SecurePtrs

Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation

Akram El-Korashy
MPI-SWS

Joint work with
Roberto Blanco, Jérémy Thibault, Adrien Durier (MPI-SP), Deepak Garg (MPI-SWS), Catalin Hritcu (MPI-SP)
Setup: Secure compilation of *partial* programs
Setup: Secure compilation of *partial* programs

Risk: Partial programs may be linked against *buggy* or *malicious contexts*. 
Setup: Secure compilation of partial programs

Risk: Partial programs may be linked against buggy or malicious contexts.

Strategy: Prove that the partial programs, when compiled properly, are protected from the contexts.
Setup: Secure compilation of *partial* programs

For example, a single module or compilation unit
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();
    }
}
Setup: Secure compilation of **partial** programs

```plaintext
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();
    }
}
```

- The context implements it
- The partial program calls it
Setup: Secure compilation of partial programs

```plaintext
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();
    }
}
```

The partial program intentionally shares the array with the context
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();
    }
}

The partial program NEVER shares the user balance with the context
import module Net

module Main {
  char iobuffer[1024];
  static long int user_balance_usd;

  int main(void) {
    Net.init_network(iobuffer)
    Net.receive();
  }
}
Setup: Secure compilation of **partial** programs

**Recall**

Risk: Partial programs may be linked against **buggy** or **malicious contexts**.

Strategy: *Prove* that the partial programs, when compiled properly, are *protected* from the contexts.
Intention is that user balance is "high integrity".

Setup: Secure compilation of partial programs

```java
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer);
        Net.receive();
    }
}
```

The partial program \textbf{NEVER shares} the user balance with the context.

A buggy/malicious context \textbf{might access} the user balance.
Setup: Secure compilation of *partial* programs

```python
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer);
        Net.receive();
    }
}
```

A buggy/malicious context might access the user balance

```assembly
init_network:
    addi $r1 $r_arg 1024
    sw $r2 0($r1)
```
Setup: Secure compilation of *partial* programs

Risk: Partial programs may be linked against *buggy* or malicious contexts.

Strategy: *Prove* that the partial programs, when compiled properly, are *protected* from the contexts.
Setup: Secure compilation of partial programs

Risk: Partial programs may be linked against buggy or malicious contexts.

Strategy: Prove that the partial programs, when compiled properly, are protected from the contexts.

"Compiled properly" means the compiler enforces isolation e.g. by relying on CHERI, micropolicies, etc.
Setup: Secure compilation of *partial* programs

Risk: Partial programs may be linked against buggy or malicious contexts.

Strategy: *Prove* that the partial programs, when compiled properly, are *protected* from the contexts.

Focus of this talk: **Proof techniques**
Strategy: **Prove** that the partial programs, when compiled properly, are **protected** from the contexts.

**Desired:** Preserve the security of the source program part  
(assuming a memory-safe source semantics)
Desired: Preserve the security of the source program part
(assuming a memory-safe source semantics)

Desired (for our example):
If no execution with a **source context** overwrites the user balance, then no execution with a **target context** overwrites it either.
Desired: Preserve the security of the source program part
(assuming a memory-safe source semantics)

Desired (for our example): Forall safety property $S$,
If no execution with a source context violates $S$,
no execution with a target context violates $S$ either.
Desired (for our example):

**Forall safety property S,**

If no execution of a source context violates S, then no execution of the target context violates S either.

called "**Preservation of Robust Safety**"
Desired (for our example):

\[
\text{Forall safety property } S, \\
\text{If no execution of a source context violates } S, \text{ then no execution of the target context violates } S \text{ either.}
\]

called "Preservation of Robust Safety"

This talk: Explain a proof technique, called data-flow back-translation.
This talk: Explain a proof technique, called data-flow back-translation.

