Formally Secure Compilation

Cătălin Hrițcu

Inria Paris

https://secure-compilation.github.io

Devastating low-level attacks

Devastating low-level attacks

inherently insecure languages like C/C++

- **e.g. memory unsafe**: any buffer overflow is catastrophic allowing remote attackers to gain complete control

Devastating low-level attacks

inherently insecure languages like C/C++

- **e.g. memory unsafe**: any buffer overflow is catastrophic allowing remote attackers to gain complete control
- ~100 different **undefined behaviors** in usual C compiler

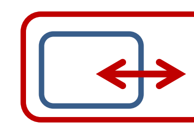
Devastating low-level attacks

inherently insecure languages like C/C++

- **e.g. memory unsafe**: any buffer overflow is catastrophic allowing remote attackers to gain complete control
- ~100 different **undefined behaviors** in usual C compiler

insecure interoperability with lower-level code

- even code in **more secure languages (Java, OCaml, Rust)** has to interoperate with **low-level code (C, C++, ASM)**
- **insecure interoperability**: all source-level guarantees lost



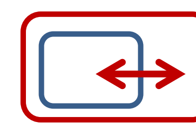
Devastating low-level attacks

inherently insecure languages like C/C++

- **e.g. memory unsafe**: any buffer overflow is catastrophic allowing remote attackers to gain complete control
- ~100 different **undefined behaviors** in usual C compiler

insecure interoperability with lower-level code

- even code in **more secure languages (Java, OCaml, Rust)** has to interoperate with **low-level code (C, C++, ASM)**



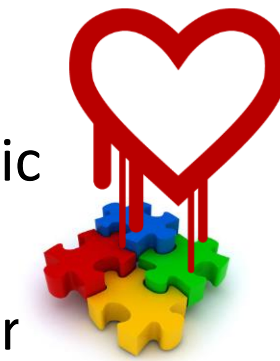
- **insecure interoperability**: all source-level guarantees lost

Part 1: formalize what it means to solve this problem

Devastating low-level attacks

Part 2: give meaning to mitigation (protected components)

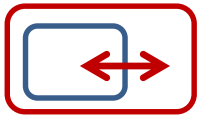
inherently insecure languages like C/C++



- **e.g. memory unsafe**: any buffer overflow is catastrophic allowing remote attackers to gain complete control
- ~100 different **undefined behaviors** in usual C compiler

insecure interoperability with lower-level code

- even code in **more secure languages (Java, OCaml, Rust)** has to interoperate with **low-level code (C, C++, ASM)**

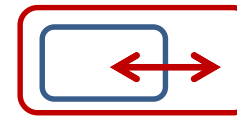


- **insecure interoperability**: all source-level guarantees lost

Part 1: formalize what it means to solve this problem

Part 1 of 2

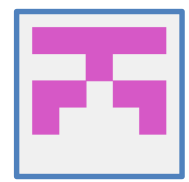
Secure Interoperability with Lower-Level Code



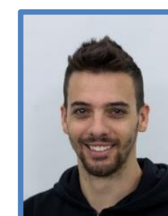
Carmine Abate
Inria Paris
U. Trento

Deepak Garg
MPI-SWS

Cătălin Hrițcu
Inria Paris



Jérémy Thibault
Inria Paris
ENS Rennes



Marco Patrignani
CISPA
Stanford

Good programming languages provide helpful abstractions for writing more secure code

Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:

Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions associated with safe C programs**

Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions associated with safe C programs**
  - **higher-level abstractions associated with ML-like languages**

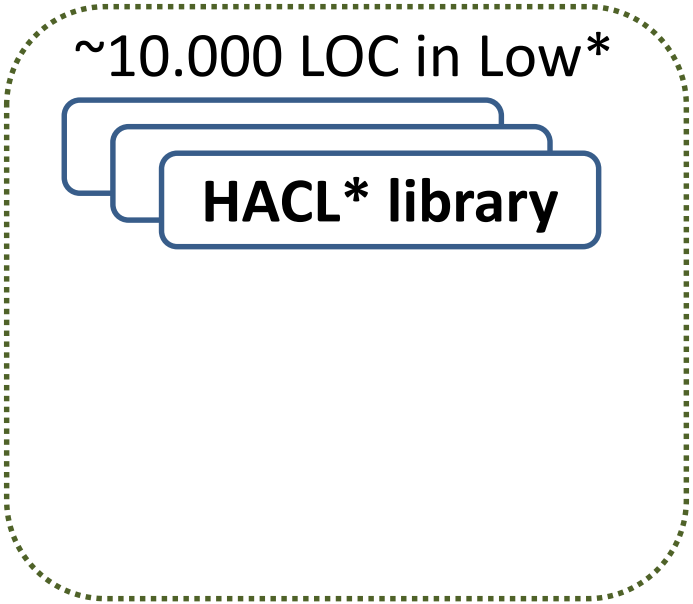
Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions associated with safe C programs**
  - **higher-level abstractions associated with ML-like languages**
  - **most features of verification systems like Coq and Dafny**

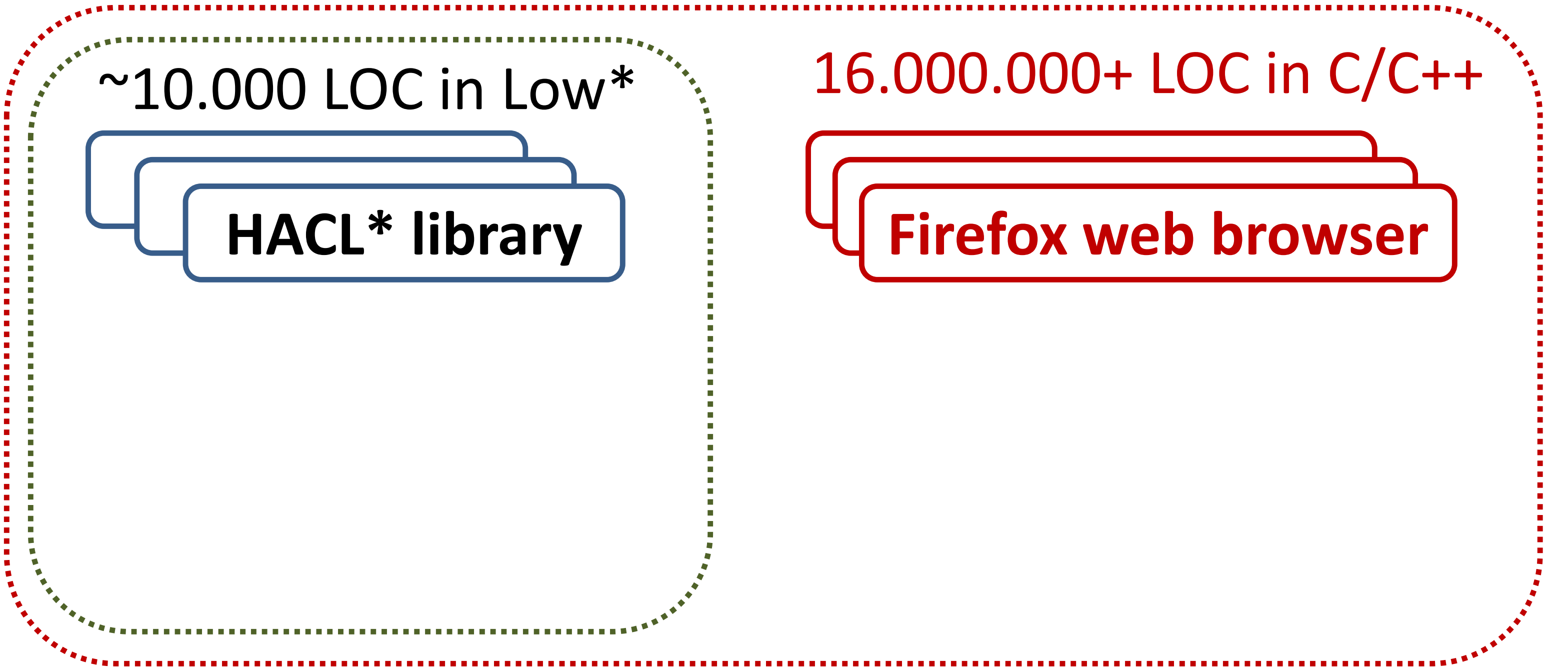
Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions associated with safe C programs**
  - **higher-level abstractions associated with ML-like languages**
  - **most features of verification systems like Coq and Dafny**
  - **patterns specific to cryptographic code**

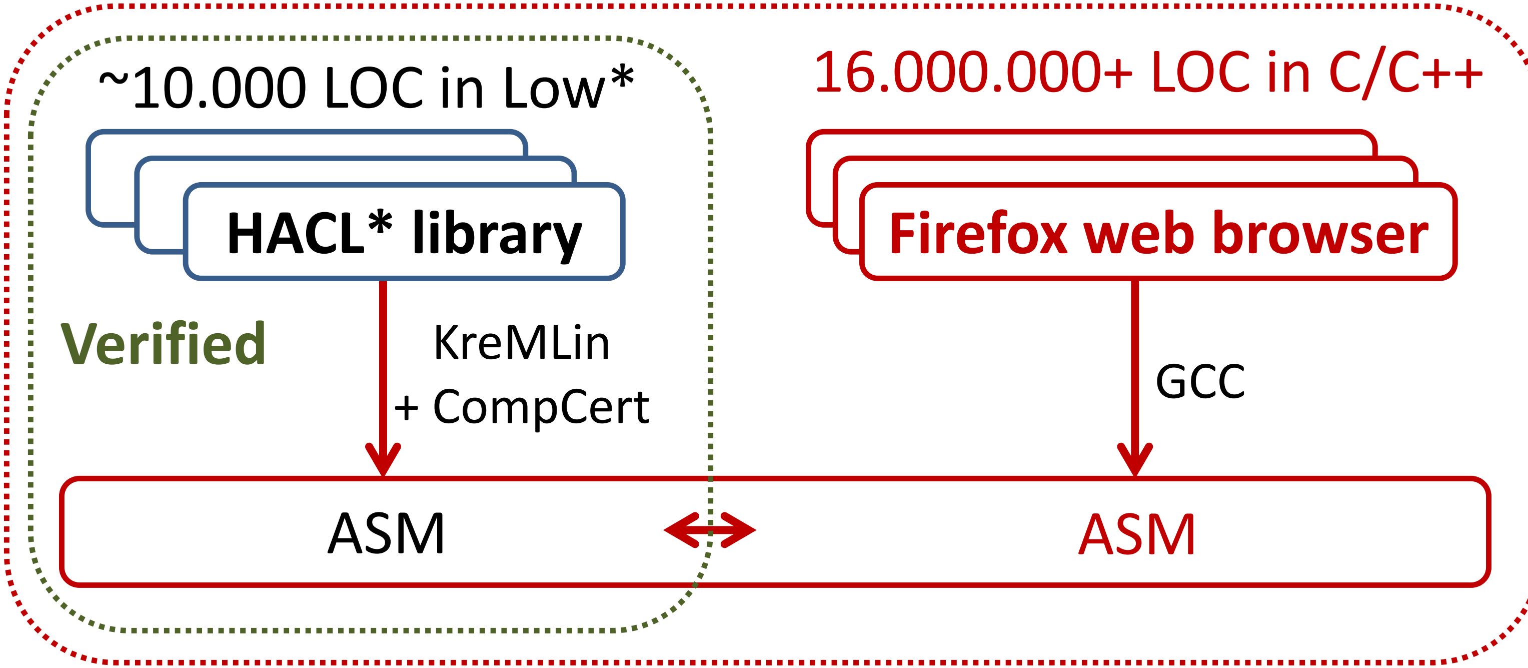
Abstractions not enforced when linking with adversarial low-level code



Abstractions not enforced when linking with adversarial low-level code



Abstractions not enforced when linking with adversarial low-level code



Insecure interoperability: compromised (or malicious) application linking in miTLS can easily **read and write miTLS's data and code**, **jump to arbitrary instructions**, **smash the stack**, ...

Secure compilation

- **Protect source-level abstractions**
  even against linked adversarial low-level code

Secure compilation

- Protect source-level abstractions
  even against linked adversarial low-level code
- Enable source-level security reasoning

Secure compilation

- **Protect source-level abstractions**
  even against linked adversarial low-level code
- **Enable source-level security reasoning**
  - even **an adversarial target-level context cannot break the security properties of the compiled program** any more than some source-level context could

Secure compilation

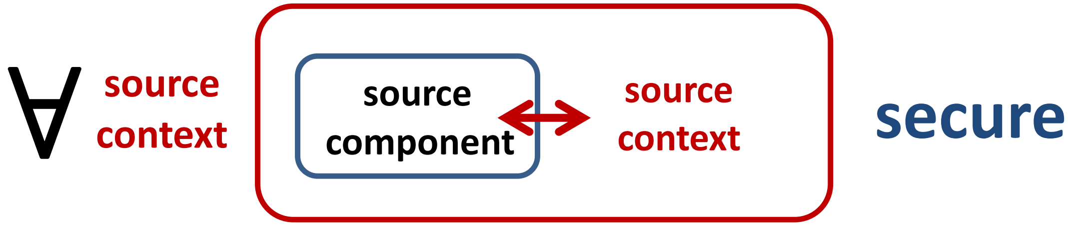
- **Protect source-level abstractions**
  even against linked adversarial low-level code
- **Enable source-level security reasoning**
  - even **an adversarial target-level context cannot break the security properties of the compiled program** any more than some source-level context could
  - **no "low-level" attacks**

Secure compilation

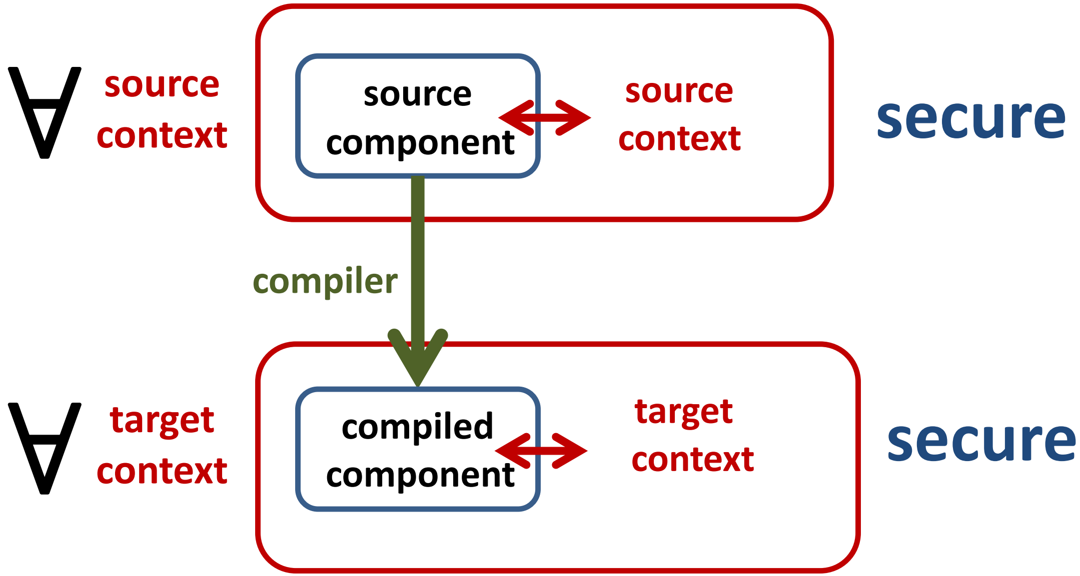
- **Protect source-level abstractions**
  even against linked adversarial low-level code
- **Enable source-level security reasoning**
  - even **an adversarial target-level context cannot break the security properties of the compiled program** any more than some source-level context could
  - **no "low-level" attacks**
  - **no need to worry about the compilation chain** (compiler, linker, loader, runtime, system, hardware)

