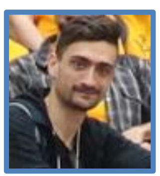
Journey Beyond Full Abstraction:

Exploring Robust Property Preservation for Secure Compilation



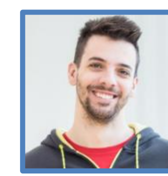
Carmine Abate
MPI-SP Bochum

Rob Blanco
MPI-SP Bochum

Deepak Garg
MPI-SWS Saarbrücken

Cătălin Hrițcu
MPI-SP Bochum

Jérémy Thibault
MPI-SP Bochum



Marco Patrignani
Stanford & CISPA

Good programming languages provide helpful abstractions for writing more secure code

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

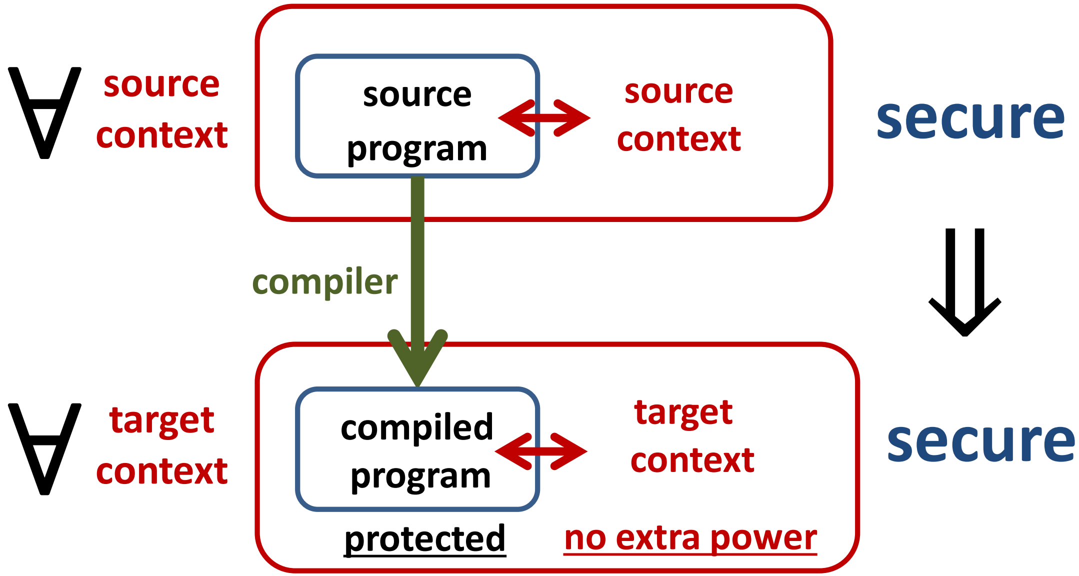
abstractions not enforced when compiling and linking with adversarial low-level code

- 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



But what should "secure" mean?

What properties should we robustly preserve?

No one-size-fits-all security criterion

relational hyperproperties
(trace equivalence)

