(Hyper)Property-Preserving Compilers

summer semester 18-19, block

Marco Patrignani\textsuperscript{1,2}
Properties and Hyperproperties

- Formalise any security property
- Established theory with practical applications

Recommended reading:

- Clarkson and Schneider. 2010. Hyperproperties.
• Property = set of traces that respect a certain condition (or predicate)
• Property = set of traces that respect a certain condition (or predicate)
• trace $t = \text{sequence of (formalised as } sth)$
Security Properties

• Property = set of traces that respect a certain condition (or predicate)
• trace $t$ = sequence of (formalised as $sth$)
  • program states $\Theta$
Security Properties

- Property = set of traces that respect a certain condition (or predicate)
- trace $t$ = sequence of (formalised as $sth$)
  - program states $\Theta$
  - component-context interactions $\alpha?\alpha!\ldots$
Security Properties

- Property = set of traces that respect a certain **condition** (or predicate)
- trace $t =$ sequence of (formalised as $sth$)
  - program states $\Theta$
  - component-context interactions $\alpha?$ $\alpha!\cdots$
  - code-environment interaction $read \; v; \; write \; v$

We use $t$ abstractly now, though mostly:

$t = \overline{\Theta}$
Security Properties

- A Trace captures a \textit{single run} of a program
Security Properties

- A Trace captures a single run of a program
- A Set of traces captures all individual runs of any program
Security Properties

• A Trace captures a single run of a program
• A Set of traces captures all individual runs of any program

This is unlike program equivalence:

• properties talk a single program
Examples

• NRW: $\{ t \mid \not\exists \Theta < \Theta'. \downarrow \text{read}\Theta \land \downarrow \text{send}\Theta' \}$

NRW: the program does not send on the network after reading a file
Examples

NRW: \{ t \mid \not\exists \Theta < \Theta'. \leftarrow \text{read}\Theta \land \leftarrow \text{send}\Theta' \}\}

NRW: the program does not send on the network after reading a file
\leftarrow \text{read}\Theta \text{ and } \leftarrow \text{send}\Theta' \text{ are abstract predicates}
Examples

• **NRW:** \( \{ t \mid \exists \Theta < \Theta'. \leftarrow \text{read}\Theta \land \leftarrow \text{send}\Theta' \} \)

NRW: the program does not send on the network after reading a file

\( \leftarrow \text{read}\Theta \) and \( \leftarrow \text{send}\Theta' \) are abstract predicates

• **GS:** \( \{ t \mid \leftarrow \text{req}\Theta_i \Rightarrow \leftarrow \text{resp}\Theta_j \text{ where } j > i \} \)

GS: the program eventually responds to the requests
Safety and Liveness

Properties are partitioned in

- **Safety**: something bad does not happen (NRW)

- **Liveness**: something good eventually happens (GS)
Safety and Liveness

Properties are partitioned in

- **Safety**: something bad does not happen (NRW)
- **Liveness**: something good eventually happens (GS)
Safety

- Safety = integrity
Safety

• Safety = integrity
• Safety ≠ confidentiality
• Safety = integrity
• Safety ≠ confidentiality
• but, Safety = weak secrecy: we don’t leak a fresh $k$ to $\mathcal{C}$
Safety as a dual

• Take the traces that define a safety property
Safety as a dual

- Take the traces that define a safety property
- Describe safety by the so-called set of bad prefixes
Safety as a dual

- Take the traces that define a safety property
- Describe safety by the so-called set of bad prefixes
- In the following: $m$ is a finite trace $t$ (a finite $\Theta$) aka a prefix
Safety as a dual

- Take the traces that define a safety property
- Describe safety by the so-called set of bad prefixes
- In the following: $m$ is a finite trace $t$ (a finite $\Theta$) aka a prefix
- NRW-dual:
  \[
  \{m \mid \Theta < \Theta'. \leftarrow \text{read}\Theta \land \leftarrow \text{send}\Theta'\}
  \]
Beyond Properties: Hyperproperties

- Properties = sets of traces
Beyond Properties: Hyperproperties

- Properties = sets of traces
- capture a *single run* (the trace) of any program (the set)
Beyond Properties: Hyperproperties

- Properties = sets of traces
- capture a **single run** (the trace) of any program (the set)

Hyperproperties = *sets of sets of traces*
Beyond Properties: Hyperproperties

- Properties = sets of traces
  - capture a **single run** (the trace) of any program (the set)