Source-level security reasoning

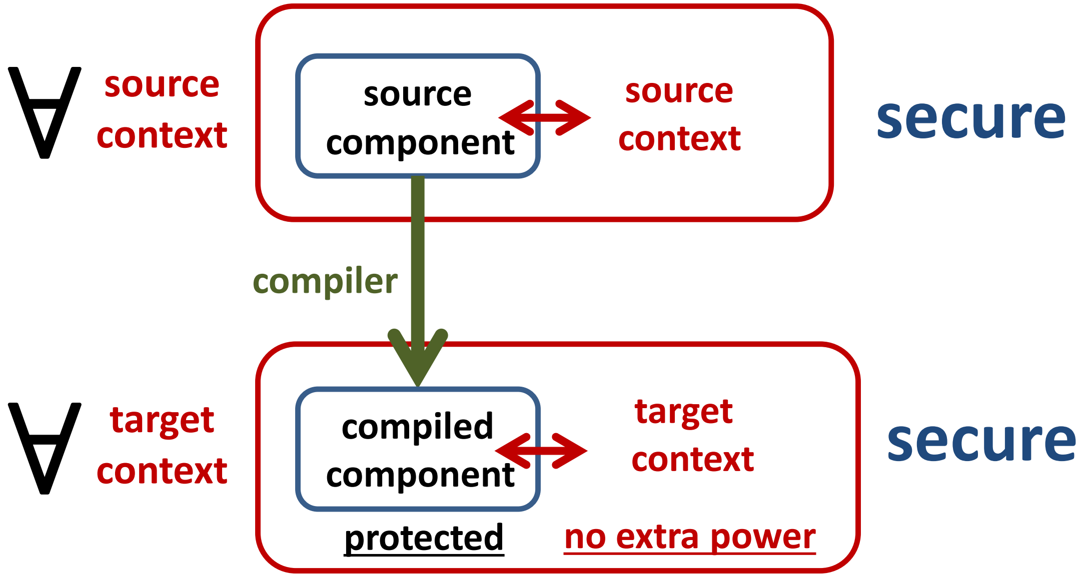
Source-level security reasoning



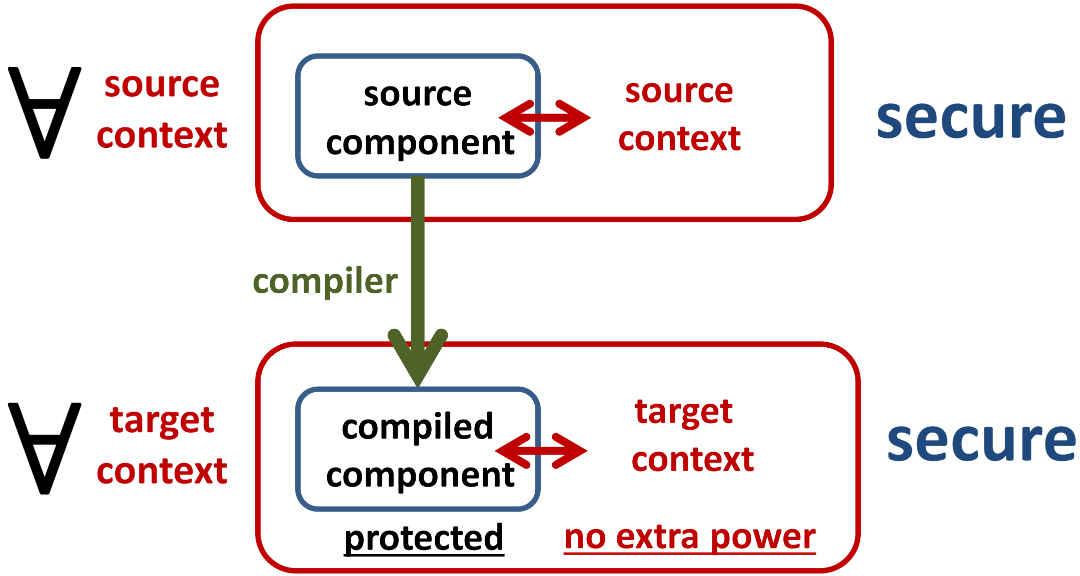
Source-level security reasoning



Source-level security reasoning



Source-level security reasoning



But what does "secure" mean?

What security properties should we preserve?

What security properties should we preserve?

- We explore a large space of security properties

What security properties should we preserve?

- We explore a large space of security properties
- Study preserving various classes of ...
  - **trace properties** (safety, liveness)
  - **hyperproperties** (e.g. noninterference)
  - **relational hyperproperties** (e.g. trace equivalence)

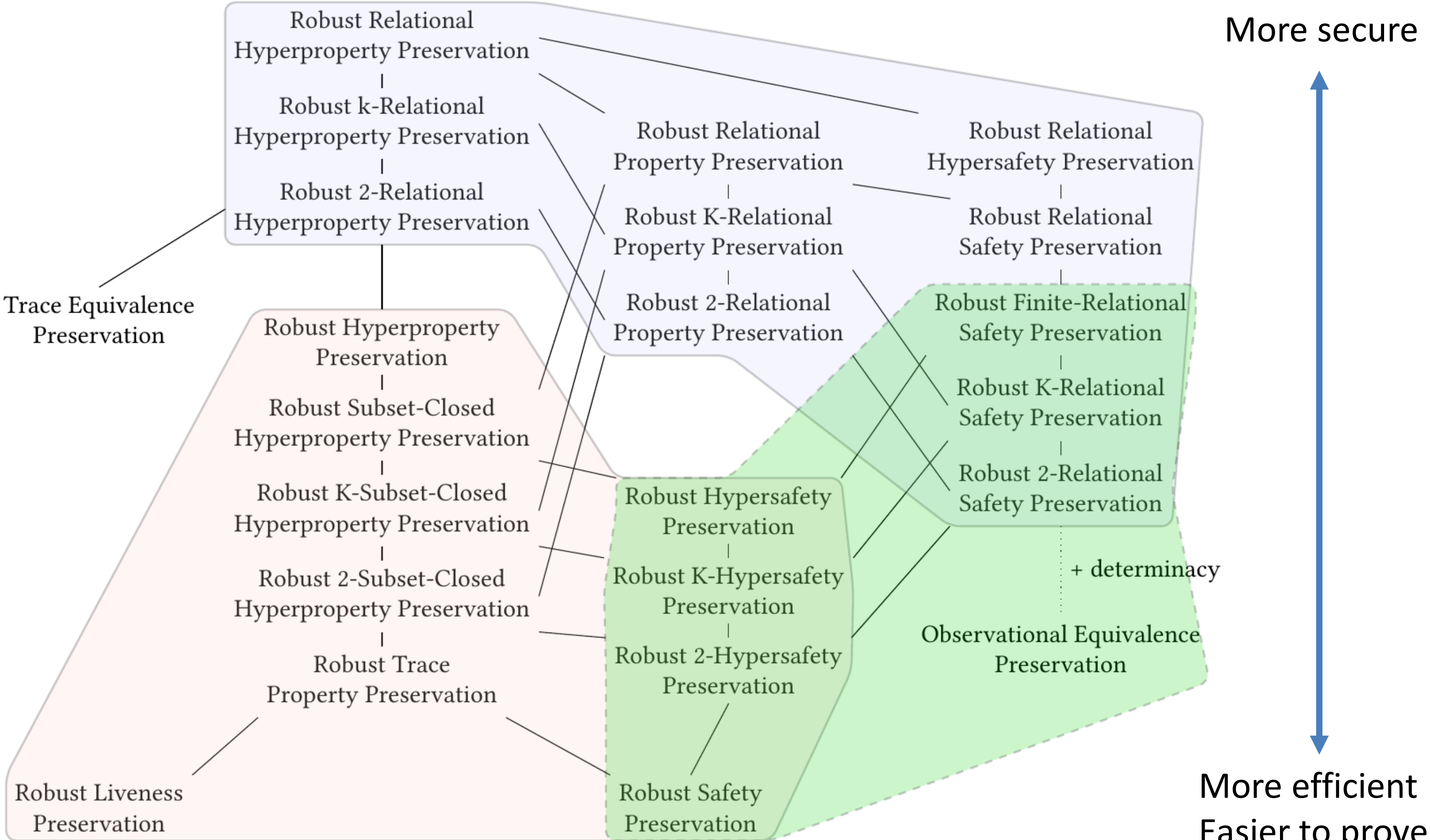
... against adversarial target-level contexts

What security properties should we preserve?

- **We explore a large space of security properties**
- **Study preserving various classes of ...**
  - **trace properties** (safety, liveness)
  - **hyperproperties** (e.g. noninterference)
  - **relational hyperproperties** (e.g. trace equivalence)

... against adversarial target-level contexts

- **No "one-size-fits-all solution"**
  - e.g. full abstraction does **not** imply the other criteria we study
  - **stronger** criteria are **harder** to achieve and prove, both challenging


Robust Relational Hyperproperty Preservation
Robust k-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation
Trace Equivalence Preservation
Robust Relational Property Preservation
Robust K-Relational Property Preservation
Robust 2-Relational Property Preservation
Robust Relational Hypersafety Preservation
Robust Relational Safety Preservation
Robust Finite-Relational Safety Preservation
Robust K-Relational Safety Preservation
Robust 2-Relational Safety Preservation
+ determinacy
Observational Equivalence Preservation
Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust K-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Trace Property Preservation
Robust Liveness Preservation
Robust Hypersafety Preservation
Robust K-Hypersafety Preservation
Robust 2-Hypersafety Preservation
Robust Safety Preservation
More secure
More efficient
Easier to prove

Robust Trace Property Preservation

Robust Trace Property Preservation

∀source component.

∀π trace property.

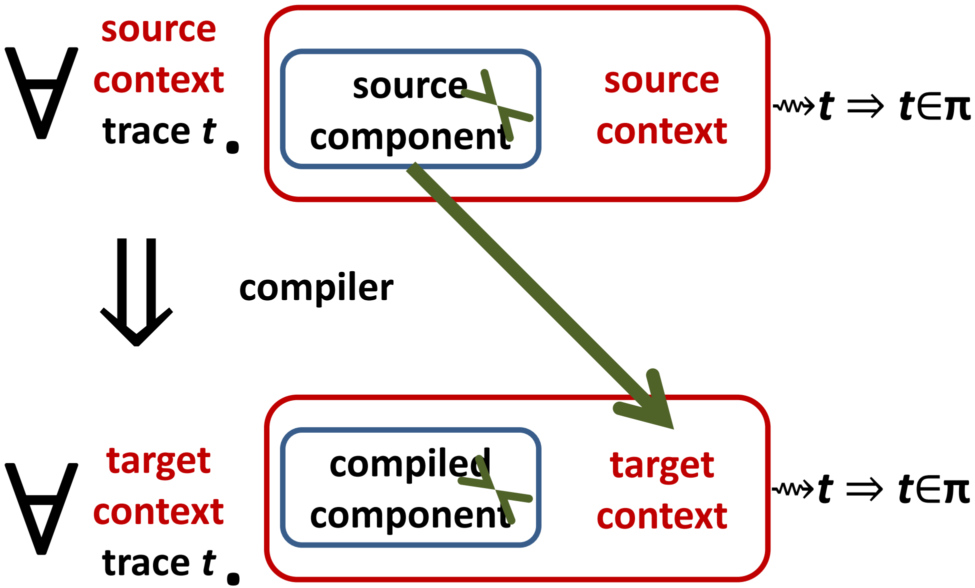
∀ source context trace *t* . [source component ⋈ source context] ⇝ *t* ⇒ *t*∈π

⇓

Robust Trace Property Preservation

∀source component.

∀π trace property.

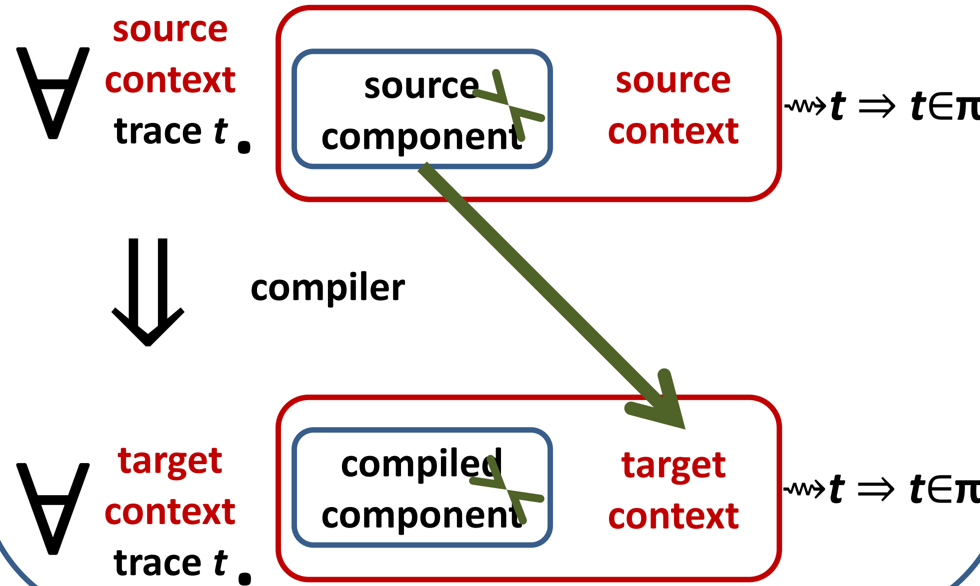


Robust Trace Property Preservation

property-based characterization

∀source component.

∀π trace property.



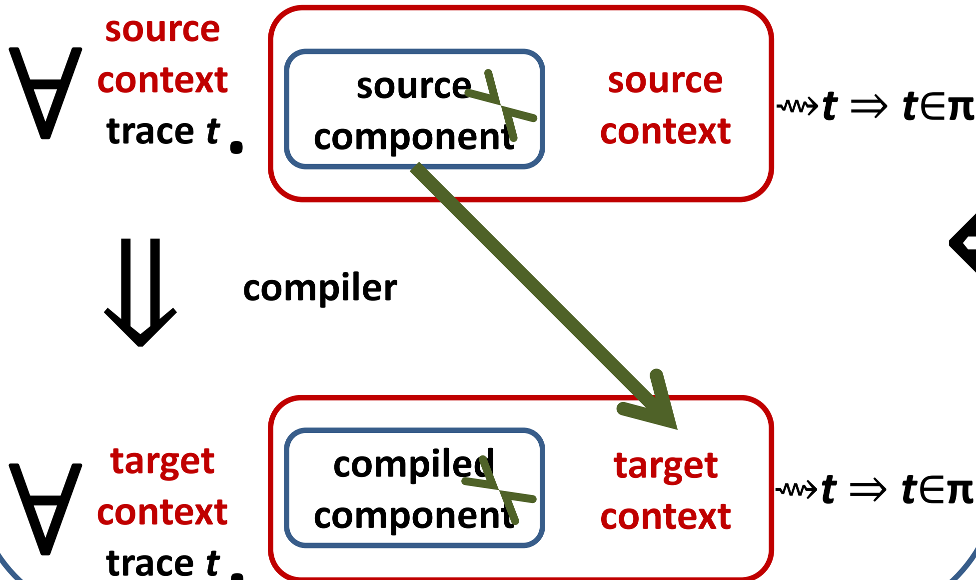
preservation of **robust** satisfaction

Robust Trace Property Preservation

property-based characterization

∀source component.

∀π trace property.