Key Benefits: Suited for memory sharing and syntactic dissimilarity
Desired (for our example):
If no execution with a source context overwrites the user balance, then no execution with a target context overwrites it either.
Alternatively, prove the contrapositive:
If there exists an execution of a target context that overwrites the user balance, then there also exists a source context and an execution in which it too overwrites the user balance.
Alternatively, prove the contrapositive:

If there exists an execution of a **target context** that overwrites the user balance, then there also exists a **source context** and an execution in which it too overwrites the user balance.

called "**Back-translation**".
Familiar from plenty of secure compilation literature
Alternatively, prove the contrapositive:

If there exists an execution of a **target context** that **overwrites the user balance**, then there also exists a **source context** and an execution in which it too **overwrites the user balance**.

Can prove a **back-translation lemma about just whole programs** [Abate et al. 2018 "When good components go bad"]:  

If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution.
If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution, called "**Back-translation**". Two techniques in the literature:
If there exists an execution of a whole target program, then there exists a whole source program and a related execution. called "Back-translation". Two techniques in the literature:

Syntax-directed
If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution. called "**Back-translation**". Two techniques in the literature:

**Syntax-directed**

: **target programs** → **source programs**
If there exists an execution of a whole target program, then there exists a whole source program and a related execution.

called "Back-translation". Two techniques in the literature:

Syntax-directed

ALL target programs, not just the image of the compiler.
If there exists an execution of a whole target program, then there exists a whole source program and a related execution. Called "Back-translation". Two techniques in the literature:

Syntax-directed:

Need to prove that is correct, i.e., satisfies the spec.
If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution. Called "Back-translation". Two techniques in the literature:

**Syntax-directed**

: target programs \rightarrow source programs

Correctness proof similar to a compiler correctness proof
If there exists an execution of a whole target program, then there exists a whole source program and a related execution.

called "Back-translation". Two techniques in the literature:

**Syntax-directed**

- target programs → source programs

  Correctness proof similar to a compiler correctness proof

  Compiling unstructured target to a structured source unclear
If there exists an execution of a whole target program, then there exists a whole source program and a related execution.

called "Back-translation". Two techniques in the literature:

**Syntax-directed**

: target programs \rightarrow source programs

Correctness proof similar to a compiler correctness proof

Compiling unstructured target to a structured source unclear

**Trace-directed**

Ignore the given program. Focus just on the given execution trace (i.e., on an individual run of the program).
If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution.

called "Back-translation". Two techniques in the literature:

**Syntax-directed**
- Target programs $\rightarrow$ Source programs
- Correctness proof similar to a compiler correctness proof
- Compiling unstructured target to a structured source unclear

**Trace-directed**
- Interaction traces $\rightarrow$ Source programs
If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution.

called "Back-translation". Two techniques in the literature:

**Syntax-directed**

- Target programs $\rightarrow$ Source programs
- Correctness proof similar to a compiler correctness proof
- Compiling unstructured target to a structured source unclear

**Trace-directed**

- Target programs $\rightarrow$ Interaction traces $\rightarrow$ Source programs
- A prefix of one target trace.
If there exists an execution of a **whole target program**, then there exists a **whole source program** and a related execution. called "Back-translation". Two techniques in the literature:

### Syntax-directed

- **Target programs** → **Source programs**
  - Correctness proof similar to a compiler correctness proof
  - Compiling unstructured **target** to a structured **source** unclear

### Trace-directed

- **Interaction traces** → **Source programs**
  - Indifferent to syntactic dissimilarity between **target** and **source**
If there exists an execution of a whole target program, then there exists a whole source program and a related execution.

Called "Back-translation". Two techniques in the literature:

- **Syntax-directed target programs**: source programs
- **Trace-directed interaction traces**: source programs

Correctness proof similar to a compiler correctness proof.

Compiling unstructured target to a structured source unclear.

Indifferent to syntactic dissimilarity between target and source.

