The Quest for Formally Secure Compartmentalizing Compilation

Cătălin Hrițcu

Habilitation Defense

My research in the last 7 years

Secure Compilation

Program Verification

Tag-based Monitoring

Property-Based Testing

My research in the last 7 years

Secure Compilation

Program Verification

Tag-based Monitoring

Property-Based Testing

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

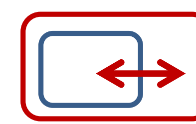
Devastating low-level attacks

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

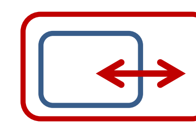
Devastating low-level attacks

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

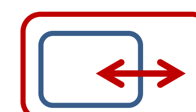
Devastating low-level attacks

Part 2: give meaning to compartmentalization mitigation

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

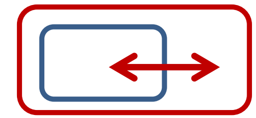


- even code in **more secure languages (Java, OCaml, Rust)** has to interoperate with **low-level code (C, C++, ASM)**
- **insecure interoperability**: all source-level guarantees lost

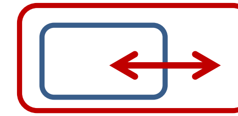
Part 1: formalize what it means to solve this problem

Part 1 of 2

Secure Interoperability with Lower-Level Code



Secure Interoperability with Lower-Level Code

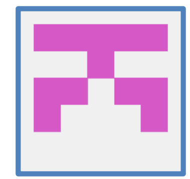


Carmine Abate
Inria Paris

Rob Blanco
Inria Paris

Deepak Garg
MPI-SWS

Cătălin Hrițcu
Inria Paris

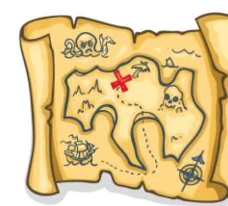


Jérémy Thibault
Inria Paris

Marco Patrignani
Stanford & CISPA

Journey Beyond Full Abstraction

https://arxiv.org/abs/1807.04603



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 of safe C programs**
    - structured control flow, procedures, abstract memory model

Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions of safe C programs**
    - structured control flow, procedures, abstract memory model
  - **higher-level abstractions of ML-like languages**
    - modules, interfaces, and parametric polymorphism

Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions of safe C programs**
    - structured control flow, procedures, abstract memory model
  - **higher-level abstractions of ML-like languages**
    - modules, interfaces, and parametric polymorphism
  - **specifications of a verification system like Coq and Dafny**
    - effects, dependent types, refinements, logical pre- and post-conditions

Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
  - **low-level abstractions of safe C programs**
    - structured control flow, procedures, abstract memory model
  - **higher-level abstractions of ML-like languages**
    - modules, interfaces, and parametric polymorphism
  - **specifications of a verification system like Coq and Dafny**
    - effects, dependent types, refinements, logical pre- and post-conditions
  - **coding patterns specific to cryptographic code**
    - abstract types and interfaces for defending against side-channel attacks

Recherche

20/09/2017

EPI Prosecco: high assurance cryptography for Mozilla Firefox

Mozilla Firefox

Mozilla Security Blog

SEP 13 2017

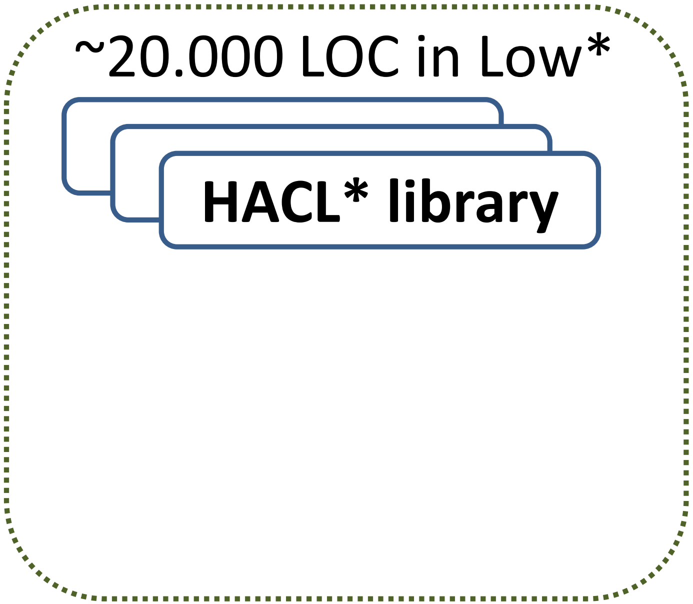
Verified cryptography for Firefox 57

Benjamin Beurdouche

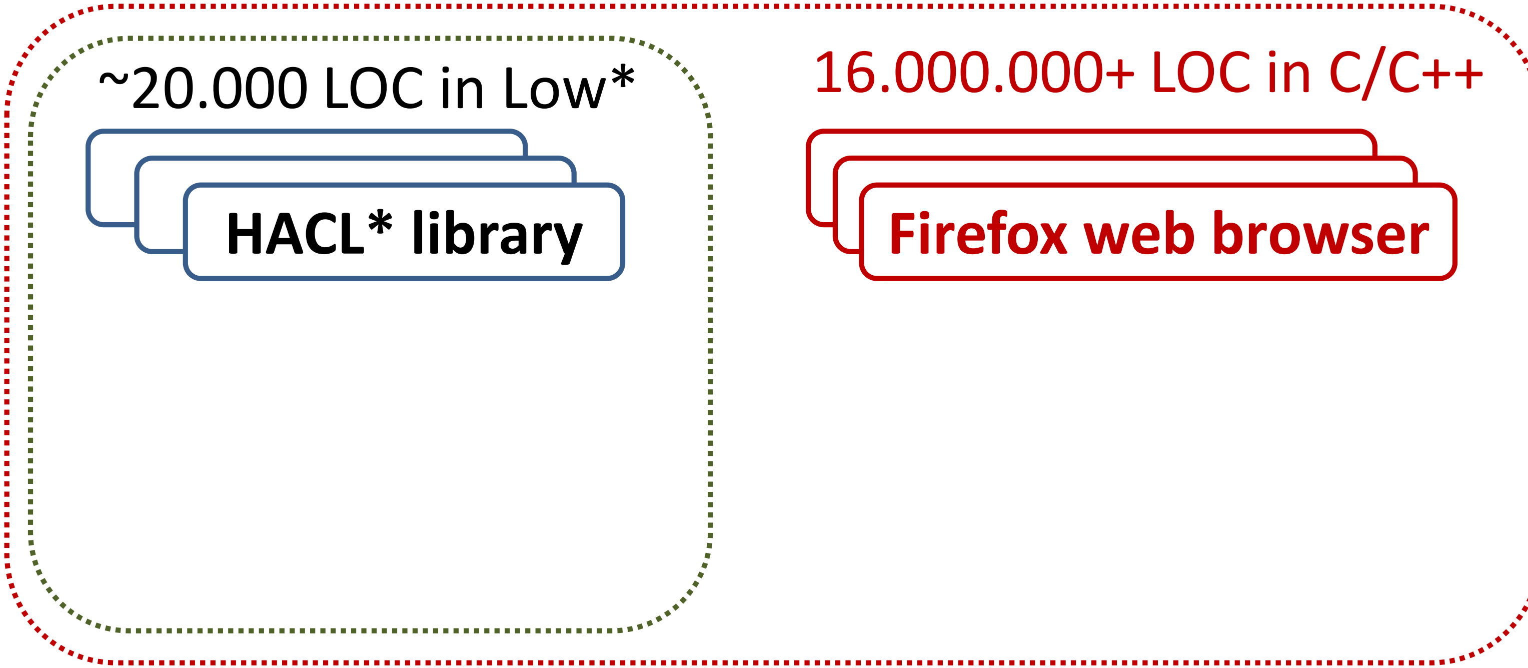
Traditionally, software is produced in this way: write some code, maybe do some code rev
run unit-tests, and then hope it is correct. Hard experience shows that it is very hard for
programmers to write bug-free software. These bugs are sometimes caught in manual tes
but many bugs still are exposed to users, and then must be fixed in patches or subsequen
versions. This works for most software, but it's not a great way to write cryptographic soft
users expect and deserve assurances that the code providing security and privacy is well

Originally, [the HACL* project] is a joint effort between
(CMU, INRIA, Microsoft Research) to produce a [Hig
written in the F* formal verification language] and ger

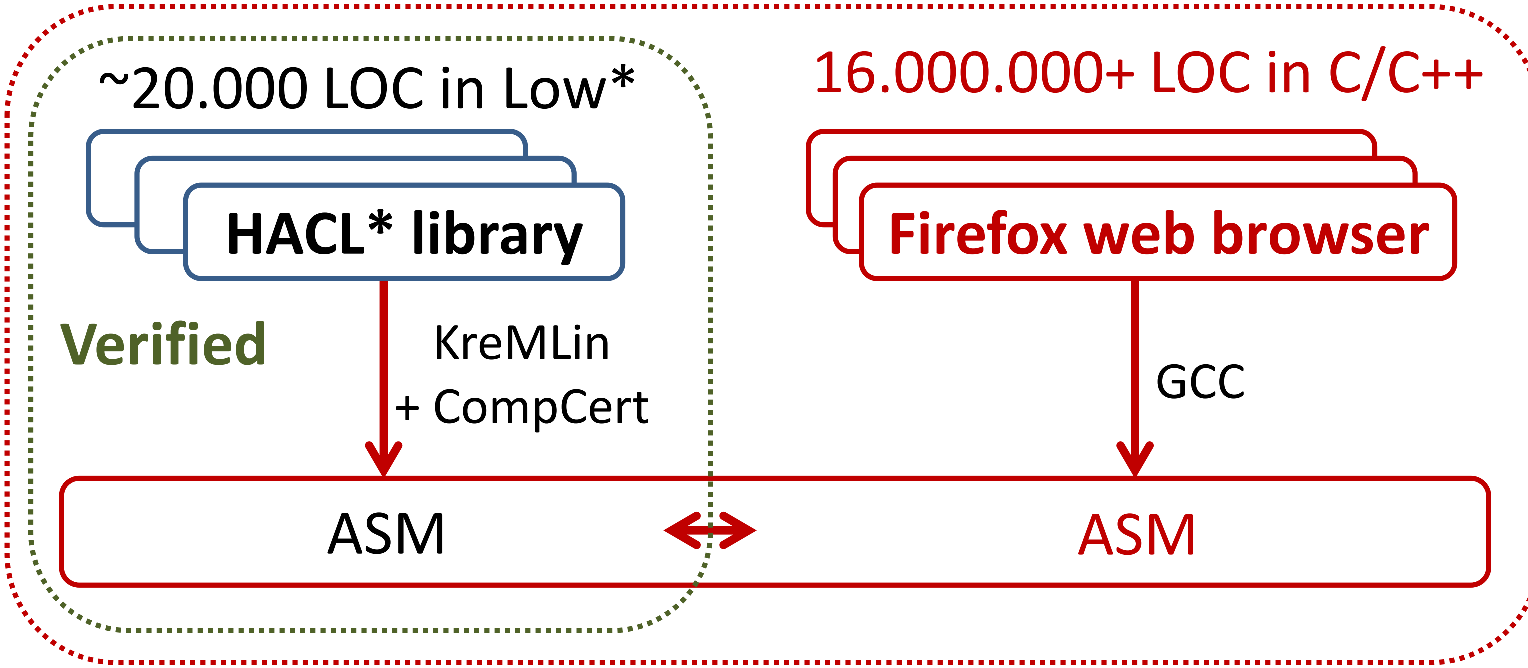
But abstractions not enforced when compiling and linking with adversarial low-level code



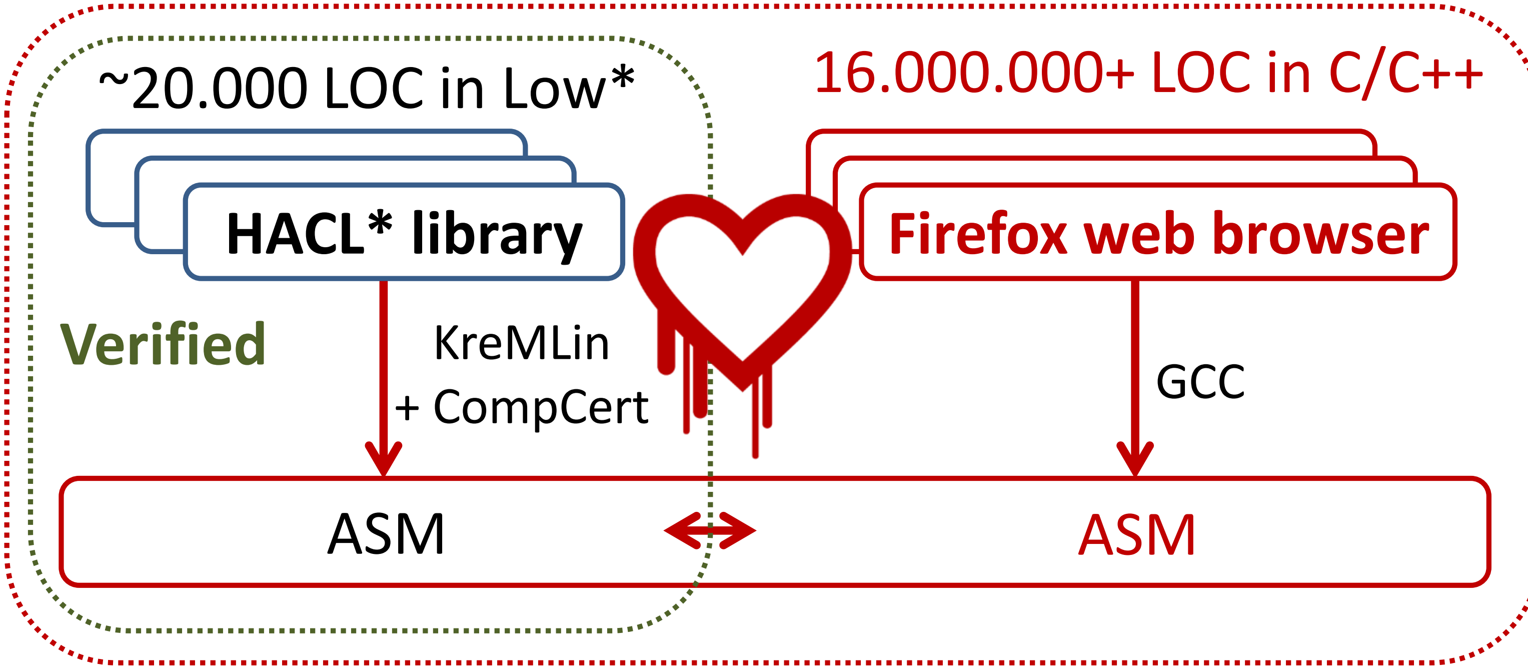
But abstractions not enforced when compiling and linking with adversarial low-level code



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



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



Insecure interoperability: linked code can read and write data and code, jump to arbitrary instructions, smash the stack, ...

