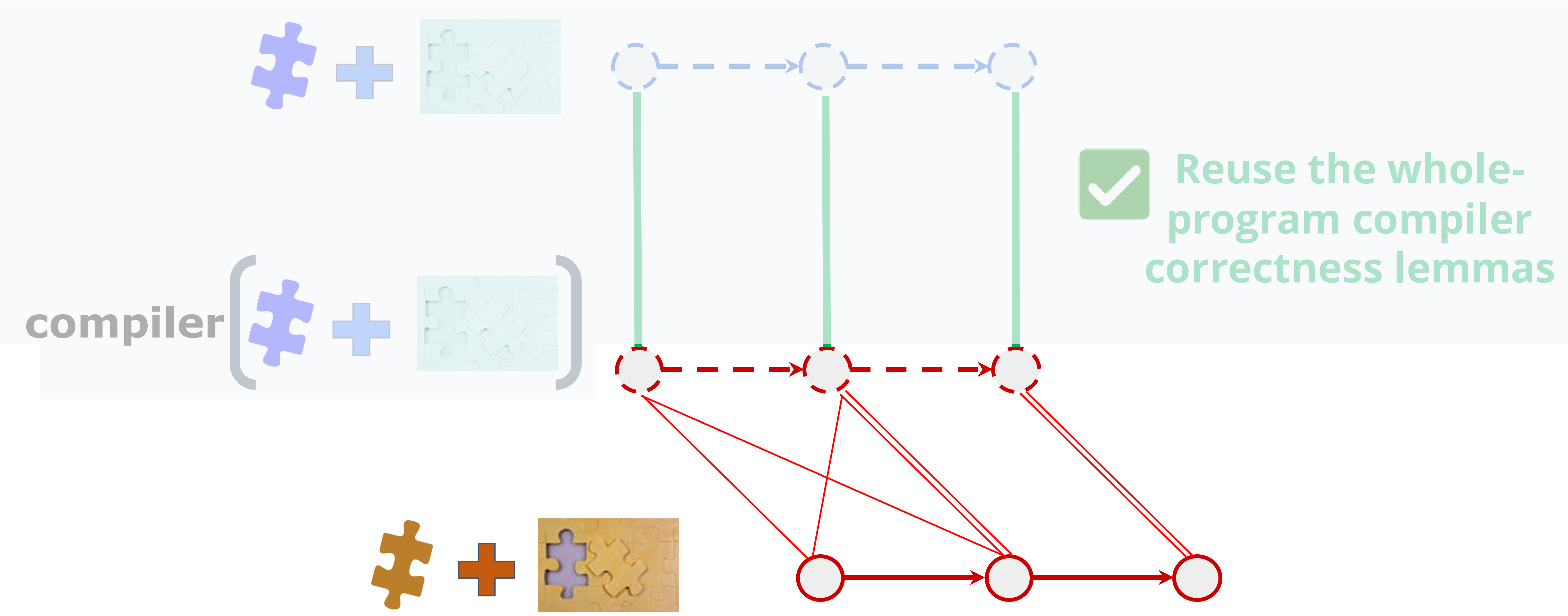
mediator
execution

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



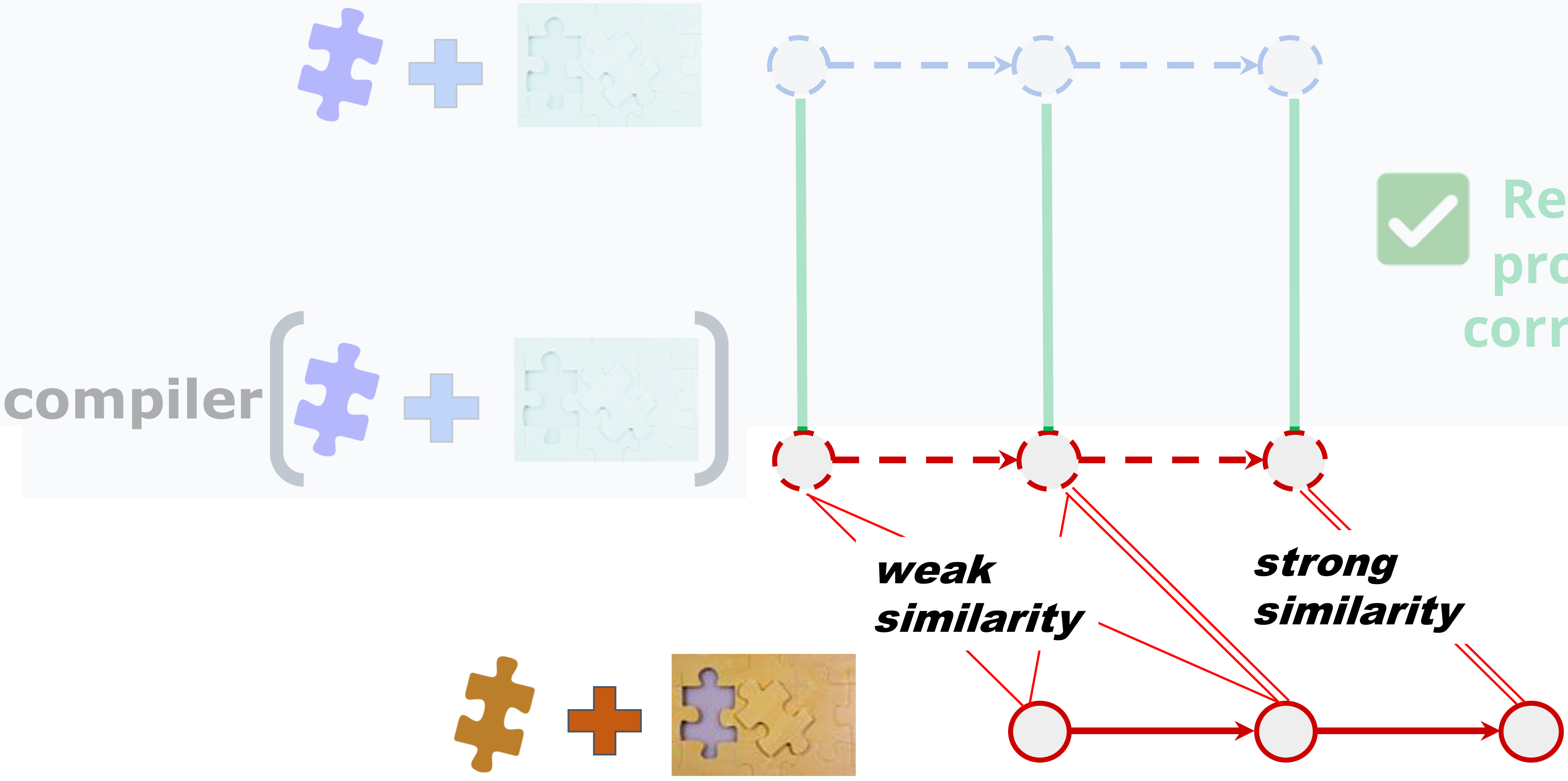
✓ Reuse the whole-program compiler correctness lemmas