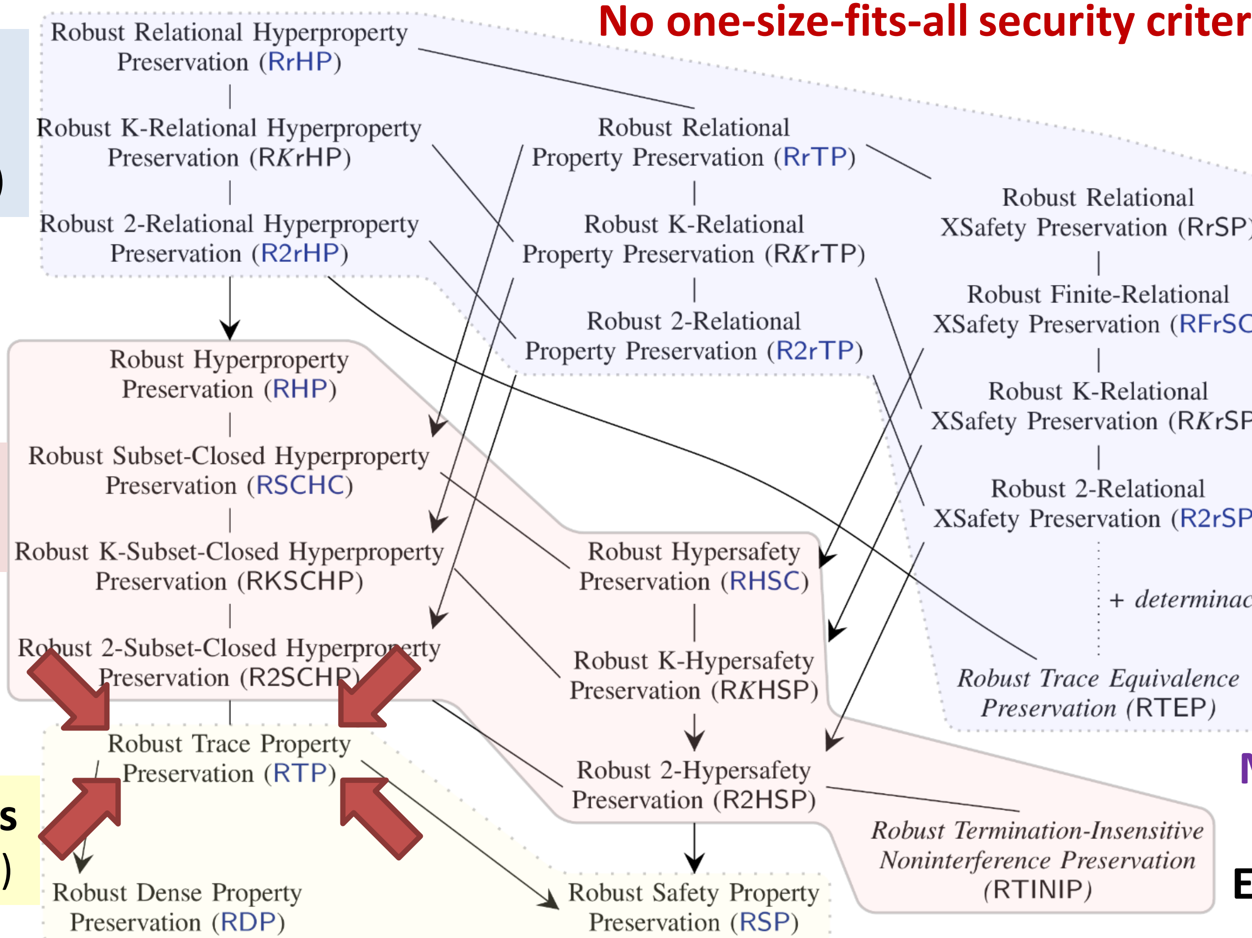
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