Secure compilation chains

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

Secure compilation chains

- **Protect source-level abstractions**
  even against linked adversarial low-level code
  - **various enforcement mechanisms**: processes, SFI, ...
  - shared responsibility: compiler, linker, loader, OS, HW
- **Goal: enable source-level security reasoning**

Secure compilation chains

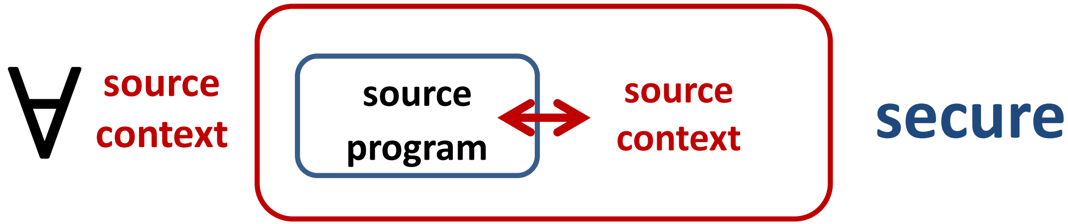
- **Protect source-level abstractions even against linked adversarial low-level code**
  - **various enforcement mechanisms**: processes, SFI, ...
  - shared responsibility: compiler, linker, loader, OS, HW
- **Goal: enable source-level security reasoning**
  - **adversarial target-level context cannot break the security of compiled program** any more than some source-level context

Secure compilation chains

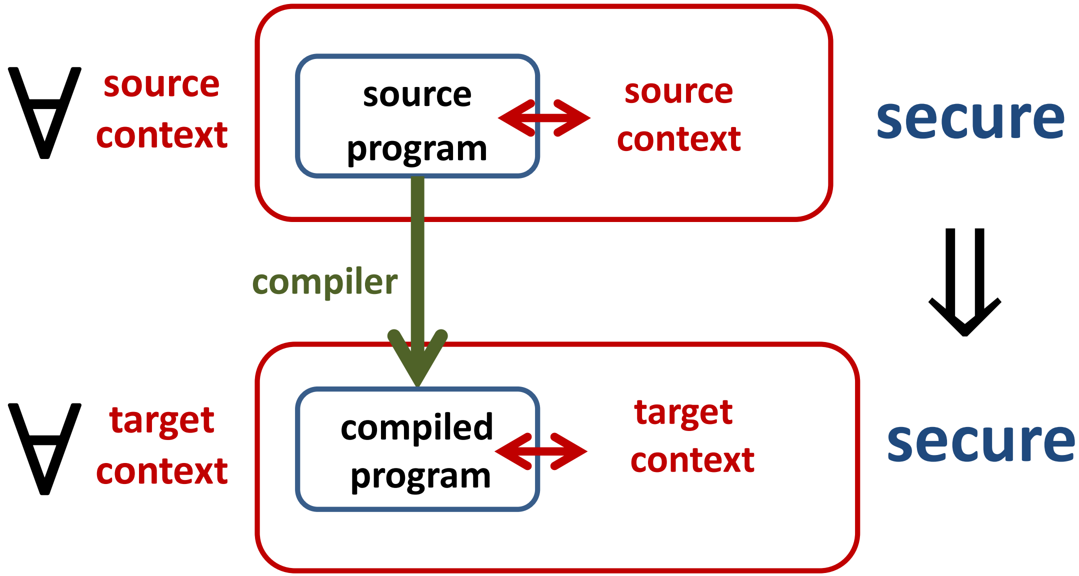
- **Protect source-level abstractions**
  even against linked adversarial low-level code
  – **various enforcement mechanisms**: processes, SFI, ...
  – shared responsibility: compiler, linker, loader, OS, HW
- **Goal: enable source-level security reasoning**
  – **adversarial target-level context cannot break the security of compiled program** any more than some source-level context
  – **no "low-level" attacks**

Robustly preserving security

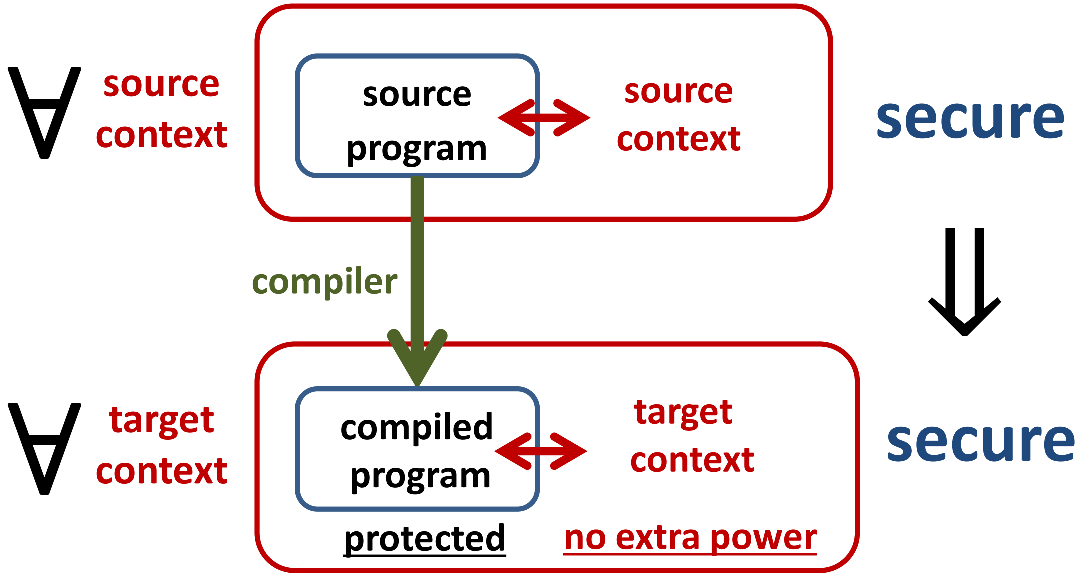
Robustly preserving security



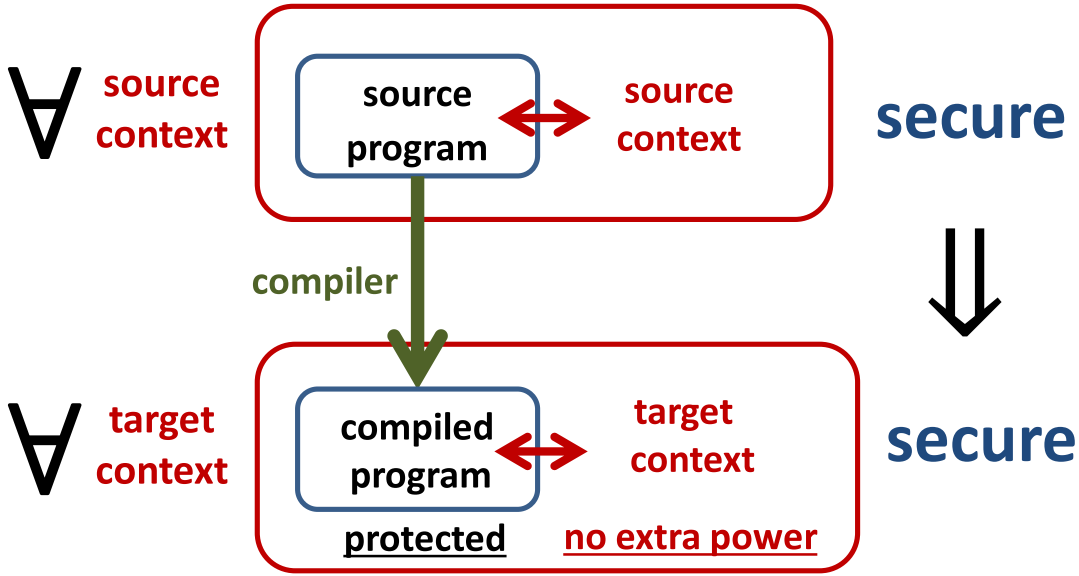
Robustly preserving security



Robustly preserving security



Robustly preserving security



But what should "secure" mean?

What properties should we robustly preserve?

What properties should we robustly preserve?

trace properties
(safety & liveness)

What properties should we robustly preserve?

hyperproperties
(noninterference)

trace properties
(safety & liveness)

What properties should we robustly preserve?

relational hyperproperties
(trace equivalence)

new

hyperproperties
(noninterference)

trace properties
(safety & liveness)

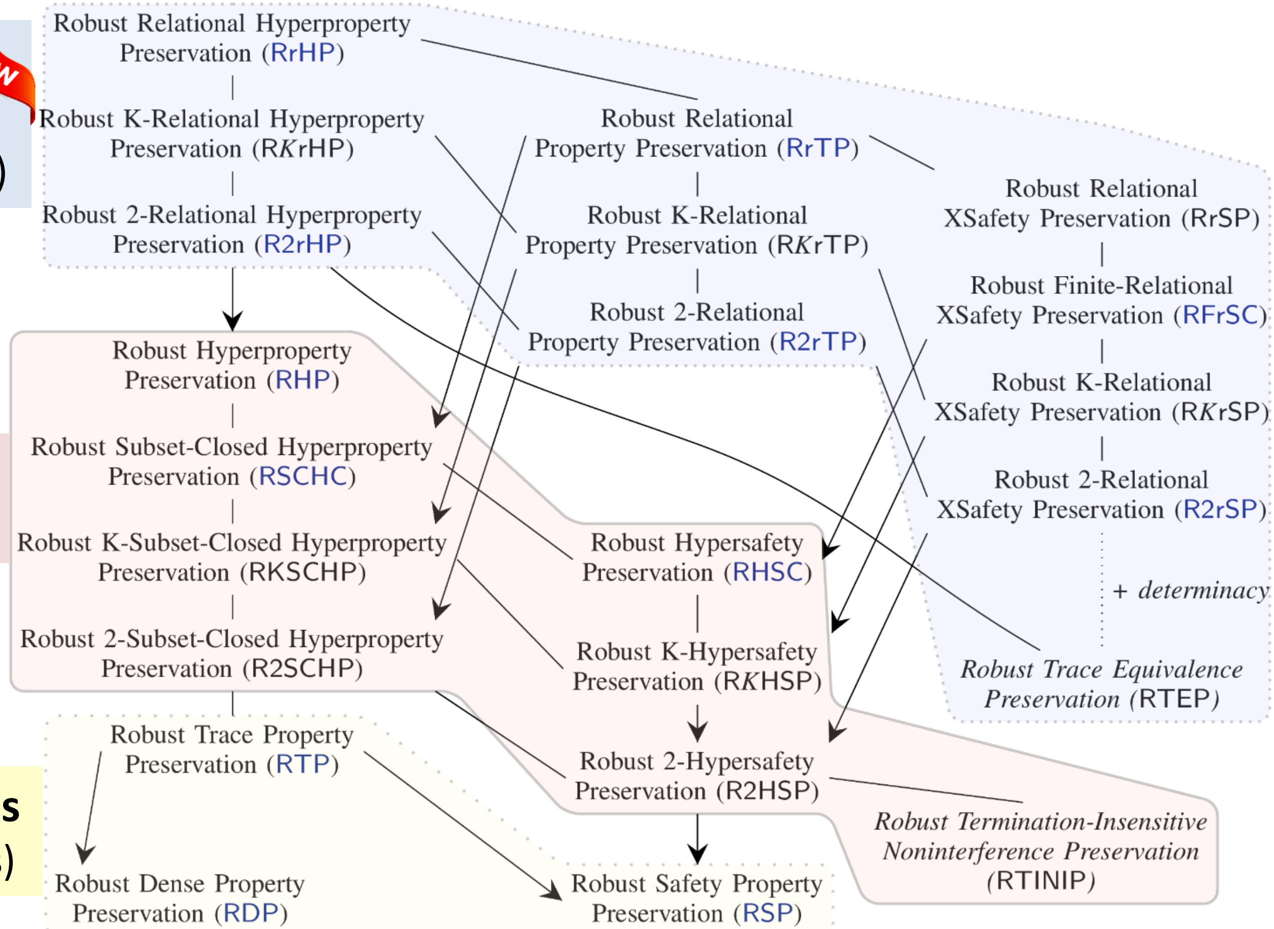
What properties should we robustly preserve?

relational hyperproperties
(trace equivalence)

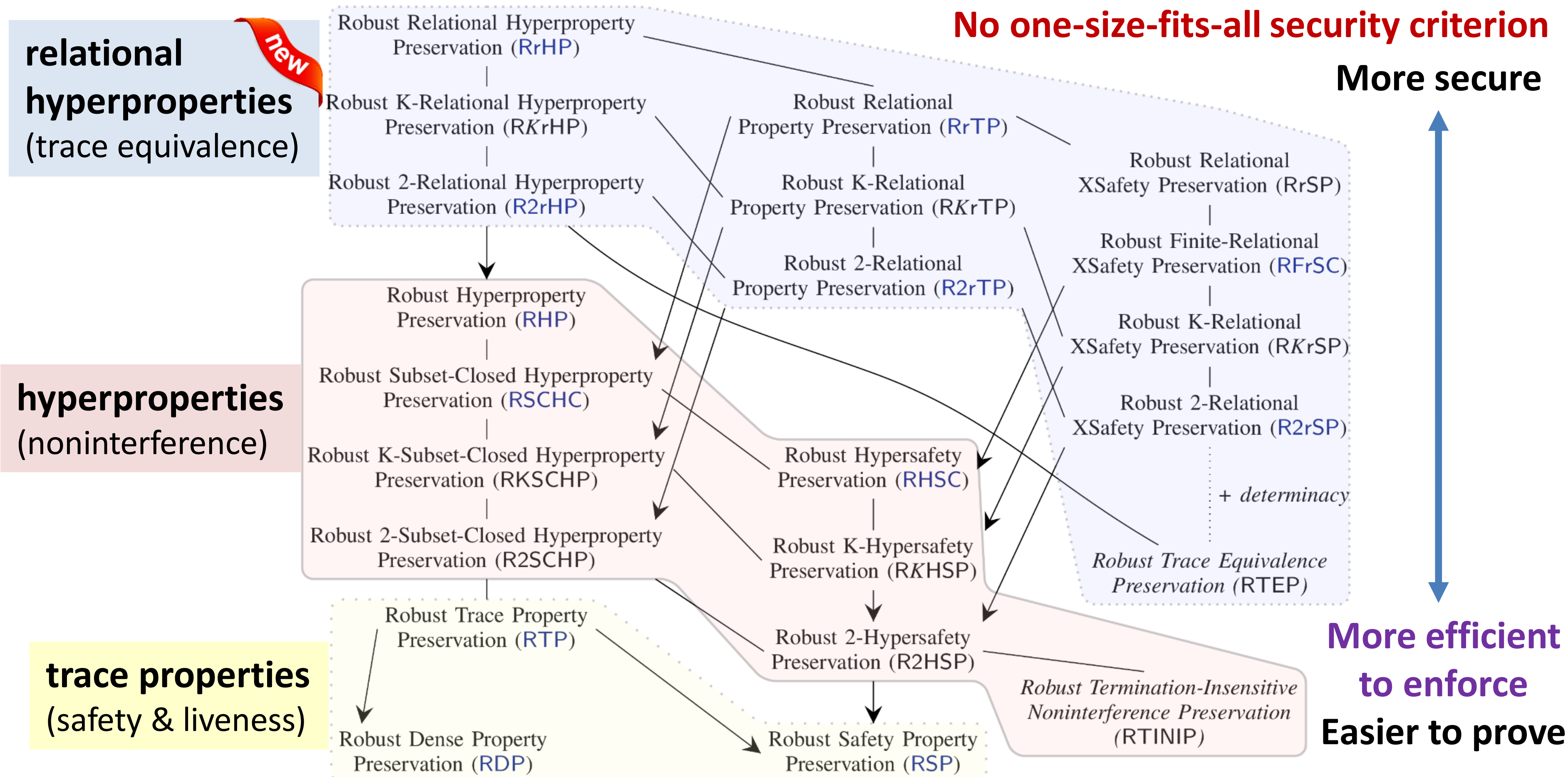
new

hyperproperties
(noninterference)

trace properties
(safety & liveness)



What properties should we robustly preserve?



What properties should we robustly preserve?

No one-size-fits-all security criterion

More secure

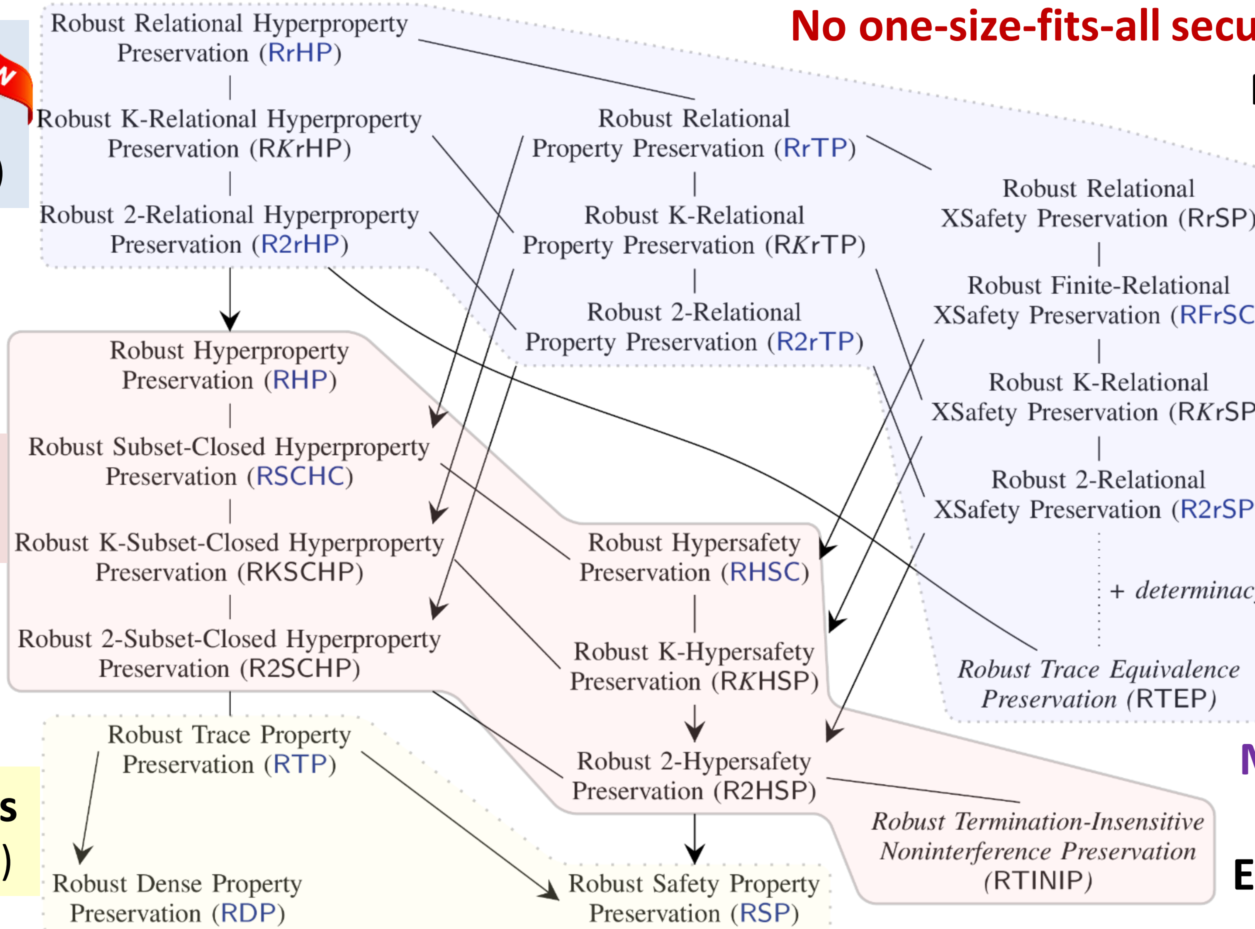
relational hyperproperties (trace equivalence)

new

hyperproperties (noninterference)

trace properties (safety & liveness)

only integrity



More efficient to enforce

Easier to prove

What properties should we robustly preserve?

