How to *define* compiler security?

(A property-centric view)

Ongoing work with:

Catalin Hritcu, **Marco Patrignani**, Jeremy Thibault, Carmine Abate, **Roberto Blanco** + others

An overview

Source

P

Program

Target

P↓

Program

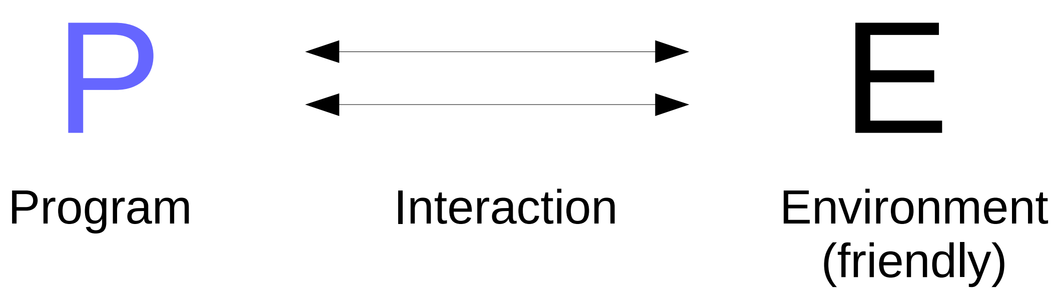
Correctness: Preserve P's properties

If P has a property, then P↓ has the property.

Security: Preserve P's properties despite adversaries

If P has a property despite all source adversaries, then P↓ has the property despite all target adversaries.

Observable behavior as traces



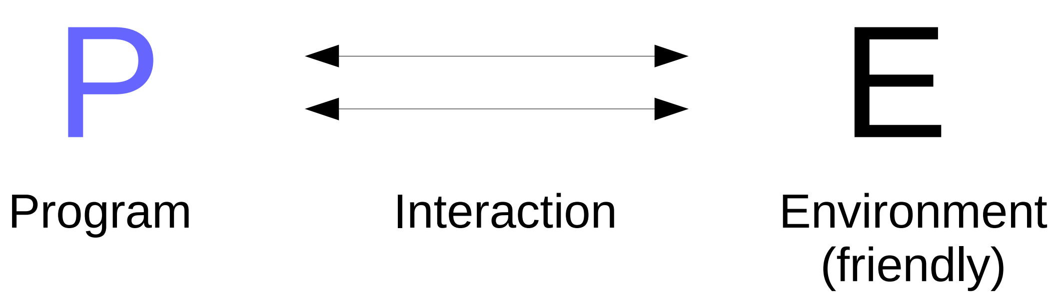
- Trace: Sequence of actions representing behavior observed by environment.

  ?0 !0 ?5 !5 ?2 !7 ?3 !10 …

  ?1 !1 ?2 !2 ?3 !6 ?7 !42 ?8 !(Halt)

- Tr(P) = Set of all traces of P
- Tr(P) describes P’s observable behavior
- Does not represent hidden state and unobservable ways of interacting with P

