A Brief Tour of
Formally Secure Compilation

Matteo Busi

Letterio Galletta

Università di Pisa

IMT Scuola Alti Studi Lucca
Overview

1. Motivations
2. Defining security
3. Secure compilers
4. Robustly secure compilers
5. Low-level enforcement mechanisms
6. Open problems and challenges
7. Conclusions
Compiler security?

**Informally:** we can say that a compiler is secure if it preserves the security properties of the source programs.

- Should we even care?
- Is this a real-world and relevant thing?

Well...
Compiler security?

**Informally:** we can say that a compiler is secure if it preserves the security properties of the source programs.

- Should we even care?
- Is this a real-world and relevant thing?

Well...
Consider the snippet

```java
pin := read_secret();
if (check(pin))
    // OK!
pin := 0; // overwrite the pin
```

This dead-store preserves the semantics
- But surprisingly breaks the security at the target level
Consider the snippet

```plaintext
code
pin := read_secret();
if (check(pin))
    // OK!
pin := 0; // overwrite the pin
```

This dead-store preserves the semantics

But surprisingly breaks the security at the target level
Consider the snippet

```c
pin := read_secret();
if (check(pin))
    // OK!
pin := 0; // overwrite the pin
```

This dead-store preserves the semantics

But *surprisingly* breaks the security at the target level
Security...

How can we even define security for programs?

Trace properties, i.e. safety and liveness properties

Hyperproperties, e.g. non-interference
Security...

How can we even define security for programs?

*Trace properties*, i.e. safety and liveness properties *Hyperproperties*, e.g. non-interference
Security...

How can we even define security for programs?

Trace properties, i.e. safety and Hyperproperties, e.g. non-interference liveness properties
Security: trace properties

- Trace property: the set of *admissible* traces, i.e. sequences of *interesting events* that a program generates.

Consider:

```plaintext
x := read();
for (i = 0; i < x; i++)
  print (i);
```

Then, for \( x = 10 \):

```
out(0)
  ...
out(8)
out(9)
```

- *Events*: outputs
- *Trace property*: admit all traces w/o the generated trace is admissible
Security: trace properties

- Trace property: the set of *admissible* traces, i.e. sequences of *interesting events* that a program generates.

Consider:

```plaintext
x := read();
for (i = 0; i < x; i++)
  print (i);
```

- *Events*: outputs

- *Trace property*: admit all traces w/o

Then, for \( x = 10 \):

```plaintext
out(0)
\vdots
out(8)
out(9)
```

the generated trace is admissible
Security: trace properties

- Trace property: the set of admissible traces, i.e. sequences of interesting events that a program generates.

Consider:

```plaintext
x := read();
for (i = 0; i < x; i++)
    print (i);
```

- Events: outputs

- Trace property: admit all traces w/o out(42)

Then, for $x = 10$:

```plaintext
out(0)
:
out(8)
out(9)
```

the generated trace is admissible
Security: trace properties

▶ Trace property: the set of admissible traces, i.e. sequences of interesting events that a program generates.

Consider:

\[
\begin{align*}
x & := \text{read}(); \\
\text{for} \ (i = 0; i < x; i++) \\
& \quad \text{print} (i);
\end{align*}
\]

▶ Events: outputs
▶ Trace property: admit all traces w/o out(42)

However for \(x = 44\):

\[
\begin{align*}
\text{out}(0) \\
\quad \vdots \\
\text{out}(42) \\
\text{out}(43)
\end{align*}
\]

the generated trace is not admissible.
Hyperproperty: a set defining the sets of admissible traces.

Consider:

```plaintext
x := read_public()
y := read_secret()
// ...
print (y);
```

Events: I/O

Hyperproperty: non-interference

Hence the pair of traces

- in(6), in(42) …, out(42)
- in(6), in(7) …, out(7)

same low inputs, different low outputs: the snippet does not enjoy non-interference!
Security: hyperproperties

- **Hyperproperty**: a set defining the sets of *admissible* traces.

Consider:

```plaintext
x := read_public()
y := read_secret()
// ...
print (y);
```

- **Events**: I/O

- **Hyperproperty**: non-interference

Hence the pair of traces

- `in(6), in(42) ..., out(42)`
- `in(6), in(7) ..., out(7)`

same low inputs, different low outputs: the snippet does not enjoy non-interference!
Security: hyperproperties

- Hyperproperty: a set defining the sets of *admissible* traces.

Consider:

```
x := read_public();
y := read_secret();
// ...
print(y);
```

Hence the pair of traces

- \(\text{in}(6), \text{in}(42) \ldots, \text{out}(42)\)
- \(\text{in}(6), \text{in}(7) \ldots, \text{out}(7)\)

same *low inputs*, different *low outputs*: the snippet *does not* enjoy non-interference!

- *Events*: I/O
- *Hyperproperty*: non-interference
Compiler security, then...

