Translation Validation for Security Properties
3rd Workshop on Principles of Secure Compilation

Matteo Busi, Pierpaolo Degano, Letterio Galletta

13/01/2018
1. Motivations
2. The key idea
3. Open questions and future
4. Conclusions
What I going to present is still a *work in progress*!

It goes without saying that any suggestion, hint, advice and collaboration offer will be greatly appreciated! 😊

That said,

let’s start!
Secure compilation

Given:

- a compiler $\mathcal{J}$
- a program $S$ that enjoys some (class of) security properties for any $C_S[\cdot]$

What can we say about the security of $T = [S]$ in a context $C_T[\cdot]$?

- ...w. (some) restriction on $[\cdot]$? Many things
- ...w/o restrictions? Not much!
A DIFFERENT PERSPECTIVE

Consider

\[ S \triangleq \lambda i. \text{if } i \geq 0 \text{ then } (\text{print } i; i) \text{ else } (-1) \]

If no networking primitives

\[ \Downarrow \]

“the program never sends a value to the network” holds in any context.

but compiling it to \( T \)

\[ \lambda i. \text{if } i \geq 0 \text{ then } (\text{sc\_print } i; i) \text{ else } (-1) \]

in a language where contexts \text{support} networking, breaks the property 😞
...or does not?

Strictly speaking there exists a target context that breaks it:

\((\lambda i.\text{let } \text{sc\_print } = \lambda x.(\text{display } x; \text{ send } x) \text{ in } \[\cdot \] i) 42\)

So, YES!
...or does not?

Strictly speaking there exists a target context that breaks it:

\[
(\lambda i. \text{let } \text{sc\_print} = \lambda x.(\text{display } x; \text{ send } x) \text{ in } \cdot i) 42
\]

So, YES!

But many others do not, e.g. those not using networking \textit{at all}!
Secure translation validation (STV)

So, what can we say \textit{a-posteriori} about a specific $T$ and a given context?

\begin{align*}
\text{Behav. of } S & \rightarrow \text{Anlyser} & \text{Bad} & \rightarrow \text{Counter example} \\
C_T[\cdot] & \rightarrow \text{Anlyser} & \text{Good} & \rightarrow \text{Proof script} \\
T \triangleq [S] & \rightarrow \text{Anlyser}
\end{align*}

Our goal is to implement an STV via load time analysis that decides automatically if a given family of properties has been preserved.
Consider the *robustly safe compilation (RSC)*

\[
\forall S, C_T[\cdot], m. \ (m \in \text{Pref}(C_T[[S]])) \Rightarrow \exists C_S[\cdot]. \ m \in \text{Pref}(C_S[S])
\]

Can we use it in our STV approach?
Indeed!

We can get rid of

- $\forall S$: just a program at a time
- $\forall C_T[\cdot]$: load-time analysis

And the analyser can *just*:

1. Take $T$ and plug it into $C_T[\cdot]$
2. Searching for a $C_S[\cdot]$ making the condition above to hold!

This is not easy (nor decidable 😞)!
We can resort to *history expressions*:

- Practically: their semantics safely over-approximates (interesting parts of) the behaviour of a program
- Formally: processes of a basic process algebra
- Most importantly: they can be obtained automatically (e.g. via a T&E system)
Their syntax usually looks like:

\[ H ::= \epsilon \mid H_1 + H_2 \mid H_1 \cdot H_2 \mid \mu h. H \mid o \in \mathcal{O} \]

- Models choice, sequencing, recursion and observable actions (possibly different in source and target)
- The semantics of a string in $H$ is a superset of the traces of the program
- Histories are finite-state processes
So, we get:

Now, the analyser just need to check whether $H_{CT[T]}$ can be back-translated to a source language’s history expression.
Consider:

```
let p = read () in
if p = REAL_P then
  (* ... *); print true
else
  (* ... *); print false
```

```
let p = sc_read () in
let r = (p = REAL_P) in
if r then
  (* ... *)
else
  (* ... *);
sc_print r
```

Under the assumptions that:

- **Source language** and contexts just has primitives for I/O
- **Target language** uses system calls, given by the context using its primitives

For any source context the source program satisfies:

"does not send anything on the network".
Can we say the same for the target?

Consider $C_T[\cdot]$:

```latex
let sc\_read = \lambda (\cdot). \text{read} (\cdot) in
let sc\_print = \lambda x. \text{display} x; \text{send} x in
\cdot
```

from which $H_{C_T[T]}$:

```
read \cdot display \cdot send
```

that has no counterpart in the source histories!

**Bad context!**
Instead $C_T'$:

\[
\begin{align*}
&\text{let } \text{sc\_read} = \lambda () . \text{read} () \text{ in} \\
&\text{let } \text{sc\_print} = \lambda x . \text{display} x \text{ in} \\
&[\cdot]
\end{align*}
\]

from which $H_{C_T'[T]}$:

\[
\text{read} \cdot \text{display}
\]

that has a counterpart in the source histories!

**Good context!**
Recall: histories safely overapproximate the behaviour of programs:

- If the analysis tells that a transformation is “secure”, we are sure it is
- Otherwise, it might be still “secure” but our analysis cannot prove it!

For example:

```plaintext
let p = read () in
if 42 < 0 then
  print true
else
  (* ... *)
```

⇝

```plaintext
let p = sc_read () in
if 42 < 0 then
  sc_print true
else
  (* ... *)
```
For the context:

```ocaml
let sc_read = \(). read () in
let sc_print = \x. display x; send x in
```

- Still no counterpart in the source
- **But** the target program satisfies the property above!

Good context, imprecise analysis!
Open questions and future

We are confident that our approach is worth, still

- A complete formalisation of our idea is missing
- Up to now just simple functional languages, i.e.
  - What happens when adding complex features to the language? (e.g. more complex types?)
  - What happens when changing the power of the attacker?
- For simplicity we just considered simple observables,
  - How hard is to remove some false alarms raising from that?
  - Are they always informative enough?
- How hard is to extend the approach to other secure compilation principles?
We proposed an approach with the goal of implementing a *secure translation validation* for RSC:

- Doing that by
  - Computing, at load time, a safe over approximation of the target program plugged in the target context
  - Backtranslating it to a source history expression
    - If the source history exists, we proved that the transformation preserved safeties
- The whole approach works because of a chain of implications
- We intuitively shown that such an approach is viable
- Of course, there still a lot to be done: hopefully new ideas after this workshop! 😊
THE END

Matteo Busi
matteo.busi@di.unipi.it
University of Pisa