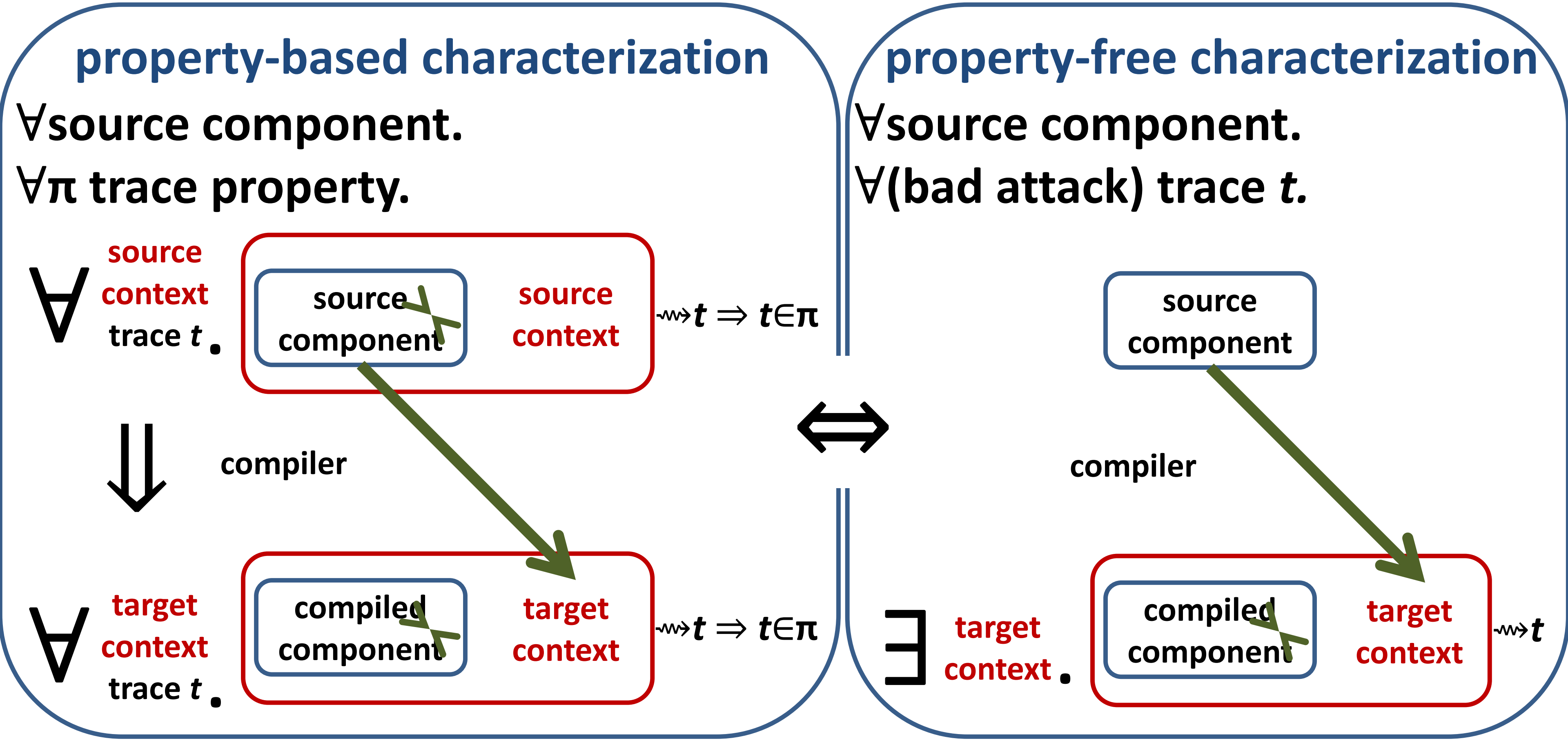
⇔

property-free characterization

preservation of **robust** satisfaction

how one can prove it

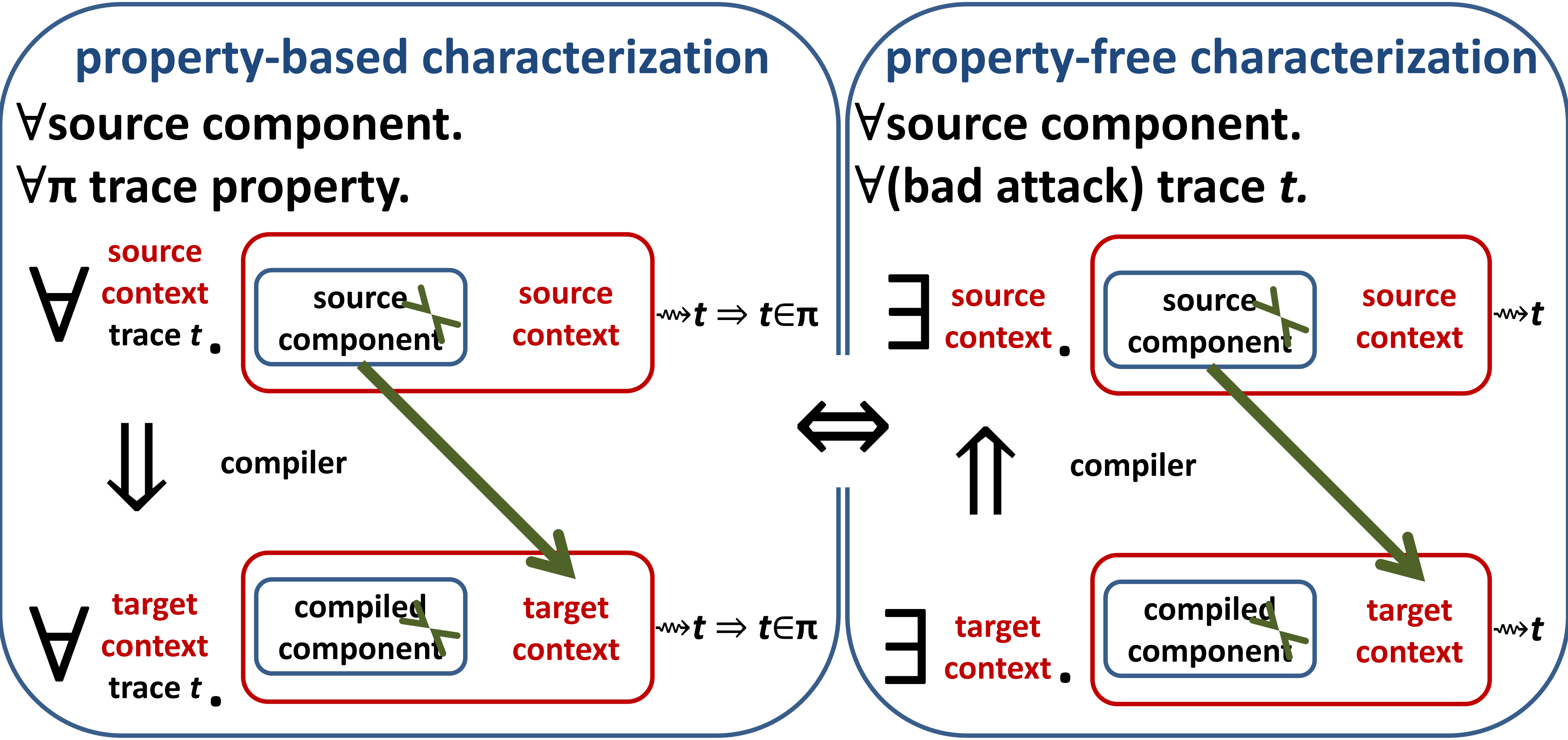
Robust Trace Property Preservation



preservation of **robust** satisfaction

how one can prove it

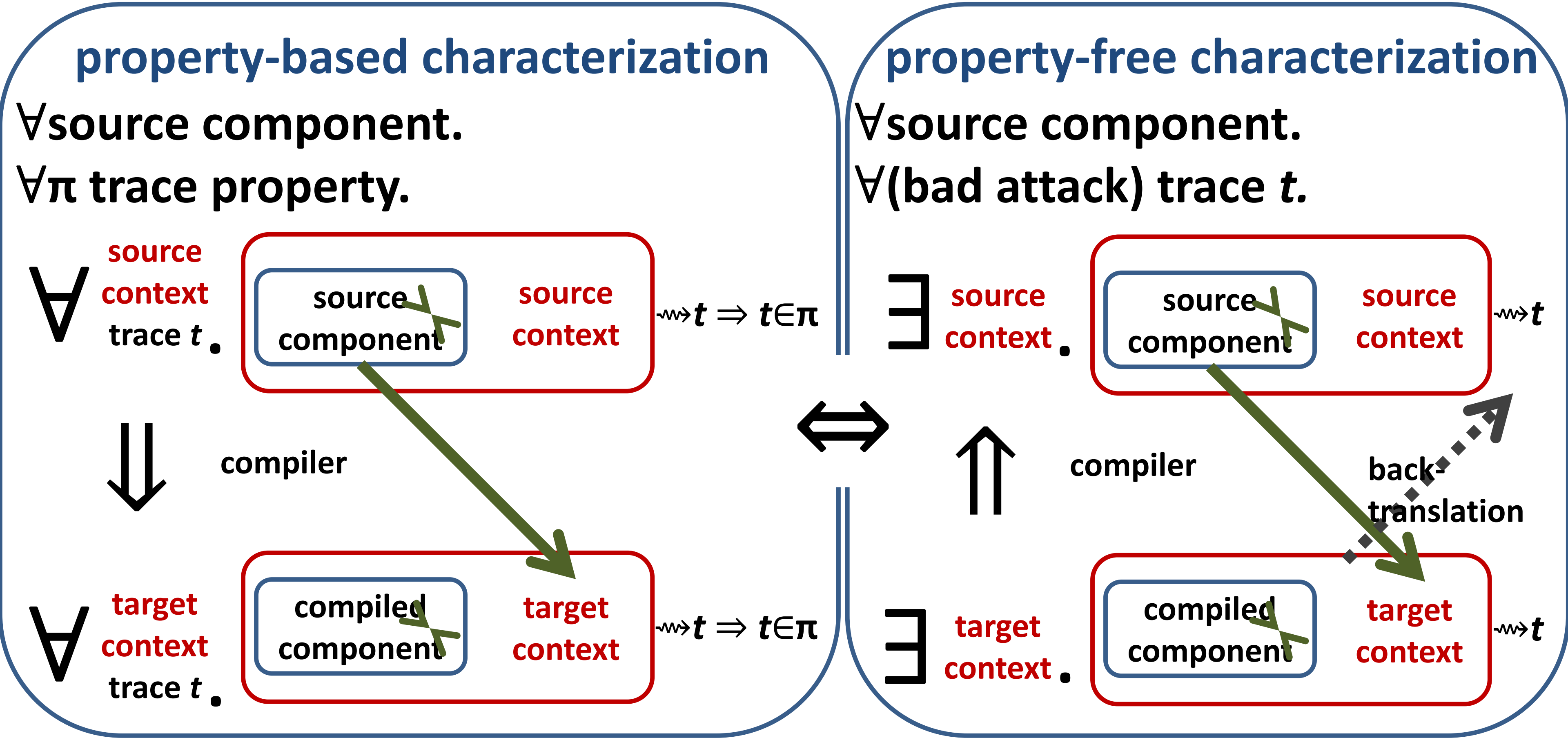
Robust Trace Property Preservation



preservation of **robust** satisfaction

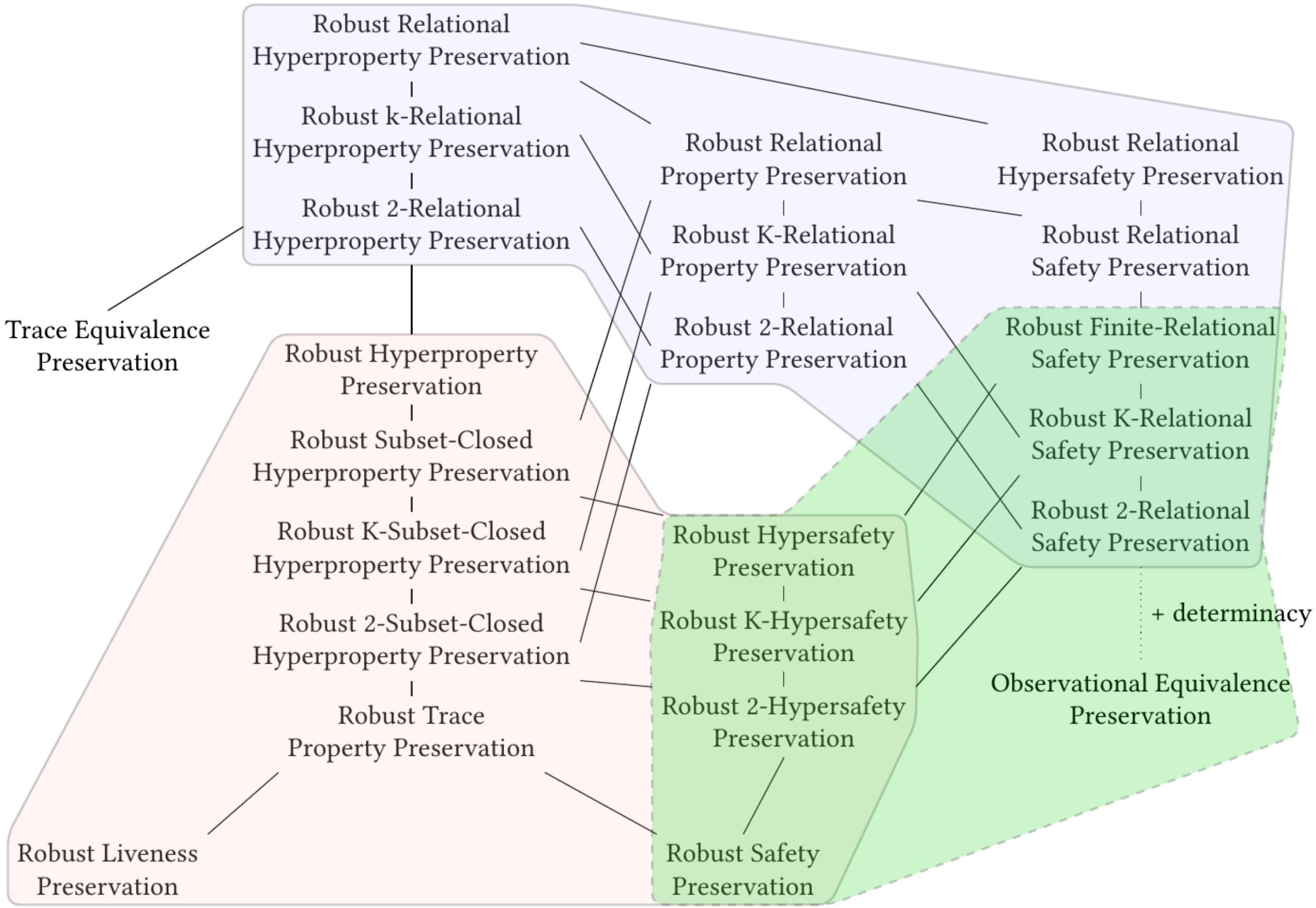
how one can prove it

Robust Trace Property Preservation

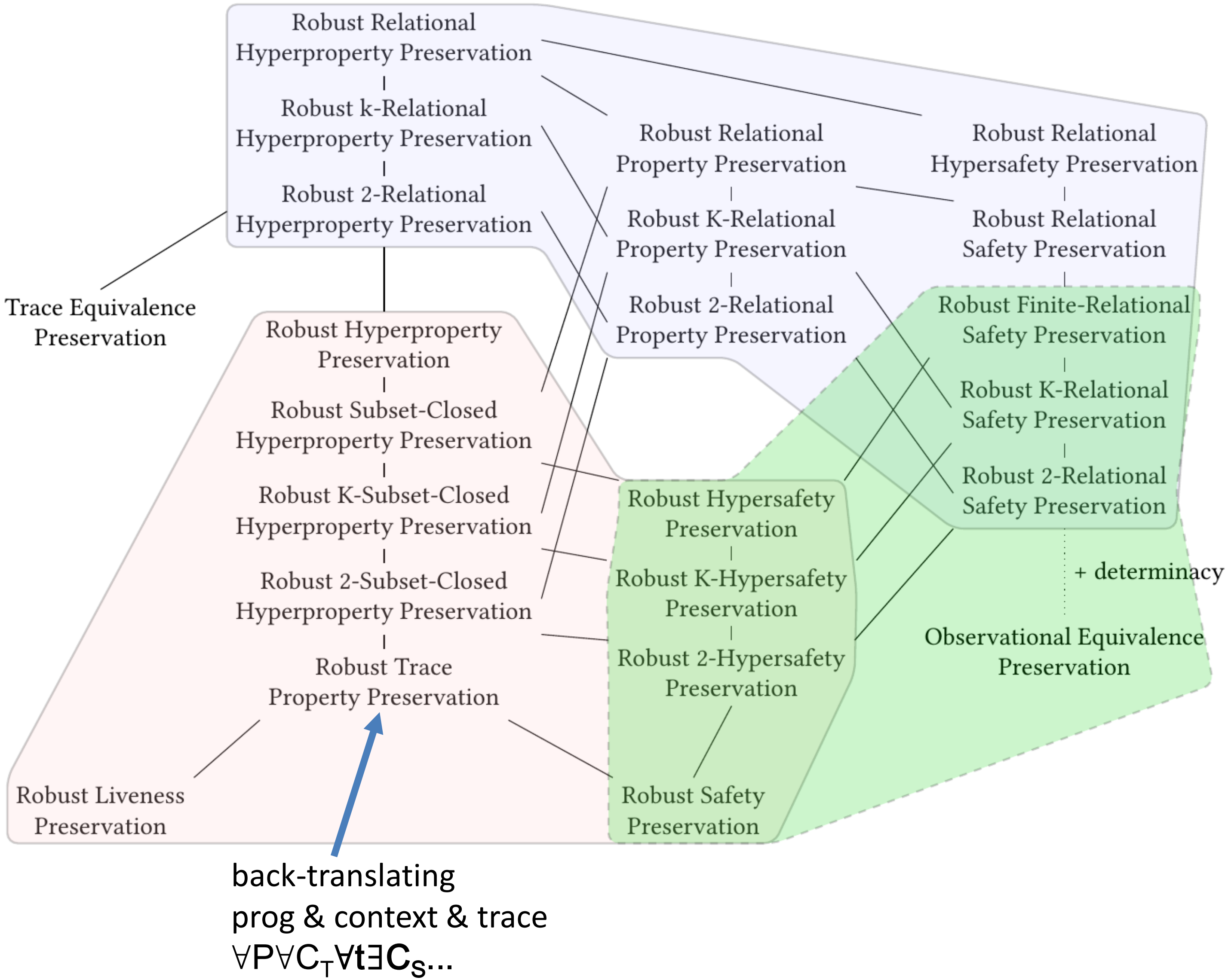


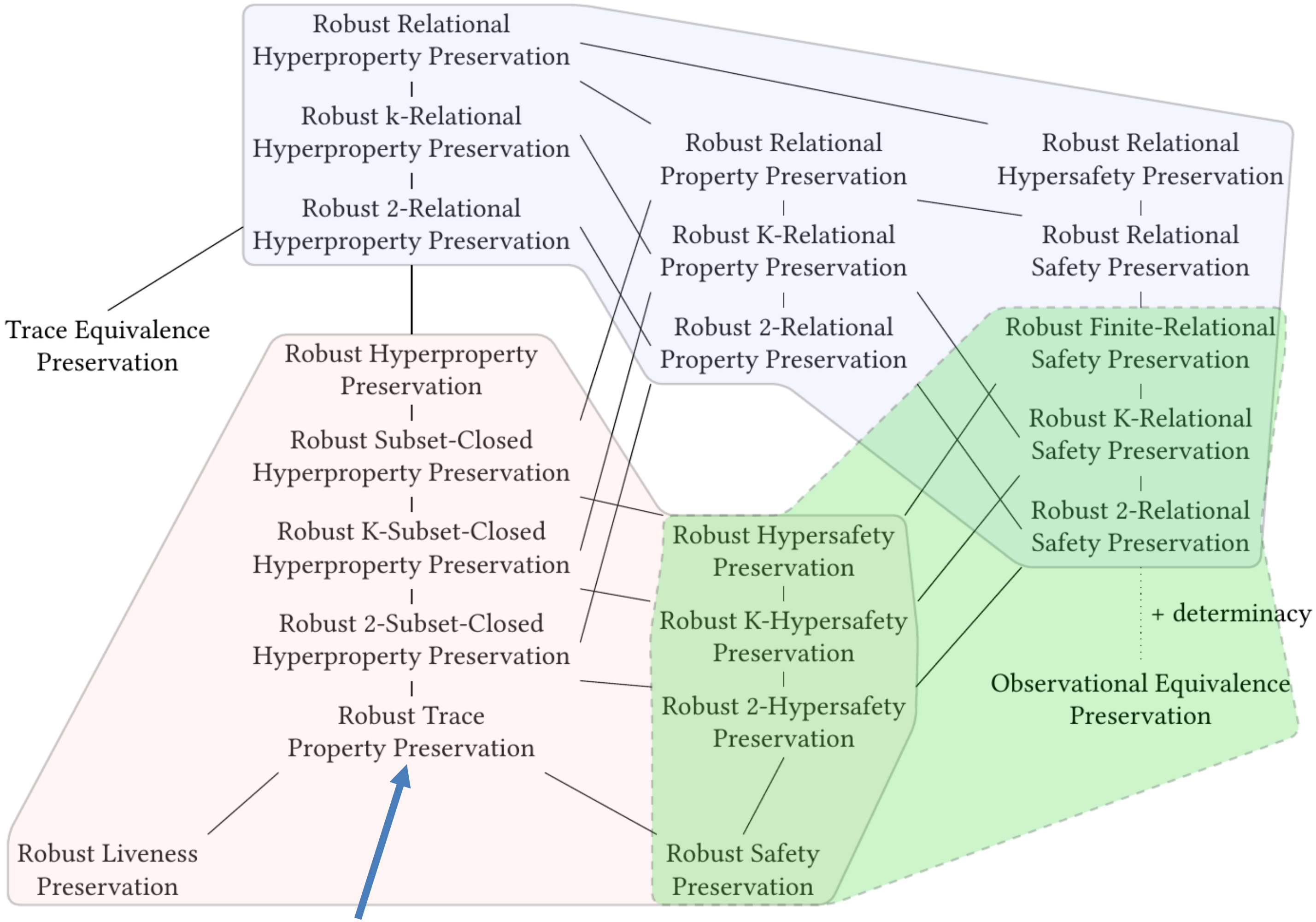
preservation of **robust** satisfaction

how one can prove it



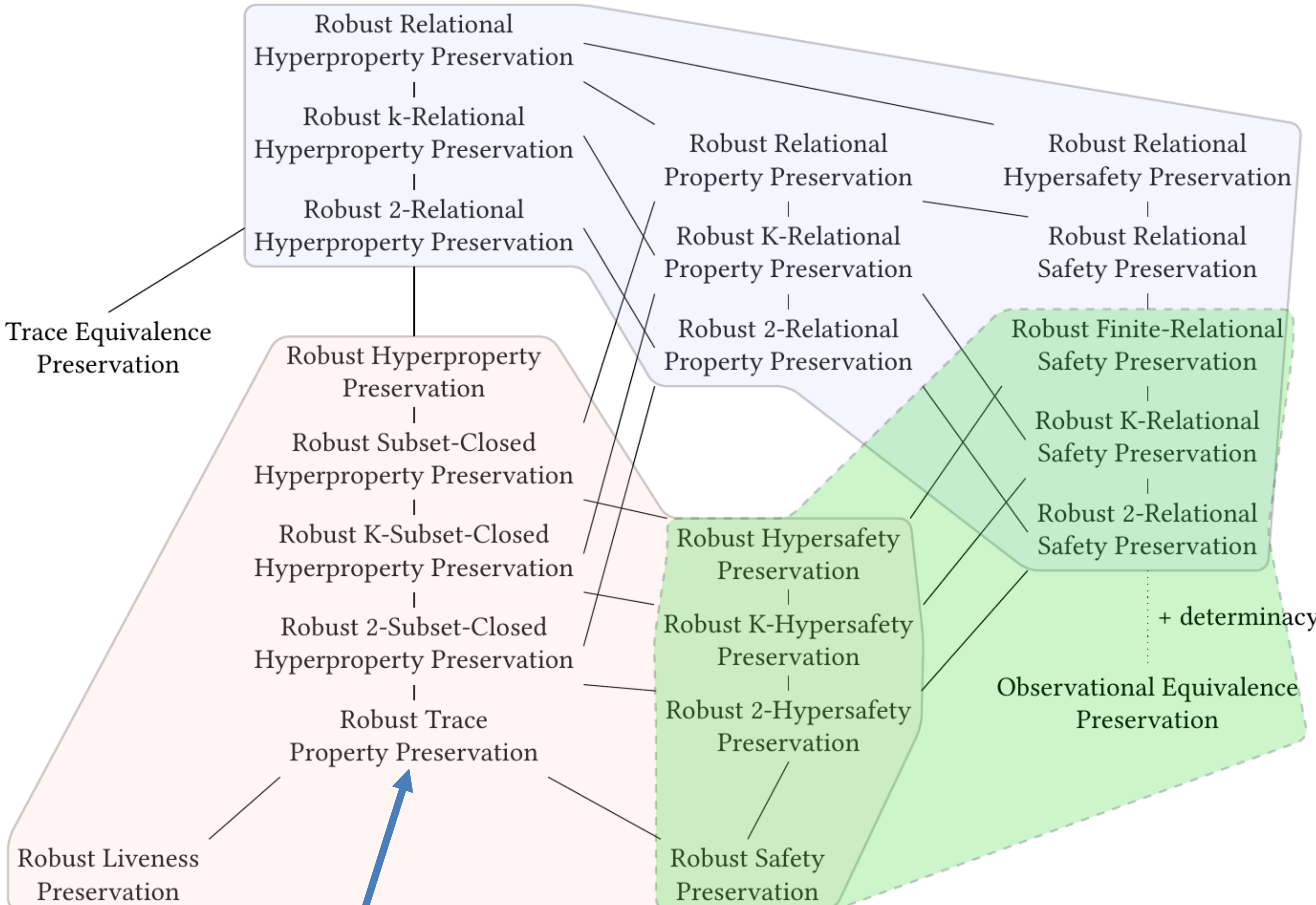
Robust Relational Hyperproperty Preservation
Robust k-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation
Robust Relational Property Preservation
Robust K-Relational Property Preservation
Robust 2-Relational Property Preservation
Robust Relational Hypersafety Preservation
Robust Relational Safety Preservation
Robust Finite-Relational Safety Preservation
Robust K-Relational Safety Preservation
Robust 2-Relational Safety Preservation
Trace Equivalence Preservation
Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust K-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Trace Property Preservation
Robust Liveness Preservation
Robust Hypersafety Preservation
Robust K-Hypersafety Preservation
Robust 2-Hypersafety Preservation
Robust Safety Preservation
+ determinacy
Observational Equivalence Preservation


Robust Relational Hyperproperty Preservation
Robust k-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation
Robust Relational Property Preservation
Robust K-Relational Property Preservation
Robust 2-Relational Property Preservation
Robust Relational Hypersafety Preservation
Robust Relational Safety Preservation
Robust Finite-Relational Safety Preservation
Robust K-Relational Safety Preservation
Robust 2-Relational Safety Preservation
Trace Equivalence Preservation
Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust K-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Trace Property Preservation
Robust Hypersafety Preservation
Robust K-Hypersafety Preservation
Robust 2-Hypersafety Preservation
+ determinacy
Observational Equivalence Preservation
Robust Liveness Preservation
Robust Safety Preservation
back-translating
prog & context & trace
∀P∀C_T∀t∃C_S...



back-translating
prog & context & trace
$\forall P \forall C_T \forall \mathbf{t} \exists \mathbf{C_S}$...

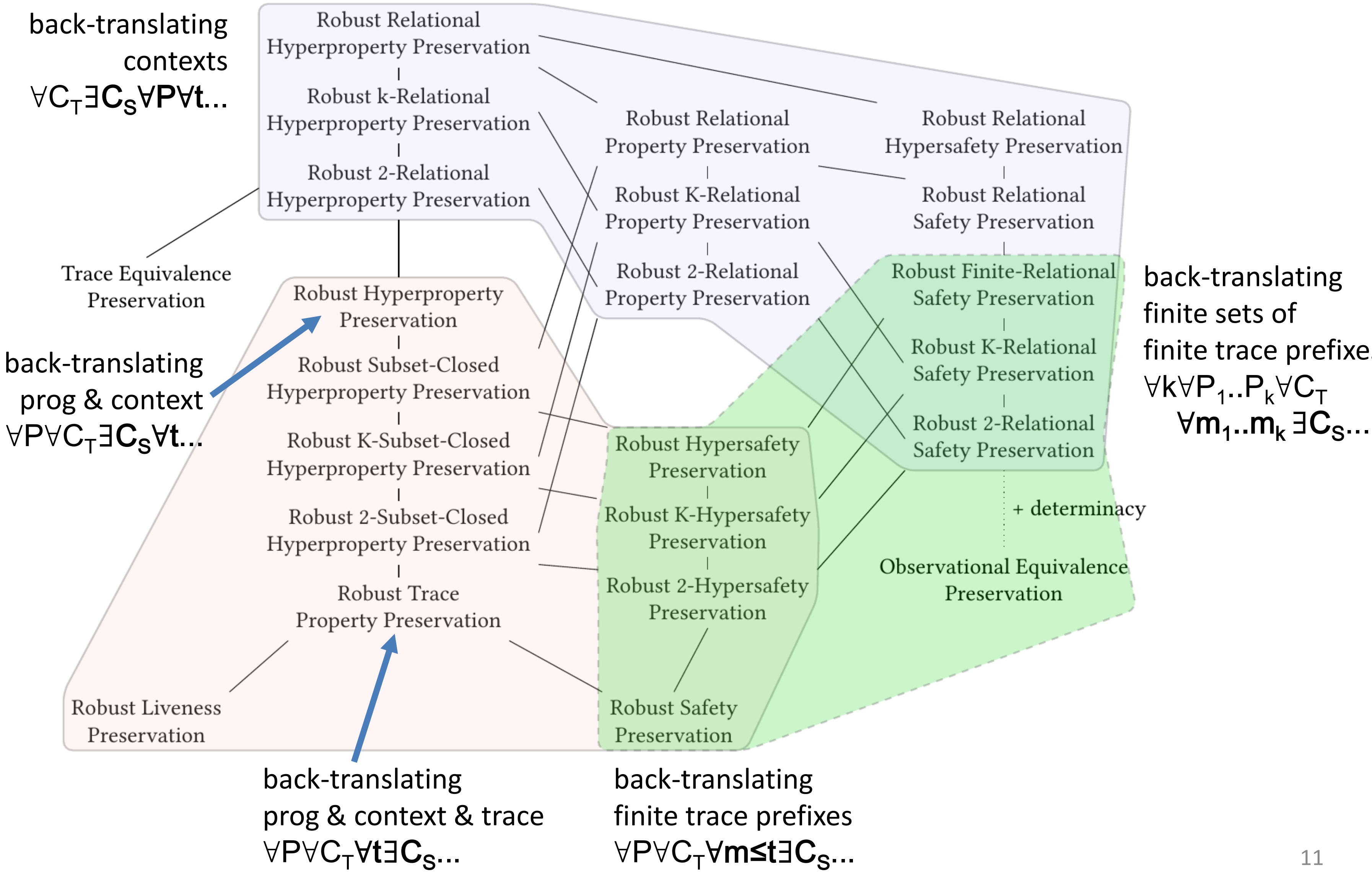
back-translating
finite trace prefixes
$\forall P \forall C_T \forall \mathbf{m \leq t} \exists \mathbf{C_S}$...



back-translating
finite sets of
finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T$
$\mathbf{\forall m_1..m_k} \exists \mathbf{C_S}$...

back-translating
prog & context & trace
$\forall P \forall C_T \mathbf{\forall t} \exists \mathbf{C_S}$...

back-translating
finite trace prefixes
$\forall P \forall C_T \mathbf{\forall m \leq t} \exists \mathbf{C_S}$...


back-translating contexts
$\forall C_T \exists C_S \forall P \forall t \ldots$
Robust Relational Hyperproperty Preservation
Robust k-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation
Robust Relational Property Preservation
Robust K-Relational Property Preservation