No one-size-fits-all security criterion

relational hyperproperties (trace equivalence)

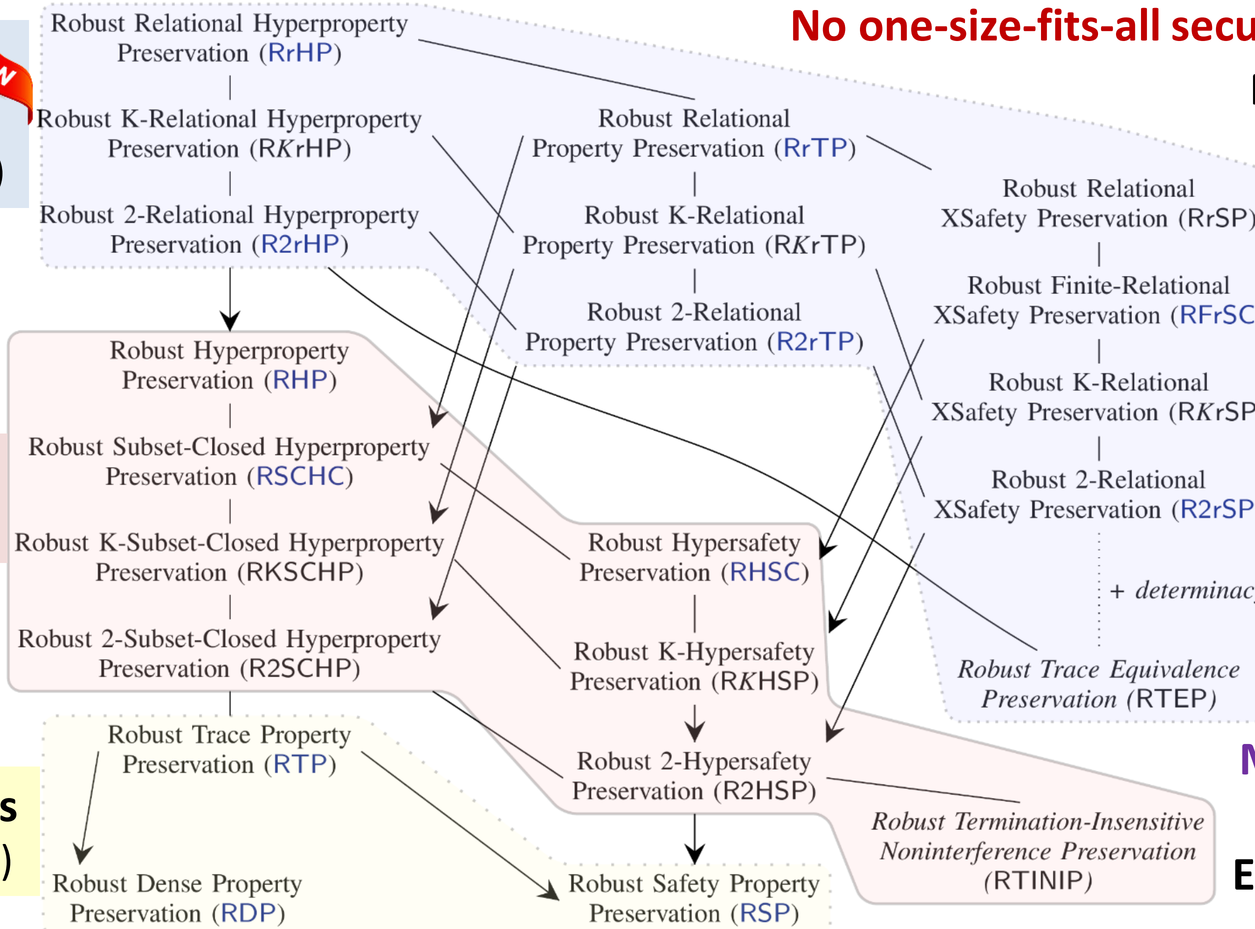
new

hyperproperties (noninterference)

+ data confidentiality

trace properties (safety & liveness)

only integrity



More secure

More efficient to enforce

Easier to prove

What properties should we robustly preserve?

relational hyperproperties
(trace equivalence)

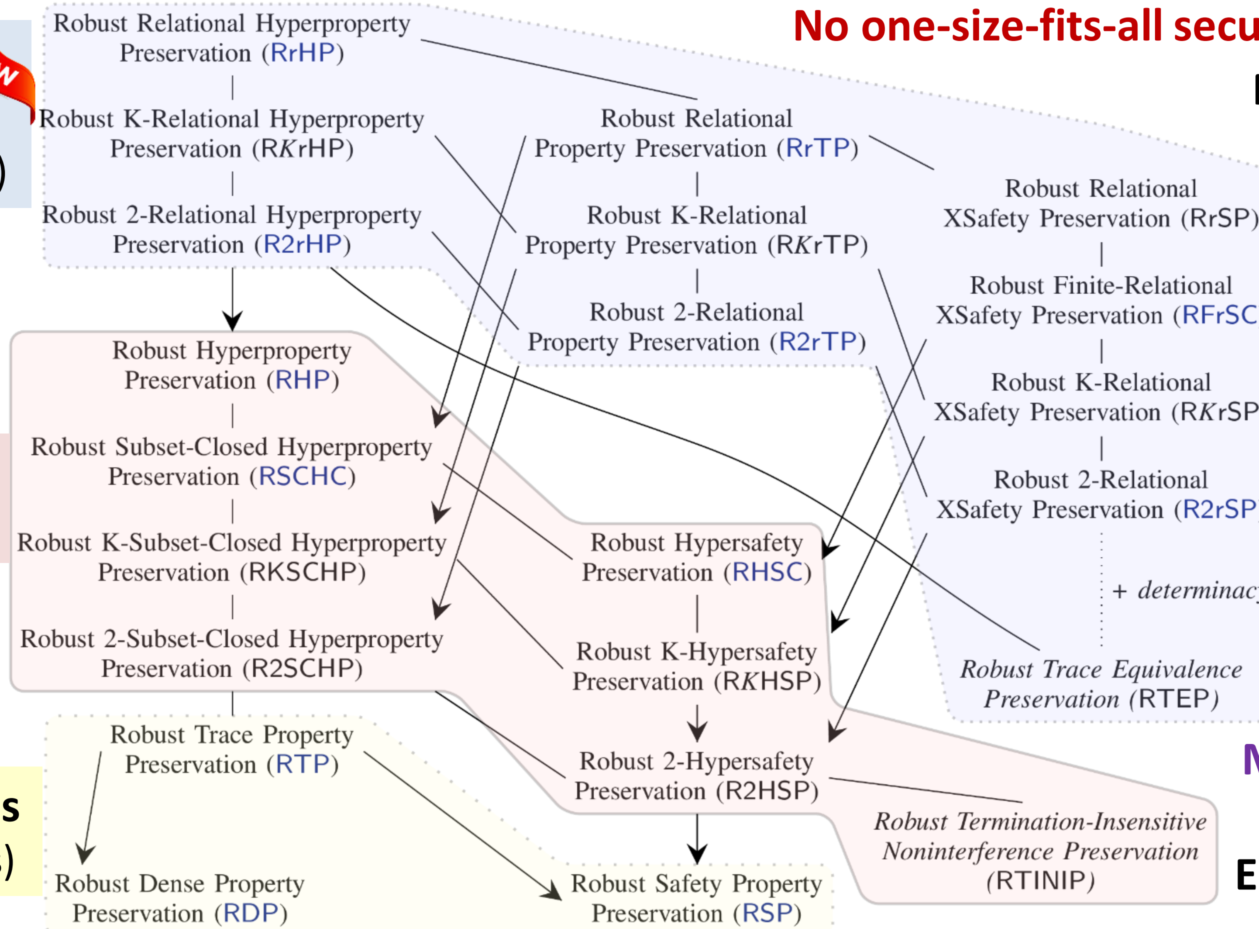
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



No one-size-fits-all security criterion

More secure

More efficient to enforce

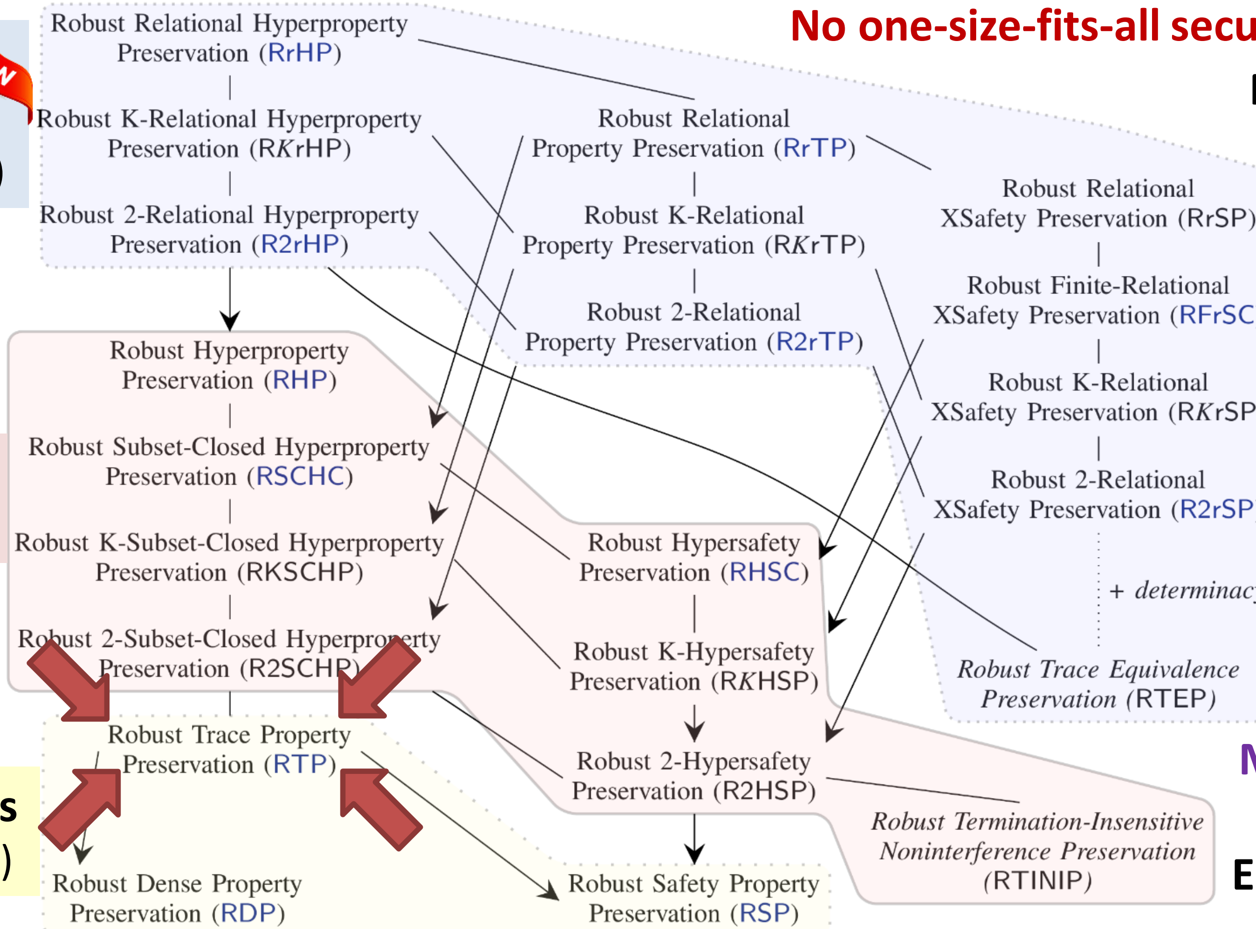
Easier to prove

What properties should we robustly preserve?