Properties as sets of traces



- Property/specification π: Set of good traces

  $\pi = \{?a_0\ !b_0\ ?a_1\ !b_1\ \ldots \mid b_i >= 0\}$

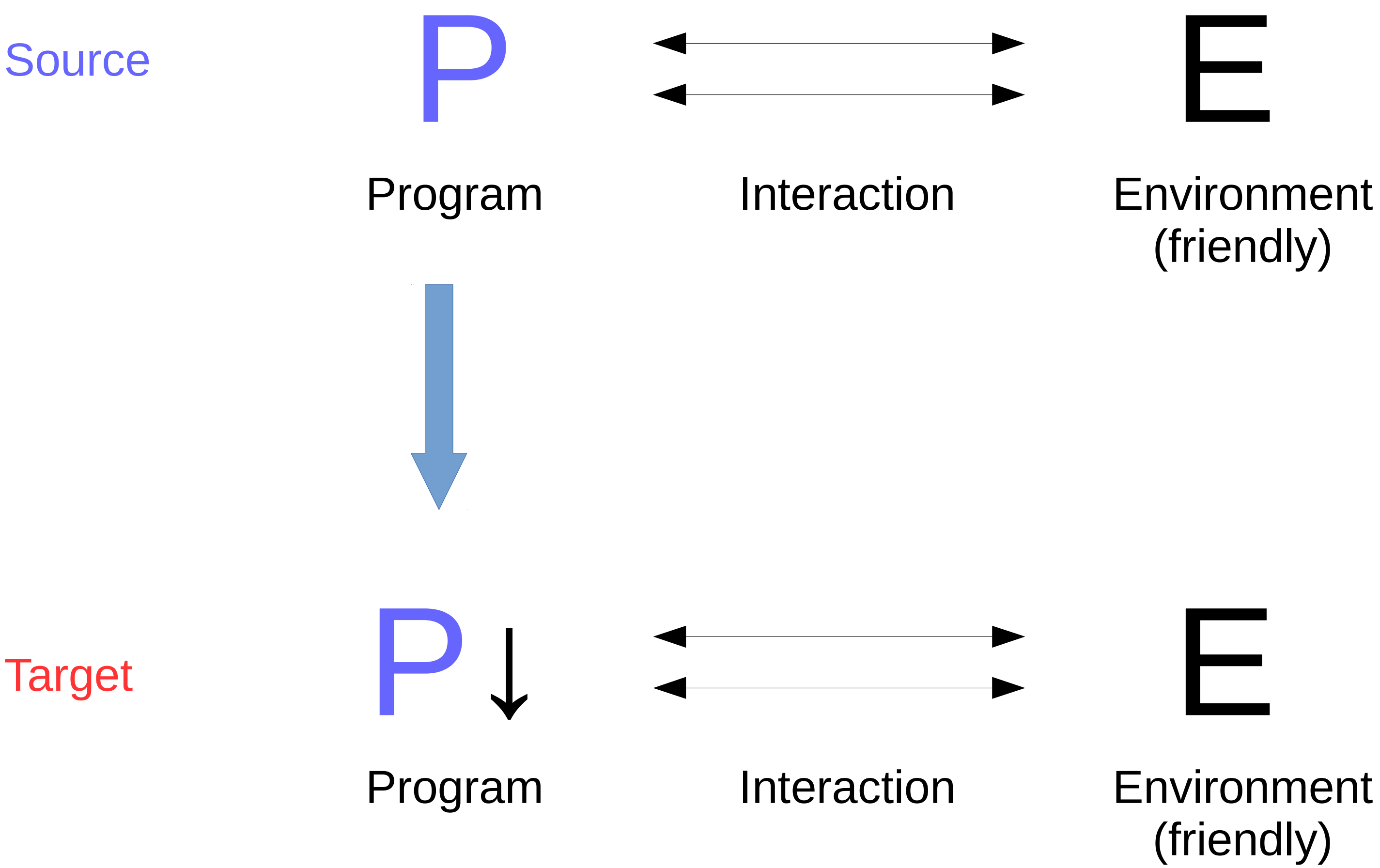
  (Non-negative outputs only)

  $?0\ !0\ ?5\ !5\ ?2\ !7\ \ldots \in \pi$

  $?0\ !0\ ?5\ !(-5)\ \ldots \notin \pi$

- P has property π iff $Tr(P) \subseteq \pi$

Compilation



Compiler <u>correctness</u> as property preservation

A compiler ↓ is <u>correct</u> if it *preserves all properties:*

$$\forall \pi.\ \forall P.\ (P \text{ has } \pi) \Rightarrow (P\!\downarrow \text{ has } \pi)$$

<u>Key idea</u>
If we checked (manually or by verification) that the source program has π, then there is no need to re-verify the compiled program.