Robust 2-Relational Property Preservation
Robust Relational Hypersafety Preservation
Robust Relational Safety Preservation
Robust Finite-Relational Safety Preservation
Robust K-Relational Safety Preservation
Robust 2-Relational Safety Preservation
back-translating finite sets of finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T$
$\forall m_1..m_k \exists C_S \ldots$
Trace Equivalence Preservation
Robust Hyperproperty Preservation
back-translating prog & context
$\forall P \forall C_T \exists C_S \forall t \ldots$
Robust Subset-Closed Hyperproperty Preservation
Robust K-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Trace Property Preservation
Robust Liveness Preservation
Robust Hypersafety Preservation
Robust K-Hypersafety Preservation
Robust 2-Hypersafety Preservation
Robust Safety Preservation
+ determinacy
Observational Equivalence Preservation
back-translating prog & context & trace
$\forall P \forall C_T \forall t \exists C_S \ldots$
back-translating finite trace prefixes
$\forall P \forall C_T \forall m \leq t \exists C_S \ldots$

Results

- **Mapped the space of secure compilation criteria** based on robust "property" preservation

Results

- **Mapped the space of secure compilation criteria** based on robust "property" preservation
  - **Property-free characterizations** and **implications** in Coq

Results

- **Mapped the space of secure compilation criteria** based on robust "property" preservation
  - **Property-free characterizations** and **implications** in Coq
  - **Separation results** (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)

Results

- **Mapped the space of secure compilation criteria** based on robust "property" preservation
  - **Property-free characterizations** and **implications** in Coq
  - **Separation results** (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)
  - **Surprising collapse** between preserving all hyperproperties and preserving just hyperliveness

Results

- **Mapped the space of secure compilation criteria** based on robust "property" preservation
  - **Property-free characterizations** and **implications** in Coq
  - **Separation results** (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)
  - **Surprising collapse** between preserving all hyperproperties and preserving just hyperliveness
- Showed that **even strongest criterion is achievable**
  - for simple translation from a statically to a dynamically typed language with first-order functions and I/O

Some open problems

- **Practically achieving secure interoperability with lower-level code**

Some open problems

- **Practically achieving secure interoperability with lower-level code**
  - More realistic languages and secure compilation chains

Some open problems

- **Practically achieving secure interoperability with lower-level code**
  - **More realistic languages and secure compilation chains**
  - Achieve **noninterference preservation** in realistic attacker model **with side-channels**

Some open problems

- **Practically achieving secure interoperability with lower-level code**
  - **More realistic languages and secure compilation chains**
  - Achieve **noninterference preservation** in realistic attacker model **with side-channels**
  - **Efficient enforcement mechanisms**

