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?
Why Securely Compiling Partial Programs?

\[
\text{compiler} \left[ \begin{array}{c}
\# \\
\end{array} \right] = \begin{array}{c}
\# \\
\end{array}
\]
Why Securely Compiling Partial Programs?

Compiler (source program part) = compiled program part

untrusted third-party library
Why Securely Compiling Partial Programs?

The compiler takes source program part as input and outputs compiled program part. The untrusted third-party library could be buggy or malicious.
Why Securely Compiling Partial Programs?

security property of the source program part

source program part

compiled program part

untrusted third-party library

compiler

could be buggy or malicious
Why Securely Compiling Partial Programs?

security property of the source program part

security property of the compiled program part

security property of the compiled program part

source program part

compiled program part

untrusted third-party library

could be buggy or malicious
Why Securely Compiling Partial Programs?

security property of the source program part

security property of the compiled program part

source program part

compiled program part

untrusted third-party library

could be buggy or malicious
Let confidentiality be confidentially.

compiler(保密) = 保密

could be buggy or malicious
Let confidentiality be confidentiality

```c
#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;
}
```

could be buggy or malicious
Let confidentiality be confidential.

```c
#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;
}
```

could be buggy or malicious
Let confidentiality be confidential

```c
#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;
}
```

owns in memory could be buggy or malicious
Let confidentiality be

```c
#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 confidentiality be

```
#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;
}
```

could be buggy or malicious

could read from the memory of

char *secret_ptr = (char*)4210756;
leak(*secret_ptr);
Let confidentiality be confidential. #include "networking.h"

```c
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;
}
```

could be buggy or malicious.
Let confidentiality be confidentially.

#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 confidentiality be

#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;
}

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

```c
#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;
}
```

Could be buggy or malicious.
Let confidentiality be confidential.

```c
#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;
}
```

could read from the memory of

could be buggy or malicious

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

could be buggy or malicious

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

could be buggy or malicious

could read from the memory of

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

    send_checksum(checksum);
}

#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 confidentiality be confidentiality.

could be buggy or malicious.

could read from the memory of.

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

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;
}
Could be buggy or malicious.

Let confidentiality be the focus.

```
#include "networking.h"

void read_secret(char* s);
void process(char* b, char* s);
static char secret[256];
char iobuffer[1024];

int main(void) {
    read_secret(secret);
    receive(iobuffer);
    process(iobuffer, secret);
    return 0;
}
```

Can use process-based isolation.

```
int receive (char* buffer) {
    ...  
    int checksum = 0;
    for (int i=0; i<1024; i++)
        checksum += buffer[i];
    send_checksum(checksum);
    return 0;
}
```
#include "networking.h"

void read_secret(char* s);

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

static char secret[256];

char iobuffer[1024];

int main(void) {
    read_secret(secret);
    receive(iobuffer);
    process(iobuffer, secret);
    return 0;
}

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

Shared memory needs to be set up ahead of time.

**No pointer passing at run-time.**

Can use **process-based isolation**.
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 the different parts of the program from each other (with low performance overhead) while allowing pointer passing.

In the program:

```c
#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);
}
```
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 memory of the program from each other (with low performance overhead) while allowing 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 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 memory** of the different parts of the program from each other (with low performance overhead) while allowing pointer passing.

**Hardware capabilities**

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

**Novel proof technique** (called TrICL "/'trɪk(ə)l/"")

**Hardware capabilities**

**Novel proof technique** (called TrICL "/'trɪk(ə)l/"
Prior work on Compiler security
Repoved correctness implicitly as part of the security proof.

Prior work on Compiler security
Prior work on Compiler security

Reproved correctness implicitly as part of the security proof.

Achieved isolation by preventing memory sharing altogether.
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
Prior work on compiler security implicitly proved correctness as part of the security proof. Achieved isolation by preventing memory sharing altogether.

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

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

Achieved isolation by preventing memory sharing altogether.

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

```c
extern struct cheri.object lib1;
struct cheri.object lib2;
__attribute__((cheri callee))
__attribute__((cheri.method_class(lib1)))
int f1(void);
__attribute__((cheri callee))
__attribute__((cheri.method_class(lib2)))
int f2(void);
__attribute__((constructor)) static void
sandboxes_init(void)
{
  lib2 = fetch_object("lib2");
}
int f1(void)
{
  f2();
}
```
First compiler security proof that achieves reuse of the compiler correctness proof while allowing memory sharing through pointer passing.

**CapablePtrs**

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

Achieved isolation by preventing memory sharing altogether.

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

```c
extern struct cheri.object lib1;
struct cheri.object lib2;
__attribute__((cheri.callee))
__attribute__((cheri.method_class(lib1)))
int f1(void);