More secure

More efficient to enforce

Easier to prove

Robust Trace Property Preservation

property-based characterization

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

what one might want to achieve

$\Leftrightarrow$

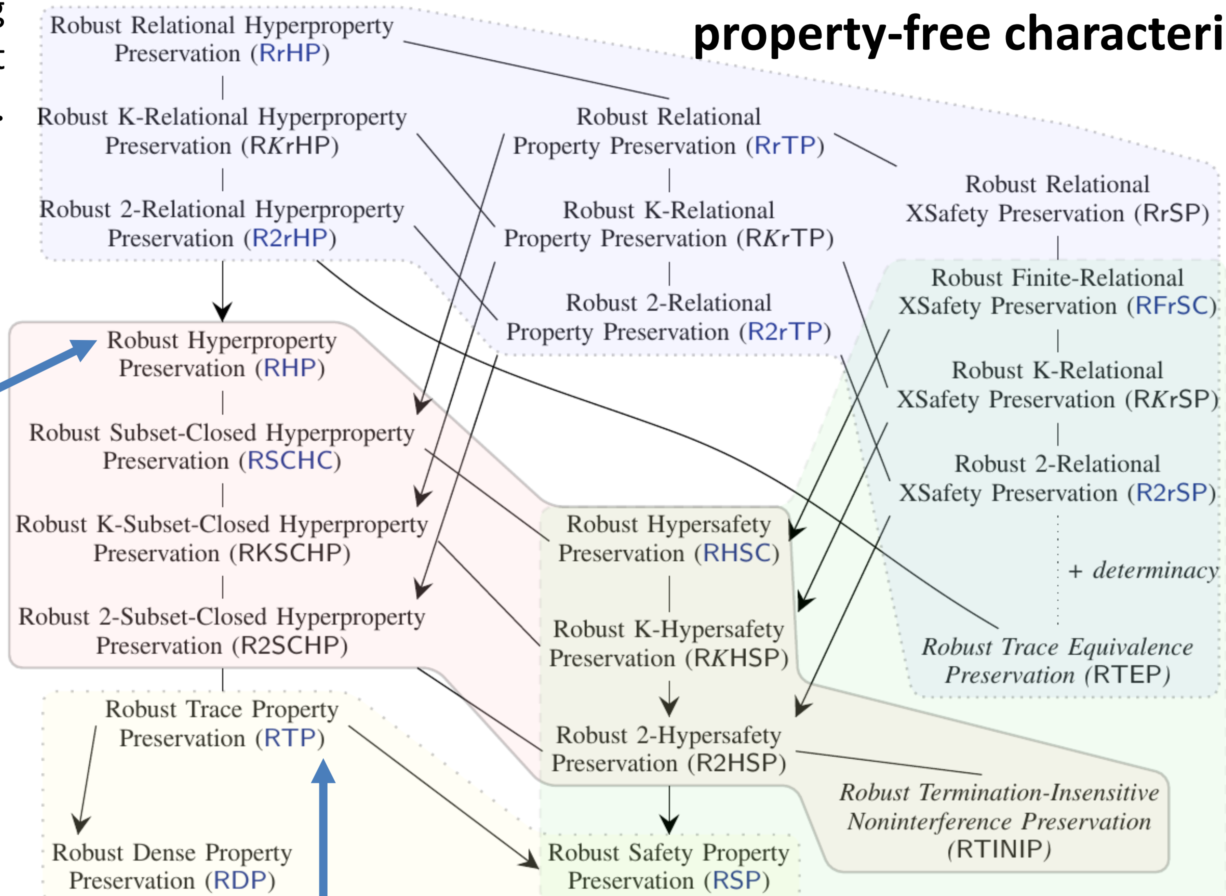
property-free characterization

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

how one can prove it

Some of the proof difficulty is manifest in property-free characterization

back-translating context
$\forall C_T \exists C_S \forall P\ \forall t\ldots$



back-translating finite set of finite trace prefixes
$\forall k \forall P_1 \ldots P_k \forall C_T$
$\forall \mathbf{m}_1 \ldots \mathbf{m}_k\ \exists C_S \ldots$

back-translating prog & context
$\forall P\ \forall C_T \exists C_S \forall t\ldots$

back-translating prog & context & trace
$\forall P\ \forall C_T \forall t\ \exists C_S \ldots$

back-translating finite trace prefix **m**
$\forall P\ \forall C_T \forall \mathbf{m}\ \exists C_S \ldots$

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

[ESOP 2020]

- **Source and target traces connected by arbitrary relation**
  - **Undefined behavior** (CompCert):
    $t_S \sim t_T \Leftrightarrow t_S = t_T \vee (\exists m \leq t_T.\ t_S = m \cdot \mathrm{Goes\_wrong})$
  - **Resource exhaustion** (CakeML):
    $t_S \sim t_T \Leftrightarrow t_S = t_T \vee (\exists m \leq t_S.\ t_T = m \cdot \mathrm{Resource\_limit\_hit})$
  - **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