No need anymore to translate the within-module control constructs. Only mimic the external interaction (flexible def of the back-translation).
If there exists an execution of a whole target program, then there exists a whole source program and a related execution. Called "Back-translation". Two techniques in the literature:

**Syntax-directed**

- **target programs** → **source programs**
- Correctness proof similar to a compiler correctness proof
- Compiling unstructured target to a structured source unclear

**Trace-directed**

- **interaction traces** → **source programs**
- Correctness proof with memory sharing is involved.
- Indifferent to syntactic dissimilarity between target and source
If there exists an execution of a whole target program, then there exists a whole source program and a related execution. called "Back-translation". Two techniques in the literature:

**Syntax-directed**
- target programs → source programs
- Correctness proof similar to a compiler correctness proof
- Compiling unstructured target to a structured source unclear

**Trace-directed**
- interaction traces → source programs
- Correctness proof with memory sharing is involved.
- Indifferent to syntactic dissimilarity between target and source
Trace-directed interaction traces: source programs

Correctness proof with memory sharing is involved.
Given a trace emitted by a **target** program
Trace-directed interaction traces: source programs

Correctness proof with memory sharing is involved.

Given a trace emitted by a target program

Find a source program emitting a related trace
Trace-directed interaction traces:

- Source programs

Correctness proof with memory sharing is involved.

Given a trace emitted by a target program, find a source program emitting a related trace.

Back-translation has to mimic the visible shared memory operations to emit a related trace.
and are traces of only the **externally observable** events
are traces of only the **externally observable** events

\[ \lambda \ ::= \ \tau \]

<table>
<thead>
<tr>
<th>\checkmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ret} \ ? \ Mem</td>
</tr>
<tr>
<td>\texttt{ret} \ ! \ Mem</td>
</tr>
<tr>
<td>\texttt{call}(\texttt{fid}) \ \overline{v} \ ? \ Mem</td>
</tr>
<tr>
<td>\texttt{call}(\texttt{fid}) \ \overline{v} \ ! \ Mem</td>
</tr>
</tbody>
</table>
and are traces of only the **externally observable** events

\[
\lambda ::= \top \\
\quad \checkmark \\
\quad \text{ret} ? \text{Mem} \\
\quad \text{ret} ! \text{Mem} \\
\quad \text{call}(\text{fid}) \overline{v} ? \text{Mem} \\
\quad \text{call}(\text{fid}) \overline{v} ! \text{Mem}
\]

**Silent labels** denote internal execution. All silent labels are eventually **dropped**.
and

are traces of only the **externally observable** events

\[
\begin{align*}
\lambda & ::= \tau \\
| & \checkmark \\
| & \text{ret ? } Mem \\
| & \text{ret ! } Mem \\
| & \text{call}(fid) \overline{v} \text{ ? } Mem \\
| & \text{call}(fid) \overline{v} \text{ ! } Mem
\end{align*}
\]

There are two kinds of **border-crossing call events** (program to context, and context to program).
are traces of only the **externally observable** events.

There are two kinds of **border-crossing return events** (program to context, and context to program).
are traces of only the **externally observable** events

\[
\lambda ::= \tau \\
\quad | \checkmark \\
\quad | \text{ret} ? \text{Mem} \\
\quad | \text{ret} ! \text{Mem} \\
\quad | \text{call}(\text{fid}) \bar{v} ? \text{Mem} \\
\quad | \text{call}(\text{fid}) \bar{v} ! \text{Mem}
\]

Calls and returns record a **snapshot** of all the memory shared so far
include module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();
    }
}
include module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer);
        Net.receive();
    }
}
include module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer);
        Net.receive();
    }
}
The call to `init_network` shares the `iobuffer`; a snapshot of its contents appears now and in all future border-crossing events.
The **return** from `init_network` still shows the `iobuffer` with the same contents.

```cpp
include module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();
    }
}
```
The call to `receive` does not (directly) share anything new, but still
The return event from receive also shows a snapshot of iobuffer, now with the received data!
The `init_network` function must have stashed the pointer to `iobuffer` somewhere in order to enable other functions of `Net` to access it.

```c
include module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer);
        Net.receive();
    }
}
```
The `init_network` function must have stashed the pointer to `iobuffer` somewhere in order to enable other functions of `Net` to access it, but this stash does NOT appear on the interaction trace because it is not part of the shared memory.
The `init_network` function must have **stashed the pointer** to `iobuffer` somewhere in order to enable other functions of `Net` to access it, but this stash does NOT appear on the interaction trace because it is not part of the shared memory.