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



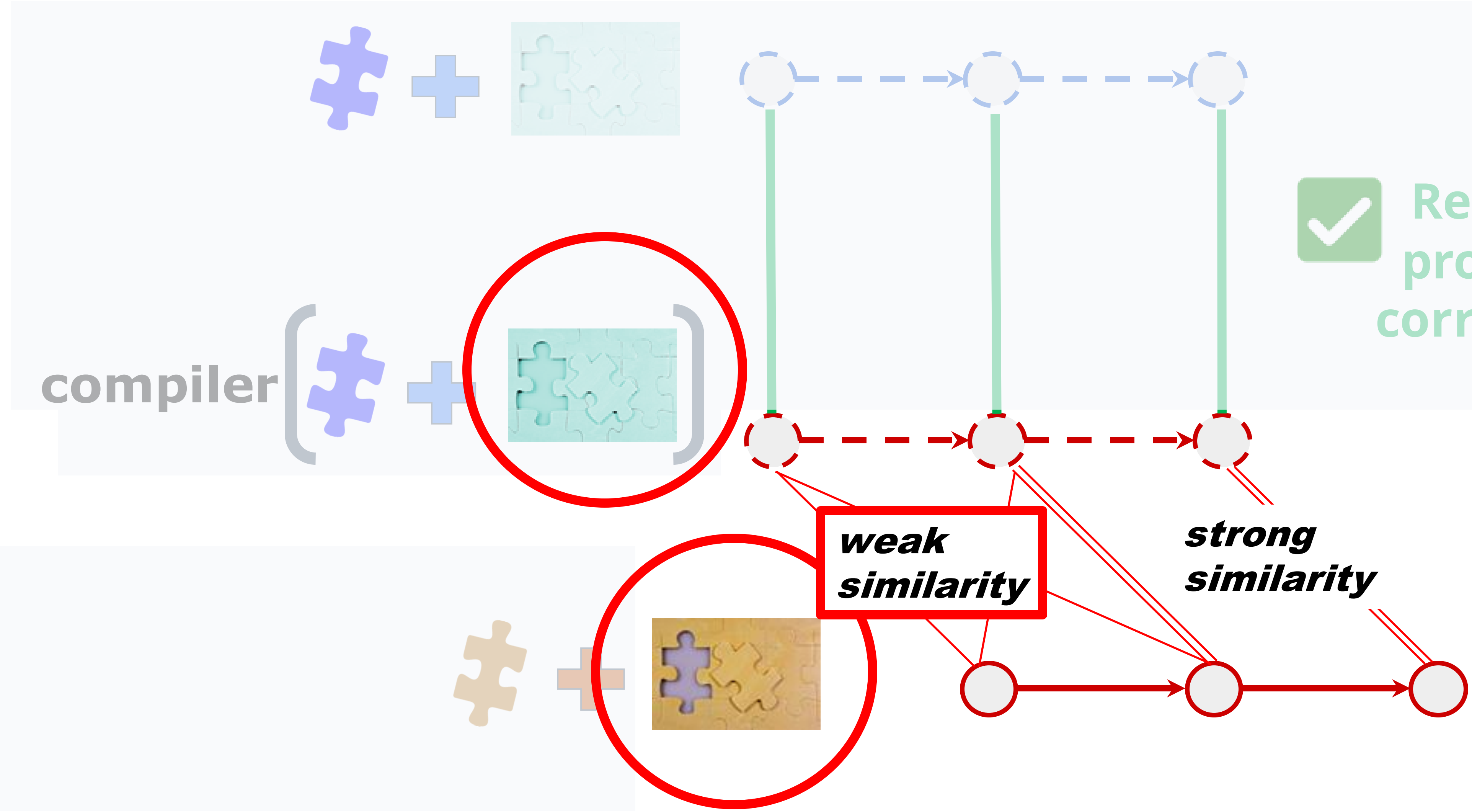
✓ Reuse the whole-program compiler correctness lemmas

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



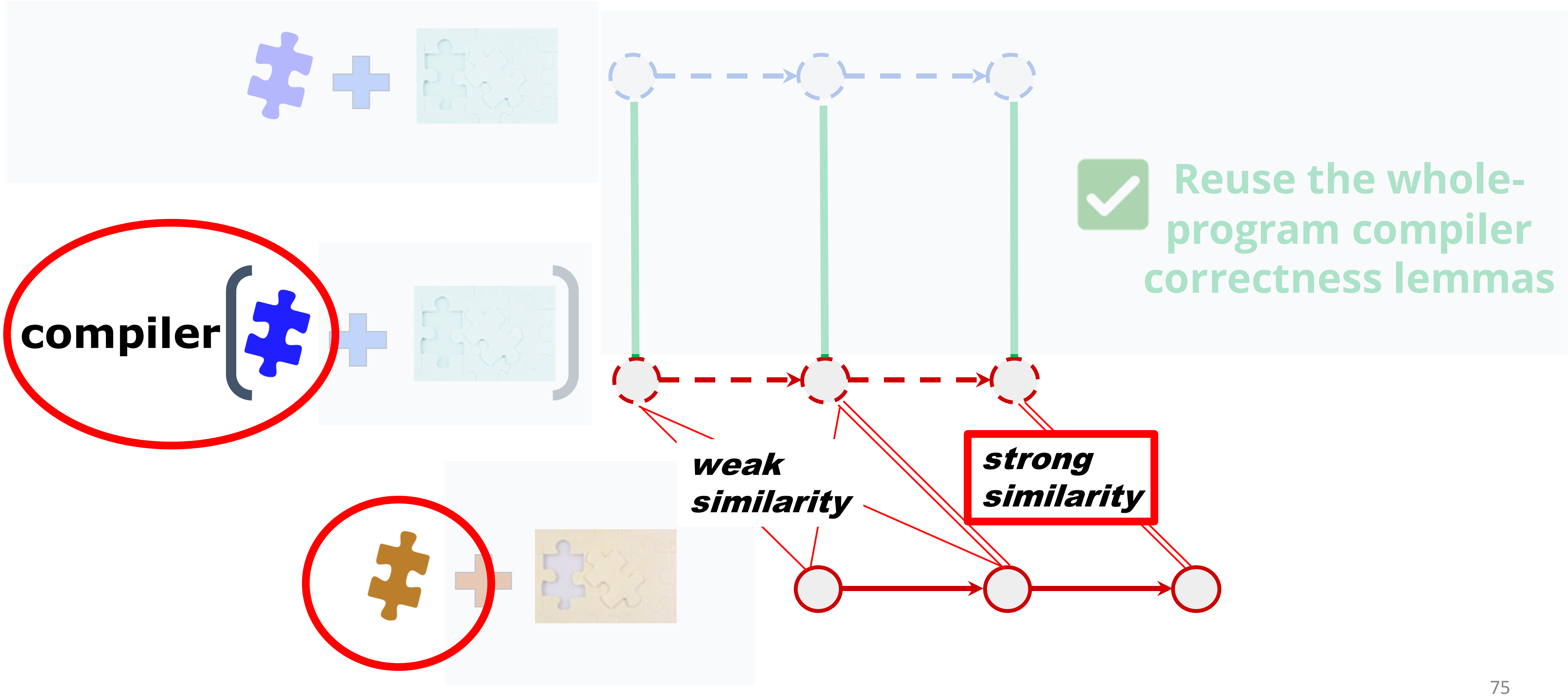
✓ Reuse the whole-program compiler correctness lemmas