Compiler correctness: Equivalent characterization

- A compiler $\downarrow$ is <u>correct</u> if it *preserves all properties*

  $$\forall \pi.\ \forall P.\ (P \text{ has } \pi) \Rightarrow (P\downarrow \text{ has } \pi)$$

- Equivalently,

  $$\forall \pi.\ \forall P.\ (Tr(P) \subseteq \pi) \Rightarrow (Tr(P\downarrow) \subseteq \pi)$$

- Equivalently,

  $$\forall P.\ Tr(P\downarrow) \subseteq Tr(P)$$

  This last statement is called “refinement”

<u>Theorem</u>
A compiler *preserves all properties* iff it *refines* behaviors (only reduces traces)

Compiler correctness: Equivalent characterization

- A compiler ↓ is <u>correct</u> if it *preserves all properties*

$$\forall \pi.\ \forall P.\ (P \text{ has } \pi) \Rightarrow (P{\downarrow} \text{ has } \pi)$$

A property-based characterization

A property-free characterization

- Equivalently,

$$\forall P.\ \mathrm{Tr}(P{\downarrow}) \subseteq \mathrm{Tr}(P)$$

Amenable to formal proof

Security!

Behavior as traces, with adversaries

$$A \parallel P \longleftrightarrow E$$

Adversary (malicious) | Program | Interaction | Environment (friendly)

- A may be:
  - A library co-linked with P (corrupts internal state of P)
  - A malicious network client (provides exploit inputs)
- A's interactions use a possibly different/broader API than E's interactions
- A's interactions are *not* what we want to specify, so they are not represented on the traces (gives compiler leeway)

Behavior as traces, with adversaries

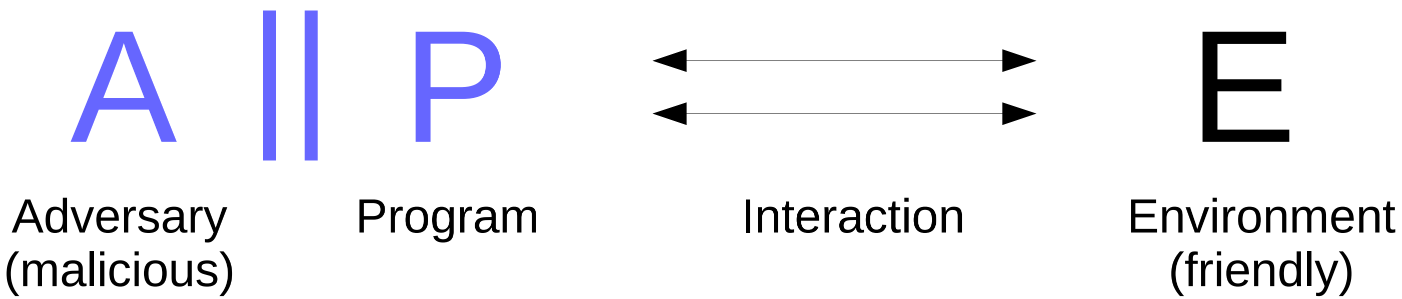
A || P ⇄ E

Adversary (malicious) | Program | Interaction | Environment (friendly)

- Example:
  - P accumulates inputs in a counter, outputs counter value at each step. Ideally,

    ?0 !0 ?5 !5 ?2 !7 ?3 !10 ...
  - A is a library that P uses to log on the side (for debugging)
  - But, due to a buffer overflow bug, A also occasionally *corrupts* P's internal counter, breaking P's expected behavior.
- The A↔P and P↔E interactions are different.
- The A↔P interaction is internal, not part of traces or properties.

Properties, with adversaries



- Properties/specifications π same as without A.
- But notion of program having property changes:

P *robustly has* π iff ∀A. Tr(A || P) ⊆ π

(*cf*. robust safety)

Compilation, with adversaries

$\forall A \parallel P \longleftrightarrow E$

Adversary (malicious) | Program | Interaction | Environment (friendly)

May be more powerful than source adversaries

A <u>secure compiler</u> ensures that adversary's increased power does not break expected P ↔ E properties

$\forall A' \parallel P\!\downarrow \longleftrightarrow E$

Adversary (malicious) | Program | Interaction | Environment (friendly)

Compiler <u>security</u> as robust property preservation

A compiler ↓ is <u>secure</u> if it *preserves all properties* ***robustly:***

$\forall\pi.\ \forall P.\ (P \text{ robustly has } \pi) \Rightarrow (P{\downarrow} \text{ robustly has } \pi)$

$\forall\pi.\ \forall P.\ (\forall A.\ \mathrm{Tr}(A \mathbin{||} P) \subseteq \pi) \Rightarrow (\forall A'.\ \mathrm{Tr}(A' \mathbin{||} (P{\downarrow})) \subseteq \pi)$

For all π, the compiled program is no more vulnerable to attacks on π than the source program. Hence, verifying the source (in source semantics) is enough to guarantee π even against low-level attacks.