**Trace-directed**

↑: interaction traces → source programs

Still needs to enable other functions of `Net` to access `iobuffer`
The `init_network` function must have stashed the pointer to `iobuffer` somewhere in order to enable other functions of `Net` to access it, but this stash does NOT appear on the interaction trace because it is not part of the shared memory.

Trace-directed implements own stash

↑: interaction traces → source programs
Drawback of trace-directed back-translation: must traverse and stash the whole shared memory CapablePtrs [El-Korashy et al. 2021]

Reason: Pointers may be shared indirectly.
Drawback of trace-directed back-translation: must traverse and stash the whole shared memory

CapablePtrs [El-Korashy et al. 2021]

Reason: Pointers may be shared indirectly.

Fatten the whole graph reachable from the shared memory and stash it:

\[
\text{init\_network\_arg\_1, init\_network\_arg\_2, \ldots, init\_network\_arg\_n}
\]

and maintain invariants between the flattening and the original.
The stashing mechanism of CapablePtrs [El-Korashy et al. 2021] is not mechanized-proof friendly.

Proving that this stashing mechanism is sufficient to mimic every possible memory snapshot is not trivial in Coq.
The stashing mechanism of CapablePtrs [El-Korashy et al. 2021] is not mechanized-proof friendly.

Proving that this stashing mechanism is sufficient to mimic every possible memory snapshot is not trivial in Coq.

e.g., **Termination** lemmas for custom **graph traversal** algorithms have to be proved.
In summary: Need a **back-translation technique** that

- supports *memory sharing* by pointer passing
- we can *mechanize* with reasonable effort
In summary: Need a **back-translation technique** that

- supports **memory sharing** by pointer passing
- we can **mechanize** with reasonable effort
- is indifferent to **syntactic dissimilarity** between **target** and **source**
In summary: Need a back-translation technique that

- supports **memory sharing** by pointer passing
- we can **mechanize** with reasonable effort
- is indifferent to **syntactic dissimilarity** between target and source

Data-Flow Back-Translation
Data-Flow Back-Translation

[Under submission]

High level idea: Make the traces more informative so that trace-directed back-translation is easier.
Data-Flow Back-Translation

[Under submission]

High level idea: Make the traces more informative so that trace-directed back-translation is easier.

Need to be careful: The validity of the top-level theorem depends on the interaction traces capturing just the externally observable behavior of a module.
High level idea: Make the traces more informative so that trace-directed back-translation is easier.

Need to be careful: The validity of the top-level theorem depends on the interaction traces capturing just the externally observable behavior of a module.

(Turns out: easy to decouple the trace alphabet of the main theorem from the trace alphabet of the back-translation. See the enrichment lemma and the projection function in the manuscript.)
Data-Flow Back-Translation

Recall alphabet of interaction traces

\[
\begin{align*}
\lambda & ::= \tau \\
& | \quad \text{ret} \ ? \ Mem \\
& | \quad \text{ret} \ ! \ Mem \\
& | \quad \text{call}(fid) \ \overline{v} \ ? \ Mem \\
& | \quad \text{call}(fid) \ \overline{v} \ ! \ Mem
\end{align*}
\]

Silent labels denote internal execution.
Silent labels are too abstract. (They beneficially hide the control steps, but unbeficentially hide data-flow steps.)
Data-Flow Back-Translation

Selectively break the silent-label abstraction
Data-Flow Back-Translation

Definition 3.2 (Events of data-flow traces).