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



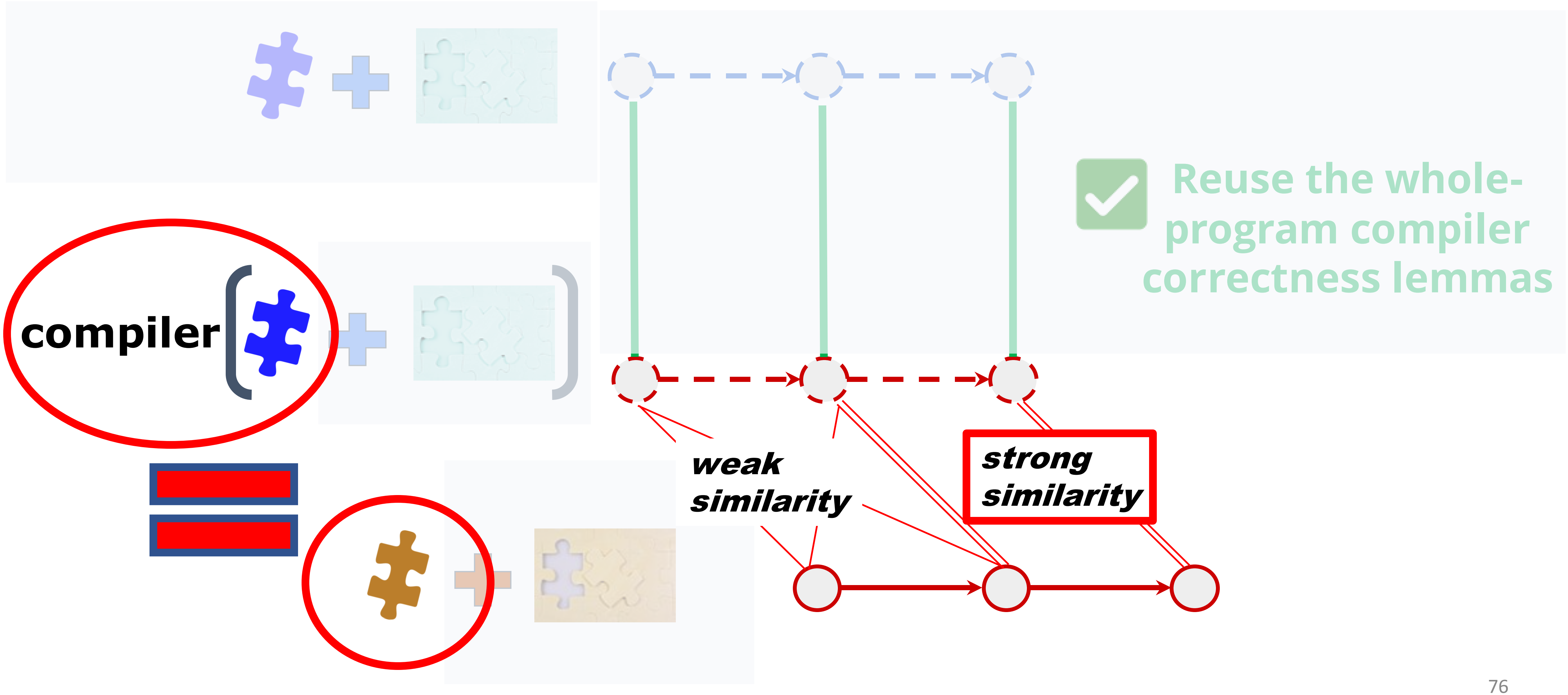
✓ Reuse the whole-program compiler correctness lemmas

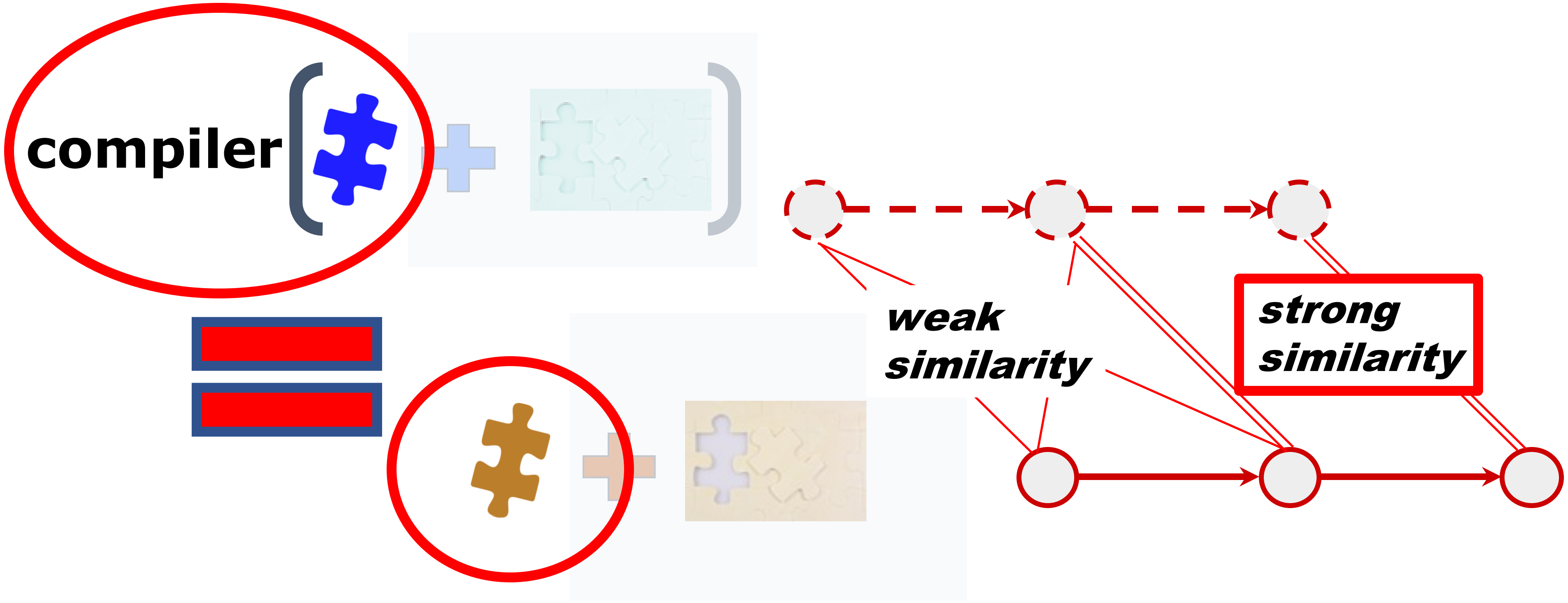
Novel proof technique
(called TrICL "/ˈtrɪk(ə)l/")