No one-size-fits-all security criterion

relational hyperproperties
(trace equivalence)
new

+ code confidentiality



hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

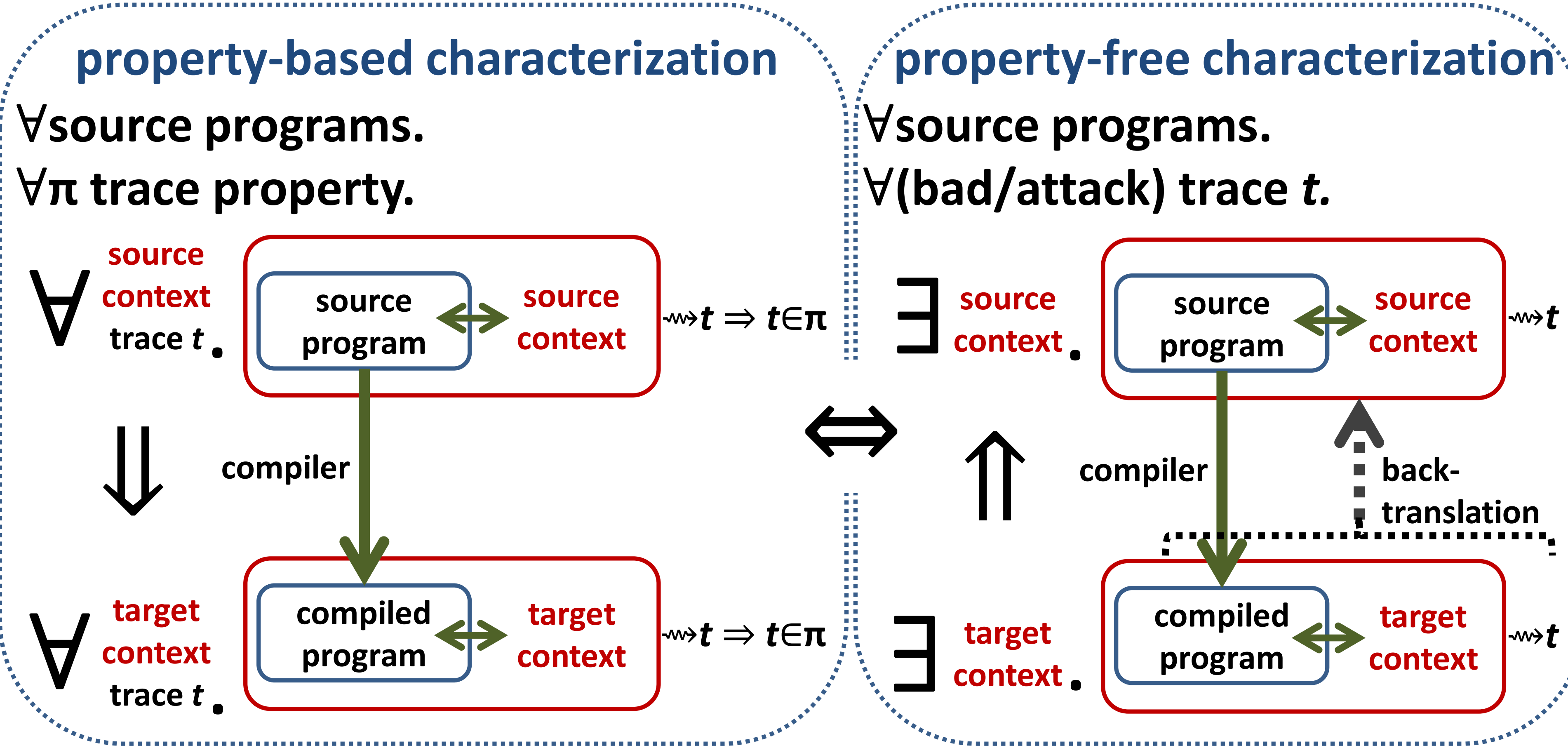
only integrity

More secure

More efficient to enforce

Easier to prove

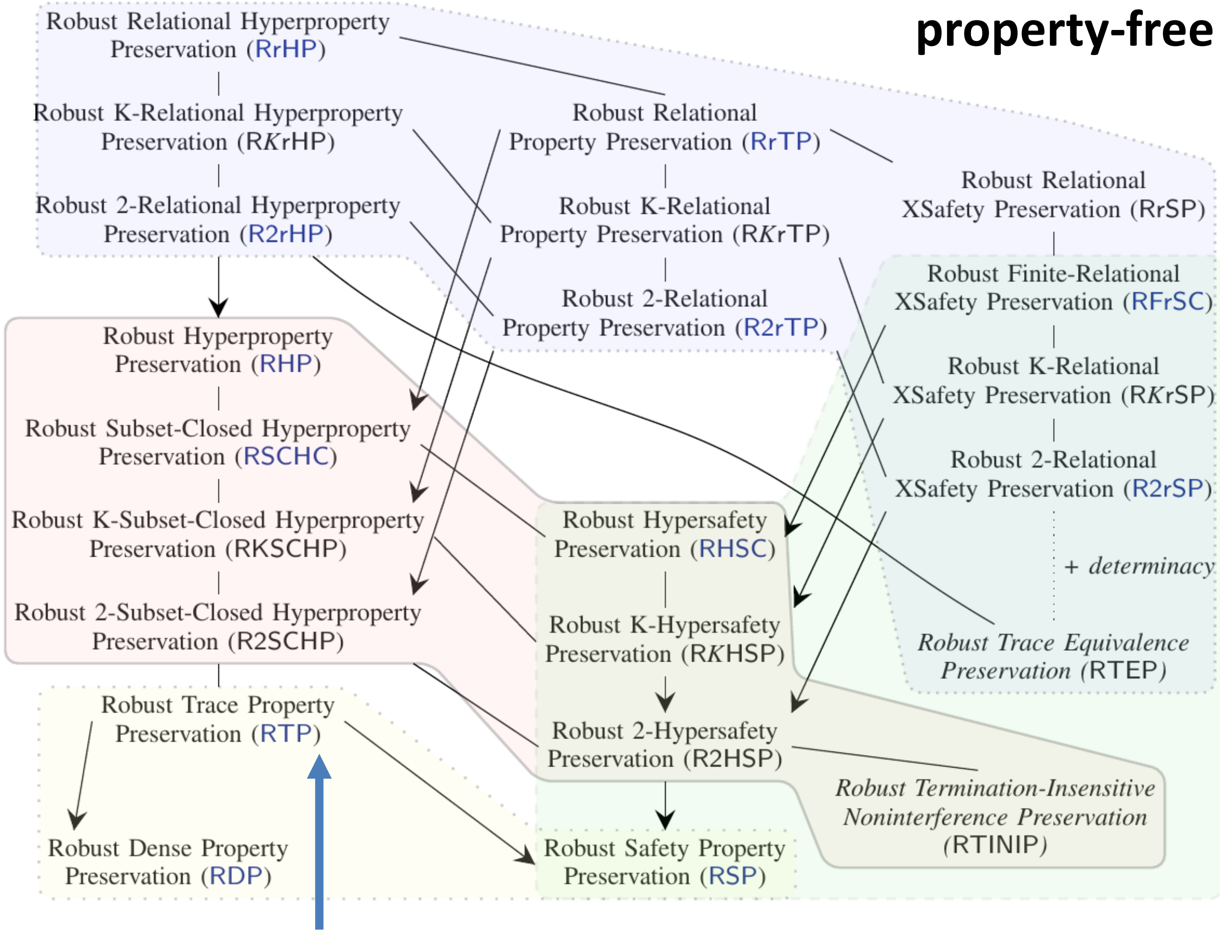
Robust Trace Property Preservation



what one can achieve

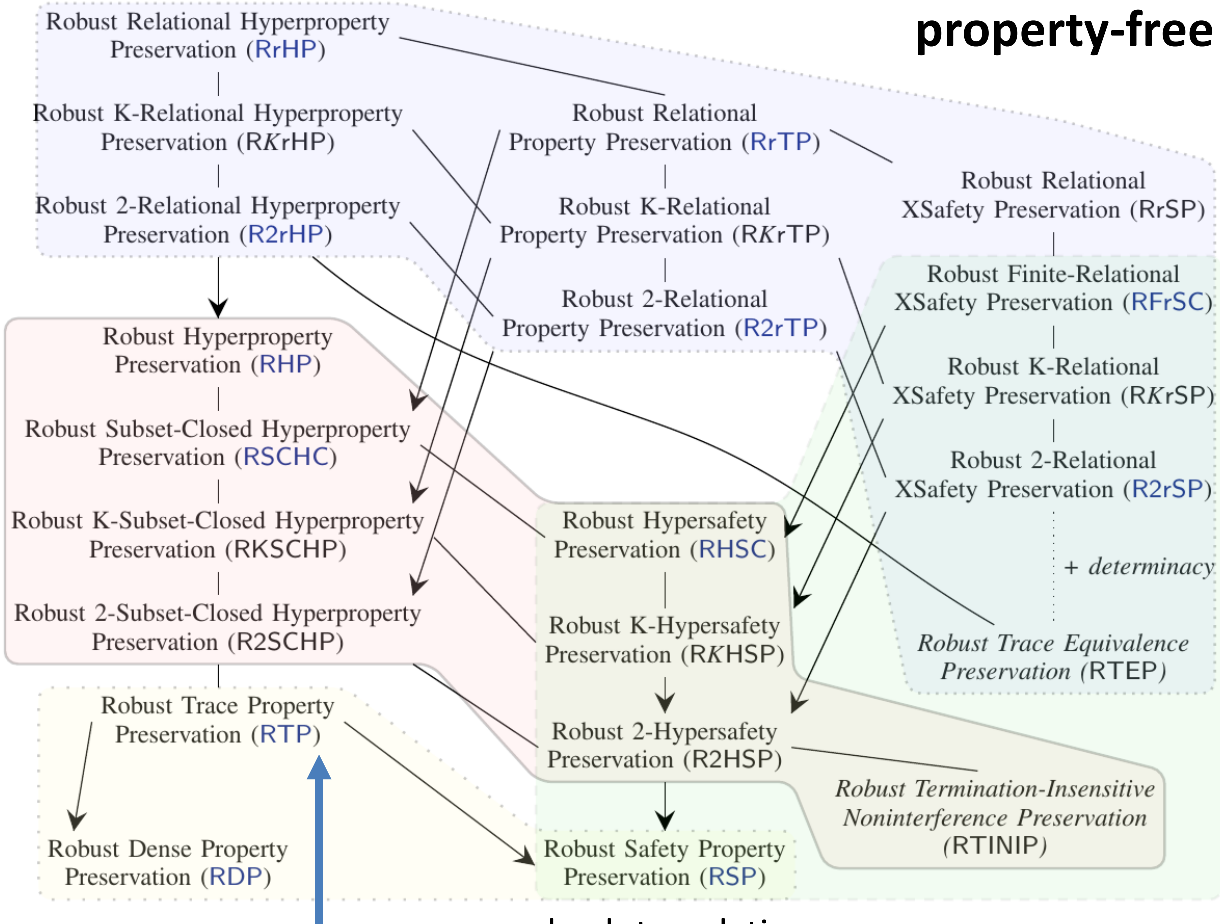
how one can prove it

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



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

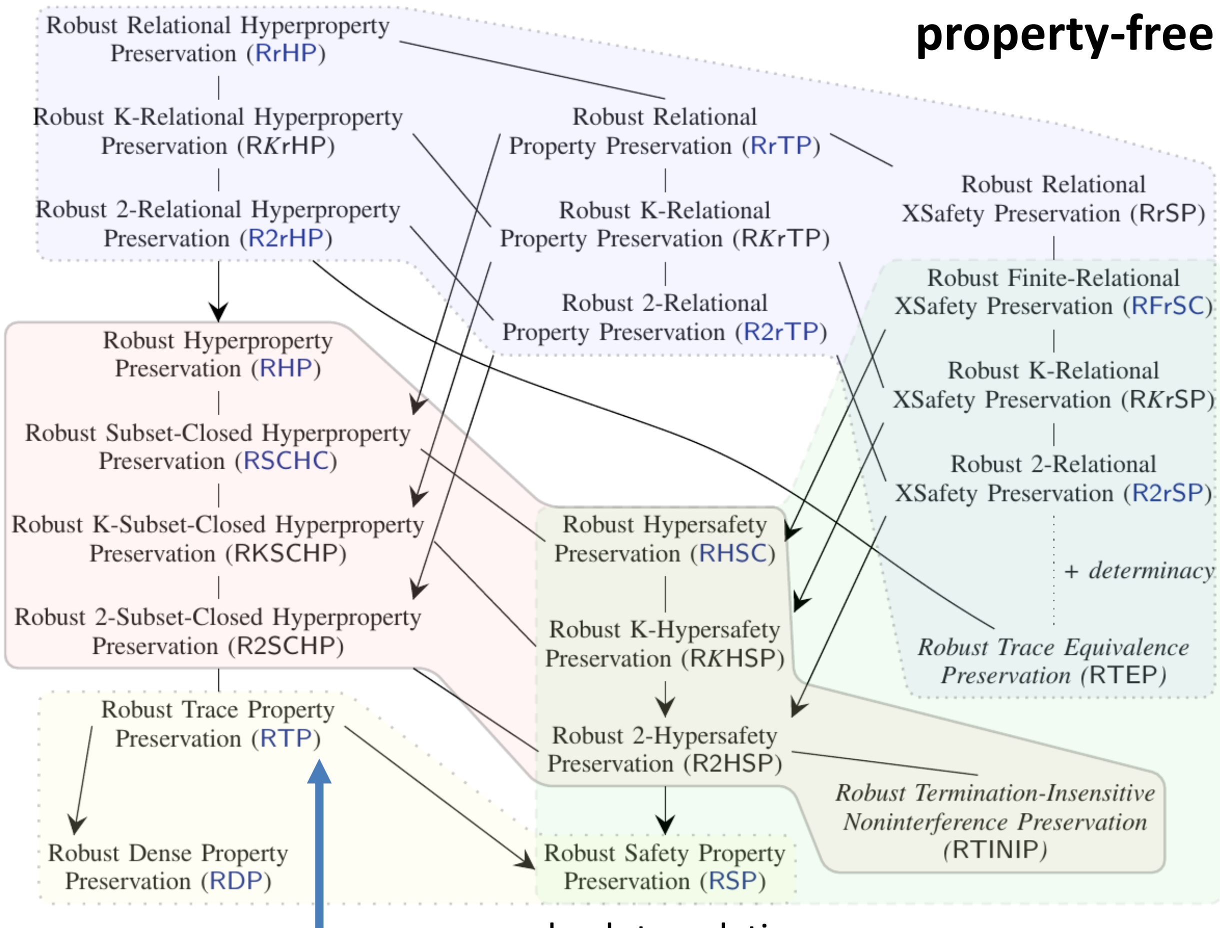
Some of the proof difficulty is manifest in property-free characterization



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

back-translating finite trace prefix $\forall P \forall C_T \forall m \leq t \exists C_S \ldots$

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



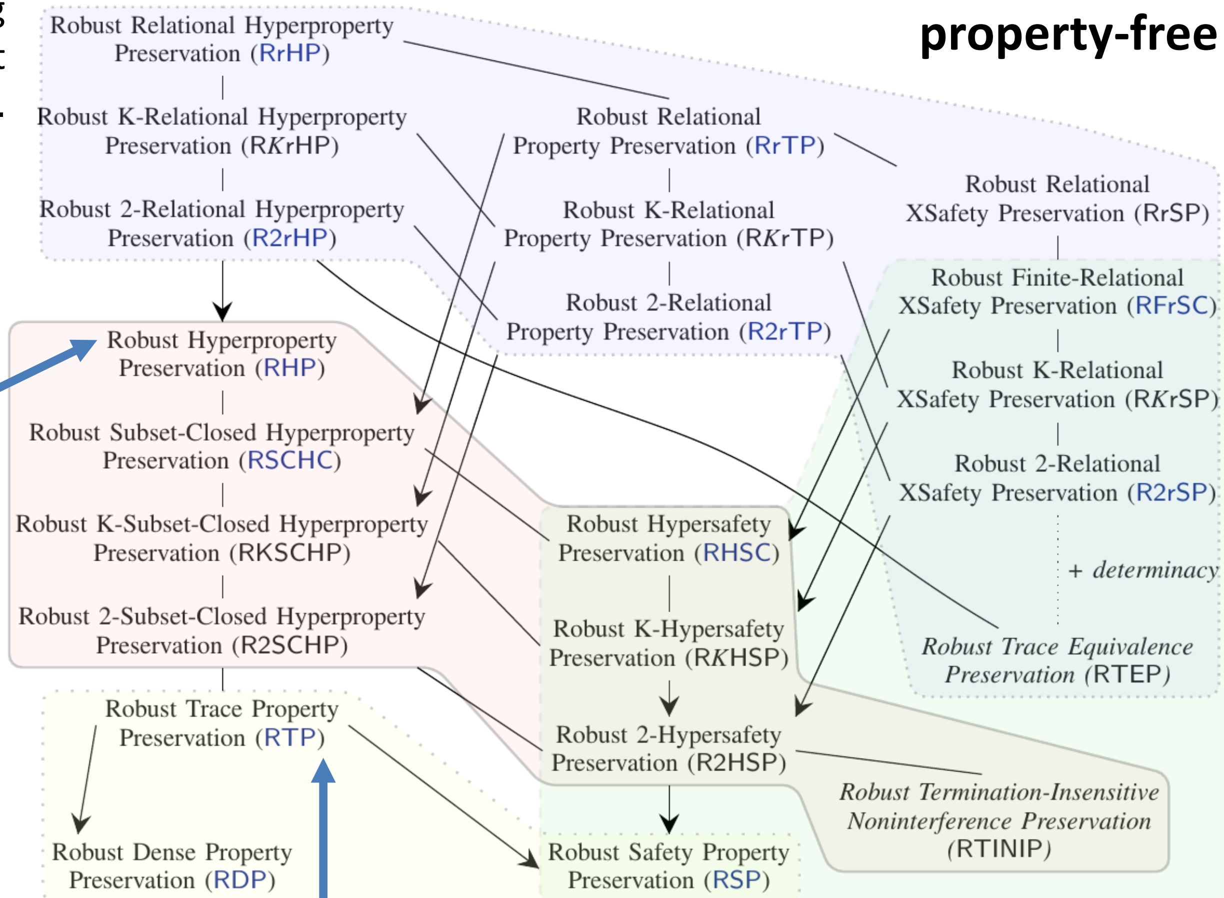
back-translating finite set of finite trace prefixes $\forall k \forall P_1..P_k \forall C_T$ $\forall m_1..m_k \exists C_S$...

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

back-translating finite trace prefix $\forall P \forall C_T \forall m \leq t \exists C_S$...

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

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



back-translating finite set of finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T$
$\forall m_1..m_k \exists C_S$...

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

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

back-translating finite trace prefix
$\forall P \forall C_T \forall m \leq t \exists C_S$...

Journey Beyond Full Abstraction

- **First to explore space of secure compilation criteria** based on robust property preservation

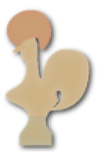
Journey Beyond Full Abstraction

- **First to explore space of secure compilation criteria** based on robust property preservation
- **Carefully study the criteria and their relations**

  - Property-free characterizations
  - implications, **collapses**, **separations results**

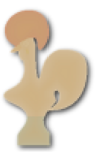
Journey Beyond Full Abstraction

- **First to explore space of secure compilation criteria** based on robust property preservation
- **Carefully study the criteria and their relations**
  - Property-free characterizations
  - implications, **collapses**, **separations results**
- **Introduce relational (hyper)properties (new classes!)**



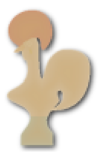
Journey Beyond Full Abstraction

- **First to explore space of secure compilation criteria** based on robust property preservation
- **Carefully study the criteria and their relations**
  - Property-free characterizations
  - implications, **collapses**, **separations results**
- **Introduce relational (hyper)properties (new classes!)**
- **Formally study relation to full abstraction ...**



Journey Beyond Full Abstraction

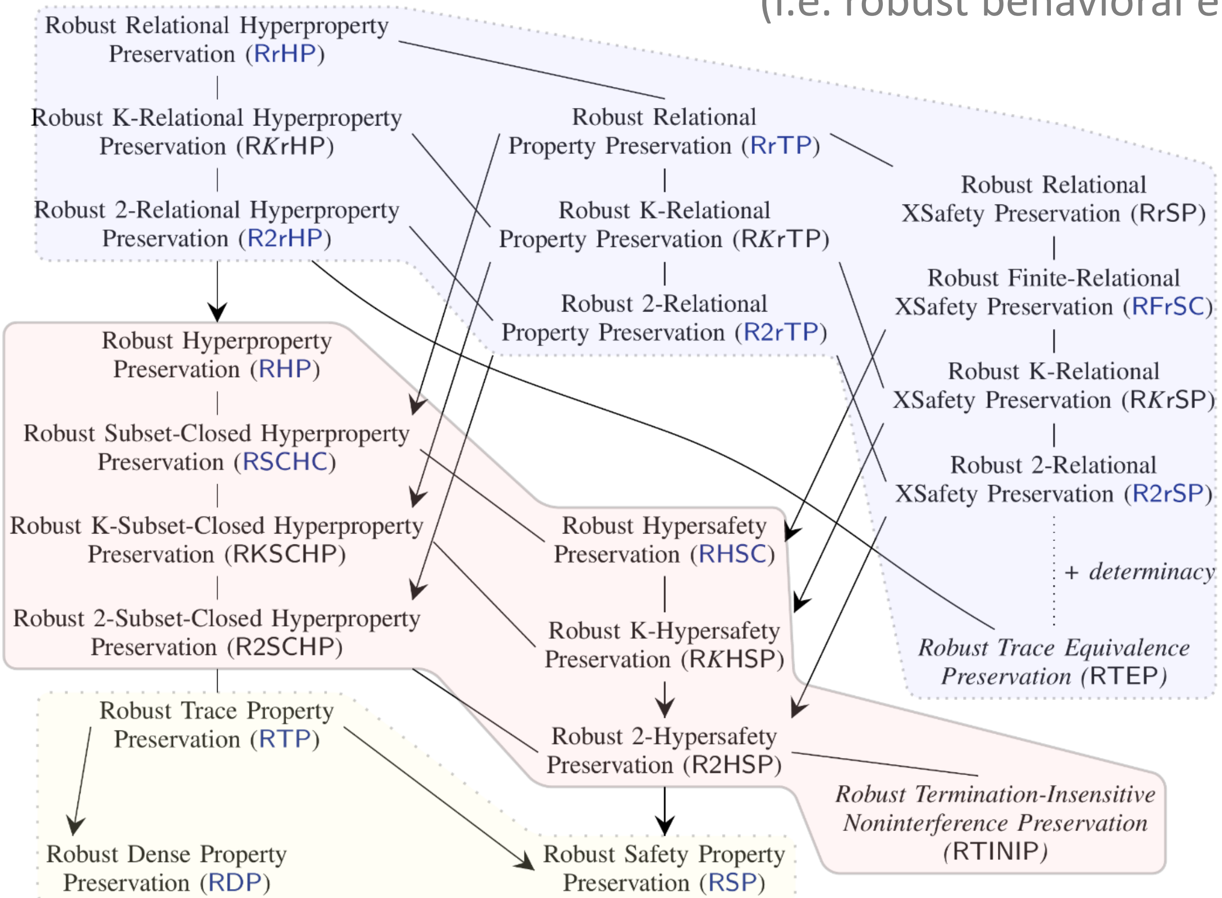
- **First to explore space of secure compilation criteria** based on robust property preservation
- **Carefully study the criteria and their relations**