Hyperproperties = sets of sets of traces

- capture **multiple runs** (the sets of traces) of any program (the sets)
Example: NonInterference

- NI: two different high inputs result in the same low outputs
Example: NonInterference

- NI: two different high inputs result in the same low outputs
- high = secret, low = public
Example: NonInterference

- NI: two different high inputs result in the same low outputs
- high = secret, low = public
- a set of traces tells all the behaviours of the same program with different high inputs

\[ \text{NI : } \begin{cases} \{t_1, t_2\} & \forall t_1, t_2 \in \{t_1, t_2\}. \\
& \text{if } \text{inputs}(t_1) =_L \text{inputs}(t_2) \\
& \text{then } \text{outputs}(t_1) =_L \text{outputs}(t_2) \end{cases} \]
Example: Average Response Time < 1

\[
\text{ART: } \left\{ \{t\ldots\} \mid \text{mean}\left( \bigcup_{t\in\{t\ldots\}} \text{response\_time}(t) \right) < 1 \right\}
\]

where \text{response\_time}() looks in trace \(t\) and checks time between \text{req}() and \text{resp}()
Hypersafety and Hyperliveness

Like Properties, Hyperproperties are partitioned in

- **Hypersafety:** something bad does not happen (NI)
Hypersafety and Hyperliveness

Like Properties, Hyperproperties are partitioned in:

- **Hypersafety**: something bad does not happen (NI)
- **Hyperliveness**: something good eventually happens (ART)
Hypersafety as a dual

- Take the sets of traces that define a hypersafety property
Hypersafety as a dual

- Take the sets of traces that define a hypersafety property
- Describe hypersafety by the so-called sets of sets of bad prefixes
Hypersafety as a dual

- Take the sets of traces that define a hypersafety property
- Describe hypersafety by the so-called sets of sets of bad prefixes

\[ \forall t_1, t_2 \in \{t_1, t_2\}. \]
\[ \text{if } \text{inputs}(t_1) = L \text{inputs}(t_2) \]
\[ \text{then } \text{outputs}(t_1) = \not L \text{outputs}(t_2) \]
How do we formalise a program having a property?
Property Satisfaction

How do we formalise a program having a property?

- $P$ generates trace $t$: $P \vdash t$
How do we formalise a program having a property?

- $P$ generates trace $t$: $P \leadsto t$
- Property $\pi = \{ t \}$
Property Satisfaction

How do we formalise a program having a property?

• $P$ generates trace $t$: $P \rightarrow t$
• Property $\pi = \{t\}$
• $\vdash P : \pi \overset{\text{def}}{=} \text{if } P \rightarrow t \text{ then } t \in \pi$
How do we formalise a program having a **hyperproperty**?

- All traces generated by $P$: $\text{Behav}(P) = \left\{ \frac{t}{\text{divides}} \right\}$.
- Hyperproperty $H = \left\{ \frac{t}{\text{union}} \right\}$
- $P : H \text{def} = \text{Behav}(P) \in H$
Hyperproperty Satisfaction

How do we formalise a program having a hyperproperty?

- All traces generated by $P$: $\text{Behav}(P) = \{t \mid P \sim t\}$. 
Hyperproperty Satisfaction

How do we formalise a program having a hyperproperty?

• All traces generated by $P$:
  
  $$\text{Behav}(P) = \{t \mid P \sim t\}.$$ 

• Hyperproperty $H = \{\{t\}\}$
Hyperproperty Satisfaction

How do we formalise a program having a hyperproperty?

• **All** traces generated by $P$:
  \[ \text{Behav}(P) = \{ t \mid P \sim t \} \]

• Hyperproperty $H = \{ \{ t \} \}$

• $\vdash P : H \overset{\text{def}}{=} \text{Behav}(P) \in H$
(Hyper)Properties must hold \textit{robustly}:

- property satisfaction for whole programs protects against our bugs
Robustness

(Hyper)Properties must hold robustly:

• property satisfaction for whole programs protects against our bugs
• robust property satisfaction protects against any active adversary
Robustness

(Hyper)Properties must hold robustly:

• property satisfaction for whole programs protects against our bugs
• robust property satisfaction protects against any active adversary

So we want our program $P$ to satisfy NRW, GS, NI or ART: $\forall C. C[P]$, so $\Theta = C[P]$
(Hyper)Properties must hold robustly:

- property satisfaction for whole programs protects against our bugs
- robust property satisfaction protects against any active adversary

So we want our program $P$ to satisfy NRW, GS, NI or ART: $\forall C. C[P]$, so $\Theta = C[P]$.