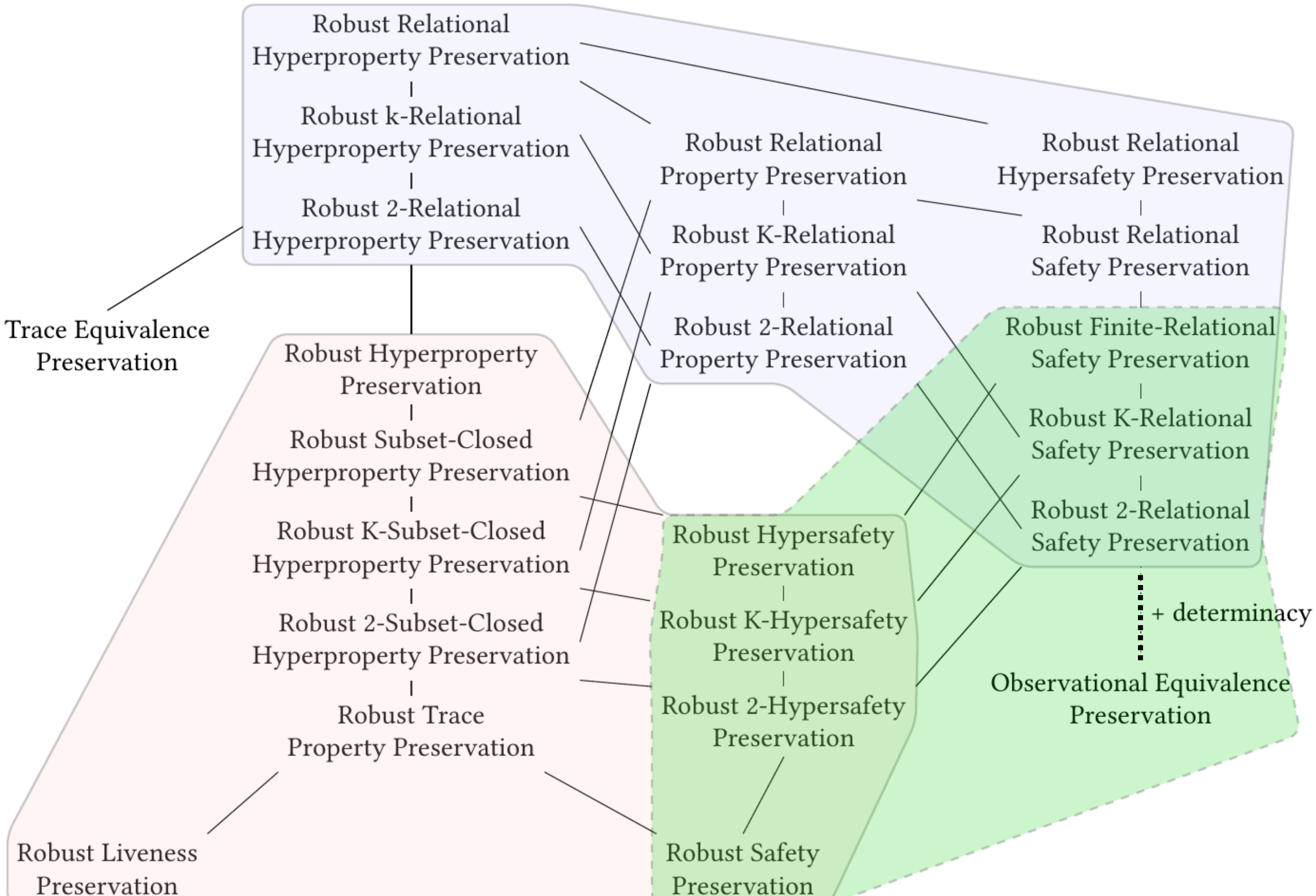
Some open problems

- **Practically achieving secure interoperability with lower-level code**
  - **More realistic languages and secure compilation chains**
  - Achieve **noninterference preservation** in realistic attacker model **with side-channels**
  - **Efficient enforcement mechanisms**
- **Scalable proof techniques for other criteria**
  - (hyper)liveness preservation (possible?)

Some open problems

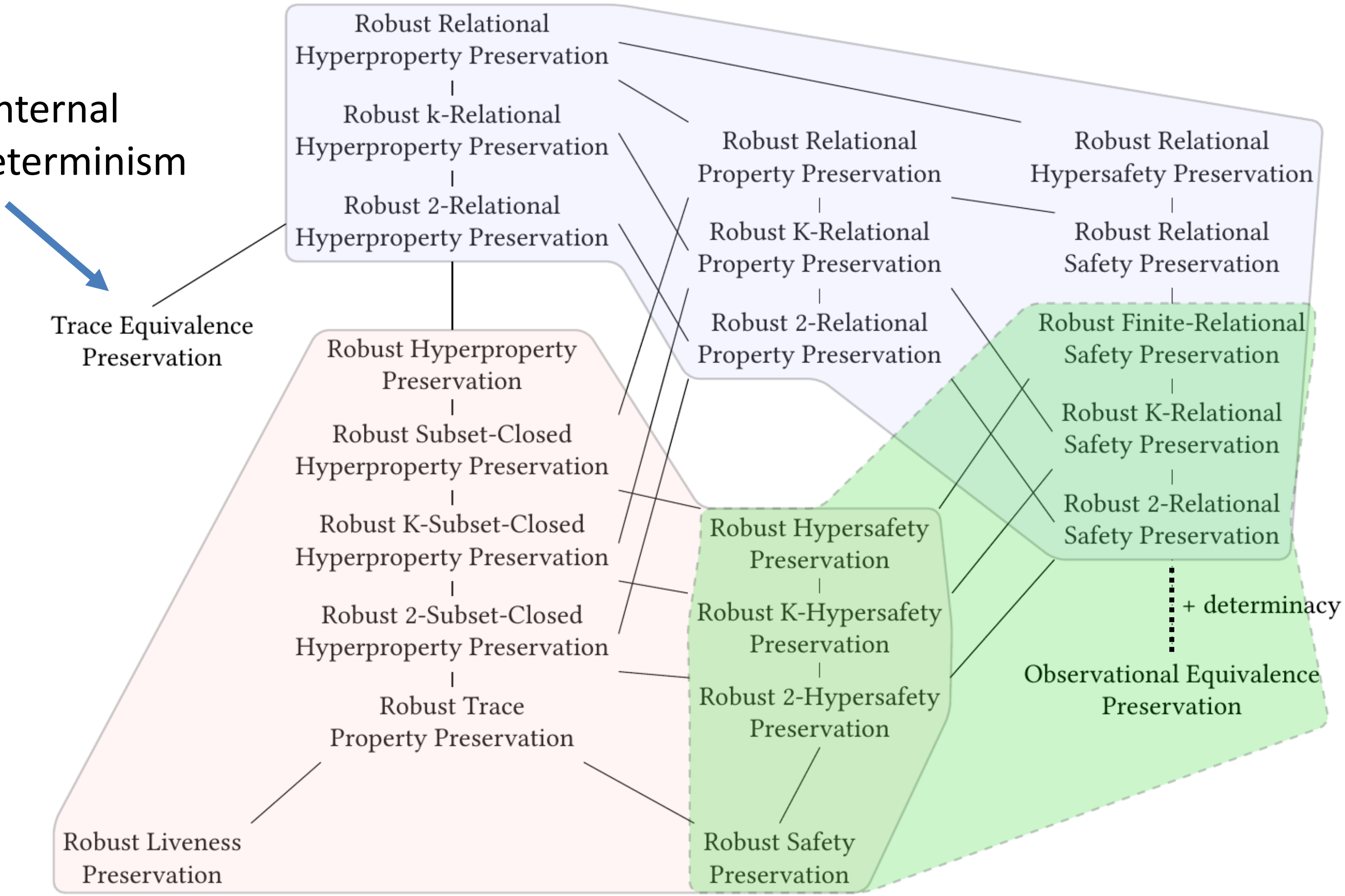
- **Practically achieving secure interoperability with lower-level code**
  - **More realistic languages and secure compilation chains**
  - Achieve **noninterference preservation** in realistic attacker model **with side-channels**
  - **Efficient enforcement mechanisms**
- **Scalable proof techniques for other criteria**
  - (hyper)liveness preservation (possible?)
- **Nontrivial relation between source and target traces**

Where is full abstraction?

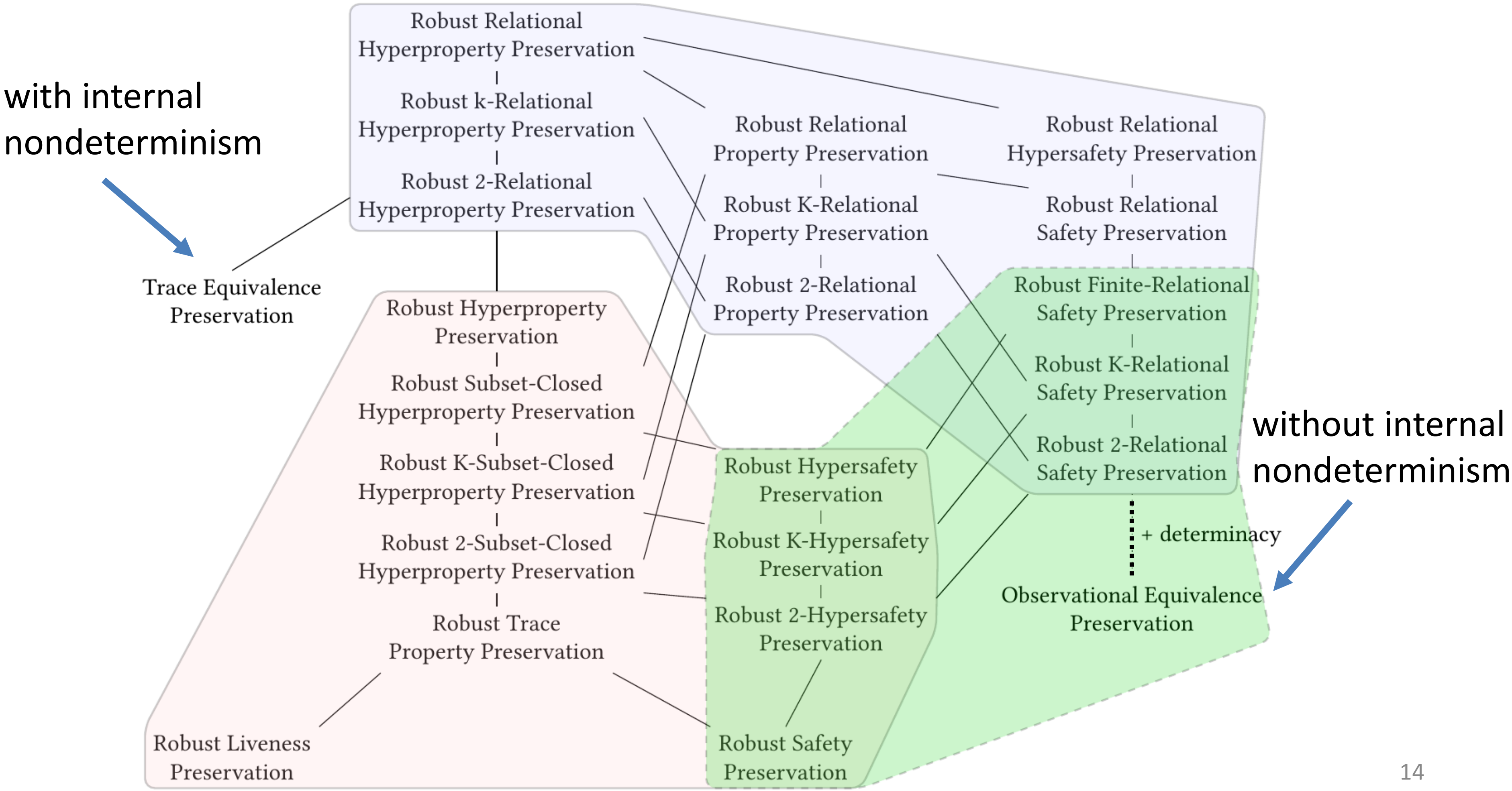


Where is full abstraction?

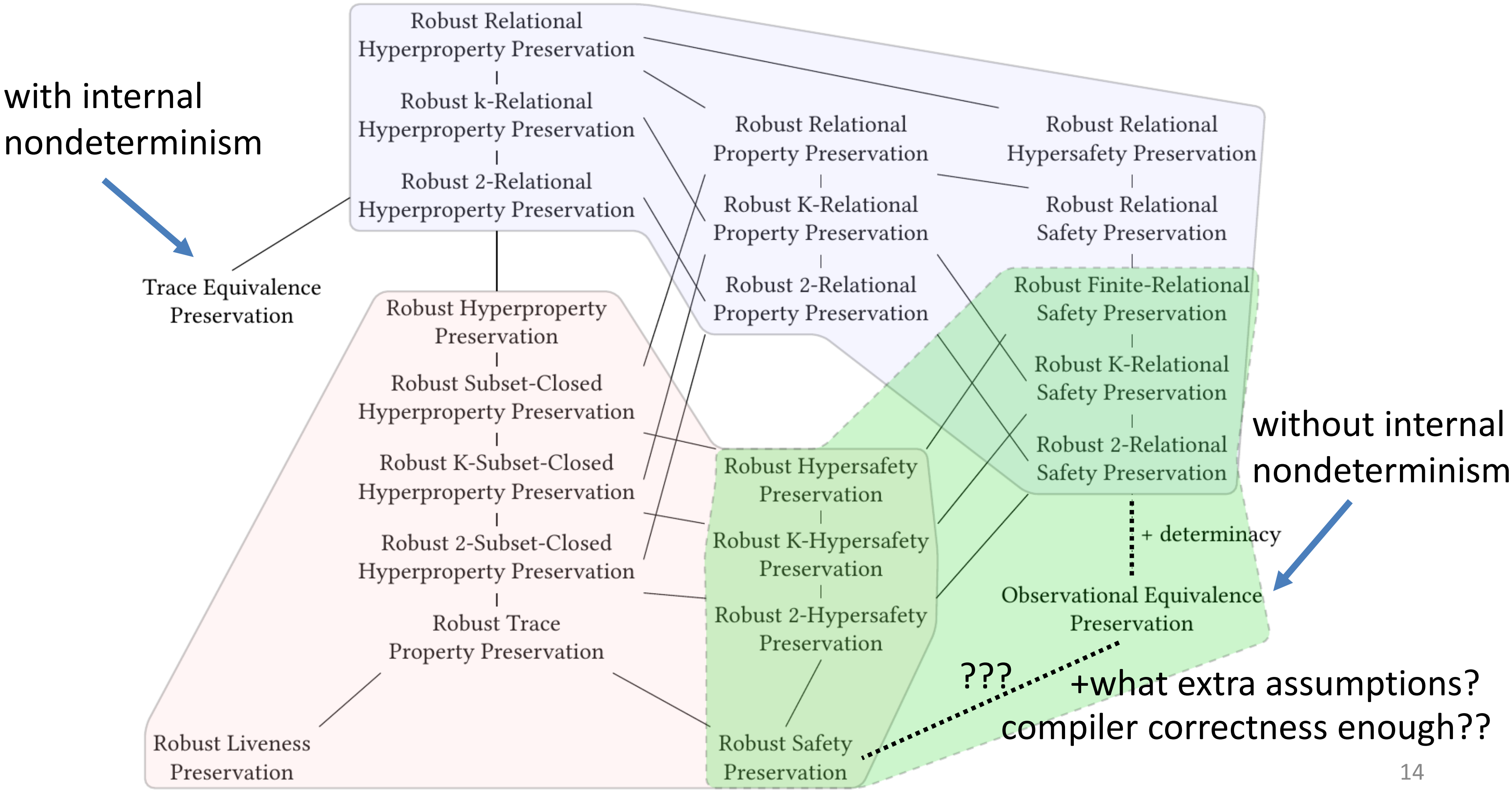
with internal nondeterminism



Where is full abstraction?



Where is full abstraction?



