Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation
Good programming languages provide helpful abstractions for writing more secure code

- structured control flow, procedures, modules, interfaces, correctness and security specifications, ...

abstractions not enforced when compiling and linking with adversarial low-level code

- all source-level security guarantees are lost
We need secure compilation chains

• **Protect source-level abstractions even against linked adversarial low-level code**
  – various enforcement mechanisms: processes, SFI, capabilities, tagged architectures, ...
  – shared responsibility: compiler, linker, loader, OS, HW

• **Goal: enable source-level security reasoning**
  – linked adversarial target code cannot break the security of compiled program any more than some linked source code
  – no "low-level" attacks introduced by compilation
Robustly preserving security

∀ source context

source program

∀ target context

compiled program

protected

no extra power

But what should "secure" mean?
What properties should we robustly preserve?

hyperproperties (noninterference)
+ data confidentiality

trace properties (safety & liveness)
only integrity

relational hyperproperties (trace equivalence)
+ code confidentiality

No one-size-fits-all security criterion

More secure

More efficient
to enforce
Easier to prove
Robust Trace Property Preservation

property-based characterization

\[ \forall P. \forall \pi \in 2^{\text{Trace}}. \left( \forall C_S \ t. C_S[P] \sim t \Rightarrow t \in \pi \right) \]
\[ \Rightarrow \left( \forall C_T \ t. C_T[P \downarrow] \sim t \Rightarrow t \in \pi \right) \]

**what** one might want to achieve

\[ \iff \]

property-free characterization

\[ \forall P \ \forall C_T \ \forall t. C_T[P \downarrow] \sim t \Rightarrow \exists C_S. C_S[P] \sim t \]

**how** one can prove it
Some of the proof difficulty is manifest in property-free characterization

∀C_T ∃C_S ∀P ∀t...

back-translating context prog & context
∀P ∀C_T ∃C_S ∀t...

back-translating prog & context
∀P ∀C_T ∃C_S ∀t...

back-translating finite trace prefixes
∀k ∀P_1...P_k ∀C_T ∃C_S...

∀m_1...m_k ∃C_S...
Journey Beyond Full Abstraction [CSF 2019]

- Thoroughly explored secure compilation criteria based on robust property preservation
- Carefully studied the criteria and their relations
  - Property-free characterizations
  - Implications, collapses, separations results
- Extended diagram to arbitrary trace relations [ESOP 2020]
- Helped better understand full abstraction and its limitations
- Embraced and extended full abstraction proof techniques

rest of this talk
Extended this to arbitrary trace relations

Source and target traces connected by arbitrary relation

- Undefined behavior (CompCert):
  \[ t_S \sim t_T \iff t_S = t_T \lor (\exists m \leq t_T. \ t_S = m \cdot \text{Goes\_wrong}) \]

- Resource exhaustion (CakeML):
  \[ t_S \sim t_T \iff t_S = t_T \lor (\exists m \leq t_S. \ t_T = m \cdot \text{Resource\_limit\_hit}) \]

- Different values, Side-channels, IO granularity, etc.

Interesting for secure compilation & compiler correctness

Main question: how are source/target properties related?
Extending Robust Trace Property Preservation

property-free characterization

\[ \forall P. \forall C_T \forall t_T. C_T[P \downarrow] \sim t_T \Rightarrow \exists C_S. \exists t_S \sim t_T. C_S[P] \sim t_S \]

\[ \forall P. \forall \pi_S. \]
\[ (\forall C_S. C_S[P] \models \pi_S) \Rightarrow (\forall C_T. C_T[P \downarrow] \models \tau_\sim(\pi_S)) \]

\[ \tau_\sim(\pi_S) = \text{target guarantee} \]
\[ (\text{existential image of } \sim) \]

\[ \forall P. \forall \pi_T. \]
\[ (\forall C_S. C_S[P] \models \sigma_\sim(\pi_T)) \Rightarrow (\forall C_T. C_T[P \downarrow] \models \pi_T) \]

\[ \sigma_\sim(\pi_T) = \text{source obligation} \]
\[ (\text{universal image of } \sim) \]

\[ \tau_\sim \rightleftharpoons \sigma_\sim \]
\[ (\text{Galois connection}) \]

property-full characterization

2 equivalent property-full characterizations
Where is Full Abstraction?
(i.e. robust behavioral equivalence preservation)

without internal nondeterminism, full abstraction is around here
doesn't imply any other criterion
Full abstraction **does not imply** any other criterion in our diagram

- **Intuitive counterexample** adapted from Marco&Deepak [CSF'17]
- **When target context passes in bad input value** (e.g. ill-typed) the compiled program:
  - *lunches the missiles* - breaks Robust Safety Preservation
  - or *loops forever* - breaks Robust Liveness Preservation
  - or *leaks secret inputs* - breaks Robust NI Preservation
- **Yet this doesn't break** full abstraction or compiler correctness!
- Full abstraction only ensures **code confidentiality**
  - *no* integrity, *no* safety, *no* data confidentiality, ...
It's actually a bit more subtle than this ...

• Seems that sometimes one can ensure that FA implies RTINIP
  – Full abstraction ensures program confidentiality, so make secrets part of the "data section" of the program [Busi et al, CSF 2020]
  – Would be good to formalize this, even if it's a very indirect way to get RTINIP

• FA implies RHP~ [Abate & Busi, FCS 2020]
  – but only for crazy ~ depending on the compiler, which is thus still in the TCB!

• All full abstraction results have the compiler in their TCB
  – For any two languages, there exists a fully abstract compiler! [Parrow, MSCS 2014] [Gorla & Nestmann, MSCS 2014]

• Still unclear to what extent full abstraction makes sense as a criterion for secure compilation
  – Fortunately now we have many other criteria
Embraced and extended™ proof techniques

for simple translation from statically to dynamically typed language with first-order functions and I/O

strongest criterion achievable

back-translating context by universal embedding

∀C_T ⊆ C_S ∀P ∀t...

[New et al, ICFP'16]

generic technique

back-translating finite set of finite trace prefixes

∀k ∀P_1..P_k ∀C_T

∀m_1..m_k ⊆ C_S...

[Jeffrey & Rathke, ESOP'05]
[Patrignani et al, TOPLAS'15]
Future directions

• Achieving provably secure interoperability with low-level code in practice
  – realistic languages and secure compilation chains

• More scalable proof techniques

• More trustworthy secure compilation proofs
  – for correct compilation all proofs are machine checked, why should this be any different for secure compilation?

• Verifying robust satisfaction for source programs