\[ \mathcal{E} ::= \text{dfCall Mem Reg } c_{\text{caller}} c_{\text{callee}} \cdot \text{proc}(v) \]

\[ \mid \text{dfRet Mem Reg } c_{\text{prev}} c_{\text{next}} v \]

\[ \mid \text{Const Mem Reg } c_{\text{cur}} v r_{\text{dest}} \]

\[ \mid \text{Mov Mem Reg } c_{\text{cur}} r_{\text{src}} r_{\text{dest}} \]

\[ \mid \text{BinOp Mem Reg } c_{\text{cur}} \text{ op } r_{\text{src1}} r_{\text{src2}} r_{\text{dest}} \]

\[ \mid \text{Load Mem Reg } c_{\text{cur}} r_{\text{addr}} r_{\text{dest}} \]

\[ \mid \text{Store Mem Reg } c_{\text{cur}} r_{\text{addr}} r_{\text{src}} \]

\[ \mid \text{Alloc Mem Reg } c_{\text{cur}} r_{\text{ptr}} r_{\text{size}} \]
Data-Flow Back-Translation

**Definition 3.2** (Events of data-flow traces).

\[ E ::= \text{dfCall} \text{ Mem Reg } c_{\text{caller}} \text{ } c_{\text{callee}} \cdot \text{proc}(v) \]
\[ \mid \text{dfRet} \text{ Mem Reg } c_{\text{prev}} \text{ } c_{\text{next}} \text{ } v \]
\[ \mid \text{Const} \text{ Mem Reg } c_{\text{cur}} \text{ } v \text{ } r_{\text{dest}} \]
\[ \mid \text{Mov} \text{ Mem Reg } c_{\text{cur}} \text{ } r_{\text{src}} \text{ } r_{\text{dest}} \]
\[ \mid \text{BinOp} \text{ Mem Reg } c_{\text{cur}} \text{ } op \text{ } r_{\text{src}1} \text{ } r_{\text{src}2} \text{ } r_{\text{dest}} \]
\[ \mid \text{Load} \text{ Mem Reg } c_{\text{cur}} \text{ } r_{\text{addr}} \text{ } r_{\text{dest}} \]
\[ \mid \text{Store} \text{ Mem Reg } c_{\text{cur}} \text{ } r_{\text{addr}} \text{ } r_{\text{src}} \]
\[ \mid \text{Alloc} \text{ Mem Reg } c_{\text{cur}} \text{ } r_{\text{ptr}} \text{ } r_{\text{size}} \]

Data-flow events are just a proof artefact. They are emitted by any execution step that **modifies the memory or the register file**.
If the target context stashes a pointer, or recovers a pointer from the stash, the data-flow events will now reveal the sequence of operations that constitute this stashing/recovery.
If the target context stashes a pointer, or recovers a pointer from the stash, the data-flow events will now reveal the sequence of operations that constitute this stashing/recovery.

Data-Flow Back-Translation

maps each individual data-flow event to one or more source-language expression/statement(s).

↑  data-flow traces  →  source programs
Example: The target context stashes the pointer that is stored at shared address "a+5" in a private address "b".
Example: The target context stashes the pointer that is **stored at shared address "a+5"** in a **private address "b"**.

Remember: On the **interaction trace (standard trace-directed back-translation)**, this stashing will just appear as the silent label.
Data-Flow Events:
Data-Flow Events:

Reg
r_arg: a
r_loc: b

a a+5
Data-Flow Events:

\textbf{Mov Mem Reg'} c \ r\_arg \ r\_l

\begin{align*}
\text{Reg'} & \\
r\_arg &: a \\
r\_l &: a \\
r\_loc &: b
\end{align*}
Data-Flow Events:

**Mov** Mem Reg' c r_arg r_1

**Const** Mem Reg'' c 5 r_ct

Reg''

r_arg: a
r_1: a
r_loc: b
r_ct: 5
Data-Flow Events:

**Mov** Mem Reg' c r_arg r_1

**Const** Mem Reg'' c 5 r_ct

**BinOp** Mem Reg''' c add r_1 r_ct r_1
Data-Flow Events:

**Mov** Mem Reg' c r_arg r_1

**Const** Mem Reg'' c 5 r_ct

**BinOp** Mem Reg''' c add r_1 r_ct r_1

**Load** Mem Reg'''' c r_1 r_1

Reg''''

r_arg: a
r_1: ptr
r_loc: b
r_ct: 5
Data-Flow Events:

**Mov** Mem Reg' c r_arg r_1

**Const** Mem Reg'' c 5 r_ct

**BinOp** Mem Reg''' c add r_1 r_ct r_1

**Load** Mem Reg'''' c r_1 r_1

**Store** Mem' Reg'''' c r_loc r_1
Mov Mem Reg' c r_arg r_1
Const Mem Reg'' c 5 r_ct
BinOp Mem Reg''' c add r_1 r_ct r_1
Load Mem Reg'''' c r_1 r_1
Store Mem' Reg'''' c r_loc r_1
module Net {

    f (arg) {
        ...
        // tmp_loc points to Net-private memory
        Mov Mem Reg' c r_arg r_1
        Const Mem Reg'' c 5 r_ct
        BinOp Mem Reg'''' c add r_1 r_ct r_1
        Load Mem Reg'''' c r_1 r_1
        Store Mem' Reg'''' c r_loc r_1
    }
}
Data-Flow Back-Translation:

```plaintext
module Net {
    f (arg) {
        ...
        // tmp_loc points to Net-private memory
        Mov Mem Reg' c r_arg r_1
        Const Mem Reg'' c 5 r_ct
        BinOp Mem Reg''' c add r_1 r_ct r_1
        Load Mem Reg'''' c r_1 r_1
        Store Mem' Reg'''' c r_loc r_1
    }
}
```

Reserve one fixed source variable to simulate each target-language register.
Data-Flow Back-Translation:

module Net {

f (arg) {
  ...

  // tmp_loc points to Net-private memory

  Mov Mem Reg' c r_arg r_1
  Const Mem Reg'' c 5 r_ct
  BinOp Mem Reg''' c add r_1 r_ct r_1
  Load Mem Reg'''' c r_1 r_1
  Store Mem' Reg''''' c r_loc r_1
}
Data-Flow Back-Translation:

module Net {

    f (arg) {
        ...
        // tmp_loc points to Net-private memory
        tmp_1 := arg;
    }

    Const Mem Reg'' c 5 r_ct
    BinOp Mem Reg''' c add r_1 r_ct r_1
    Load Mem Reg'''' c r_1 r_1
    Store Mem' Reg''''' c r_loc r_1
}
Data-Flow Back-Translation:

module Net {

    f (arg) {

        ... 

        // tmp_loc points to Net-private memory
        tmp_1 := arg;

        tmp_ct := 5;

        BinOp Mem Reg'''' c add r_1 r_ct r_1
        Load Mem Reg'''' c r_1 r_1
        Store Mem' Reg'''' c r_loc r_1
    }

}
Data-Flow Back-Translation:

module Net {

  f (arg) {

    ... 

    // tmp_loc points to Net-private memory
    tmp_1 := arg;

    tmp_ct := 5;
    tmp_1 := tmp_1 + tmp_ct

    \[\begin{align*}
    \text{Load} & \quad \text{Mem} \quad \text{Reg}''''' \quad c \quad r_1 \quad r_1 \\
    \text{Store} & \quad \text{Mem'} \quad \text{Reg}''''' \quad c \quad r\_loc \quad r_1
    \end{align*}\]
module Net {
  f (arg) {
    ...
    // tmp_loc points to Net-private memory
    tmp_1 := arg;
    tmp_ct := 5;
    tmp_1 := tmp_1 + tmp_ct
    tmp_1 := *(tmp_1)
    ↑ [ Store Mem' Reg''' c r_loc r_1 ]
  }
}
Data-Flow Back-Translation:

```plaintext
module Net {
  f (arg) {
    ...
    // tmp_loc points to Net-private memory
    tmp_1 := arg;
    tmp_ct := 5;
    tmp_1 := tmp_1 + tmp_ct
    tmp_1 := *(tmp_1)
    *(tmp_loc) := tmp_1
    ...
  }
}
```
Data-Flow Back-Translation:

```java
module Net {

    f (arg) {
        ...
        // tmp_loc points to Net-private memory
        tmp_1 := arg;
        tmp_ct := 5;
        tmp_1 := tmp_1 + tmp_ct
        tmp_1 := *(tmp_1)
        *(tmp_loc) := tmp_1
        ...
    }
}

Stashing pointers is for free.

No need to implement a traversal of the whole reachable memory.
```
Data-Flow Back-Translation:

module Net {

   f (arg) {
      ...
      // tmp_loc points to Net-private memory
      tmp_1 := arg;
      tmp_ct := 5;
      tmp_1 := tmp_1 + tmp_ct
      tmp_1 := *(tmp_1)
      *(tmp_loc) := tmp_1
      ...
Data-Flow Back-Translation

supports **memory sharing** (without the need for graph traversal)
Data-Flow Back-Translation

- supports **memory sharing** (without the need for graph traversal)
- comes with a **mechanized** back-translation lemma in Coq (12k LoC)
Data-Flow Back-Translation

- supports **memory sharing** (without the need for graph traversal)
- comes with a **mechanized** back-translation lemma in Coq (12k LoC)
- works for **syntactically dissimilar** languages: a safe untyped **target** with **unstructured** control and a safe untyped **source** language with **structured** control
More in the paper

More in the paper


Our secure compilation proof allows reuse of whole-program compiler correctness lemmas (enabled by a novel turn-taking simulation).
Why reuse whole-program compiler correctness lemmas?

Some kind of a compiler correctness obligation usually shows up in a secure compilation proof.
Why reuse whole-program compiler correctness lemmas?

Some kind of a compiler correctness obligation usually shows up in a secure compilation proof.

If we hope to scale secure compilation proofs to a verified compiler, it will be easier to reuse rather than redo years-worth of manual proof effort.
Why reuse **whole-program compiler correctness** lemmas?

Some kind of a compiler correctness obligation usually shows up in a secure compilation proof.

If we hope to scale secure compilation proofs to a verified compiler, it will be easier to reuse rather than redo *years-worth of manual proof effort*.

**Whole-program** compiler correctness makes no assumptions about the **context** (because there is no context). So, a priori, there will be no difficulty in instantiating it (as opposed to partial-program correctness lemmas).
Summary: Proof technique for robust safety preservation

- Mechanized in Coq (approx. 30 kLoC)
- Supports languages with memory sharing
- Reuses whole-program compiler correctness lemmas
- Handles syntactically dissimilar target and source languages.

Backup
State-of-the-art reuse of whole-program compiler correctness lemmas in secure compilation

[Abate et al. 2018 "When good components go bad"]

Mechanized in Coq

Languages have static memory partition with only primitive values passable as arguments.

[El-Korashy et al. 2021 "CapablePtrs"]

Detailed technique but not machine checkable

Supports memory sharing by pointer passing
Scaled the proof of Abate et al. 2018 to languages with memory sharing

- Mechanized in Coq
- Languages have static memory partition with only primitive values passable as arguments.

- Detailed technique but not machine checkable
- Supports memory sharing by pointer passing

[Abate et al. 2018 "When good components go bad"]

[El-Korashy et al. 2021 "CapablePtrs"]
Scaled the proof of Abate et al. 2018 to languages with memory sharing

[Abate et al. 2018 "When good components go bad"]

Mechanized in Coq

Novel ternary turn-taking relation to support memory sharing.
(13k LoC in Coq)

[El-Korashy et al. 2021 "CapablePtrs"]

Detailed technique but not machine checkable

Supports memory sharing by pointer passing
Borrowed some intuitions from CapablePtrs

[Abate et al. 2018 "When good components go bad"]

**Key Ingredient**: Rely on a ternary relation.

[El-Korashy et al. 2021 "CapablePtrs"]

**Key Ingredient**: Rely on a ternary relation.
Borrowed some intuitions from CapablePtrs

[Abate et al. 2018 "When good components go bad"]

*Key Ingredient:* Rely on a ternary relation (called recomposition) between **three target-language executions**.