✓ Reuse the whole-program compiler correctness lemmas

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

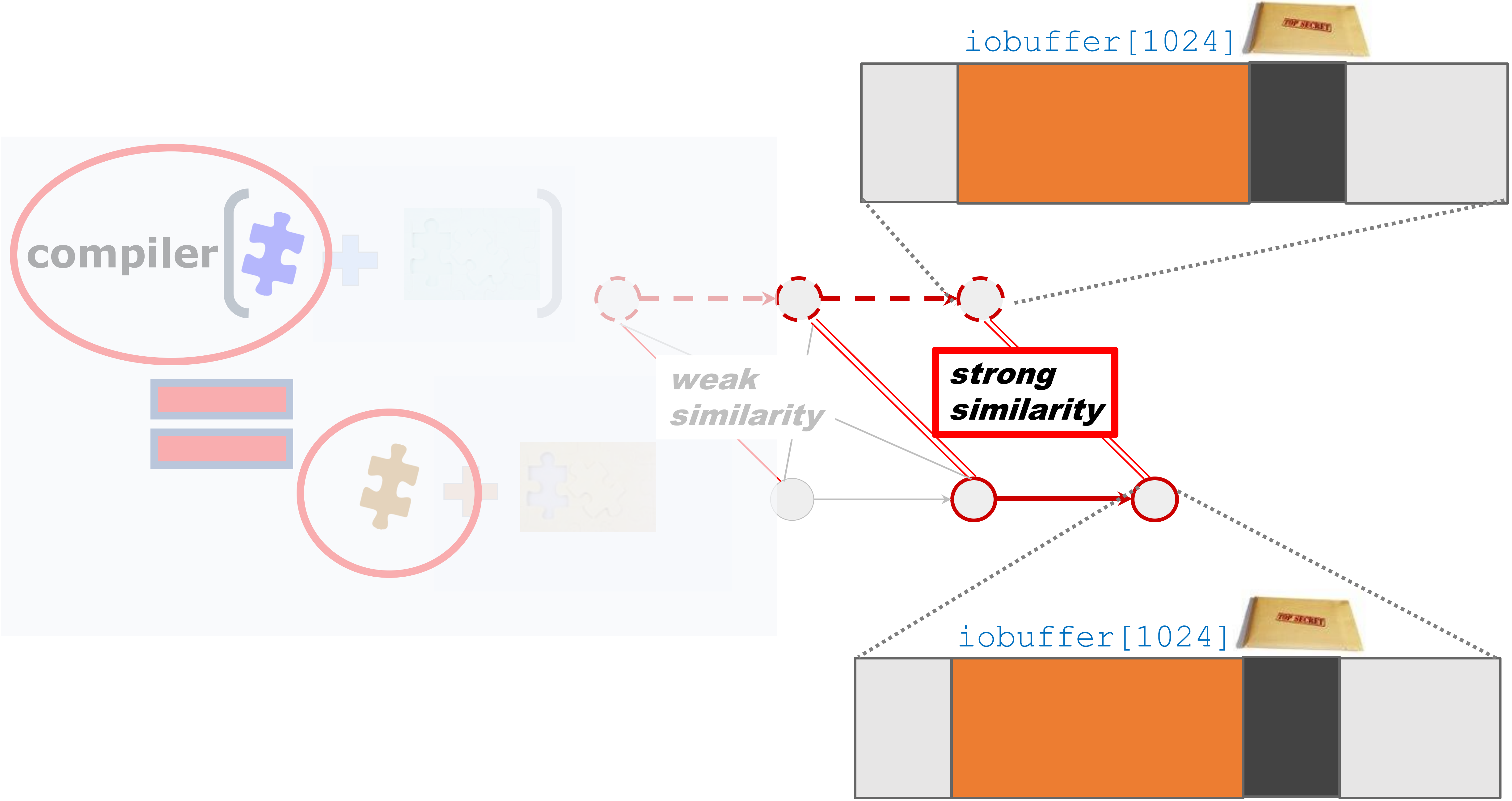


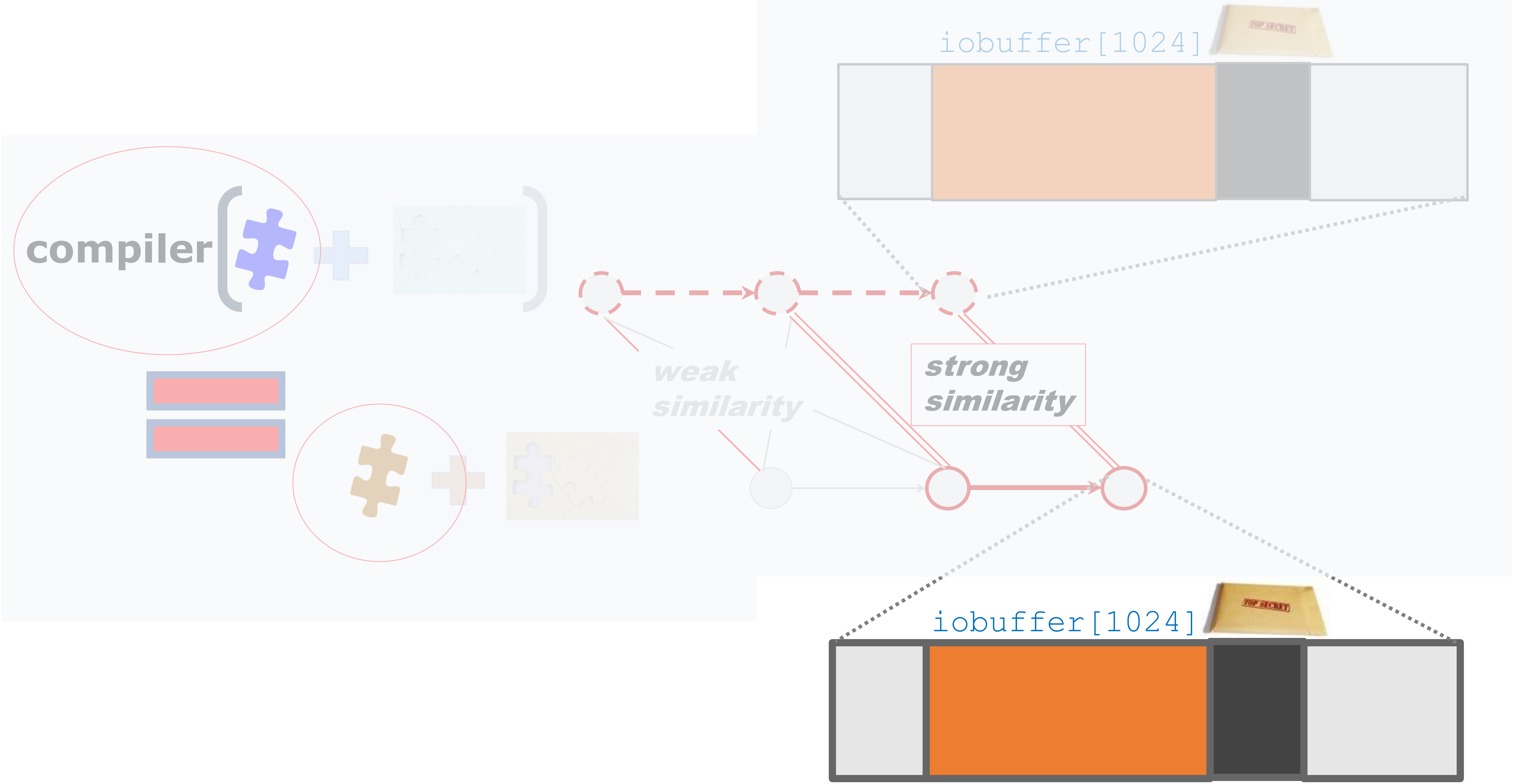


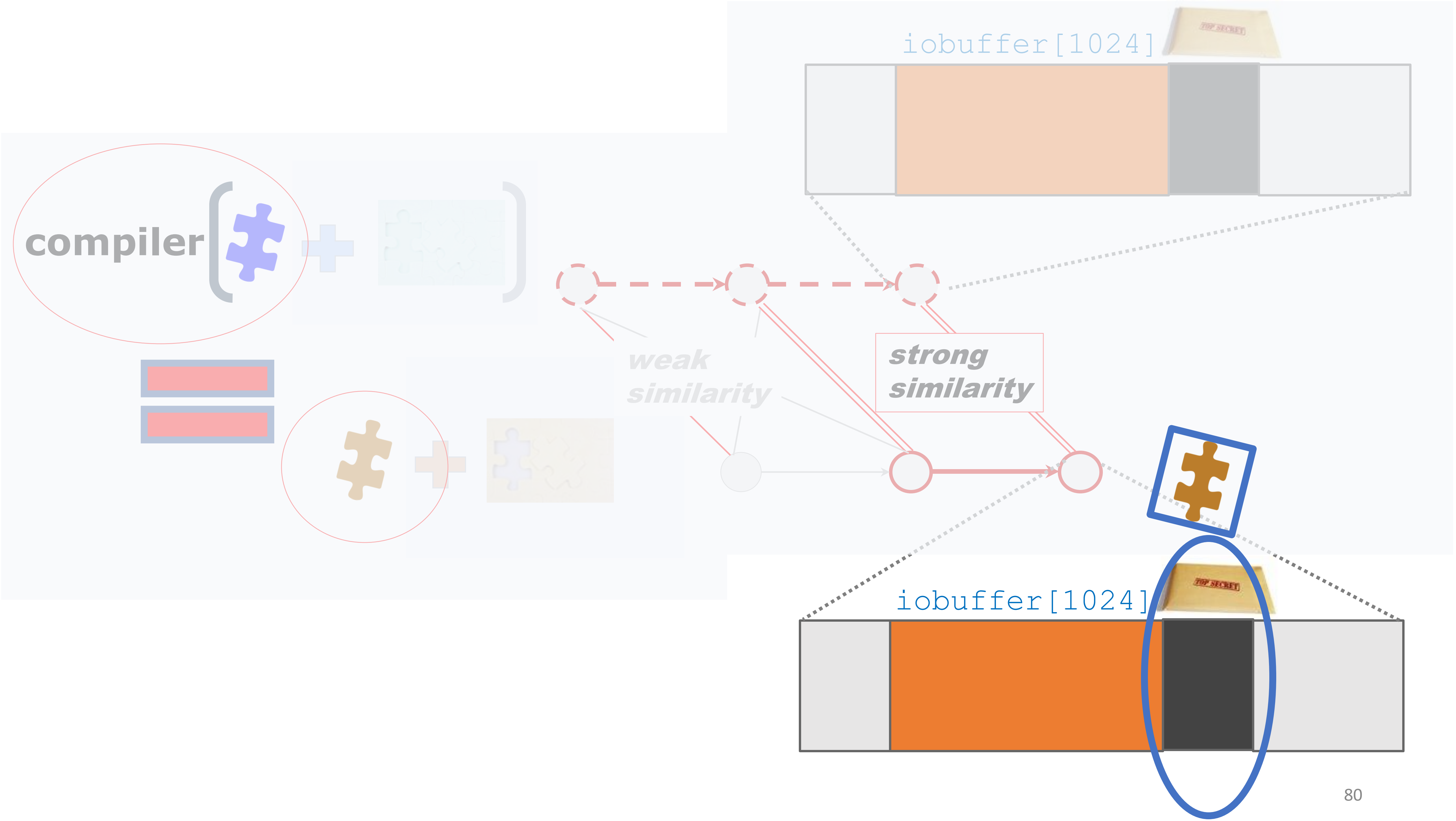
compiler

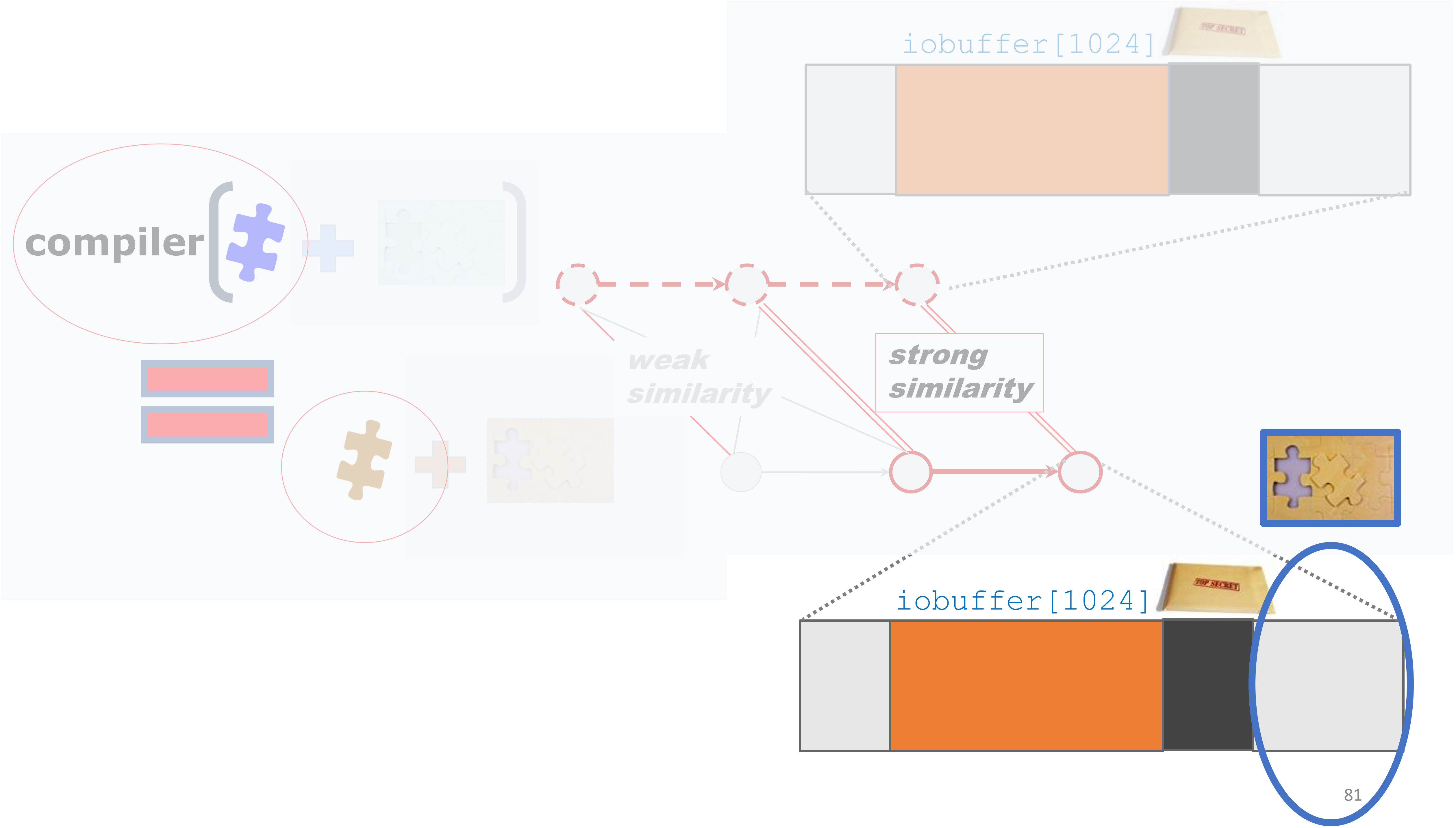
weak similarity

strong similarity


iobuffer[1024]
TOP SECRET
compiler
weak similarity
strong similarity
iobuffer[1024]
TOP SECRET


iobuffer[1024]
TOP SECRET
compiler
weak similarity
strong similarity
iobuffer[1024]
TOP SECRET


iobuffer[1024]
TOP SECRET
compiler
weak similarity
strong similarity
iobuffer[1024]
TOP SECRET