  - Property-free characterizations
  - implications, **collapses**, **separations results**
- **Introduce relational (hyper)properties (new classes!)**
- **Formally study relation to full abstraction ...**
- **Embraced and extended proof techniques ...**

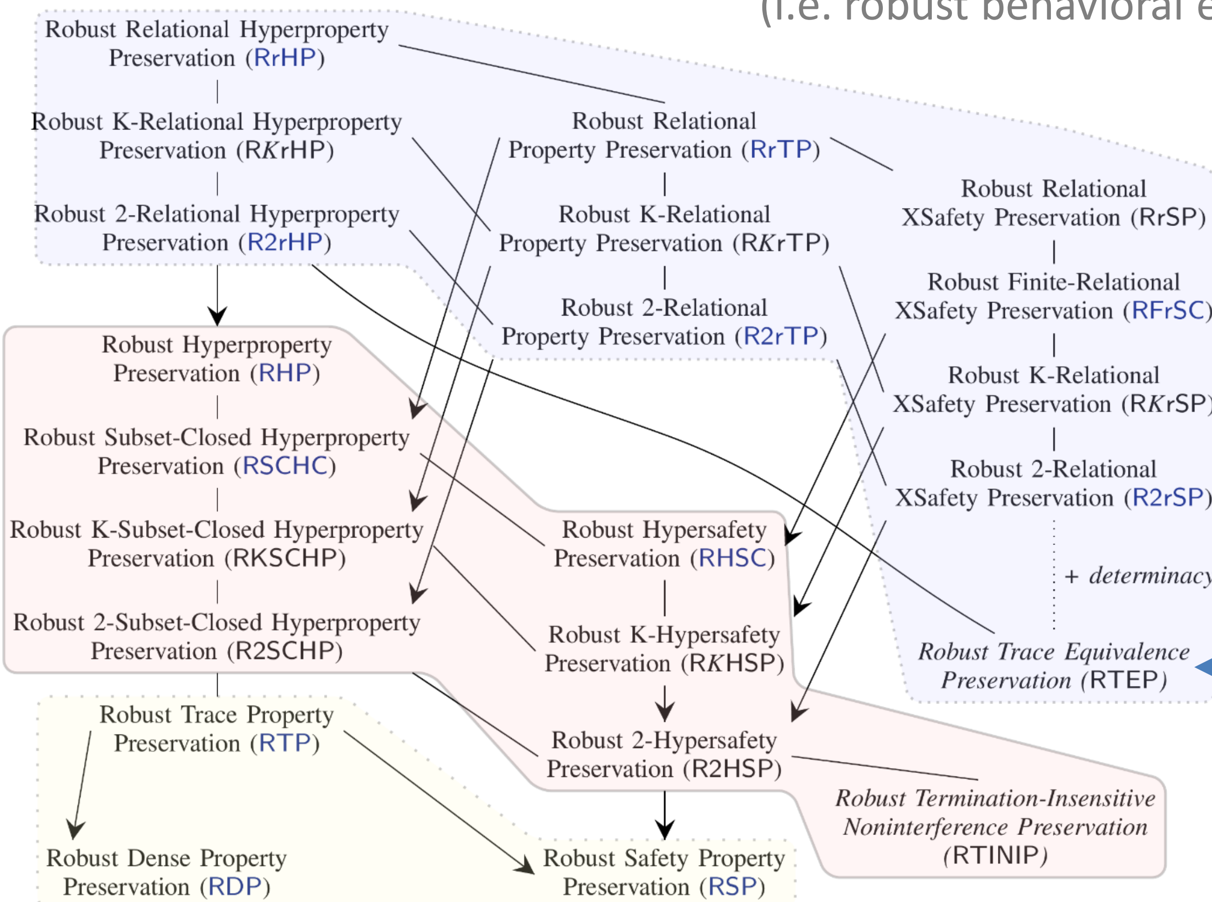
Where is Full Abstraction?

(i.e. robust behavioral equivalence preservation)



Where is Full Abstraction?

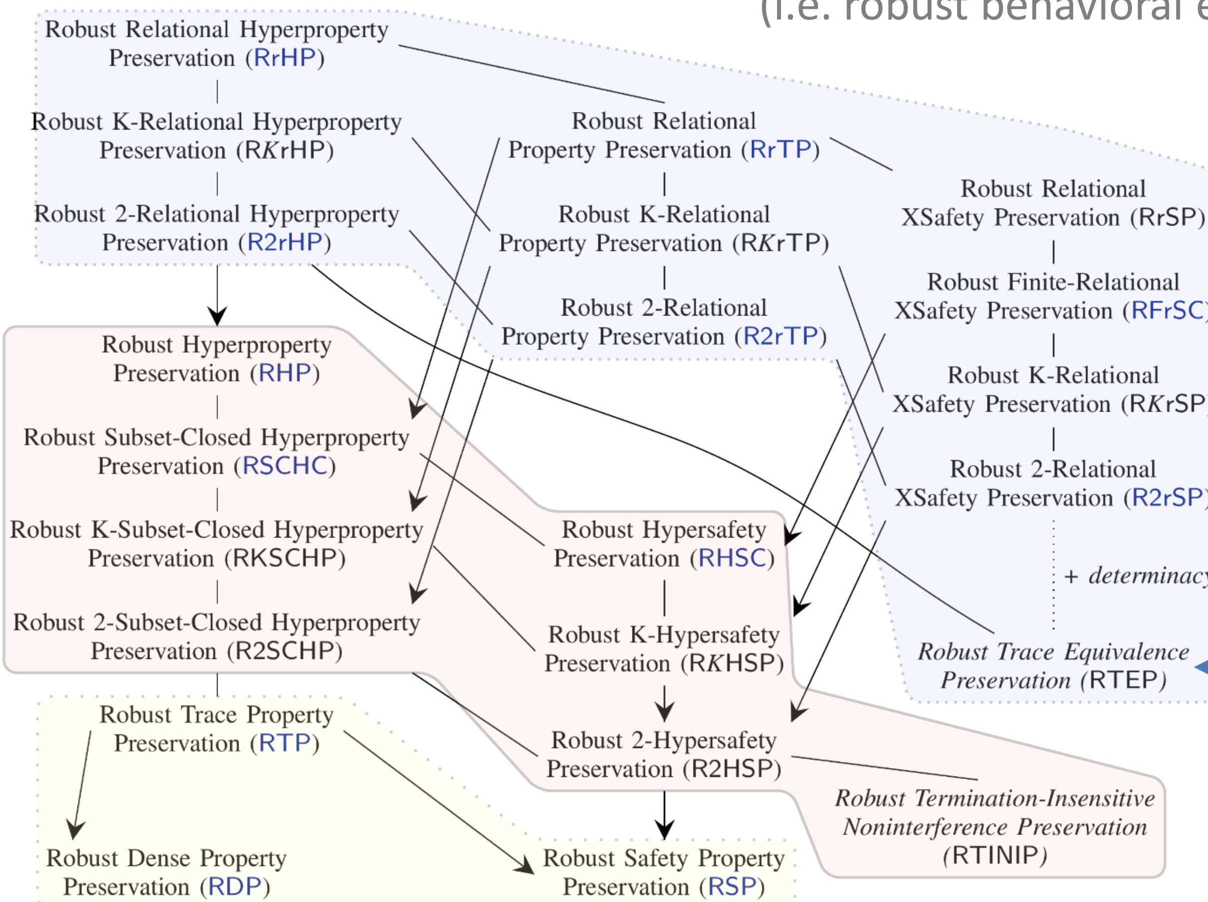
(i.e. robust behavioral equivalence preservation)



without internal nondeterminism, **full abstraction is here**

Where is Full Abstraction?

(i.e. robust behavioral equivalence preservation)

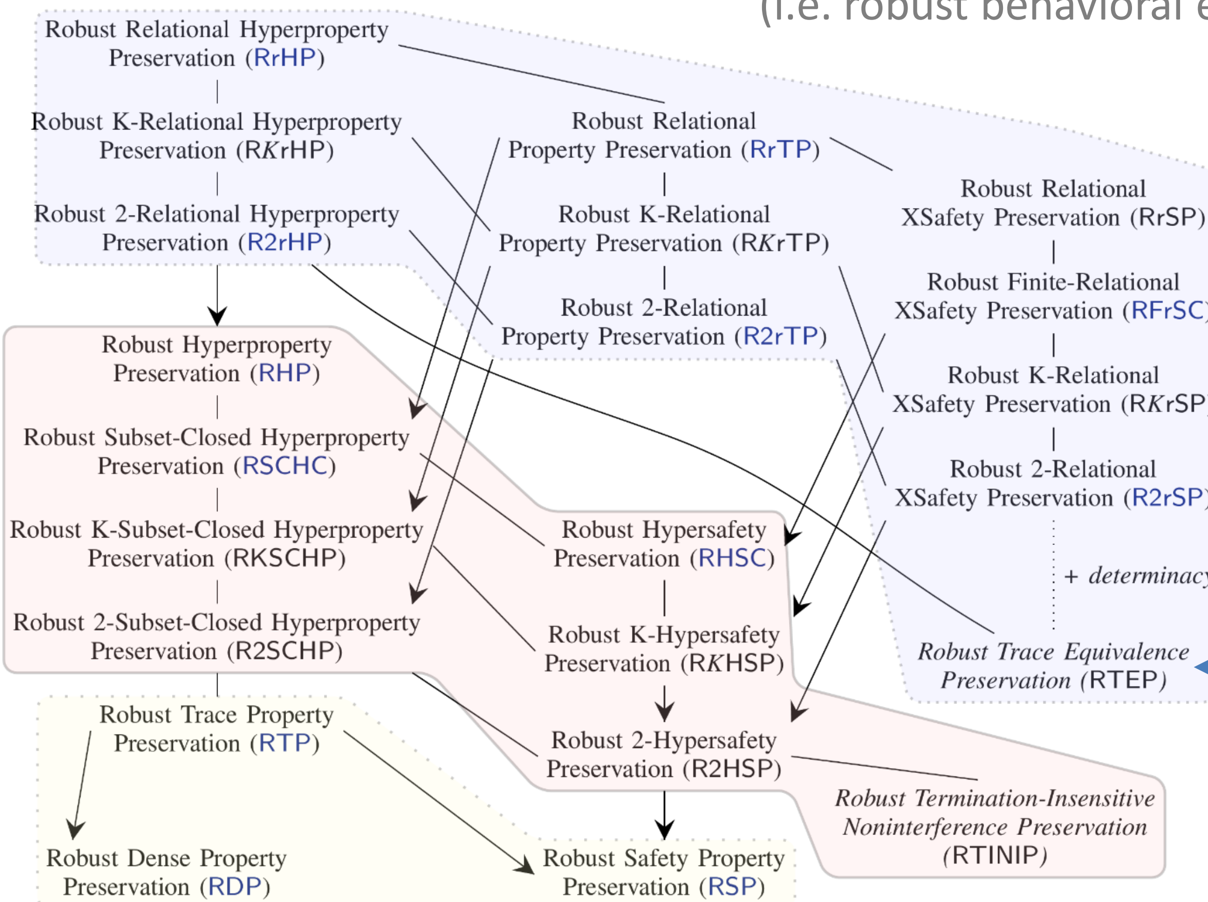


without internal nondeterminism, **full abstraction is here**

doesn't imply any of our criteria
(even assuming compiler correctness)

Where is Full Abstraction?

(i.e. robust behavioral equivalence preservation)



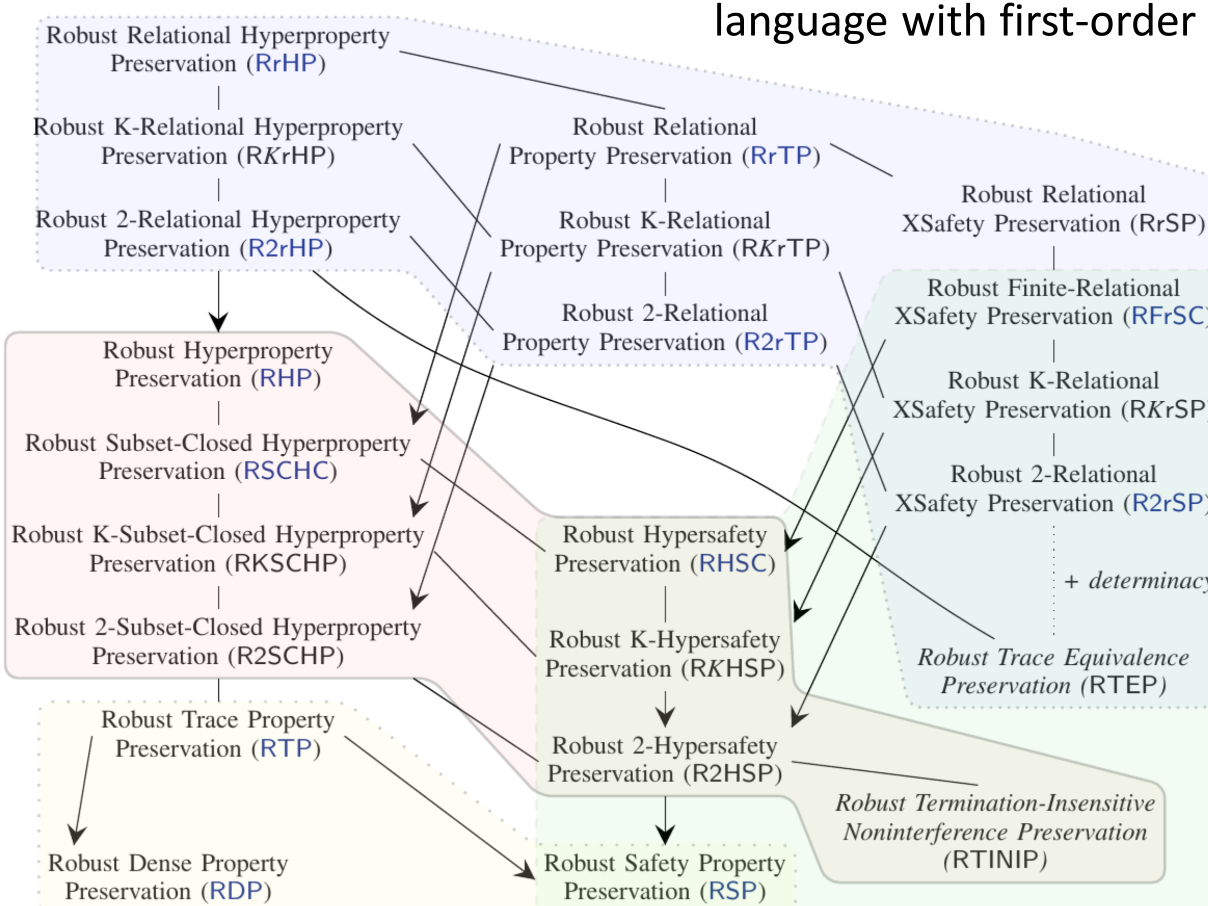
without internal nondeterminism, **full abstraction is here**

doesn't imply any of our criteria
(even assuming compiler correctness)

no one-size-fits-all criterion!