Reminiscent of contextual equivalence!
Robust (Hyper)Property Satisfaction

How do we formalise a program having a hyperproperty _robustly_?
How do we formalise a program having a hyperproperty \textit{robustly}?

- $P$ now is a partial program
Robust (Hyper)Property Satisfaction

How do we formalise a program having a hyperproperty **robustly**?

- $P$ now is a partial program
- $\mathcal{C}$ is what $P$ is linked against
Robust (Hyper)Property Satisfaction

How do we formalise a program having a hyperproperty robustly?

- \( P \) now is a partial program
- \( C \) is what \( P \) is linked against
- \( \vdash_R P : \pi \overset{\text{def}}{=} \forall C. \text{if } C[P] \Rightarrow t \text{ then } t \in \pi \)
How do we formalise a program having a hyperproperty robustly?

- $P$ now is a partial program
- $\mathcal{C}$ is what $P$ is linked against
- $\vdash_R P : \pi \overset{\text{def}}{=} \forall \mathcal{C}. \text{if } \mathcal{C}[P] \rightsquigarrow t \text{ then } t \in \pi$
- $\vdash_R P : H \overset{\text{def}}{=} \forall \mathcal{C}. \text{Behav}(\mathcal{C}[P]) \in H$
A Note on Robustness

- Contexts can generate property-relevant events now
A Note on Robustness

- Contexts can generate property-relevant events now
- so they can trivially invalidate any property
A Note on Robustness

- Contexts can generate property-relevant events now
- so they can **trivially** invalidate any property
- we must **filter** events and consider only those generated by $P$
Example: Robust Safety

- \( \pi \in Safety \)
- \( \vdash_R P : \pi \overset{\text{def}}{=} \forall C. \text{if } C[P] \leadsto t \text{ then } t \in \pi \)
Example: Robust Safety

• $\pi \in Safety$
• $\vdash_R P : \pi \overset{\text{def}}{=} \forall C. \text{ if } C[P] \leadsto t \text{ then } t \in \pi$
• dually: $\{m\} :: \pi \in Safety$
• $m \leq t = m \text{ is a prefix of } t$
• $\vdash_R P : \{m\} \overset{\text{def}}{=} \forall C. \text{ if } C[P] \leadsto t \text{ then } \not\exists m \in \{m\}. m \leq t$
Example: Robust Liveness …?

• can this hold robustly?
Example: Robust Liveness …?

- can this hold robustly?
- we need a *fair* context in our setup: a context that will interact with us
Example: Robust Liveness …?

- can this hold robustly?
- we need a fair context in our setup: a context that will interact with us
- avoid DOS: the attacker wants to violate our code, not starve it
1. specify (hyper)properties on programs through traces
Robust Compilation

1. specify (hyper)properties on programs through traces
2. specify (hyper)properties robustly
Robust Compilation

1. specify (hyper)properties on programs through traces
2. specify (hyper)properties robustly

Q: can we preserve them through compilation?
Robust Compilation

1. specify (hyper)properties on programs through traces
2. specify (hyper)properties robustly

Q: can we preserve them through compilation?

Yes!
Assumptions

- same alphabet of traces between $S$ and $T$ (I/O or syscalls)
- we lift this (partially) later
• Assume the source has a property robustly
Example: Robust Property Preservation

• **Assume** the source has a property robustly
• **Prove** the compiled program has the same property robustly
Example: Robust Property Preservation

- **Assume** the source has a property robustly
- **Prove** the compiled program has the same property robustly

\[ RTP : \forall \pi. \forall P. (\forall C. C[P] \sim t \Rightarrow t \in \pi) \Rightarrow (\forall C. C[[P]] \sim t \Rightarrow t \in \pi) \]
Example: Robust Safety Preservation

• **Same as RTP, restrict to safety**
Example: Robust Safety Preservation

- Same as RTP, restrict to safety

\[ RSP : \forall \pi \in Safety. \ \forall P. \]
\[
(\forall C \ t. \ C[P] \leadsto t \Rightarrow t \in \pi) \Rightarrow
(\forall C \ t. \ C[[P]] \leadsto t \Rightarrow t \in \pi)
\]
Evaluation