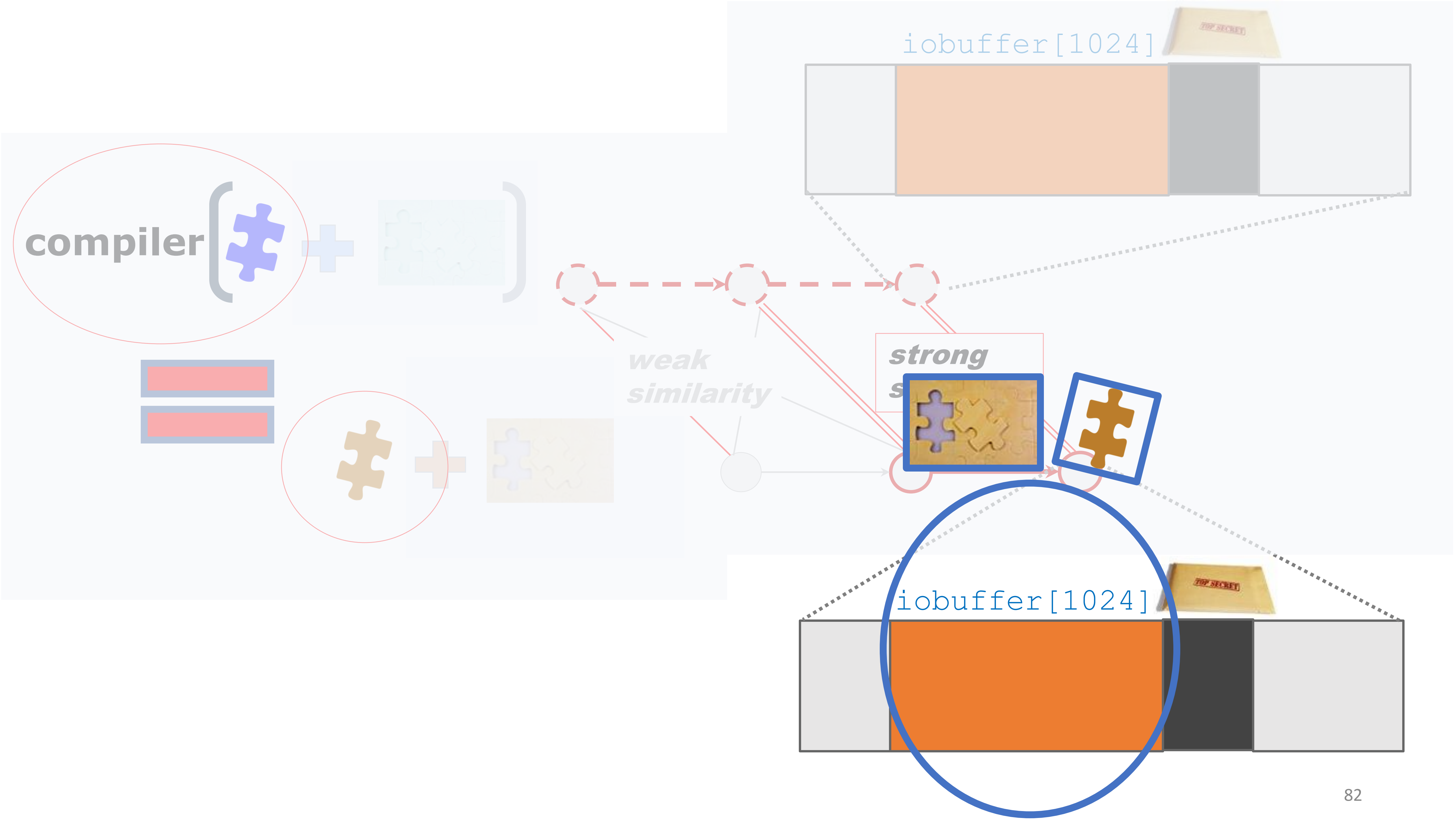
iobuffer[1024]