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

⇙⇗ ⇖⇘

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

⇔

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

$\tau_\sim(\pi_S)$ = target guarantee (existential image of ~)

$\sigma_\sim(\pi_T)$ = source obligation (universal image of ~)

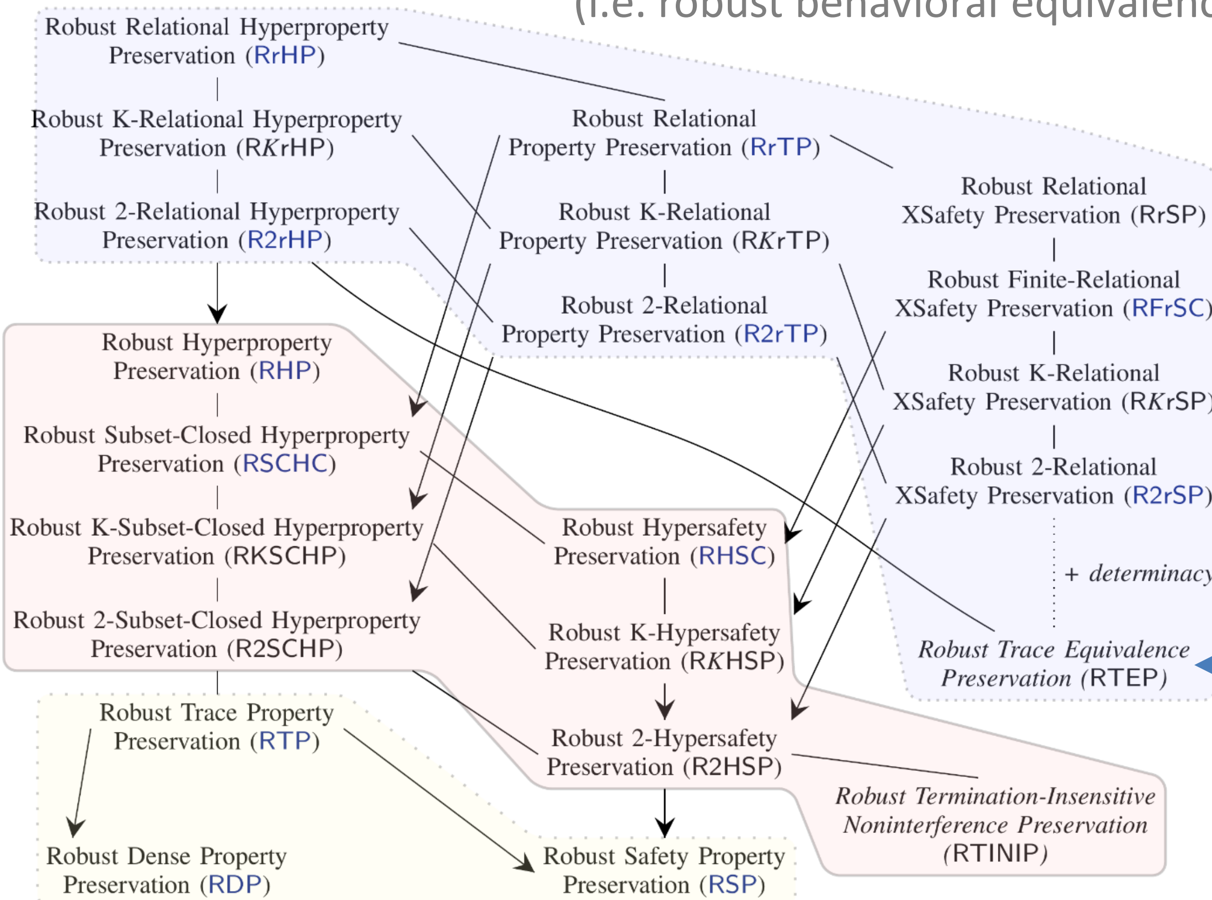
$\tau_\sim \rightleftarrows \sigma_\sim$
(Galois connection)