Embraced and extended™ proof techniques

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

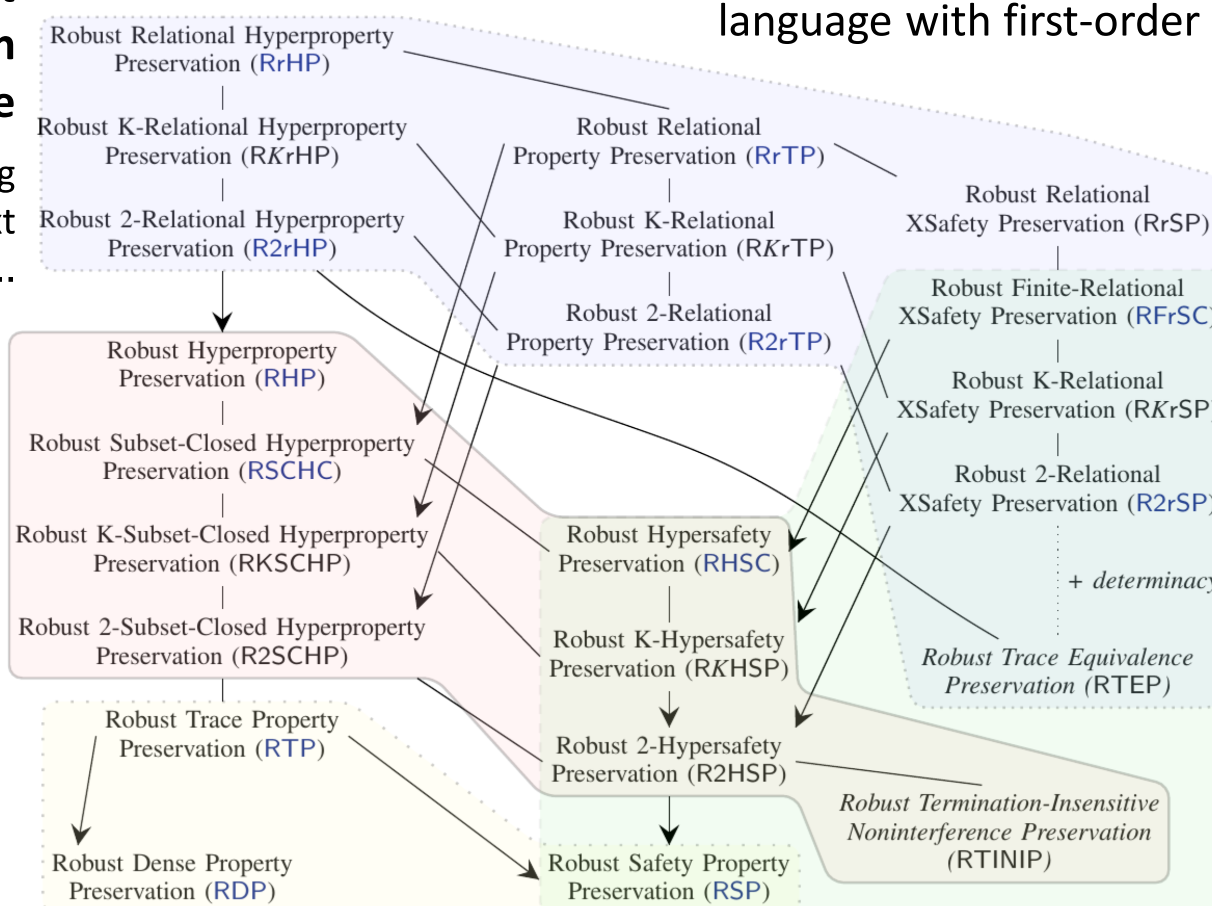


Embraced and extended™ proof techniques

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

strongest criterion achievable

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

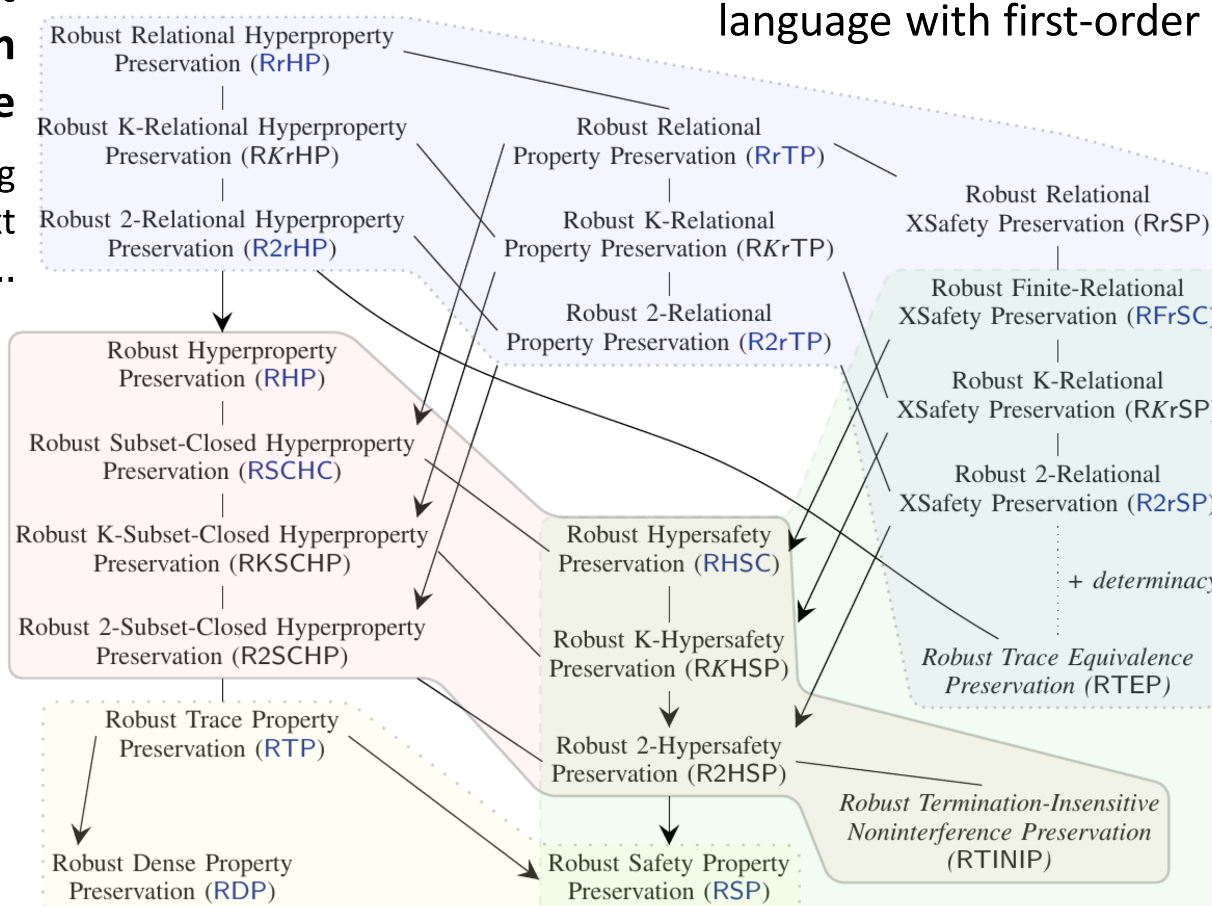


Embraced and extended™ proof techniques

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

strongest criterion achievable

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



generic technique applicable

back-translating finite set of finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T$
$\forall m_1..m_k \exists C_S \ldots$

Some open problems

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

Some open problems

- Practically achieving
  secure interoperability with lower-level code
  - More realistic languages and compilation chains
- Verifying robust satisfaction for source programs
  - partial semantics, program logics, logical relations, ...

Some open problems

- **Practically achieving**
  secure interoperability with lower-level code
  - **More realistic languages and compilation chains**
- **Verifying robust satisfaction for source programs**
  - partial semantics, program logics, logical relations, ...
- **Exploring other kinds of secure compilation**
  - target observations richer than source observations
  - generalize **noninterference preservation** **with side-channels**?

Part 2 of 2

Secure Compilation for Unsafe Languages

When Good Components Go Bad (CCS 2018)

Beyond Good and Evil (CSF 2016)

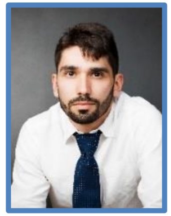
Micro-Policies (IEEE S&P 2015)

A verified information-flow architecture (POPL 2014)

When Good Components Go Bad

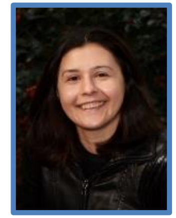
Computer and Communications Security (CCS 2018)

Carmine Abate



Arthur Azevedo de Amorim

Rob Blanco



Ana Nora Evans

Guglielmo Fachini

Cătălin Hrițcu

Théo Laurent

Benjamin Pierce

Marco Stronati

Andrew Tolmach

Inria Paris CMU U. Virginia Portland State UPenn

Inherently insecure languages like C

– any **buffer overflow** can be catastrophic

Inherently insecure languages like C

– any **buffer overflow** can be catastrophic

– ~100 different **undefined behaviors** in the usual C compiler:

- **use after frees and double frees, invalid casts, signed integer overflows, ...**

Inherently insecure languages like C

- any **buffer overflow** can be catastrophic
- ~100 different **undefined behaviors** in the usual C compiler:
  - **use after frees and double frees, invalid casts, signed integer overflows, ...**
- **root cause**, but very challenging to fix:
  - **efficiency**, precision, scalability, backwards compatibility, deployment

Compartmentalization mitigation

- **Break up security-critical applications** into **mutually distrustful components** with **clearly specified privileges**

Compartmentalization mitigation

- **Break up security-critical applications** into **mutually distrustful components** with **clearly specified privileges**
- **Protect source abstractions all the way down**
  - separation, static privileges, call-return discipline, types, ...

Compartmentalization mitigation

- **Break up security-critical applications** into **mutually distrustful components** with **clearly specified privileges**
- **Protect source abstractions all the way down**
  - separation, static privileges, call-return discipline, types, ...
- **Compartmentalizing compilation chain:**
  - compiler, linker, loader, runtime, system, hardware

Compartmentalization mitigation

- **Break up security-critical applications** into **mutually distrustful components** with **clearly specified privileges**
- **Protect source abstractions all the way down**
  - separation, static privileges, call-return discipline, types, ...
- **Compartmentalizing compilation chain:**
  - compiler, linker, loader, runtime, system, hardware
- **Base this on efficient enforcement mechanisms:**
  - **OS processes (all web browsers)**
  - **WebAssembly (web browsers)**
  - **software fault isolation (SFI)**
  - hardware enclaves (SGX)
  - capability machines
  - tagged architectures

Strong security!?

Strong security!?

- **Security guarantees one can make fully water-tight**
  - beyond just "increasing attacker effort"

Strong security!?

- **Security guarantees one can make fully water-tight**
  - beyond just "increasing attacker effort"
- **Intuitively, if we use compartmentalization ...**

... **a vulnerability in one component does not immediately destroy the security of the whole application**

Strong security!?

- **Security guarantees one can make fully water-tight**
  - beyond just "increasing attacker effort"
- **Intuitively, if we use compartmentalization ...**

... **a vulnerability in one component does not immediately destroy the security of the whole application**

... **since each component is protected from all the others**

Strong security!?

- **Security guarantees one can make fully water-tight**
  - beyond just "increasing attacker effort"
- **Intuitively, if we use compartmentalization ...**

... **a vulnerability in one component does not immediately destroy the security of the whole application**

... **since each component is protected from all the others**

... and each component receives protection as long as it has not been compromised (e.g. by a buffer overflow)

Can we formalize this intuition?

Can we formalize this intuition?

What is a compartmentalizing compilation chain supposed to enforce precisely?

Can we formalize this intuition?

What is a compartmentalizing compilation chain supposed to enforce precisely?

We answer this question:

Formal definition expressing the end-to-end security guarantees of compartmentalization

Challenge formalizing security of mitigations

- **We want source-level security reasoning principles**
  - easier to **reason about security in the source language** if and application is compartmentalized

Challenge formalizing security of mitigations

- **We want source-level security reasoning principles**
  - easier to **reason about security in the source language** if and application is compartmentalized
- **... even in the presence of undefined behavior**
  - can't be expressed at all by source language semantics!

Challenge formalizing security of mitigations

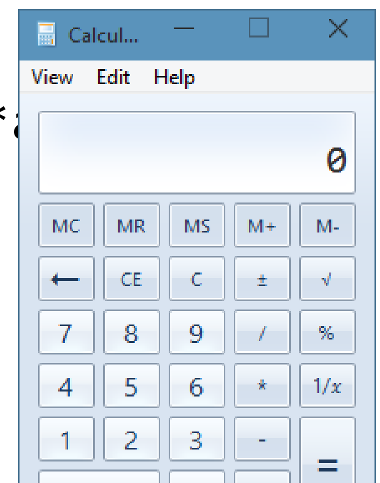
- **We want source-level security reasoning principles**
  - easier to **reason about security in the source language** if and application is compartmentalized
- **... even in the presence of undefined behavior**
  - can't be expressed at all by source language semantics!
  - **what does the following program do?**

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

Challenge formalizing security of mitigations

- **We want source-level security reasoning principles**
  - easier to **reason about security in the source language** if and application is compartmentalized
- **... even in the presence of undefined behavior**
  - can't be expressed at all by source language semantics!
  - **what does the following program do?**

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



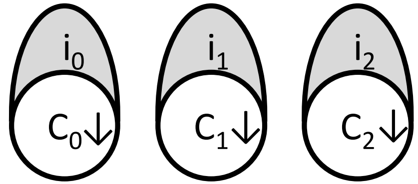
Compartmentalizing compilation should ...

- **Restrict spatial scope** of undefined behavior
  - **mutually-distrustful components**
    - **each component protected from all the others**

Compartmentalizing compilation should ...

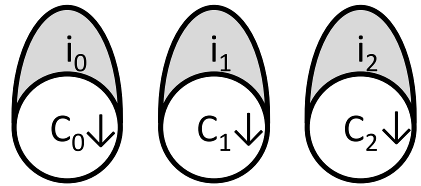
- **Restrict spatial scope** of undefined behavior
  - **mutually-distrustful components**
    - **each component protected from all the others**
- **Restrict temporal scope** of undefined behavior
  - **dynamic compromise**
    - **each component gets guarantees as long as it has not encountered undefined behavior**
    - i.e. the mere existence of vulnerabilities doesn't necessarily make a component compromised

Security definition: If



$\rightsquigarrow t$ then

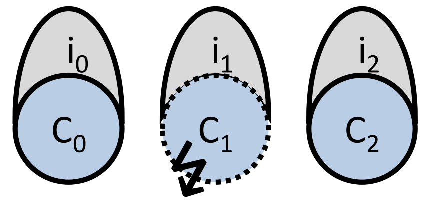
Security definition: If



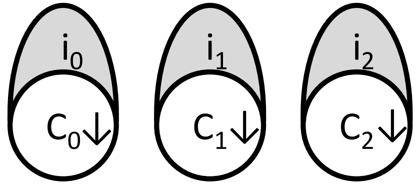
$\rightsquigarrow t$ then

∃ a sequence of component compromises explaining the finite trace t in the source language, for instance $t=m_1 \cdot m_2 \cdot m_3$ and

(1)

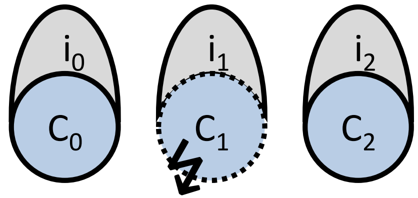


$\rightsquigarrow^* m_1 \cdot \text{Undef}(C_1)$

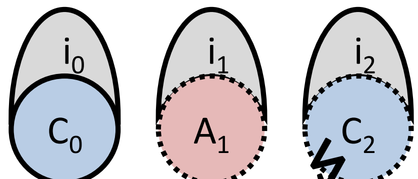
Security definition: If 

⇝ t then

∃ a sequence of component compromises explaining the finite trace t in the source language, for instance $t = m_1 \cdot m_2 \cdot m_3$ and

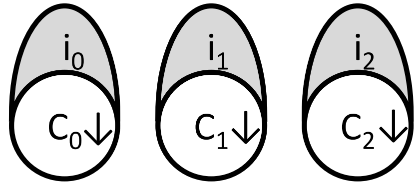
(1) 

⇝* $m_1 \cdot \mathrm{Undef}(C_1)$

(2) $\exists A_1$. 

⇝* $m_1 \cdot m_2 \cdot \mathrm{Undef}(C_2)$

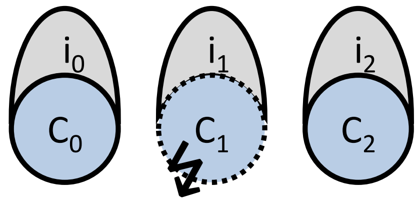
Security definition: If



$\rightsquigarrow t$ then

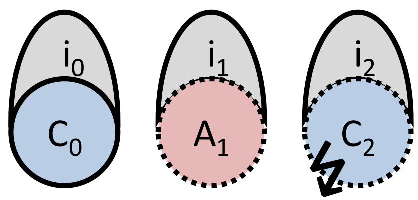
∃ a sequence of component compromises explaining the finite trace t in the source language, for instance $t=m_1 \cdot m_2 \cdot m_3$ and

(1)



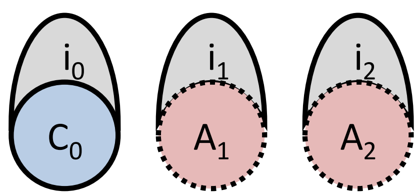
$\rightsquigarrow^* m_1 \cdot \mathrm{Undef}(C_1)$

(2) $\exists A_1.$



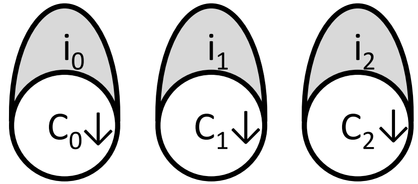
$\rightsquigarrow^* m_1 \cdot m_2 \cdot \mathrm{Undef}(C_2)$

(3) $\exists A_2.$



$\rightsquigarrow m_1 \cdot m_2 \cdot m_3$

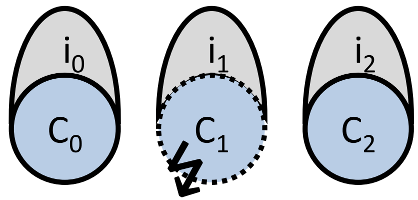
Security definition: If



$\rightsquigarrow t$ then

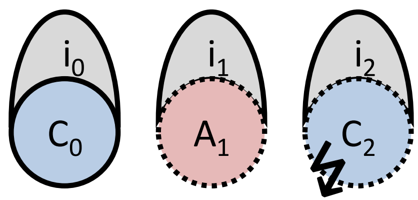
$\exists$ a sequence of component compromises explaining the finite trace t in the source language, for instance $t = m_1 \cdot m_2 \cdot m_3$ and

(1)



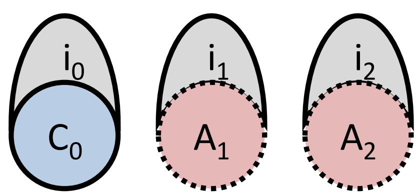
$\rightsquigarrow^* \; m_1 \cdot \text{Undef}(C_1)$

(2) $\exists A_1$.



$\rightsquigarrow^* \; m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

(3) $\exists A_2$.