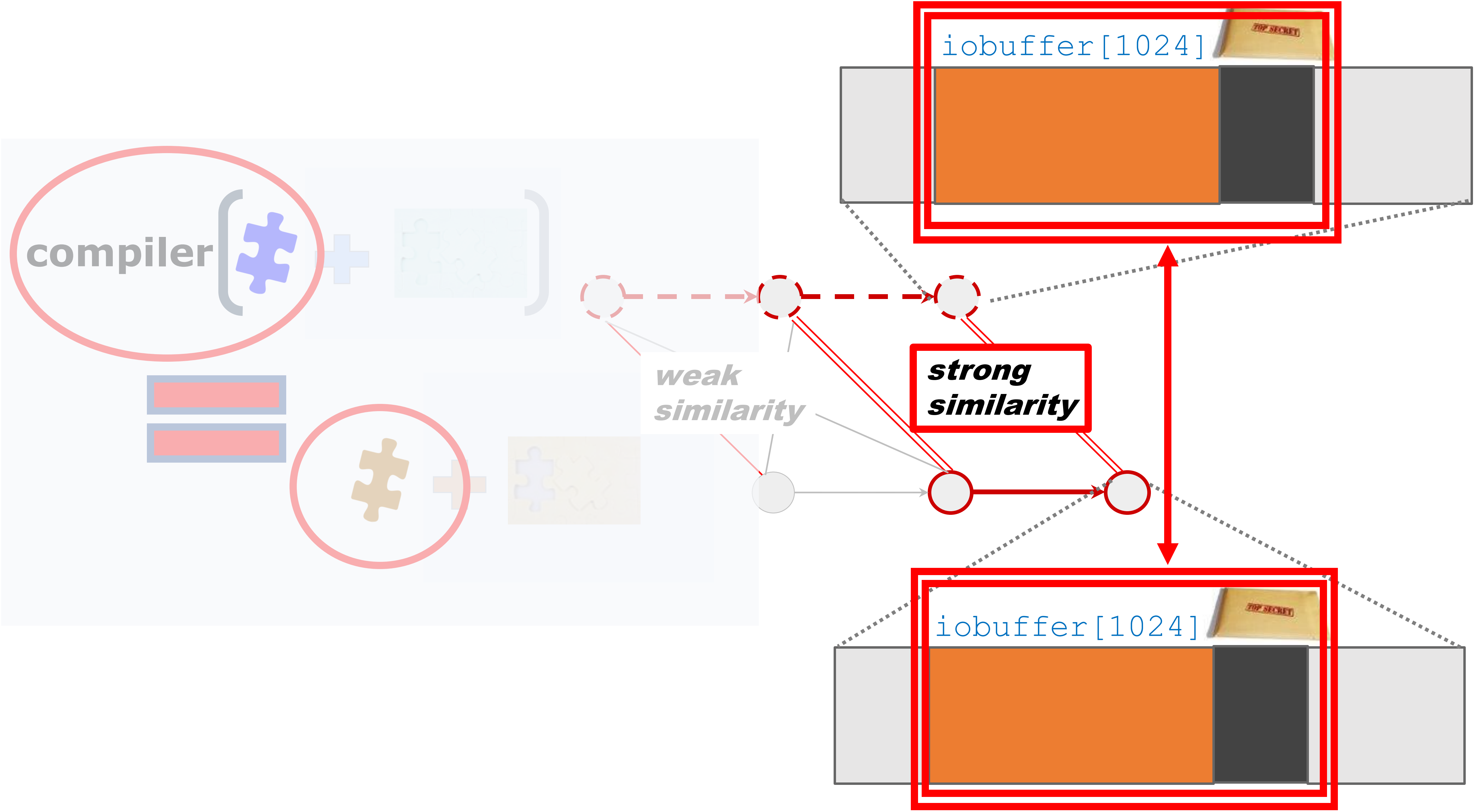
compiler

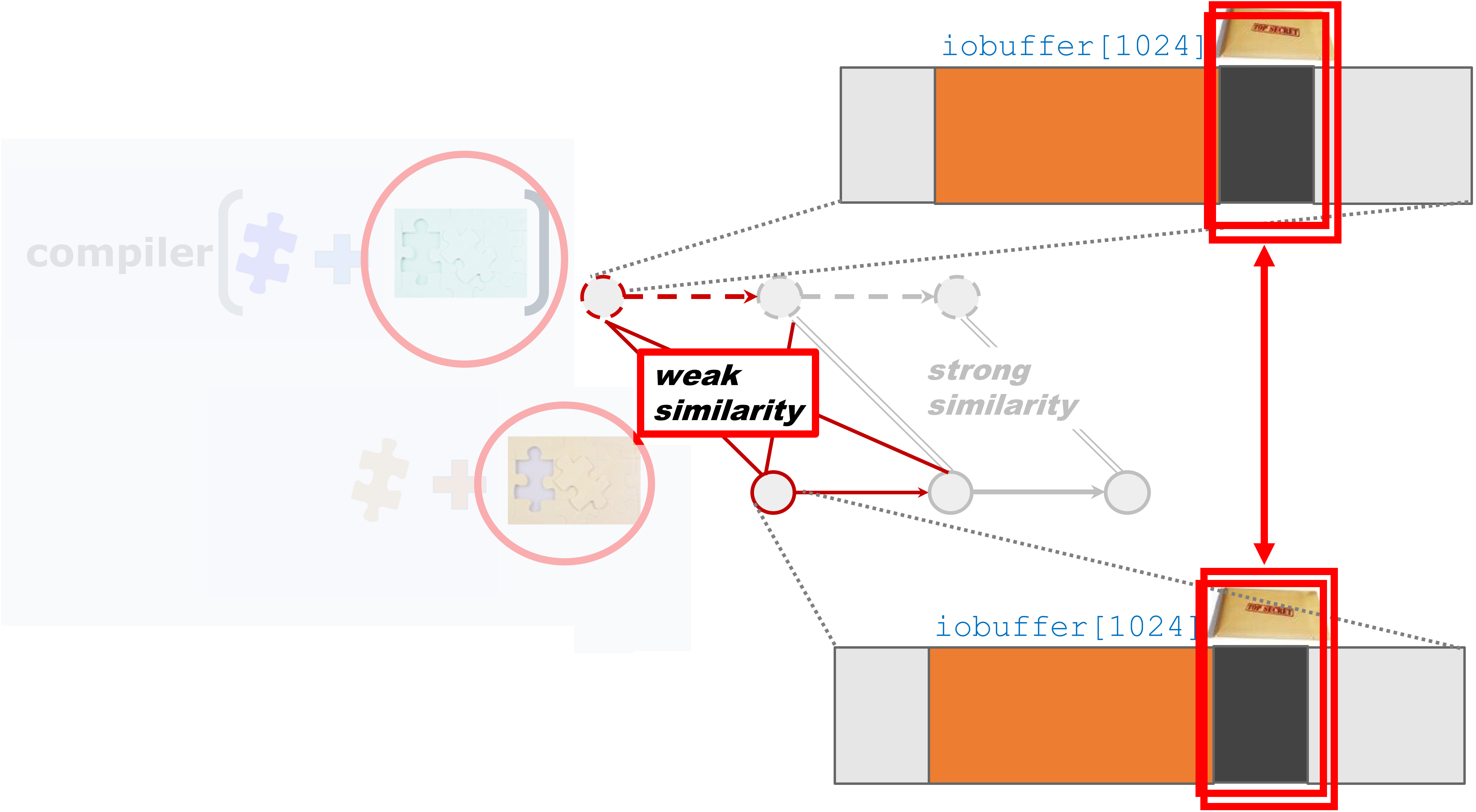
weak similarity

strong similarity

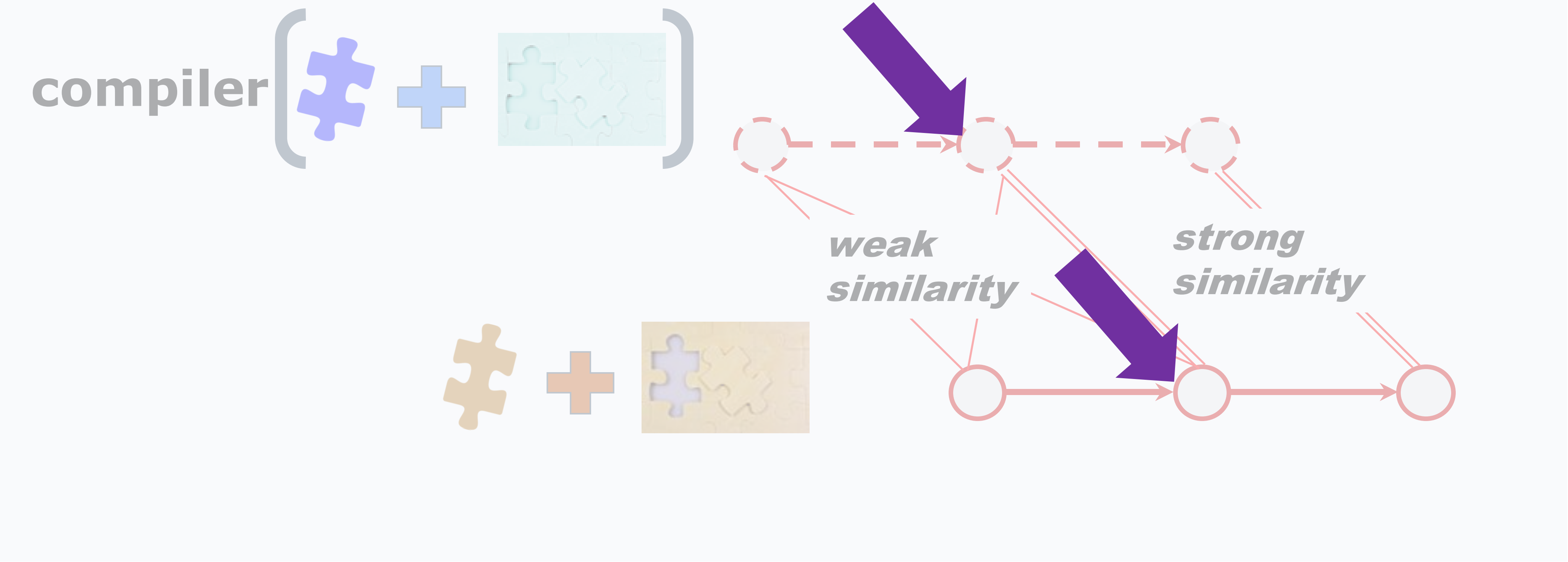
iobuffer[1024]


compiler
weak
similarity
strong
s
iobuffer[1024]
TOP SECRET
iobuffer[1024]
TOP SECRET