__attribute__((cheri.call))
__attribute__((cheri.method_class(lib2)))
int f2(void);

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

int f1(void)
{
    f2();
}
```
First compiler security proof that achieves reuse of the compiler correctness proof while allowing memory sharing through pointer passing.

Novel proof technique (called TrICL "/'trɪk(ə)l/"")

Achieved isolation by preventing memory sharing altogether.

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

<table>
<thead>
<tr>
<th>Library</th>
<th>CapablePtrs</th>
<th>Reproved</th>
<th>Achieved isolation</th>
<th>Preventing memory sharing altogether</th>
</tr>
</thead>
<tbody>
<tr>
<td>libpng</td>
<td>0.15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LibYAML</td>
<td>0.89%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zlib</td>
<td>1.15%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNU-barcode</td>
<td>3.5%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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/")

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

```c
extern struct cheri.object lib1;
struct cheri.object lib2;
__attribute__((cheri_callee))
__attribute__((cheri_method_class(lib1)))
int f1(void);
__attribute__((cheri_callee))
__attribute__((cheri_method_class(lib2)))
int f2(void);
__attribute__((constructor)) static void
sandboxes_init(void)
{
    lib2 = fetch_object("lib2");
}
int f1(void)
{
    f2();
}
```
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/"")
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** instead of implicitly re-proving it.

**Hardware capabilities**

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

No program part can forge capabilities.

Every memory access instruction expects a capability as an argument.
Hardware capabilities

No program part can forge capabilities.

Every memory access instruction expects a capability as an argument.
The compiler implements **pointer passing as capability passing**.

**Hardware capabilities**

Virtual memory

Every memory access instruction expects a capability as an argument.

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**

```c
char *secret_ptr = (char*)4210756;
leak(*secret_ptr);
```
Hardware capabilities

could be buggy or malicious

char *secret_ptr = (char*)4210756;
leak(*secret_ptr);
Hardware capabilities

could be buggy or malicious

char *secret_ptr = (char*)4210756;

leak(*secret_ptr);
Load Integer via Capability Register

```c
if not (cb_val.tag) then
    raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
    raise_c2_exception(CapEx_Sea1Violation, 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*size);
    let vAddr64 = to_bits(64, vAddr);
    if (vAddr + size) > getCapTop(cb_val) then
        raise_c2_exception(CapEx_LengthViolation, cb)
```
Hardware capabilities could be buggy or malicious.

Character *secret_ptr = (char*)4210756;

leak(*secret_ptr);

```c
if not (cb_val.tag) then
    raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
    raise_c2_exception(CapEx_SeaIValViolation, 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*subscription);
        let vAddr64 = to_bits(64, vAddr);
        if (vAddr + size) > getCapTop(cb_val) then
            raise_c2_exception(CapEx_LengthViolation, cb)
```
 Hardware capabilities

Virtual memory could be buggy or malicious.

Register file iobuffer[1024]

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

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)
```

WidthBytes(width);
Cursor(cb_val);

if (vAddr + size) > getCapTop(cb_val) then
```
raise_c2_exception(CapEx_LengthViolation, cb)
```
The compiler implements **pointer passing as capability passing**.

**Hardware capabilities**

Every memory access instruction expects a capability as an argument.

No program part can forge capabilities.

Virtual memory
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**.

**Hardware capabilities**

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

The same definition for the target language too.
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 program while allowing **pointer passing** without high **performance overhead**.

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

**Hardware capabilities**

**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/"")

compiled program part

untrusted third-party library

mimicking source library

Trace-directed Back-translation
Novel proof technique (called TrICL "/ʼtrɪk(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/")

- Compiled program part
- Untrusted third-party library

Mimicking source library

Trace-directed Back-translation
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 "/'trɪk(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

weak similarity

strong similarity
Novel proof technique (called TrICL "/ˈtrɪkl(ə)l/"")

Reuse the whole-program compiler correctness lemmas

weak similarity

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

Reuse the whole-program compiler correctness lemmas

compiler

weak similarity

strong similarity
Novel proof technique (called TrICL "/'trɪk(ə)l/"")

Reuse the whole-program compiler correctness lemmas

weak similarity

strong similarity
compiler

weak similarity

strong similarity
compiler

iobuffer[1024]

weak similarity

strong similarity

iobuffer[1024]
compiler

weak similarity

strong similarity

iobuffer[1024]

iobuffer[1024]
compiler

iobuffer[1024]

weak similarity

strong similarity

iobuffer[1024]
The diagram illustrates the concept of weak and strong similarity in the context of compiler optimization. The diagram shows two input blocks labeled "compiler" connected to two output blocks labeled "iobuffer[1024]". The "weak similarity" is indicated by dashed arrows, suggesting a less direct relationship, while "strong similarity" is indicated by solid arrows, implying a stronger relationship between the components. The diagram visually represents the flow of information and the level of similarity between different parts of the compiler process.
Strengthening lemma

compiler

weak similarity

strong similarity
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 "/trɪkl(ə)l/"
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.