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

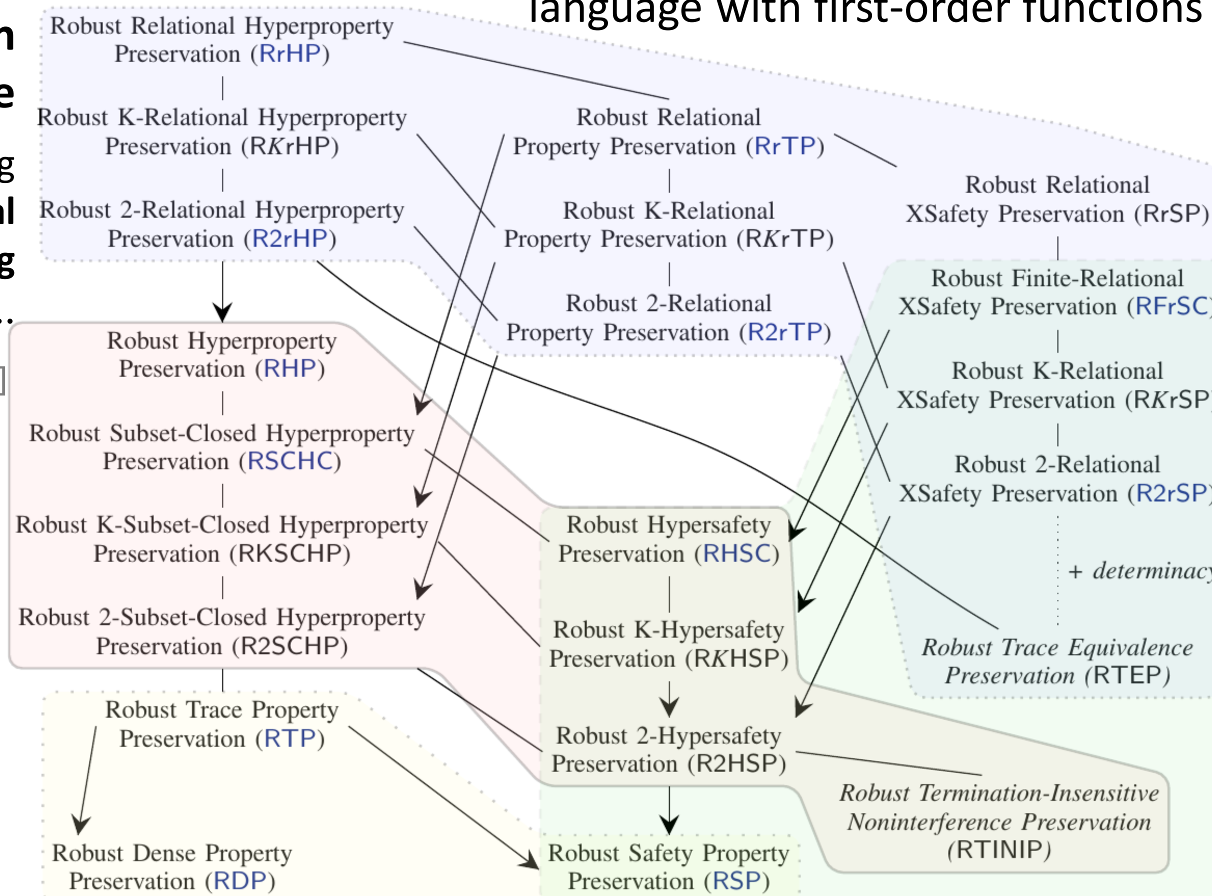
for simple translation from statically to dynamically typed language with first-order functions and I/O

strongest criterion achievable

back-translating context by **universal embedding**

$\forall C_T \exists C_S \forall P\ \forall t$...

[New et al, ICFP'16]



generic technique

back-translating finite set of finite trace prefixes

$\forall k \forall P_1..P_k \forall C_T$
$\forall m_1..m_k \exists C_S$...

[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**