Correct definitions
Correct definitions
Hard to use: no proof support
Correct definitions

Hard to use: no proof support

We want equivalent criteria that are easy to prove
Example: Robust Property Preservation #2

\[ RTP : \forall \pi. \forall P. (\forall C \ t. C[P] \sim t \Rightarrow t \in \pi) \Rightarrow (\forall C \ t. C[[P]] \sim t \Rightarrow t \in \pi) \]
Example: Robust Property Preservation #2

\[ RTP : \forall \pi. \forall P. (\forall \mathcal{C} t. \mathcal{C}[P] \sim t \Rightarrow t \in \pi) \Rightarrow \\
(\forall \mathcal{C} t. \mathcal{C}[[P]] \sim t \Rightarrow t \in \pi) \]
Example: Robust Property Preservation #2

\[ RTP : \forall \pi. \forall P. (\forall C t. C[P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\
(\forall C t. C[[P]] \rightsquigarrow t \Rightarrow t \in \pi) \]

\[ PFRTP : \forall P. \forall C. \forall t. C[[P]] \rightsquigarrow t \Rightarrow \\
\exists C. C[P] \rightsquigarrow t \]
RTP Intuition

If any trace in the target is also done in the source, and the source has the property, so does the target.
Example: Robust Safety Preservation #2

\[ RSP : \forall \pi \in Safety. \forall P. \]
\[ (\forall C t. C[P] \sim t \Rightarrow t \in \pi) \Rightarrow \]
\[ (\forall C t. C[[P]] \sim t \Rightarrow t \in \pi) \]
Example: Robust Safety Preservation #2

\[ RSP : \forall \pi \in \text{Safety}. \ \forall P. \]
\[
(\forall C \ t. \ C[P] \sim t \Rightarrow t \in \pi) \Rightarrow \\
(\forall C \ t. \ C[[P]] \sim t \Rightarrow t \in \pi)
\]
Example: Robust Safety Preservation #2

\[ RSP : \forall \pi \in Safety. \ \forall P. \]
\[ (\forall C. \ C[P] \sim t \Rightarrow t \in \pi ) \Rightarrow \]
\[ (\forall C. \ C[[P]] \sim t \Rightarrow t \in \pi ) \]

\[ PFRSP : \forall P. \ \forall C. \ \forall m. \]
\[ C[[P]] \sim m \Rightarrow \]
\[ \exists C. \ C[P] \sim m \]
RSP Intuition

Safety is defined *dually* as a set of bad prefixes. If any prefix done in the target is also done in the source and the source has the safety property, that prefix is not bad, so the target also has the safety property.
Relating RTP and RSP

- $RTP \iff PFRTP$
- $RSP \iff PFRSP$
- $RTP \implies RSP$
- $RTP \iff PFRTP$
- $RSP \iff PFRSP$

RLP (ish) /two.osf/nine.osf//four.osf/zero.osf
Relating RTP and RSP

- $RTP \iff \text{PFRTP}$
- $RSP \iff \text{PFRSP}$

RLP (ish)
Relating RTP and RSP

- \( RTP \iff PFRTP \)
- \( RSP \iff PFRSP \)
- \( RTP \Rightarrow RSP \)
Relating RTP and RSP

- \( RTP \iff PFRTP \)
- \( RSP \iff PFRSP \)
- \( RTP \implies RSP \)
Example: Robust HP Preservation

- as before: **Assume** the source has a hyperproperty robustly
Example: Robust HP Preservation

• as before: **Assume** the source has a hyperproperty robustly
• **Prove** the compiled program has the same hyperproperty robustly
Example: Robust HP Preservation

- as before: **Assume** the source has a hyperproperty robustly
- **Prove** the compiled program has the same hyperproperty robustly

\[
RHP : \forall H. \forall P. (\forall C. \text{Behav}(C[P]) \in H) \Rightarrow \\
(\forall C. \text{Behav}(C[[P]]) \in H)
\]
Example: Robust HP Preservation #2

\[ RHP : \forall H. \forall P. (\forall C. \text{Behav}(C[P]) \in H) \Rightarrow (\forall C. \text{Behav}(C[[P]]) \in H) \]
$RHP : \forall H. \forall P. (\forall C. \text{Behav}(C[P]) \in H) \Rightarrow$ 
$(\forall C. \text{Behav}(C[[P]]) \in H)$
Example: Robust HP Preservation #2