[El-Korashy et al. 2021 "CapablePtrs"]

*Key Ingredient:* Rely on a ternary relation (called TrICL) between **two target-language executions**, and a **third source execution**.
Borrowed some intuitions from CapablePtrs

Use ideas from the strong/weak binary similarity in CapablePtrs to make the ternary recomposition relation aware of memory sharing.
if not (cb_val.tag) then
   raise_c2_exception(CapEx_TagViolation, cb)
else if cb_val.sealed then
   raise_c2_exception(CapEx_SeamViolation, cb)
else if not (cb_val.permit_store) then
   raise_c2_exception(CapEx_PermitStoreViolation, cb)
else
{
   let size = wordWidthBytes(width);
   let cursor = getCapCursor(cb_val);
   let vAddr = (cursor + unsigned(rGPR(rt)) + size * signed(offset)) % pow2(64);
   let vAddr64= to_bits(64, vAddr);

   if (vAddr + size) > getCapTop(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
   else if vAddr < getCapBase(cb_val) then
      raise_c2_exception(CapEx_LengthViolation, cb)
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer)
        Net.receive();

        init_network:
        addi $r1 $r_arg 1024
        sw $r2 0($r1)
Setup: Secure compilation of partial programs

```
import module Net

module Main {
    char iobuffer[1024];
    static long int user_balance_usd;

    int main(void) {
        Net.init_network(iobuffer);
        Net.receive();
    }
}
```

**Compiler should ensure that the context CANNOT access the user balance**
Example: stash of the **Net** module

CapablePtrs [El-Korashy et al. 2021]
Example: stash of the **Net** module

CapablePtrs [El-Korashy et al. 2021]

Given a **target interaction trace** with **Net**

Call `?init_network`
Example: stash of the **Net** module

CapablePtrs [El-Korashy et al. 2021]

Given a **target interaction trace** with **Net**

Call ?*init_network*

Ret !
Example: stash of the **Net** module

CapablePtrs [El-Korashy et al. 2021]

Given a **target interaction trace** with **Net**

- Call ?*init_network*
- Ret !
- Call ?*receive*
Example: stash of the **Net** module
CapablePtrs [El-Korashy et al. 2021]

Given a *target interaction trace* with **Net**

```
Call ?init_network
Ret !
Call ?receive
Ret !
```
Example: stash of the Net module
CapablePtrs [El-Korashy et al. 2021]

Given a target interaction trace with Net

Find a source implementation of Net that emits a related interaction trace.
Example: stash of the **Net** module

CapablePtrs [El-Korashy et al. 2021]

Given a **target interaction trace** with **Net**

Before the **source** implementation of `init_network` returns, it stashes its argument in private memory, e.g. in a variable called `init_network_arg`. 
Example: stash of the **Net** module
CapablePtrs [El-Korashy et al. 2021]

Given a **target interaction trace** with **Net**

- **Call ?init_network**
- **Ret !**

Before **receive** returns, it uses the pointer stashed in **init_network_arg** to hardcode in the **iobuffer** all the **green values** that appeared on the **given trace**.
In general, must stash the whole shared memory CapablePtrs [El-Korashy et al. 2021]

The same function may have been called more than once:

\[
\text{init}_\text{network}_\text{arg}_1\_\text{c1}, \quad \text{init}_\text{network}_\text{arg}_2\_\text{c1}, \quad \ldots \quad \text{init}_\text{network}_\text{arg}_n\_\text{c1}
\]
In general, must stash the whole shared memory

CapablePtrs [El-Korashy et al. 2021]

The same function may have been called more than once:

init_network_arg_1\_c1, init_network_arg_2\_c1, \ldots, init_network_arg_n\_c1

init_network_arg_1\_c2, init_network_arg_2\_c2, \ldots, init_network_arg_m\_c2