Compiler security: Equivalent characterization

- A compiler $\downarrow$ is <u>secure</u> if it *preserves all properties* ***robustly***

  $\forall\pi.\ \forall P.\ (P \text{ robustly has } \pi) \Rightarrow (P{\downarrow} \text{ robustly has } \pi)$

- Equivalently,

  $\forall\pi.\ \forall P.\ (\forall A.\ Tr(A \mid\mid P) \subseteq \pi) \Rightarrow$

  $(\forall A'.\ Tr(A' \mid\mid (P{\downarrow})) \subseteq \pi)$

- Equivalently,

  $\forall A'\ P\ t.\ (t \in Tr(A' \mid\mid (P{\downarrow}))) \Rightarrow \exists A.\ t \in Tr(A \mid\mid P)$

  We call this last statement "robust refinement"

<u>Theorem</u>

A compiler *preserves all properties robustly* iff it *refines behaviors robustly*.

Compiler security: Equivalent characterization

- A compiler ↓ is <u>secure</u> if it *preserves all properties* ***robustly***

  $\forall\pi.\ \forall P.\ (P \text{ robustly has } \pi) \Rightarrow (P\!\downarrow \text{ robustly has } \pi)$

A property-based characterization

A property-free characterization

- Equivalently,

  $\forall A'\ P\ t.\ (t \in Tr(A' \mid\mid (P\!\downarrow))) \Rightarrow \exists A.\ t \in Tr(A \mid\mid P)$

Related to “back-translation”

Amenable to formal proof

Vertically composable

Compiler security, parametrized

- Does security coincide with preserving all properties robustly?
  - Not necessarily: It may require less (or more ... later)
- A compiler ↓ is secure wrt a class C of properties if it *robustly preserves all properties in C*

  $\forall \underline{\pi \in C}.\ \forall P.\ (P \text{ robustly has } \pi) \Rightarrow (P{\downarrow} \text{ robustly has } \pi)$
- C may be “safety properties” or “liveness properties”
- Standard notions from literature

Safety properties

- Intuitively, a property that specifies “nothing bad will happen”
- “Bad” = Set M of finite trace prefixes
- Corresponding safety property:

  $\pi_M = \{t \mid \forall m \in M.\ \text{not } (m \leq t)\}$

  (Traces that don’t extend anything from M)
- Example:

  $M = \{t \mid t \text{ finite and } t \text{ ends in a negative output}\}$

  $\pi_M = \{t \mid t \text{ does } \textit{not} \text{ have a negative output}\}$

Compiler security for safety

- A compiler $\downarrow$ is secure for safety if it *preserves all safety properties robustly*

  $\forall \pi \in \text{Safety}.\ \forall P.\ (P \text{ robustly has } \pi)$

  $\Rightarrow (P\!\downarrow \text{ robustly has } \pi)$

- Equivalently,

  $\forall A'\ P\ t.\ (t \in \text{Fin}(A' \parallel (P\!\downarrow))) \Rightarrow \exists A.\ t \in \text{Fin}(A \parallel P)$