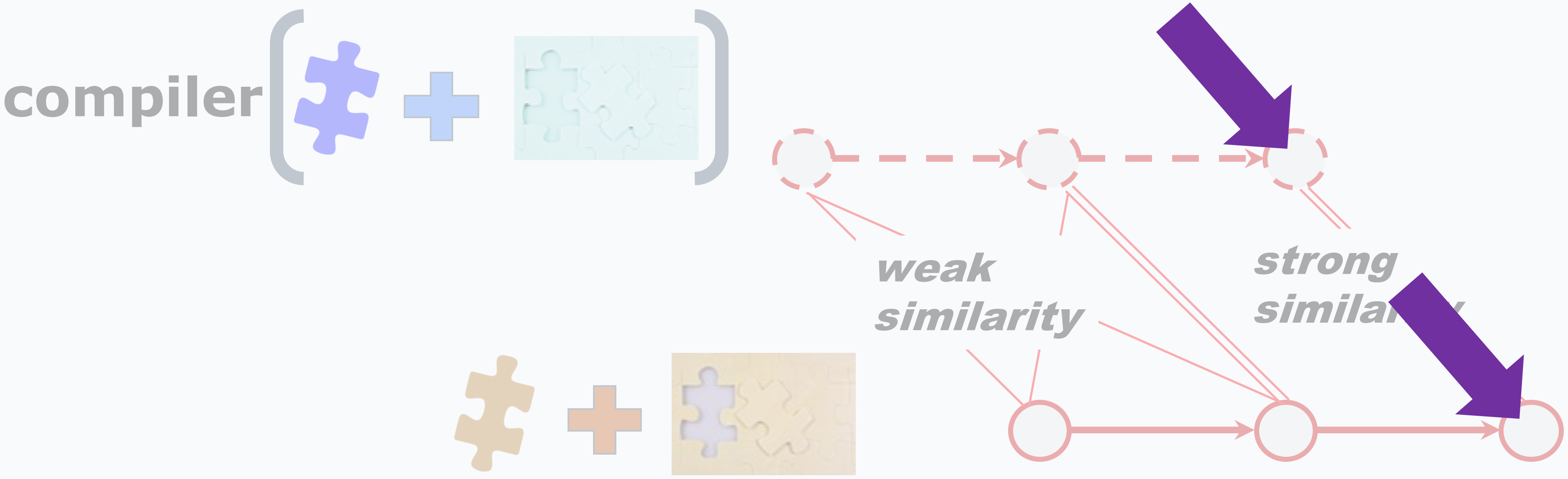

iobuffer[1024]
TOP SECRET
compiler
weak similarity
strong similarity
iobuffer[1024]
TOP SECRET


iobuffer[1024]
TOP SECRET
compiler
weak similarity
strong similarity
iobuffer[1024]
TOP SECRET

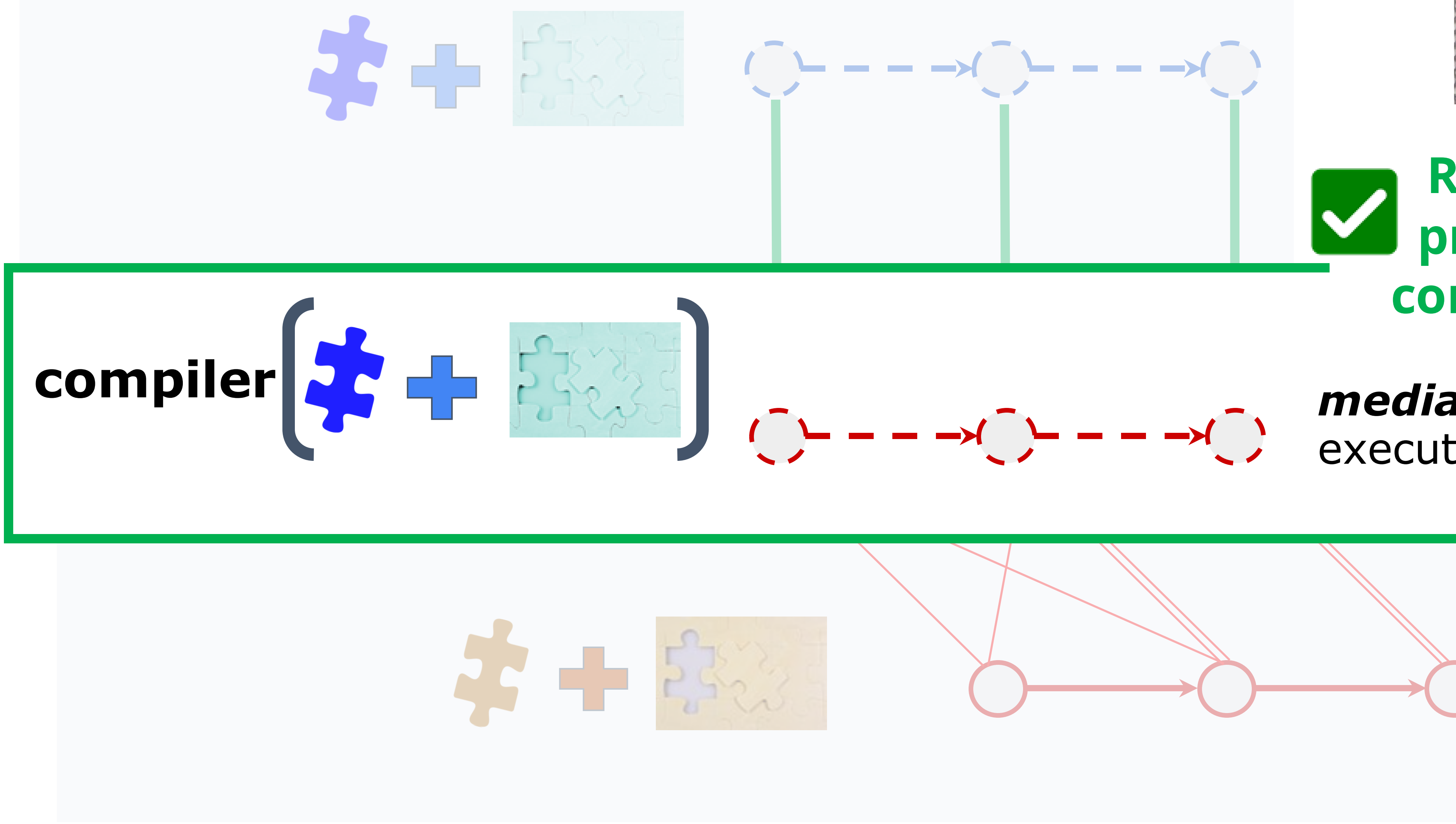
Strengthening lemma



Weakening lemma



Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")



✓ **Reuse the whole-program compiler correctness lemmas**

More in the paper

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

More in the paper

Novel proof technique
(called TrICL "/ˈtrɪk(ə)l/")

Trace-directed
Back-translation

example

More in the paper

Novel proof technique (called TrICL "/ˈtrɪk(ə)l/")

Trace-directed Back-translation example

Summary: In **CapablePtrs**, we present a proof of compiler full abstraction that achieves **reuse of the compiler correctness lemmas** while allowing **memory sharing through pointer passing.**