\[ RHP : \forall H. \forall P. (\forall C. \text{Behav}(C[P]) \in H) \Rightarrow (\forall C. \text{Behav}(C[[P]]) \in H) \]

\[ PFRHP : \forall P. \forall C. \exists C. \text{Behav}(C[[P]]) = \text{Behav}(C[P]) \]

\[ PFRHP : \forall P. \forall C. \exists C. \forall t. C[[P]] \sim t \iff C[P] \sim t \]
Quiz: Spot the Differences

\[ PFRTP : \forall P. \forall C. \forall t. C[[P]] \rightarrow t \Rightarrow \exists C. C[P] \rightarrow t \]

\[ PFRHP : \forall P. \forall C. \exists C. \forall t. C[[P]] \rightarrow t \iff C[P] \rightarrow t \]
• Quantifier ordering
Answers

- Quantifier ordering
- Implication
Answers

- Quantifier ordering
- Implication

Intuition

- Quantifier ordering: lifts to sets of traces since a $\in$ in PFRHP works for a set of traces
Answers

- Quantifier ordering
- Implication

Intuition

- Quantifier ordering: lifts to sets of traces since a $\exists$ in PFRHP works for a set of traces
- Implication: a single implication means refinement, so the target can have more behaviours.
Answers

• Quantifier ordering
• Implication

Intuition

• Quantifier ordering: lifts to sets of traces since a $\mathcal{C}$ in PFRHP works for a set of traces
• Implication: a single implication means refinement, so the target can have more behaviours. Co-implication means no refinement, we need the exact same traces to ensure inclusion in the $H$
Example: Robust Hypersafety Preservation

\[ PFRHSP : \forall P. \forall C. \forall \{m\}. \]
\[ \{m\} \leq \text{Behav}(C[\![P]\!]]) \Rightarrow \exists C. \{m\} \leq \text{Behav}(C[\!P\!]) \]
Example: Robust Hypersafety Preservation

\[ PFRHSP : \forall P. \forall C. \forall \{ m \}. \]
\[ \{ m \} \leq \text{Behav} (C[[P]]) \Rightarrow \exists C. \{ m \} \leq \text{Behav} (C[P]) \]

Where \( \leq \) means all prefixes of \( \{ m \} \) are extended by the behaviour of the (compiled) program.
Subclasses of Hyperproperties

- **K-Hypersafety**: hypersafety for sets of cardinality $k$ (if $k = 4$, NMIF)

- **/two.osf-Hypersafety**: hypersafety for sets of cardinality 2: set of pairs of traces: NI

- **Subset-closed HP**: set of traces closed under subsetting

- **K-, /two.osf- Subset-closed HP**: as before, curtail set cardinality to $k$, 2

- **Hyperliveness**: not present: RHLP collapses with RHP
Subclasses of Hyperproperties

- K-Hypersafety: hypersafety for sets of cardinality \( k \) (if \( k = 4 \), NMIF)
- 2-Hypersafety: hypersafety for sets of cardinality 2: set of pairs of traces: NI
Subclasses of Hyperproperties

- **K-Hypersafety**: hypersafety for sets of cardinality $k$ (if $k = 4$, NMIF)
- **2-Hypersafety**: hypersafety for sets of cardinality 2: set of pairs of traces: NI
- **Subset-closed HP**: set of traces closed under subsetting
- **Hyperliveness**: not present: RHLP collapses with RHP
Subclasses of Hyperproperties

- K-Hypersafety: hypersafety for sets of cardinality $k$ (if $k = 4$, NMIF)
- 2-Hypersafety: hypersafety for sets of cardinality 2: set of pairs of traces: NI
- Subset-closed HP: set of traces closed under subsetting
- K-, 2- Subset-closed HP: as before, curtail set cardinality to $k$, 2
Subclasses of Hyperproperties

- K-Hypersafety: hypersafety for sets of cardinality $k$ (if $k = 4$, NMIF)
- 2-Hypersafety: hypersafety for sets of cardinality 2: set of pairs of traces: NI
- Subset-closed HP: set of traces closed under subsetting
- K-, 2- Subset-closed HP: as before, curtail set cardinality to $k$, 2
- Hyperliveness: not present: RHLP collapses with RHP
Robust Compilation (RC) Diagram
Robust Compilation (RC) Diagram
RC vs FAC