<u>Theorem</u>

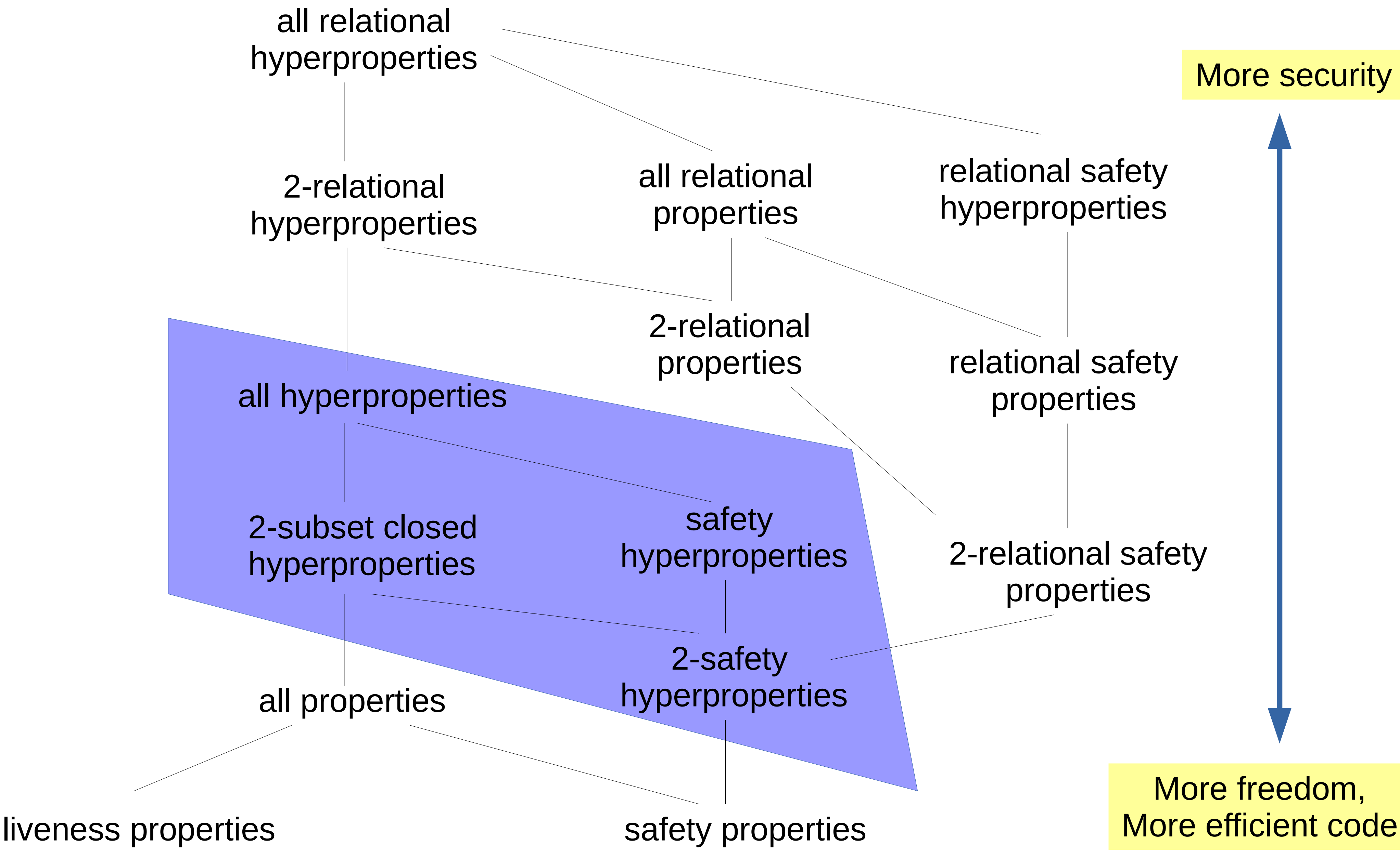
A compiler *preserves all safety properties robustly* iff it *refines finite behaviors robustly*.

Defining compiler security: General principle

- Define compiler security as preservation of some robust security class X.
- Obtain an equivalent, X-free characterization that is proof-amenable.
- Surprisingly, can be done for many different X.