- Consider a family $\mathbb{F}$ of trace properties/hyperproperties
- A compiler $[\cdot]_O^S$ is secure for $\mathbb{F}$ iff for all $p$

\[
\forall \mathcal{F} \in \mathbb{F}. \ p \models \mathcal{F} \Rightarrow [p]_O^S \models \mathcal{F}
\]

This concept is *nicely* linked with compiler correctness:
- a correct compiler preserves *all* the trace properties!
- but also *all* the subset-closed hyperproperties!
Compiler security, then...

- Consider a family $\mathcal{F}$ of trace properties/hyperproperties
- A compiler $\llbracket \cdot \rrbracket^S_O$ is secure for $\mathcal{F}$ iff for all $p$

$$\forall \mathcal{F} \in \mathcal{F}. \; p \models \mathcal{F} \Rightarrow \llbracket p \rrbracket^S_O \models \mathcal{F}$$

This concept is *nicely* linked with *compiler correctness*:
- a correct compiler preserves *all* the trace properties!
- but also *all* the subset-closed hyperproperties!
Attackers...where are they?

So far everything seems to be nice, but we have not considered any active attacker!

Idea:

Attacker as context, e.g. an untrusted OS providing syscalls to the programs.
Attackers...where are they?

So far everything seems to be nice, but we have not considered any active attacker!

Idea:

Attacker as context, e.g. an untrusted OS providing syscalls to the programs.
Consider again a family $\mathcal{F}$ of trace properties/hyperproperties.

A compiler $\llbracket \cdot \rrbracket_O^S$ is robustly secure for $\mathcal{F}$ iff for any source program $p$,

$$\forall \mathcal{F} \in \mathcal{F}. \ (\forall C_S[\cdot]. \ C_S[p] \models \mathcal{F}) \Rightarrow (\forall C_O[\cdot]. \ C_O[\llbracket p \rrbracket_O^S] \models \mathcal{F})$$

Robustly secure compilers are the subject of a very active research area:

- historically: mainly about full abstraction
- currently: many different principles
- more details in the paper, much more in the literature!
Consider again a family $F$ of trace properties/hyperproperties.

A compiler $[[\cdot]]_O$ is robustly secure for $F$ iff for any source program $p$

$$\forall F \in F. (\forall C_S[\cdot]. C_S[p] \models F) \Rightarrow (\forall C_O[\cdot]. C_O[[p]]_O \models F)$$

Robustly secure compilers are the subject of a very active research area:

- historically: mainly about full abstraction
- currently: many different principles
- more details in the paper, much more in the literature!
Low-level enforcement mechanism

Changing the compiler to make it secure is *just* one of the options! Another is to change the object language and make it more secure:

- TAL
- Capability machines
- Protected module architectures
- Micro-tagged architectures
Open problems and challenges

Indeed, many open problems in the area

- Principles and proof techniques to help the designers to prove the security of their compilers
- Automatic validation techniques for existing (and realistic!) compilers
- Secure compilation in case of separate compilation
Conclusions

Summarising:

- Current, real-world compilers do not preserve security properties
- First take: *correct compilation* and *secure compilation*
- A more realistic approach: *robust secure compilation*
- A re-design of low level mechanisms helps

Since this is a relatively young field:

- Many open problems and questions yet to be tackled
- Interesting for people from Cybersecurity, PL and Verification!
- **Our goal**: help newcomers to grasp the basics of the field!
Conclusions

Summarising:

- Current, real-world compilers do not preserve security properties
- First take: *correct compilation* and *secure compilation*
- A more realistic approach: *robust secure compilation*
- A re-design of low level mechanisms helps

Since this is a relatively young field:

- Many open problems and questions yet to be tackled
- Interesting for people from Cybersecurity, PL and Verification!
- **Our goal:** help newcomers to grasp the basics of the field!
The End
All the principles in a nutshell

See https://arxiv.org/abs/1807.04603
Full abstraction

See https://doi.org/10.1007/3-540-48749-2_2

A compiler $\llbracket \cdot \rrbracket^S_o$ is fully abstract iff

$$\forall p_1, p_2 \in S. p_1 \simeq p_2 \iff \llbracket p_1 \rrbracket^S_o \simeq \llbracket p_2 \rrbracket^S_o.$$

Nice, but:

- Not always satisfactory, see the paper and Patrignani & Garg, 2017 (https://ieeexplore.ieee.org/document/8049734)
- Sometimes hard to either prove or disprove
Robust safety property preservation

See https://arxiv.org/abs/1807.04603

A compiler $\llbracket \cdot \rrbracket^S_0$ robustly preserves all the safety properties iff

$$\forall p. \forall C_0[\cdot]. \forall m. (m \text{ finite trace prefix of } C_0[p]) \Rightarrow (\exists C_S[\cdot]. m \text{ finite trace prefix of } C_S[p])$$
Robust hyperproperty preservation

See https://arxiv.org/abs/1807.04603

A compiler $\lbrack \cdot \rbrack^S_o$ robustly preserves all the hyperproperties in a family $\mathcal{F}$ iff

$$\forall \mathcal{F} \in \mathcal{F}. \forall p. (\forall C_S. C_S[p] \models \mathcal{F}) \Rightarrow (\forall C_o. C_o[\lbrack p \rbrack^S_o] \models \mathcal{F})$$