$\rightsquigarrow \; m_1 \cdot m_2 \cdot m_3$

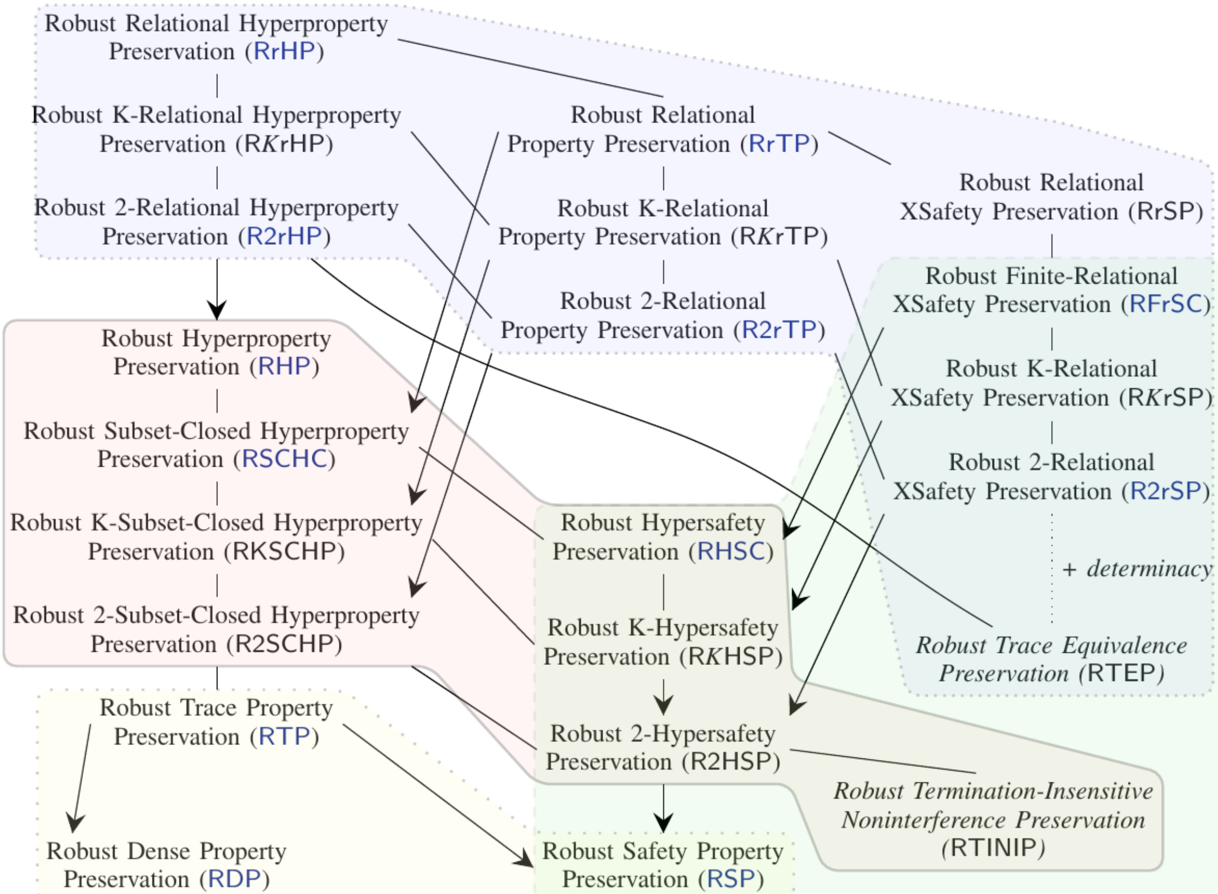
Finite trace records which component encountered undefined behavior and allows us to rewind execution

How can we prove this?

How can we prove this?

When compilation and back-translation are **compositional**, previous definition reduces to (a variant of)
Robust Safety Preservation

How can we prove this?

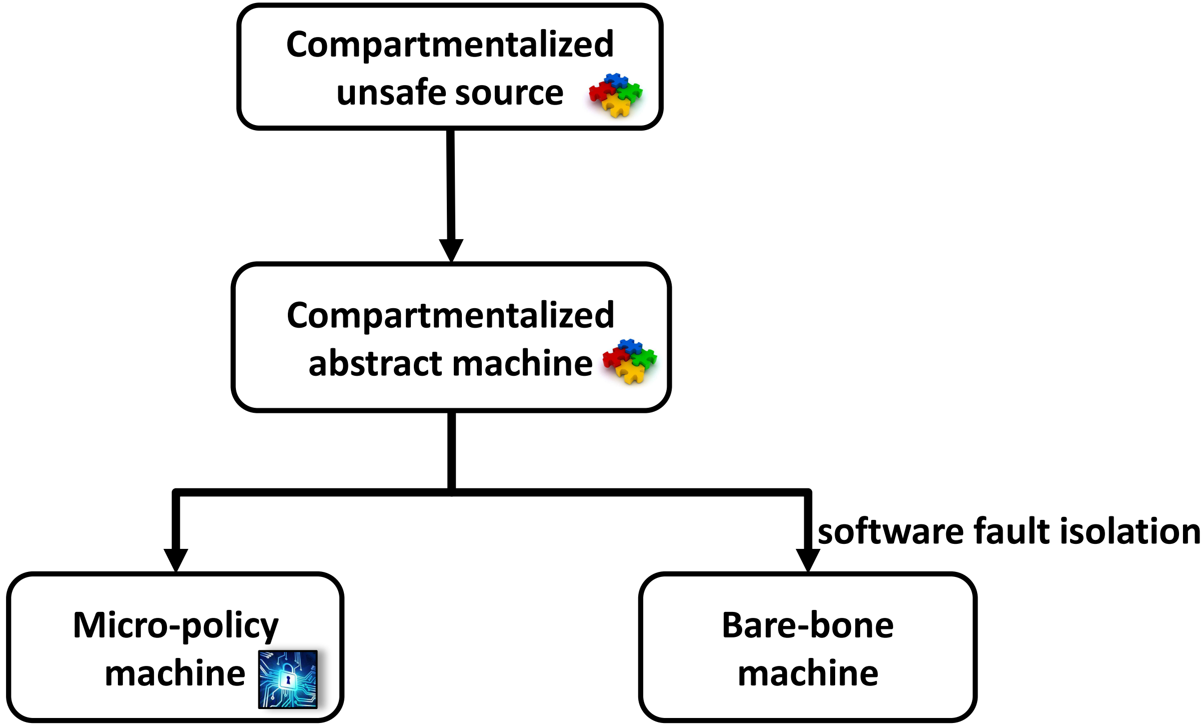


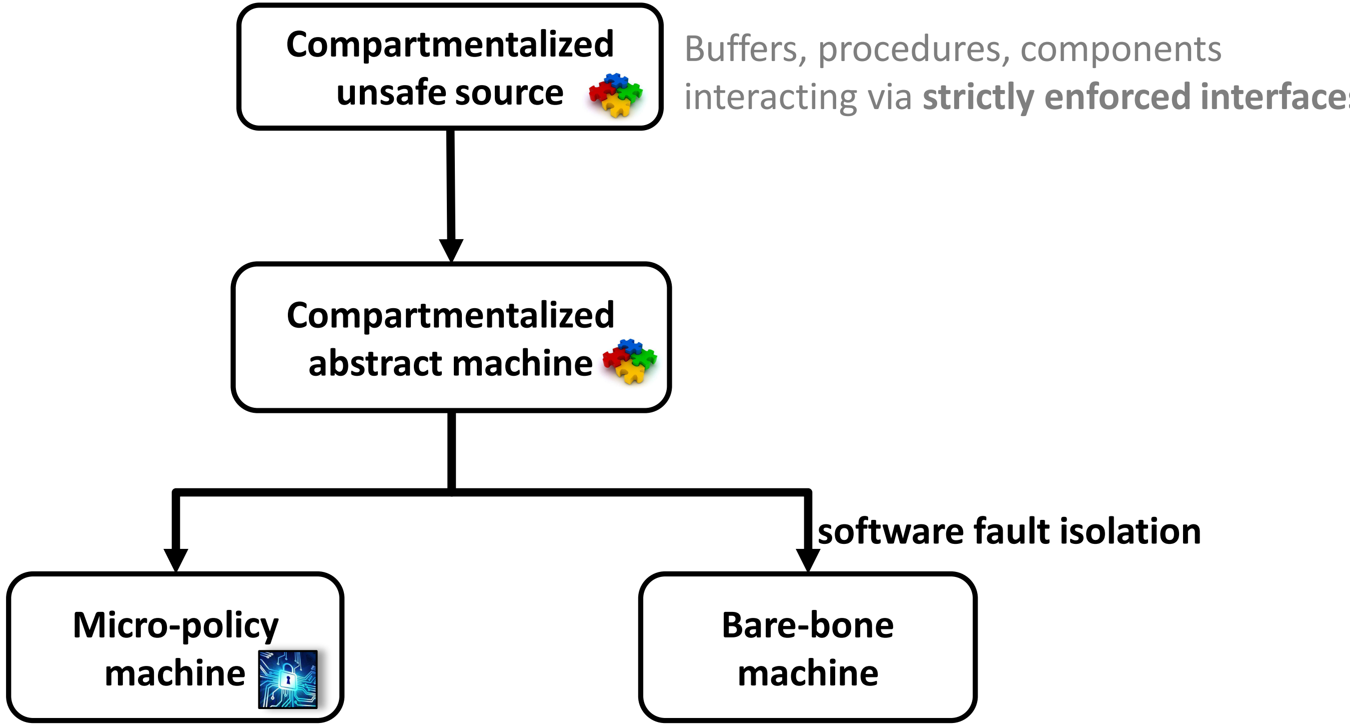
back-translating finite trace prefix $\forall P \forall C_T \forall m \leq t \exists C_S \ldots$

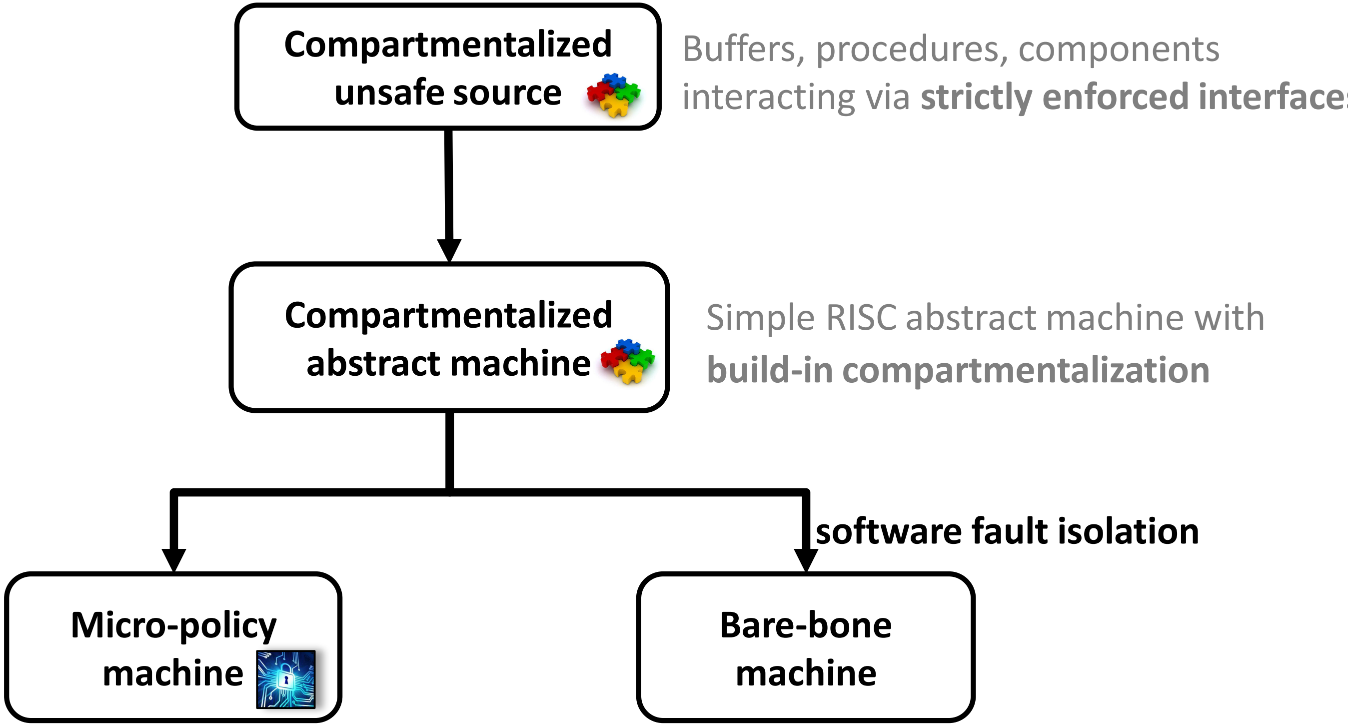
When compilation and back-translation are **compositional**, previous definition reduces to (a variant of) **Robust Safety Preservation**

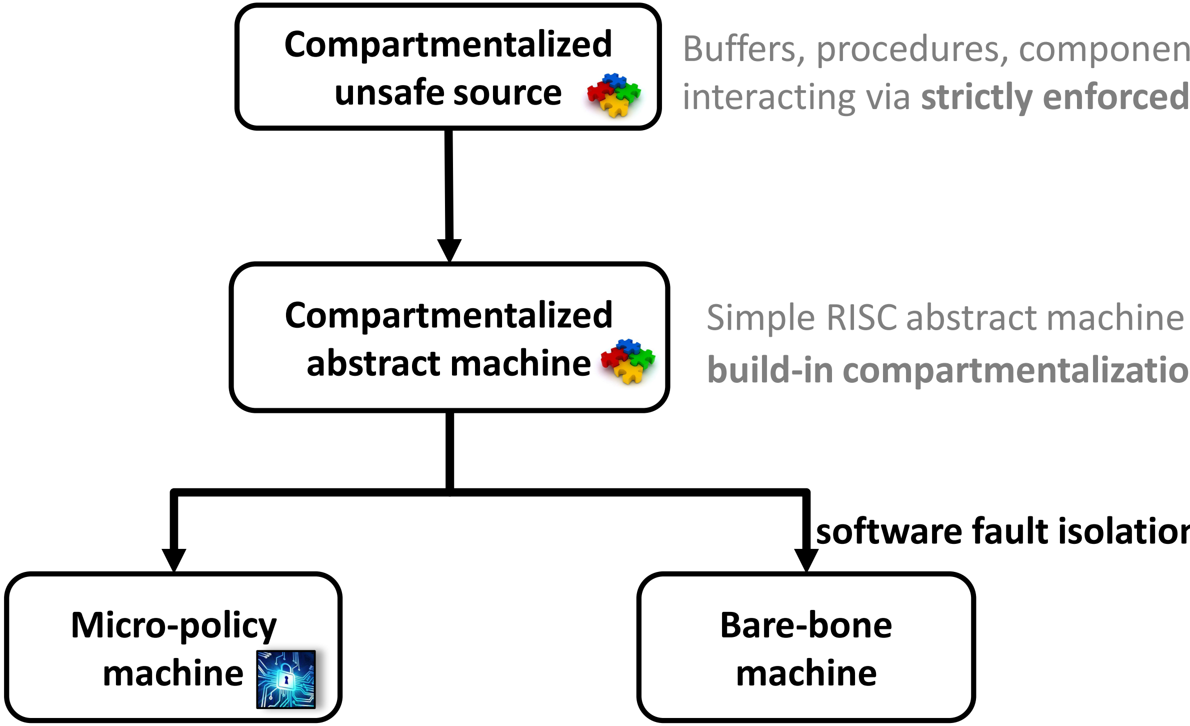
Proof-of-concept formally secure compilation chain in Coq

Illustrates our formal definition


Compartmentalized unsafe source
Compartmentalized abstract machine
software fault isolation
Micro-policy machine
Bare-bone machine


Compartmentalized unsafe source
Buffers, procedures, components interacting via **strictly enforced interfaces**
Compartmentalized abstract machine
software fault isolation
Micro-policy machine
Bare-bone machine

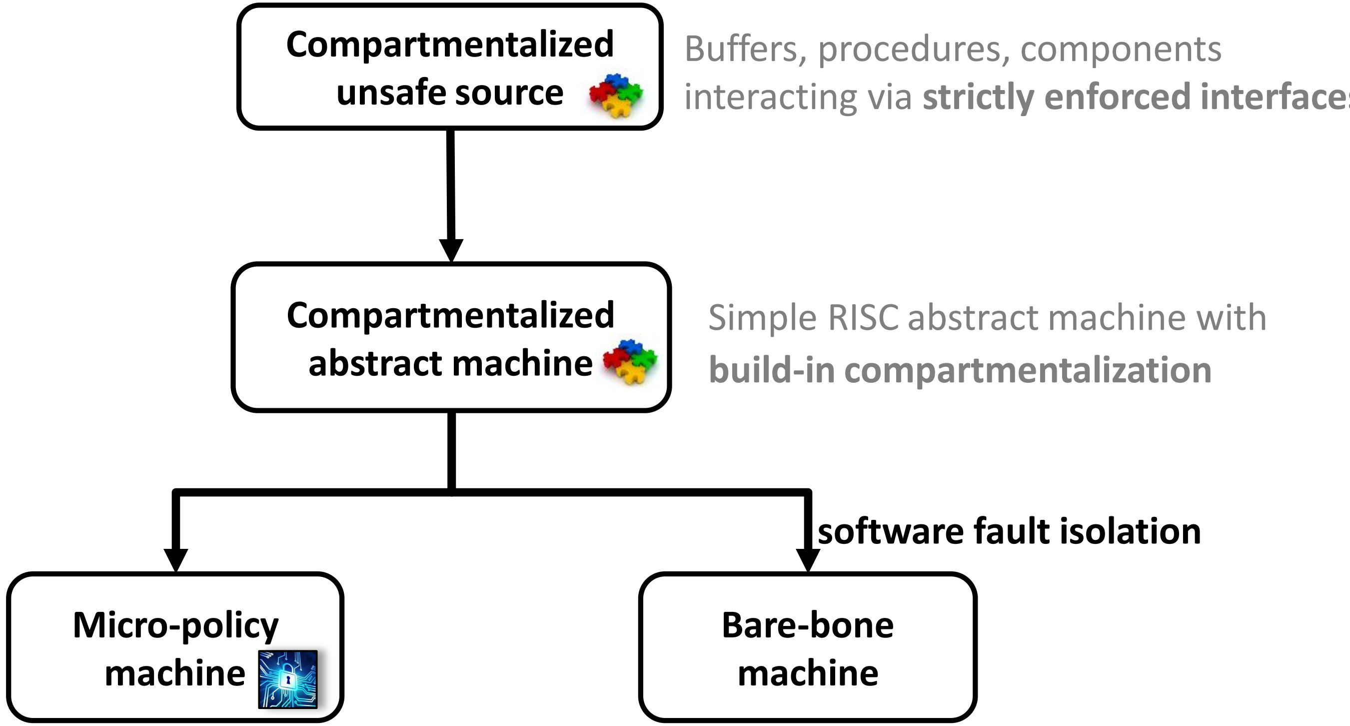

Compartmentalized unsafe source
Buffers, procedures, components interacting via **strictly enforced interfaces**
Compartmentalized abstract machine
Simple RISC abstract machine with **build-in compartmentalization**
software fault isolation
Micro-policy machine
Bare-bone machine