Part 2 of 2

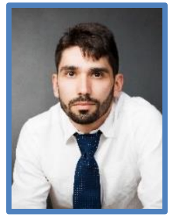
When Good Components Go Bad

Secure Compilation Despite Dynamic Compromise

https://arxiv.org/abs/1802.00588

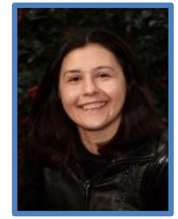
Collaborators for Part 2

Carmine Abate



Arthur Azevedo de Amorim

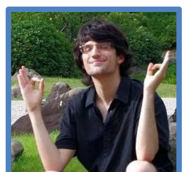
Rob Blanco



Ana Nora Evans

Guglielmo Fachini

Cătălin Hrițcu



Yannis Juglaret

Théo Laurent

Benjamin Pierce

Marco Stronati

Andrew Tolmach

Inria Paris CMU U. Virginia U. Trento Paris 7 ENS Paris Portland State UPenn

Undefined behavior

```
#include <string.h>
int main (int argc, char **argv) {
   char c[12];
   strcpy(c, argv[1]);
   return 0;
}
```

Undefined behavior

```
#include <string.h>
int main (int argc, char **argv) {
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```

Buffer overflow

```
$ gcc target.c
$ ./a.out haha
```

Undefined behavior

```
#include <string.h>
int main (int argc, char **argv) {
   char c[12];
   strcpy(c, argv[1]);
   return 0;
}
```

Buffer overflow

```
$ gcc target.c
$ ./a.out haha
$ ./a.out hahahahahahahahahaha
zsh: segmentation fault (core dumped)
```

Undefined behavior

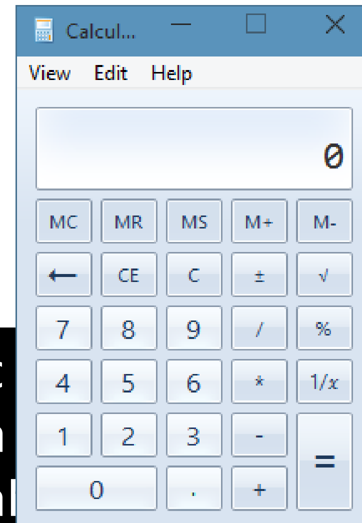
```
#include <string.h>
int main (int argc, char **argv) {
   char c[12];
   strcpy(c, argv[1]);
   return 0;
}
```

Buffer overflow

```
$ gcc target.c
$ ./a.out haha
$ ./a.out hahahahahahahahahaha
zsh: segmentation fault (core dumped)
$ ./exploit.sh | a.out
```

Undefined behavior

```
#include <string.h>
int main (int argc, char **argv) {
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```



erflow

```
$ gcc target.c
$ ./a.out haha
$ ./a.out hahal                haha
zsh: segmentation fault (core dumped)
$ ./exploit.sh | a.out
```

Practical mitigation: compartmentalization

Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces

Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"

Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"

Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"
  - "each components receives guarantees as long as it has not encountered undefined behavior"

Practical mitigation: compartmentalization

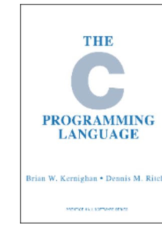
- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"
  - "each components receives guarantees as long as it has not encountered undefined behavior"

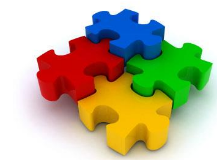
Goal 1: Formalize this

Goal 2: Build secure compilation chains

Goal 2: Build secure compilation chains

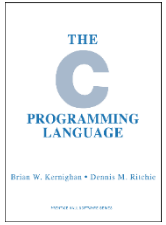
- **Add components to C**
  - interacting only via **strictly enforced interfaces**





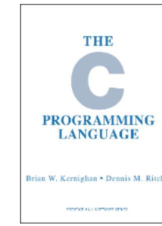
Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions**:
  - component separation, call-return discipline, ...



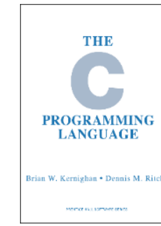
Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions**:
  - component separation, call-return discipline, ...
- **Secure compilation chain:**
  - compiler, linker, loader, runtime, system, hardware



Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions:**
  - component separation, call-return discipline, ...
- **Secure compilation chain:**
  - compiler, linker, loader, runtime, system, hardware
- **Use efficient enforcement mechanisms:**
  - OS processes (all web browsers) — WebAssembly (web browsers)
  - software fault isolation (SFI) — capability machines
  - hardware enclaves (SGX) — tagged architectures



Goal 1: Formalizing the security of compartmentalizing compilation

Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior

Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior

Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...

Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior** shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...
  - CompCert **already offers** this saner temporal model

Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...
  - CompCert **already offers** this saner temporal model
  - GCC and LLVM **currently violate** this model

Dynamic compromise

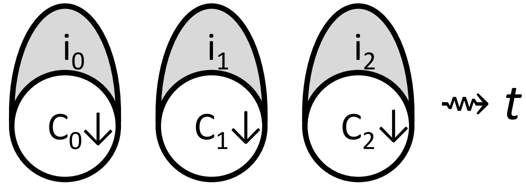
- each component gets guarantees as long as it has not encountered undefined behavior

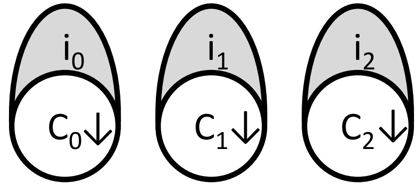
Dynamic compromise

- each component gets guarantees as long as it has not encountered undefined behavior
- a component only loses guarantees after an attacker discovers and exploits a vulnerability

Dynamic compromise

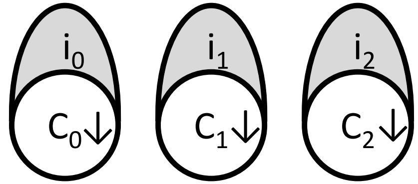
- each component gets guarantees as long as it has not encountered undefined behavior
- a component only loses guarantees after an attacker discovers and exploits a vulnerability
- the mere existence of vulnerabilities doesn't immediately make a component compromised

If  then

If 

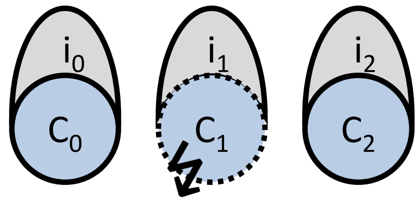
⇝ *t* then

∃ a **dynamic compromise scenario** explaining *t* in source language

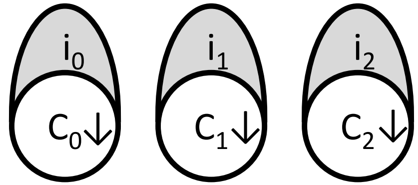
If 

$\rightsquigarrow$ *t* then

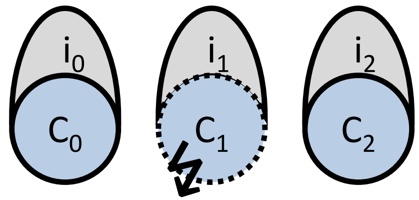
∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0) 

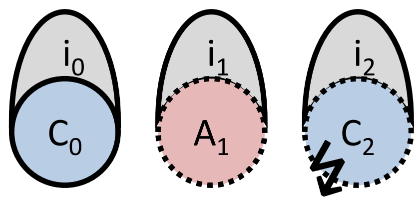
$\rightsquigarrow^*$ m_1;Undef(C_1)

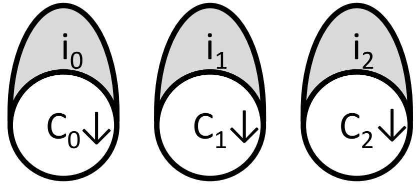
If  ⇝ t then

∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

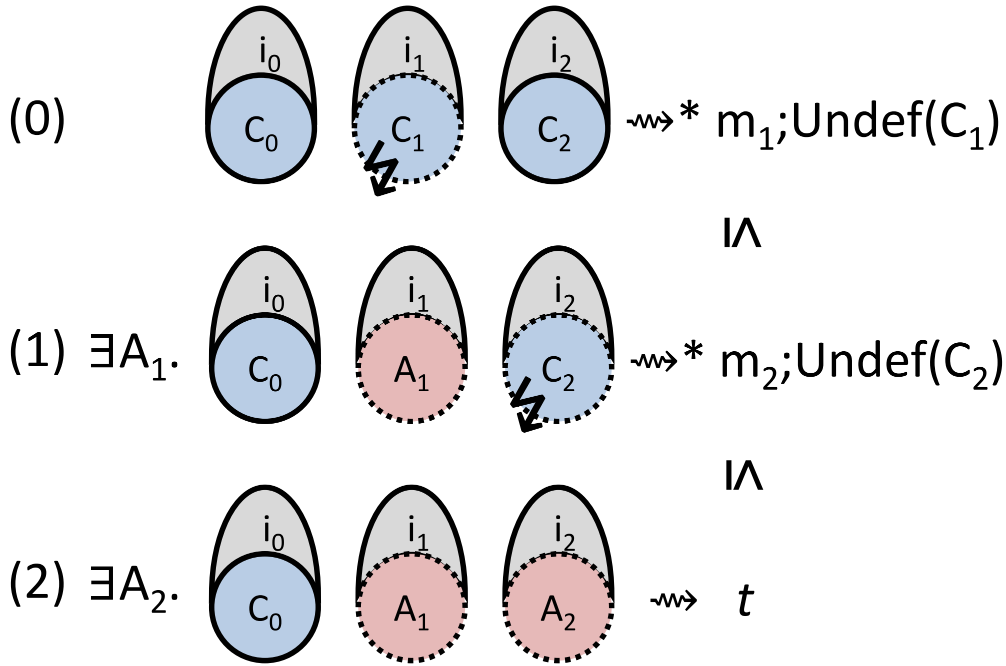
(0)  ⇝* m_1;Undef(C_1)

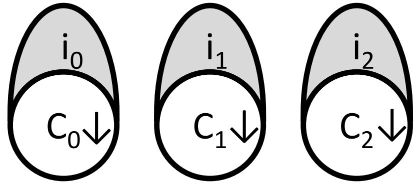
IΛ

(1) ∃A_1.  ⇝* m_2;Undef(C_2)

If  ⇝ t then

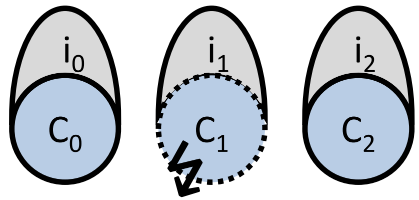
∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:



If 

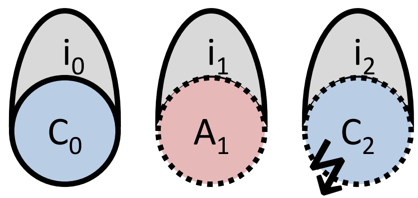
$\rightsquigarrow t$ then

$\exists$ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0) 

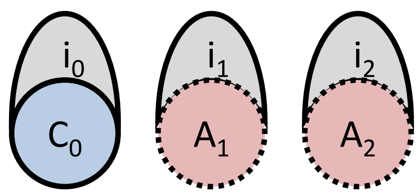
$\rightsquigarrow^* m_1;\mathrm{Undef}(C_1)$

$\leq$

(1) $\exists A_1.$ 

$\rightsquigarrow^* m_2;\mathrm{Undef}(C_2)$

$\leq$

(2) $\exists A_2.$ 

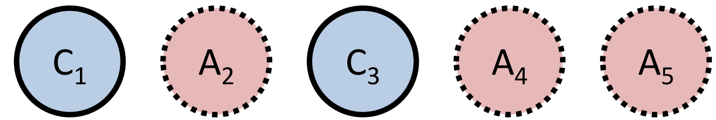
$\rightsquigarrow t$

Trace is very helpful
- detect undefined behavior
- rewind execution

Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe low-level languages)

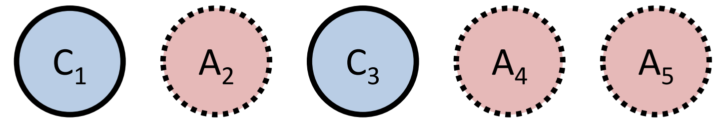
Mutual distrust



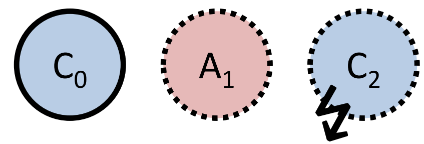
Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust



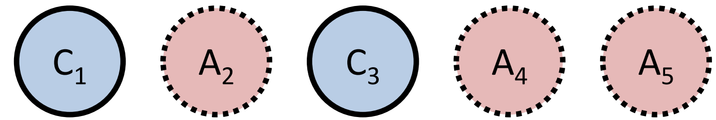
Dynamic compromise

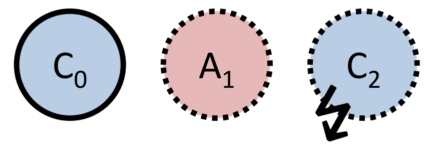


$\Downarrow m_2; \text{Undef}(C_2)$

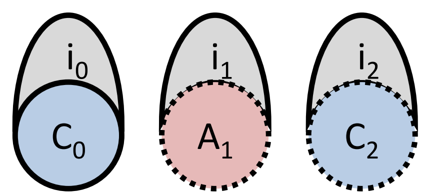
Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe low-level languages)

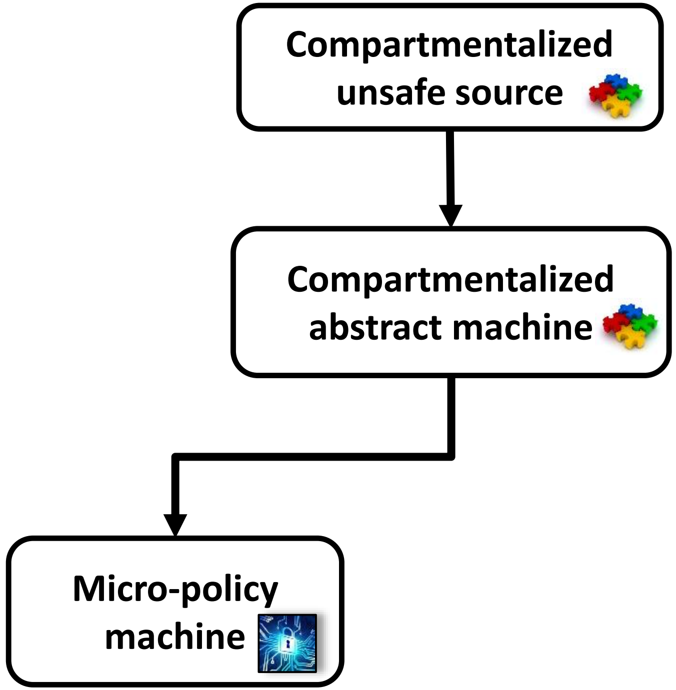
Mutual distrust 

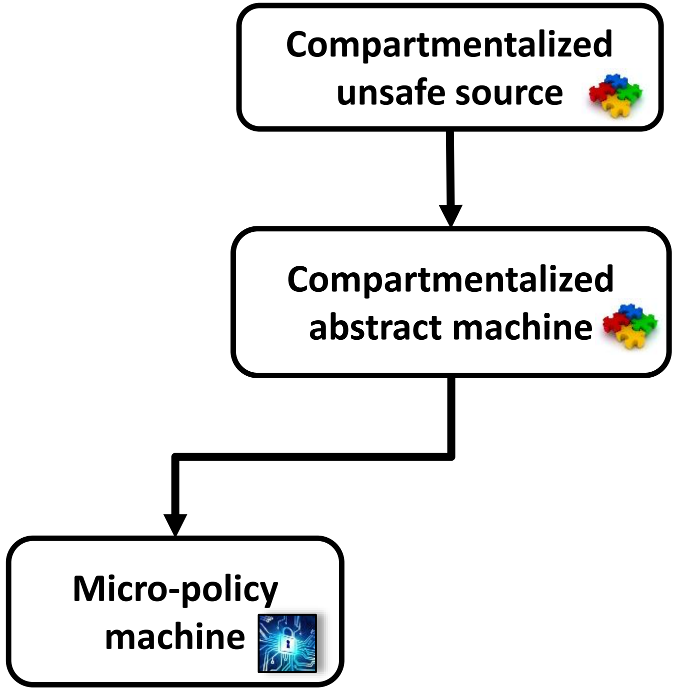
Dynamic compromise 

$\Downarrow m_2; \mathrm{Undef}(C_2)$

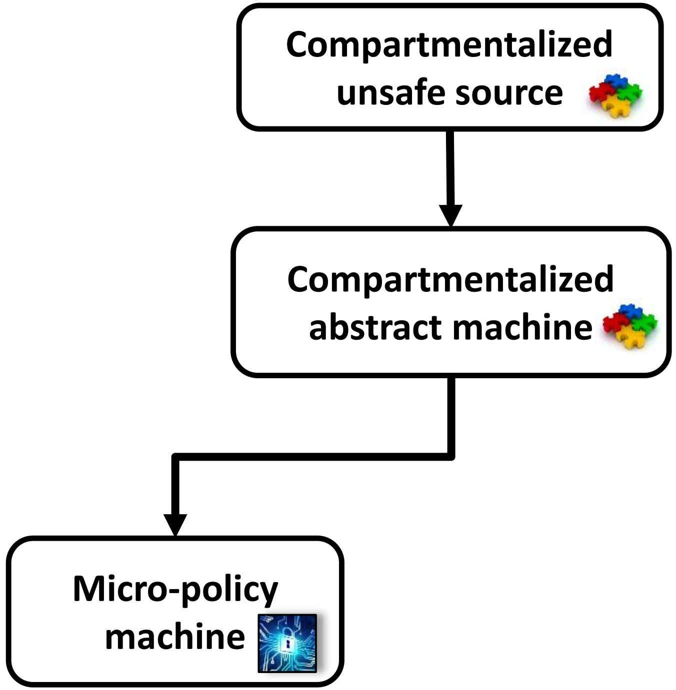
Static privilege 

Goal 2: Towards building secure compilation chains


Compartmentalized unsafe source
Compartmentalized abstract machine
Micro-policy machine