Summary of results (abridged)

A compiler is secure for X if it robustly preserves all X, where X is:



2-safety hyperproperties (2-SHP)

- A 2-safety hyperproperty H is characterized by a set of *pairs* of bad trace prefixes, T as follows:

  P has H iff there is no pair in T such that P has both traces of the pair.
- Example: Program takes two inputs and produces one ouput:
  - 2-SHP: “Output must be independent of first input”
  - $T = \{(?i_1\ ?i_2\ !o_1,\ ?i_1'\ ?i_2\ !o_1') \mid o_1 <> o_1'\}$
  - Confidentiality property

Compiler security for 2-SHP

- A compiler $\downarrow$ is secure for 2-SHP if it *preserves all 2-SHPs robustly*

$$\forall H \in \text{2-SHP}.\ \forall P.\ (P \text{ robustly has } H) \Rightarrow (P{\downarrow} \text{ robustly has } H)$$

- Equivalently,

$$\forall A'\ P\ t_1\ t_2.\ (t_1, t_2 \in \mathrm{Fin}(A' \parallel (P{\downarrow}))) \Rightarrow \exists A.\ t_1, t_2 \in \mathrm{Fin}(A \parallel P)$$

Compare to compiler security for safety:
$\forall A'\ P\ t.\ (t \in \mathrm{Fin}(A' \parallel (P{\downarrow}))) \Rightarrow \exists A.\ t \in \mathrm{Fin}(A \parallel P)$

More examples

- Security for all hyperproperties

  $\forall A'\ P.\ \exists A.\ Tr(A' \mid\mid (P\downarrow)) = Tr(A \mid\mid P)$

- Security for all relational hyperproperties

  $\forall A'.\ \exists A.\ \forall P.\ Tr(A' \mid\mid (P\downarrow)) = Tr(A \mid\mid P)$

Current status

One compiler security criterion for each point in the partial order. Equivalent, back-translation-like characterization.

What must a compiler do to attain each point?
- Safety: Protect integrity of P↓'s private state
- 2-safety: Above + protect confidentiality of P↓'s private state
- Relational properties: Above + protect code identity

Compilers that realize these criteria.
- Significant progress on compiler security for safety

Proof techniques

Open questions

How should adversaries be represented? As contexts?

Is there a story for compositional or modular compilation?

We have a class of criteria. Which one(s) should we actually pursue?

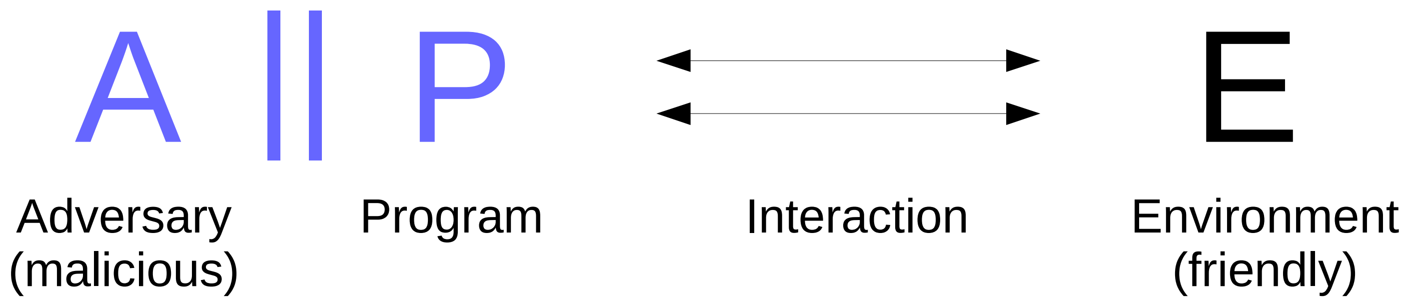
What is the connection to other notions of compiler security?

Other notions of compiler security ...

- Gracefully avoiding undefined behavior, *a la* memory-safety (related to our compiler security for safety?)
- Specific defense mechanisms:
  - Stack canaries
  - Code-pointer integrity
  - Address space layout randomization

  Is there a uniform, parametric definition of compiler security for these?
- Full abstraction: How does it connect to our notions?
- …

Conclusion



- Represent external interactions (as traces)
- Represent properties on interactions
- Compiler security = Preservation of a class of properties despite adversaries
- End-to-end security, but secure compilation seems to be broader than this