• (some) RC criteria are propositional
RC vs FAC

• (some) RC criteria are propositional
  (some are relational but they are not presented here)

• FAC is only relational
RC vs FAC

• (some) RC criteria are propositional
  (some are relational but they are not presented here)

• FAC is only relational

• both are robust
RC vs FAC

- (some) RC criteria are propositional
  (some are relational but they are not presented here)
- FAC is only relational
- both are robust
- FAC is only as precise as the equivalence
- RC do not preserve abstractions beyond the related security (hyper)property
Proving RC

$PFRT{P} : \forall P. \forall C. \forall t. \quad \mathcal{C}[\mathcal{P}] \rightsquigarrow t \Rightarrow \exists C. \mathcal{C}[P] \rightsquigarrow t$

$PFRS{P} : \forall P. \forall C. \forall m. \quad \mathcal{C}[\mathcal{P}] \rightsquigarrow m \Rightarrow \exists C. \mathcal{C}[P] \rightsquigarrow m$
Proving RC

\[ PFRTP : \forall P. \forall C. \forall t. \]
\[ C[[P]] \sim t \Rightarrow \exists C. C[P] \sim t \]

\[ PFRSP : \forall P. \forall C. \forall m. \]
\[ C[[P]] \sim m \Rightarrow \exists C. C[P] \sim m \]

Recall $\Rightarrow$ for FAC (contrapositive):
Proving RC

**PFRTTP**: \( \forall P. \forall C. \forall t. \)  
\( C[[P]] \vdash t \implies \exists C. C[P] \vdash t \)

**PFRTSP**: \( \forall P. \forall C. \forall m. \)  
\( C[[P]] \vdash m \implies \exists C. C[P] \vdash m \)

Recall \( \implies \) for FAC (contrapositive):

\( \forall P_1, P_2 \)  
\( \exists C. C[[P_1]] \uparrow \iff C[[P_2]] \implies \exists C. C[P_1] \uparrow \iff C[P_2] \uparrow \)
Backtranslation!

- generate a $C$ starting from what we have
Backtranslation!

• generate a $C$ starting from what we have
Backtranslation!

- generate a $C$ starting from what we have
- $C, t$ for PFRTP
Backtranslation!

- generate a $C$ starting from what we have
- $C, t$ for PFRTP
- $C, m$ for PFRSP
Backtranslation!

- generate a $\mathcal{C}$ starting from what we have
- $\mathcal{C}, t$ for PFRTP
- $\mathcal{C}, m$ for PFRSP
- $\mathcal{C}, \text{only!!}$ for PFRHP
Backtranslation!

- generate a $c$ starting from what we have
- $c$, $t$ for PFRTP
- $c$, $m$ for PFRSP
- $c$, only!! for PFRHP
- $c$, {m} for PFRHSP
Backtranslation!

- $m/\{m\}$ yields trace-based BT
Backtranslation!

- \( m/\{m\} \) yields trace-based BT

\[ m/\{m\} \quad \text{yields trace-based BT} \]
Backtranslation!

• \( m/\{m\} \) yields trace-based BT
• \( t \) is infinite, \( C \) is finite, so only use \( C \) there
Backtranslation!

- \( m / \{m\} \) yields **trace-based BT**
- \( t \) is infinite, \( c \) is finite, so only use \( c \) there
- \( c \) yields **context-based BT**
Backtranslation!

- $m/\{m\}$ yields trace-based BT
- $t$ is infinite, $c$ is finite, so only use $c$ there
- $c$ yields context-based BT
  - can be precise BT
Backtranslation!

- $m / \{m\}$ yields **trace-based BT**
- $t$ is infinite, $c$ is finite, so only use $c$ there
- $c$ yields **context-based BT**
  - can be **precise BT**
  - or **approximate BT** (intuitively analogous to trace-based BT)
Backtranslation!

- $m/\{m\}$ yields trace-based BT
- $t$ is infinite, $c$ is finite, so only use $c$ there
- $c$ yields context-based BT
  - can be precise BT
  - or approximate BT (intuitively analogous to trace-based BT)
- BT is not the inverse of compilation
We have seen:

• Properties and Hyperproperties: to formalise a program having a security property
• Robust compilation criteria, which preserve classes of (hyper)properties
• Backtranslation-equivalent Robust compilation criteria