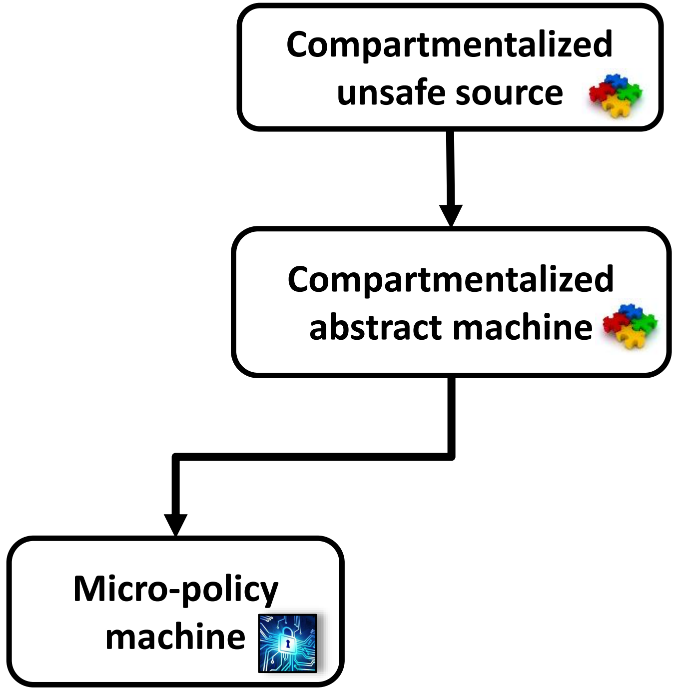


Buffers, procedures, components interacting via **strictly enforced interfaces**



Buffers, procedures, components interacting via **strictly enforced interfaces**

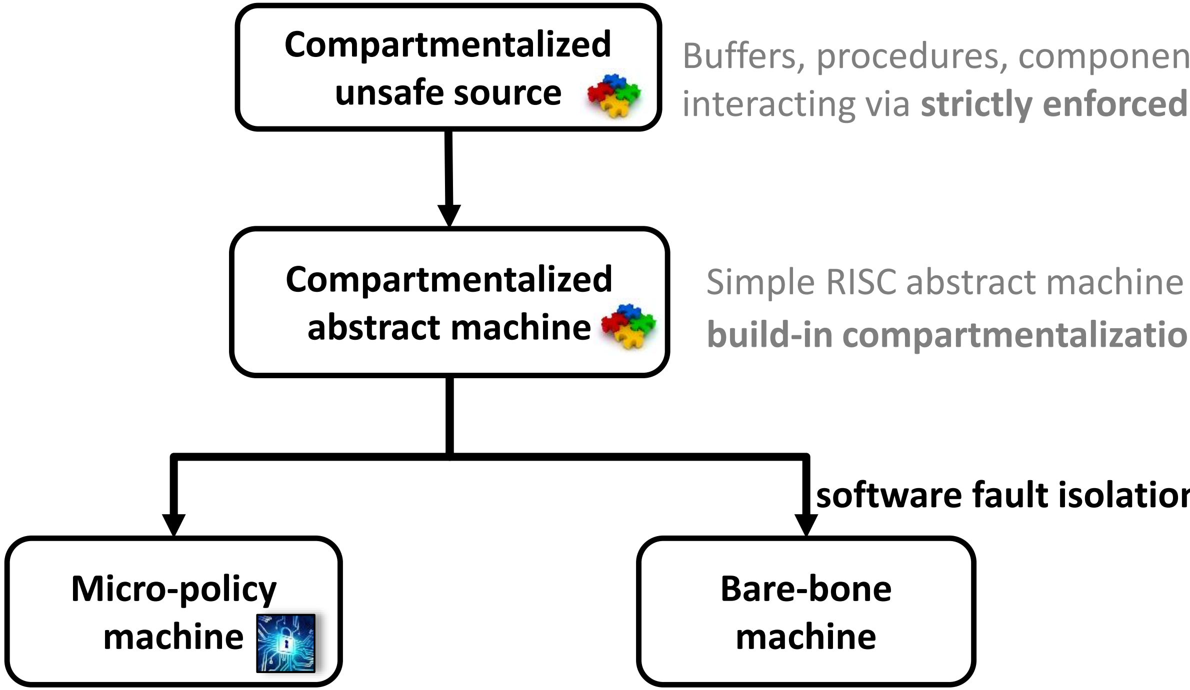
Simple RISC abstract machine with **build-in compartmentalization**



Buffers, procedures, components interacting via **strictly enforced interfaces**

Simple RISC abstract machine with **build-in compartmentalization**

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

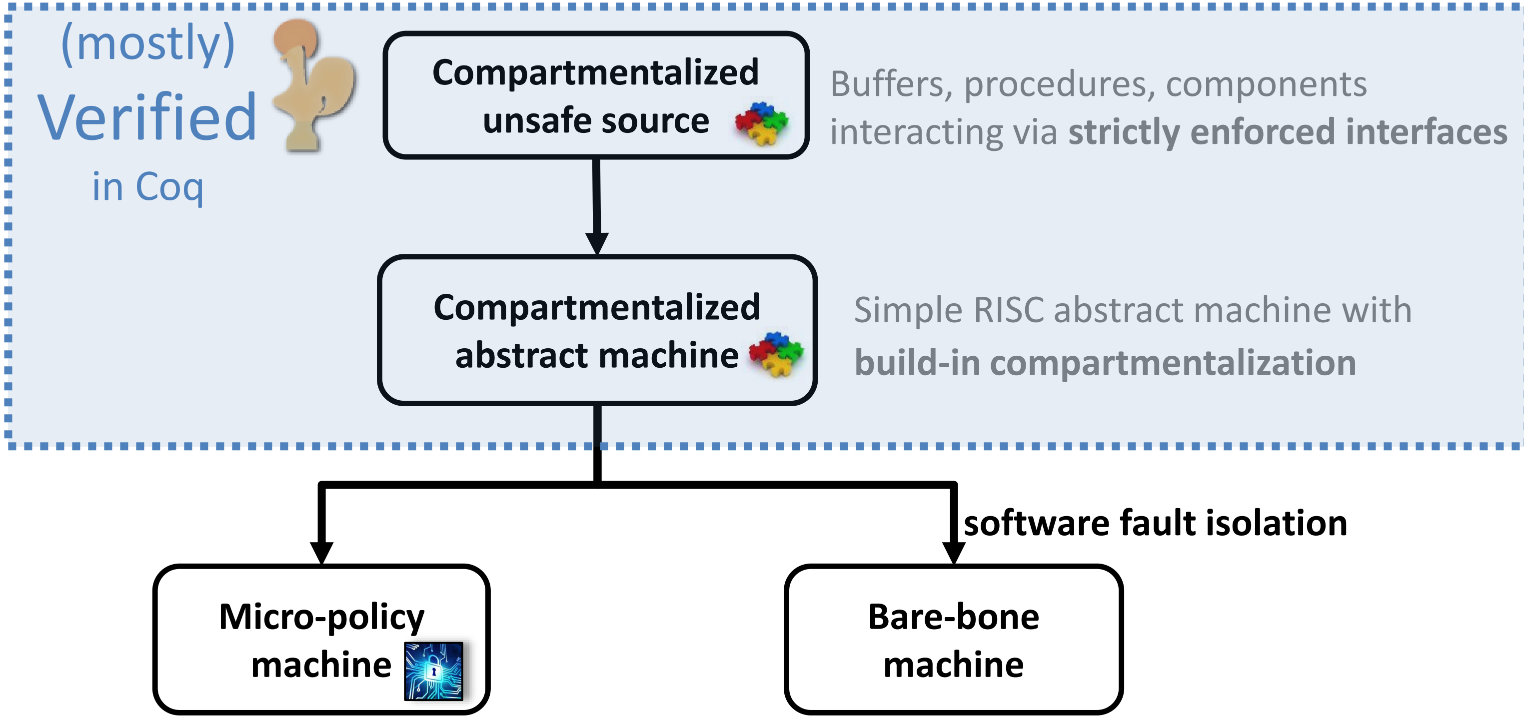


Buffers, procedures, components interacting via **strictly enforced interfaces**

Simple RISC abstract machine with **build-in compartmentalization**

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)



Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)

(mostly) **Verified** in Coq

Compartmentalized unsafe source

Buffers, procedures, components interacting via **strictly enforced interfaces**

Compartmentalized abstract machine

Simple RISC abstract machine with **build-in compartmentalization**

Micro-policy machine

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline

(linear capabilities / linear entry points)

software fault isolation

Bare-bone machine

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline

(program rewriting, shadow call stack)

Systematically tested (with QuickChick)

Making this more practical ... next steps:

Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)

Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work

Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**

Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...

Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...
- **Support other enforcement mechanisms (back ends)**

Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...
- **Support other enforcement mechanisms (back ends)**
- **Measure & lower overhead**

Wrapping up

- **Secure interoperability with lower-level code**
  - **exploring a continuum, security vs efficiency tradeoff**
- **Secure compilation despite dynamic compromise**
  - **restrict scope of undefined behavior**
    - **spatially** to the component that caused it
    - **temporally** by treating UB as an observable trace event

Wrapping up

- **Secure interoperability with lower-level code**
  - **exploring a continuum, security vs efficiency tradeoff**
- **Secure compilation despite dynamic compromise**
  - **restrict scope of undefined behavior**
    - **spatially** to the component that caused it
    - **temporally** by treating UB as an observable trace event
- **We're hiring!**
  - PostDocs, Young Researchers, Interns, PhD students

BACKUP SLIDES

More goals of secure compilation

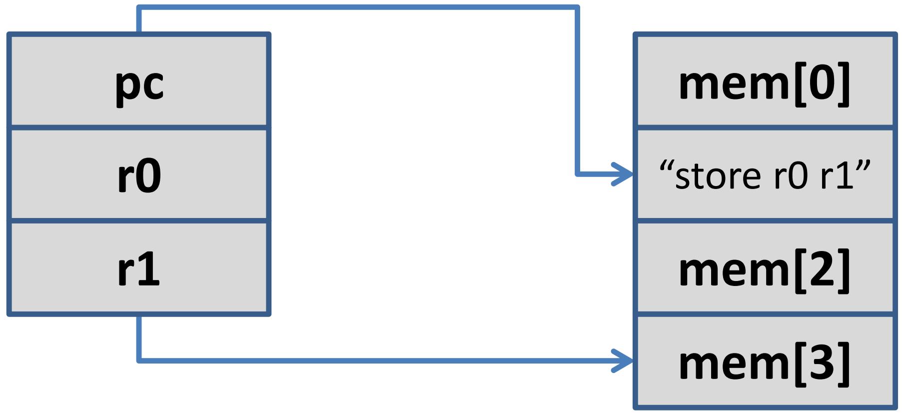
- **Enabling source-level security reasoning**
- **Making the source language safer**
  – memory and type safety, less/no undefined behavior
- **Making it easier to express security intent**
  – marking secrets, specifying security properties
- **Making exploits more difficult**
  – CFI, CPI, stack protection, randomization, diversity

Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring

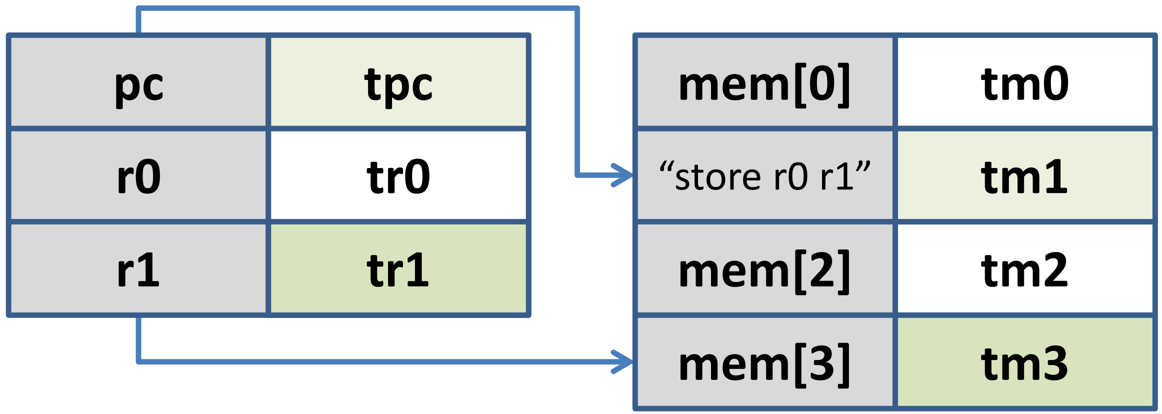
Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring



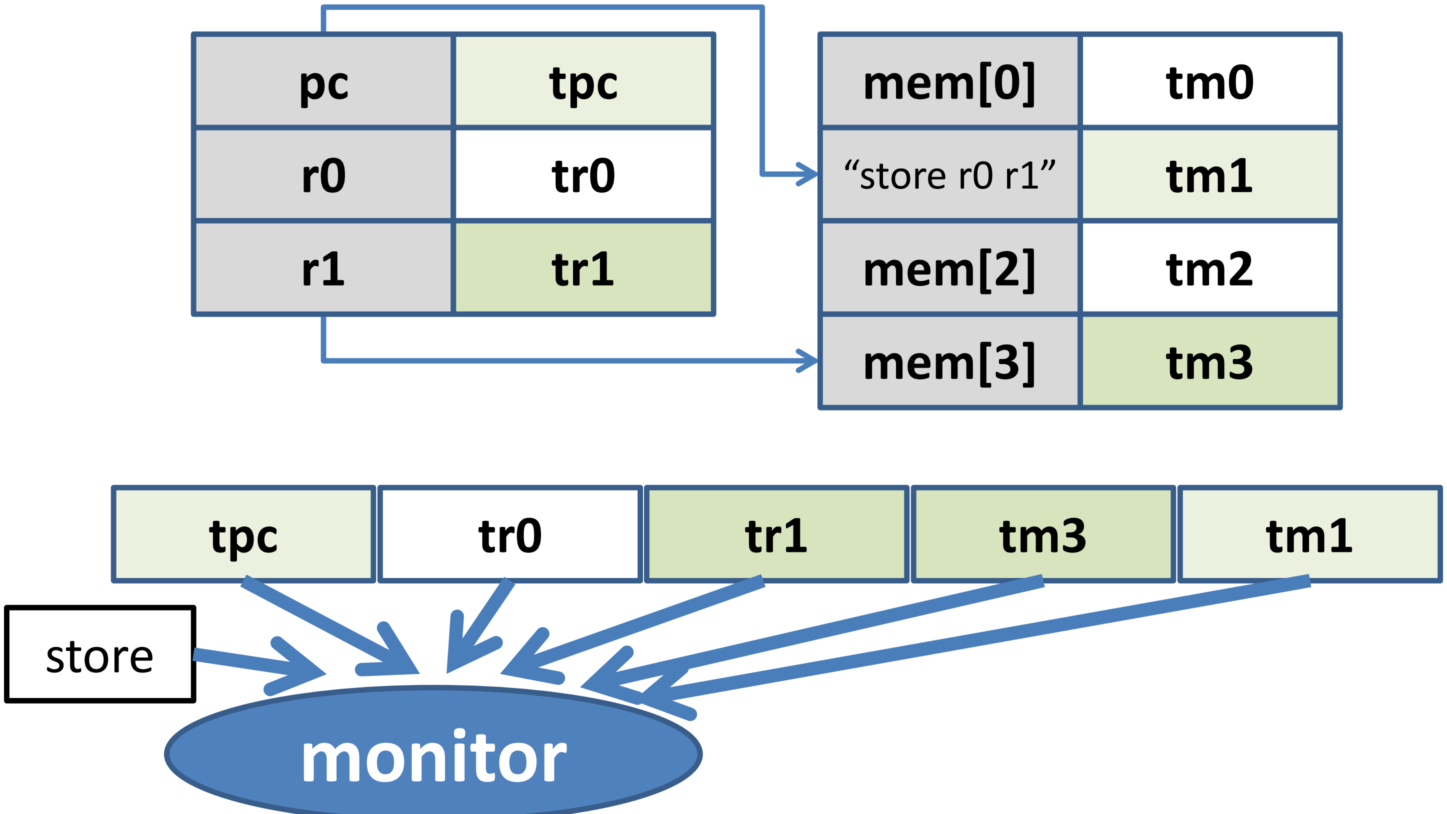
Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring



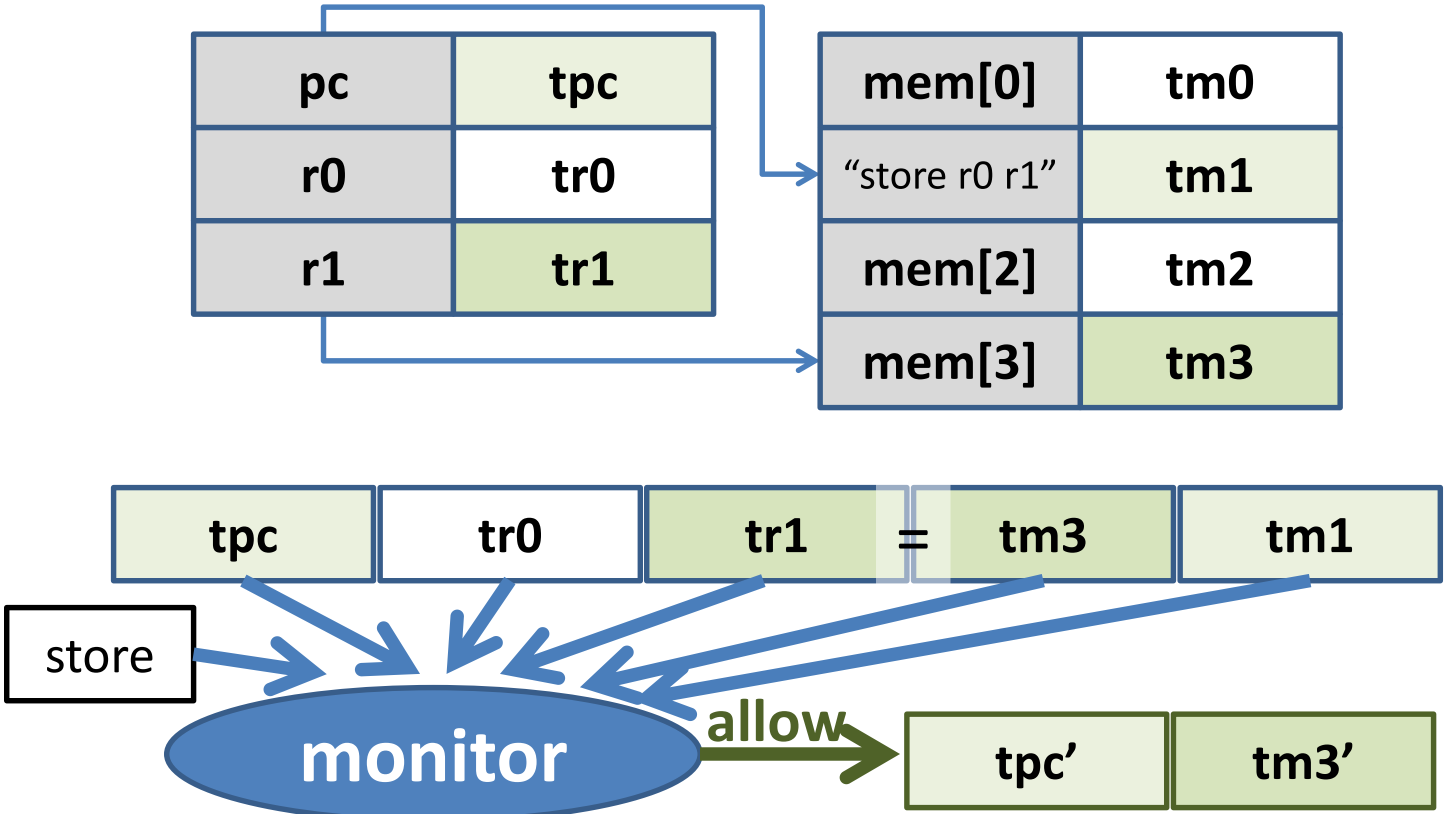
Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring



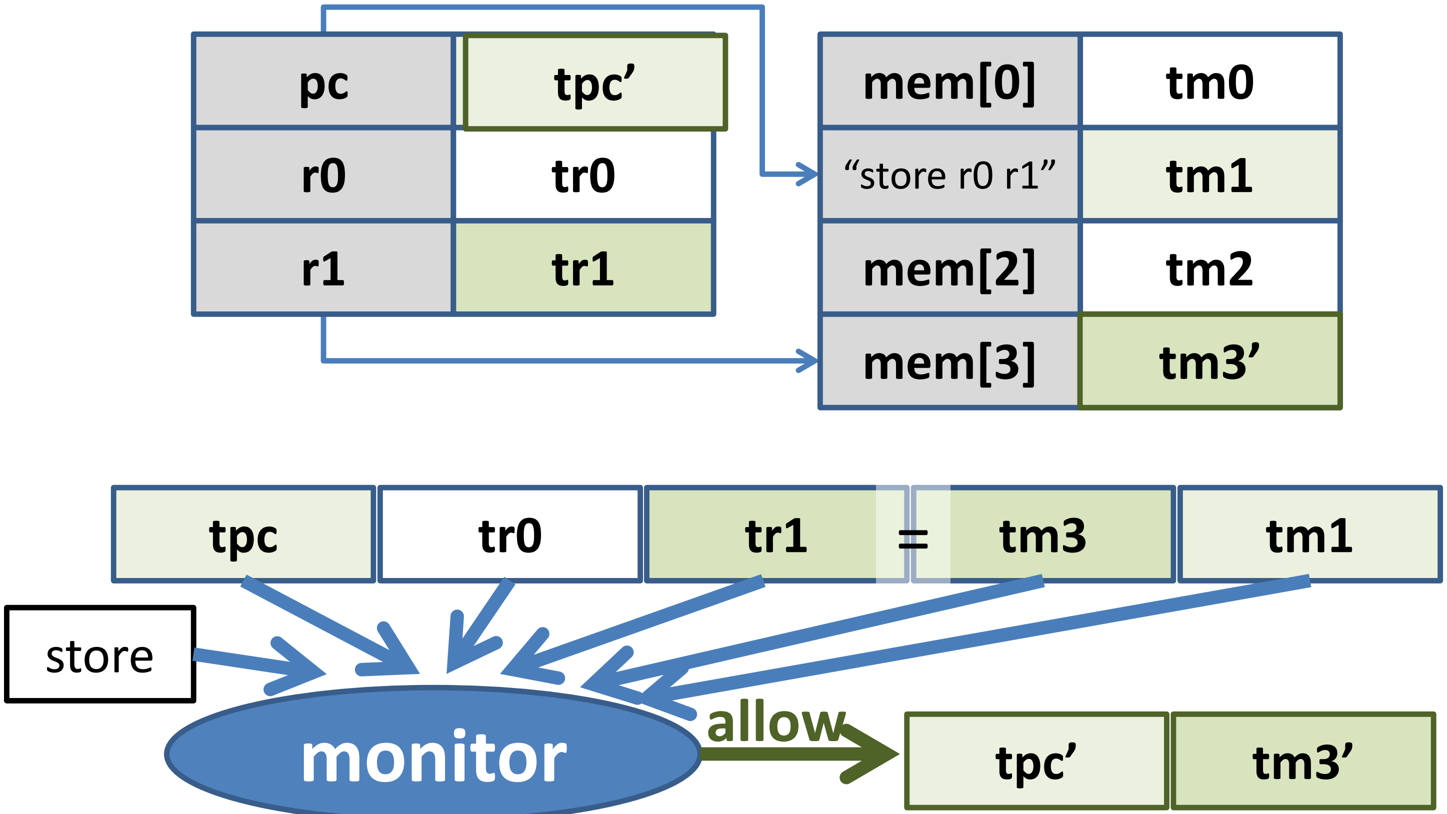
Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring



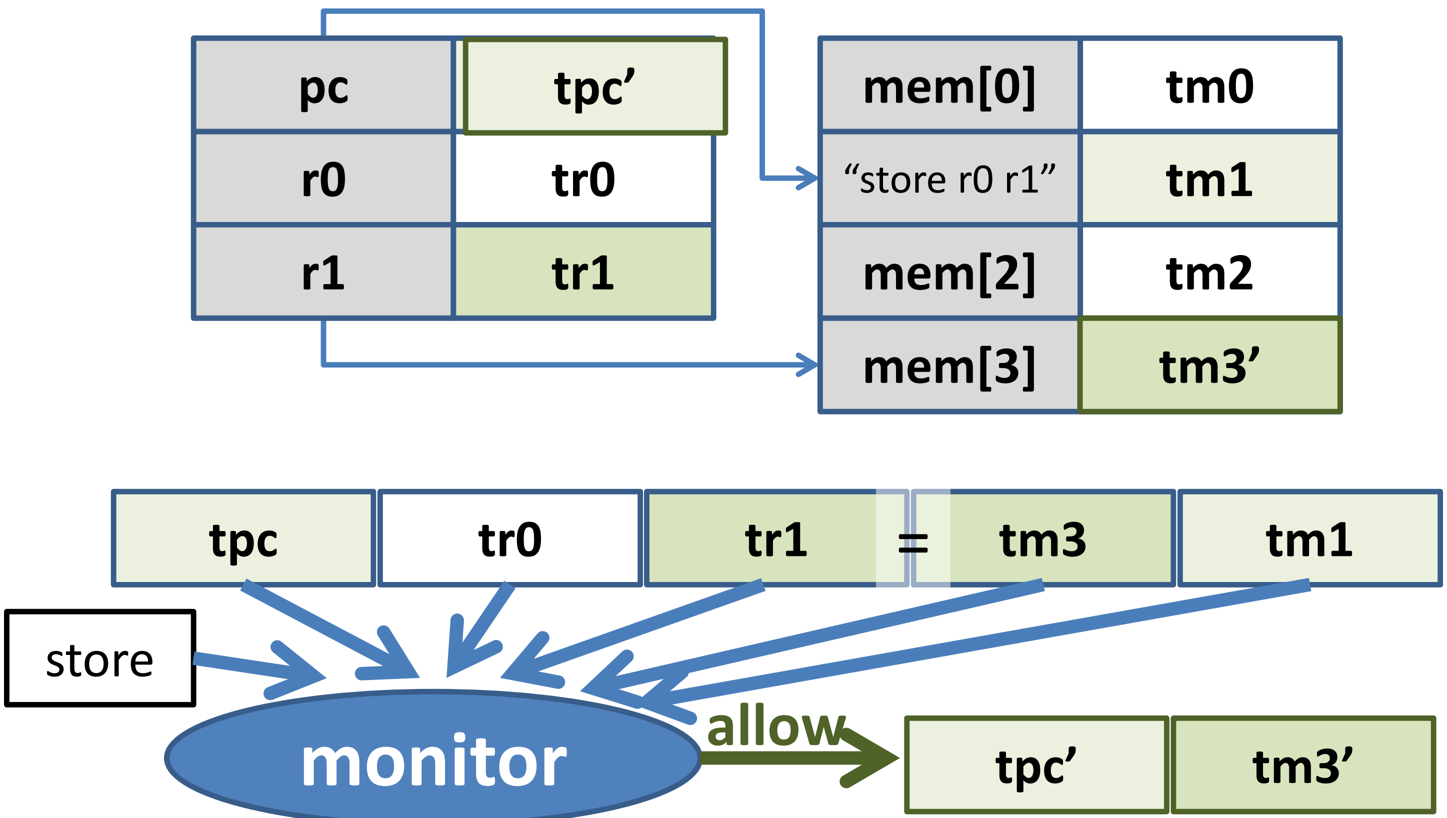
Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring



Micro-Policies

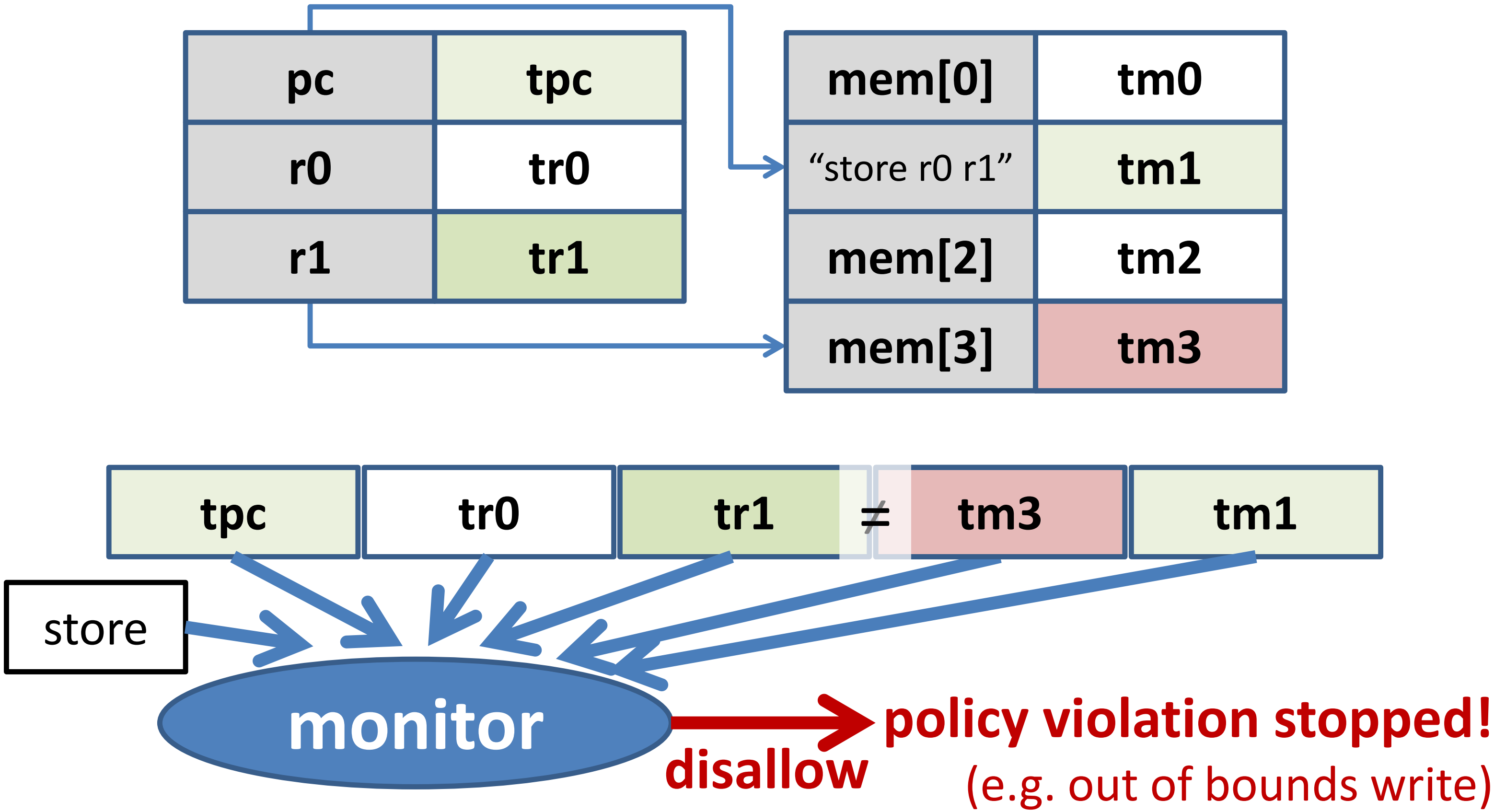
software-defined, hardware-accelerated, tag-based monitoring



software monitor’s decision is hardware cached

Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring



Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction

Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V

Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V

DRAPER DOVER MICROSYSTEMS

Expressiveness

- information flow control (IFC) [POPL'14]

Expressiveness

- information flow control (IFC) [POPL'14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

Expressiveness

- information flow control (IFC) [POPL'14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation

Verified (in Coq) [Oakland'15]

- control-flow integrity (CFI)
- taint tracking
- …

Expressiveness

- information flow control (IFC) [POPL'14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking

Verified (in Coq) [Oakland'15]

Evaluated (<10% runtime overhead) [ASPLOS'15]

spec