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

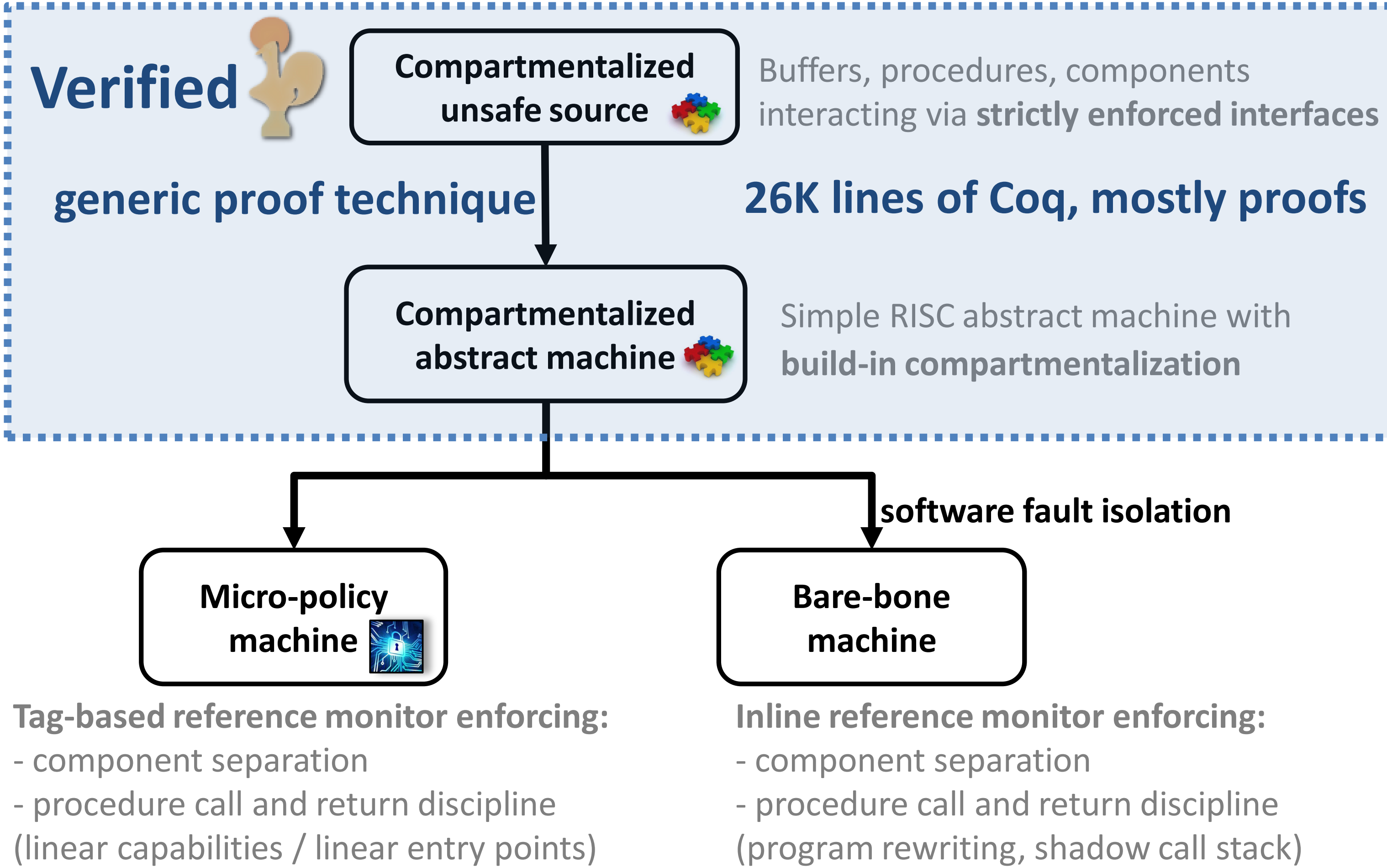
Simple RISC abstract machine with **build-in compartmentalization**

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



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)


Verified
Compartmentalized unsafe source
Buffers, procedures, components interacting via **strictly enforced interfaces**
generic proof technique
26K lines of Coq, mostly proofs
Compartmentalized abstract machine
Simple RISC abstract machine with **build-in compartmentalization**
software fault isolation
Micro-policy machine
Bare-bone machine
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)

Verified

Compartmentalized unsafe source

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

generic proof technique

26K lines of Coq, mostly proofs

Compartmentalized abstract machine

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

software fault isolation

Micro-policy machine

Bare-bone machine

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

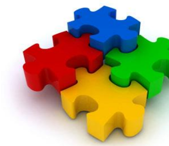
Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)

Systematically tested (with QuickChick)

When Good Components Go Bad

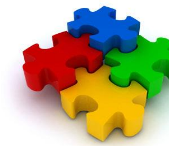
- **Formalized security of compartmentalization**
  - **first definition** supporting **dynamic compromise**
  - **restricting undefined behavior spatially** and **temporally**

When Good Components Go Bad



- **Formalized security of compartmentalization**
  - **first definition** supporting **dynamic compromise**
  - **restricting undefined behavior spatially** and **temporally**
- **Proof-of-concept secure compilation chain in Coq**
  - **software fault isolation** or **tag-based reference monitor**

When Good Components Go Bad



- Formalized security of compartmentalization
  - first definition supporting dynamic compromise
  - restricting undefined behavior spatially and temporally

- Proof-of-concept secure compilation chain in Coq
  - software fault isolation or tag-based reference monitor
- Generic definition and proof technique
  - we expect them to extend and scale well

Making this **more practical** ... next steps:

Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **shared memory** (ongoing) and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster

Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **shared memory** (ongoing) and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster
- **Extend all this to dynamic component creation**
  - rewind to when compromised component was created

Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **shared memory** (ongoing) and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster
- **Extend all this to dynamic component creation**
  - rewind to when compromised component was created
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, history-based access control, ...

Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **shared memory** (ongoing) and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster
- **Extend all this to dynamic component creation**
  - rewind to when compromised component was created
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, history-based access control, ...
- **From robust safety to hypersafety (e.g. confidentiality)**

Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **shared memory** (ongoing) and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster
- **Extend all this to dynamic component creation**
  - rewind to when compromised component was created
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, history-based access control, ...
- **From robust safety to hypersafety (e.g. confidentiality)**
- **Secure compilation of Low*** using components, contracts, sealing, ...

My dream: secure compilation at scale

Low* language
(safe C subset in F*)

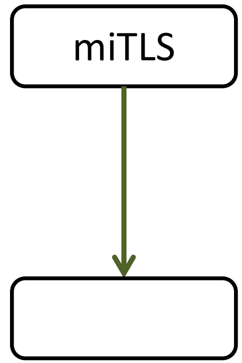
miTLS

C language
+ components
+ memory safety

My dream: secure compilation at scale

Low* language
(safe C subset in F*)

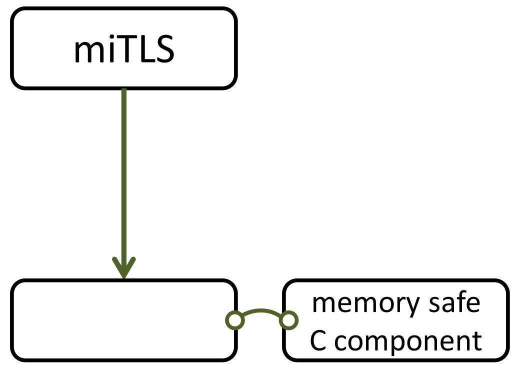
C language
+ components
+ memory safety



My dream: secure compilation at scale

Low* language
(safe C subset in F*)

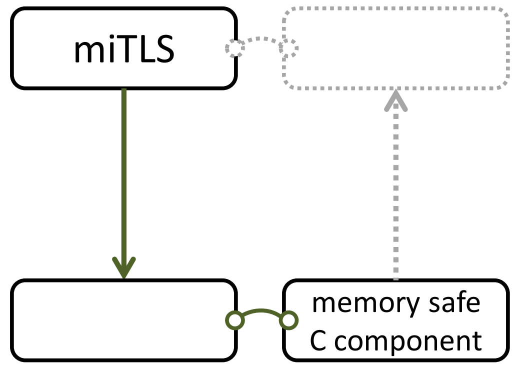
C language
+ components
+ memory safety



My dream: secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

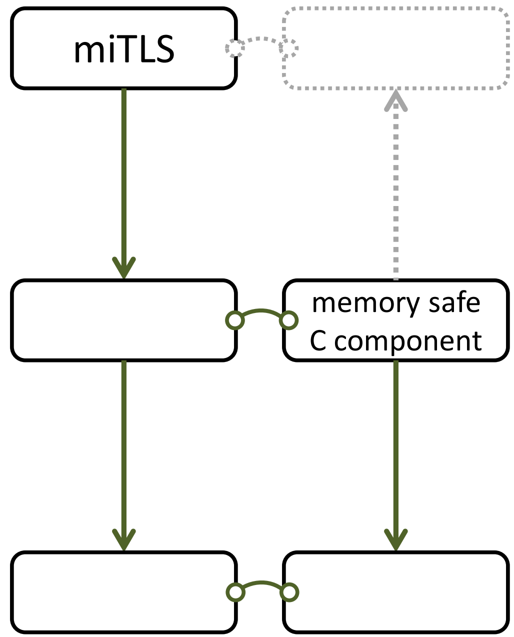


My dream: secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

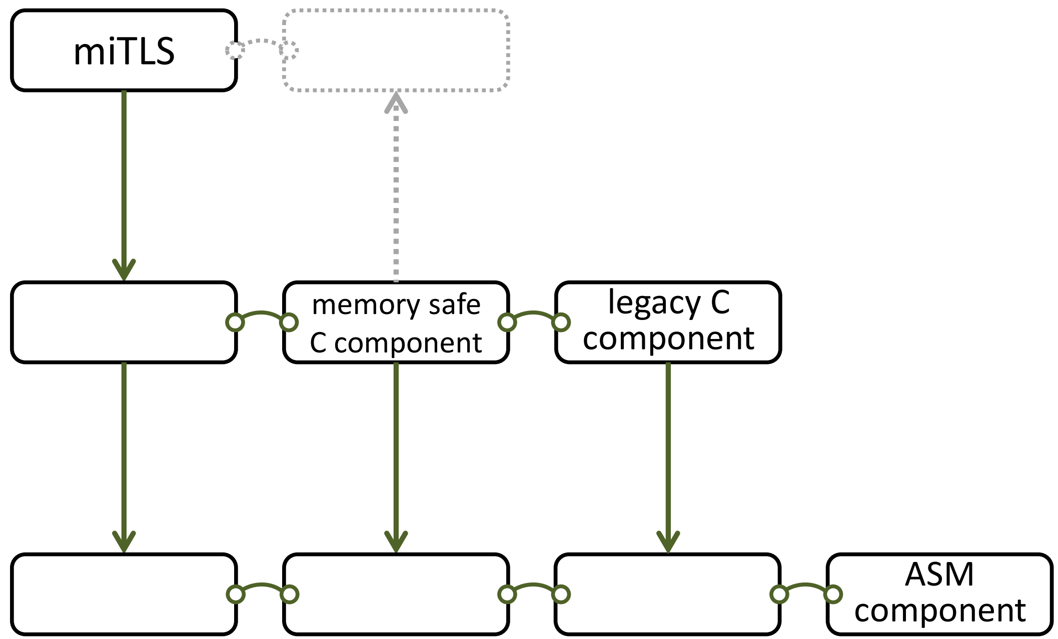


My dream: secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

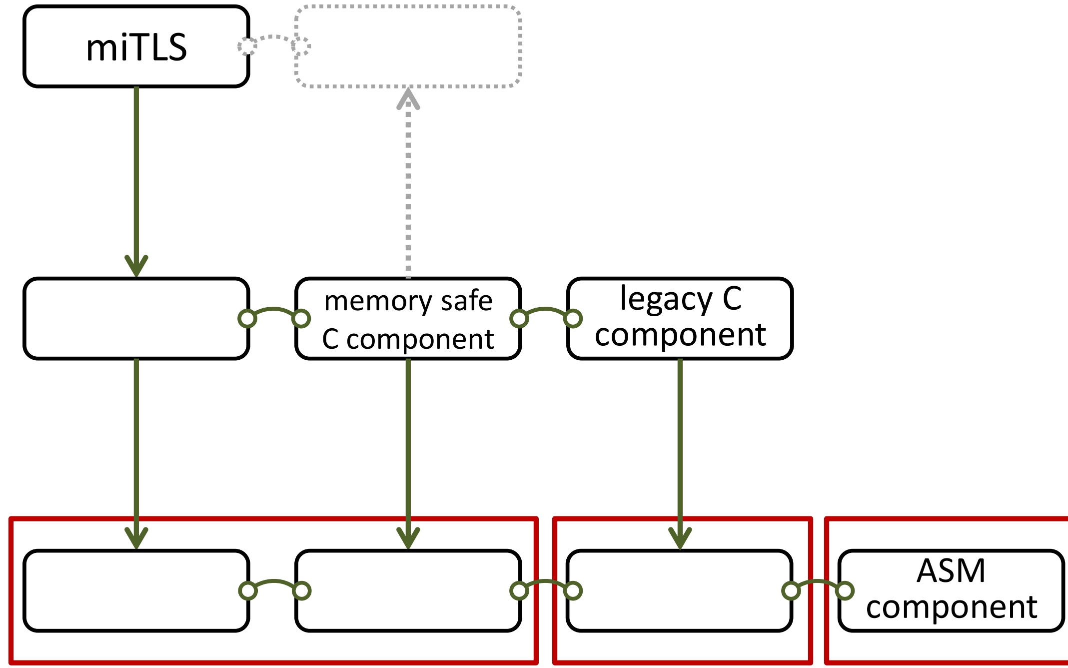


My dream: secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

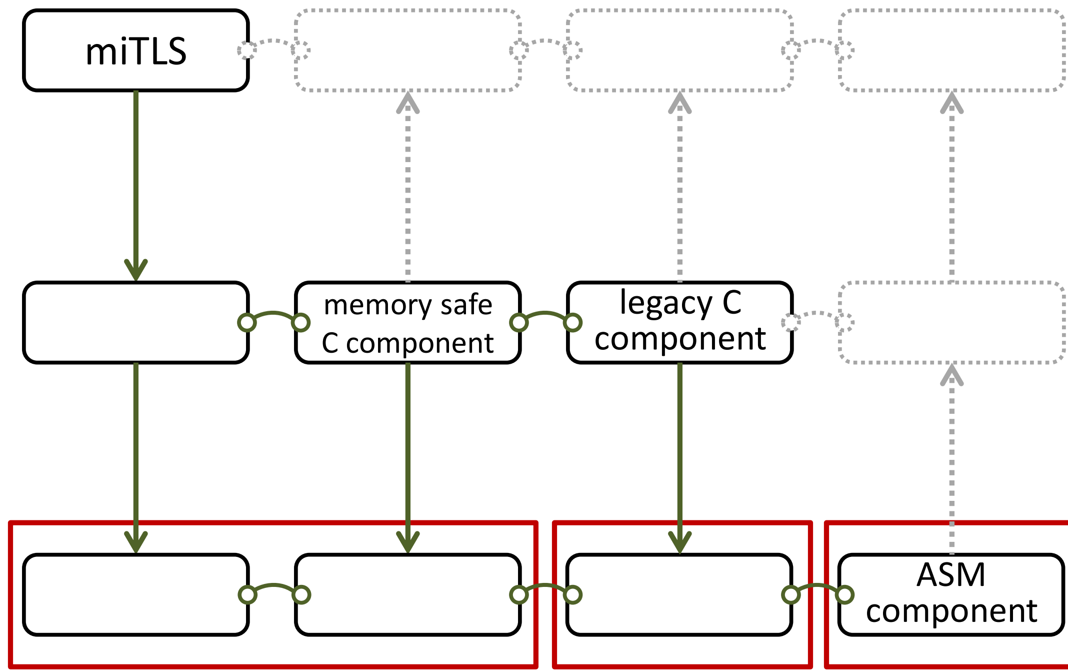


My dream: secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)



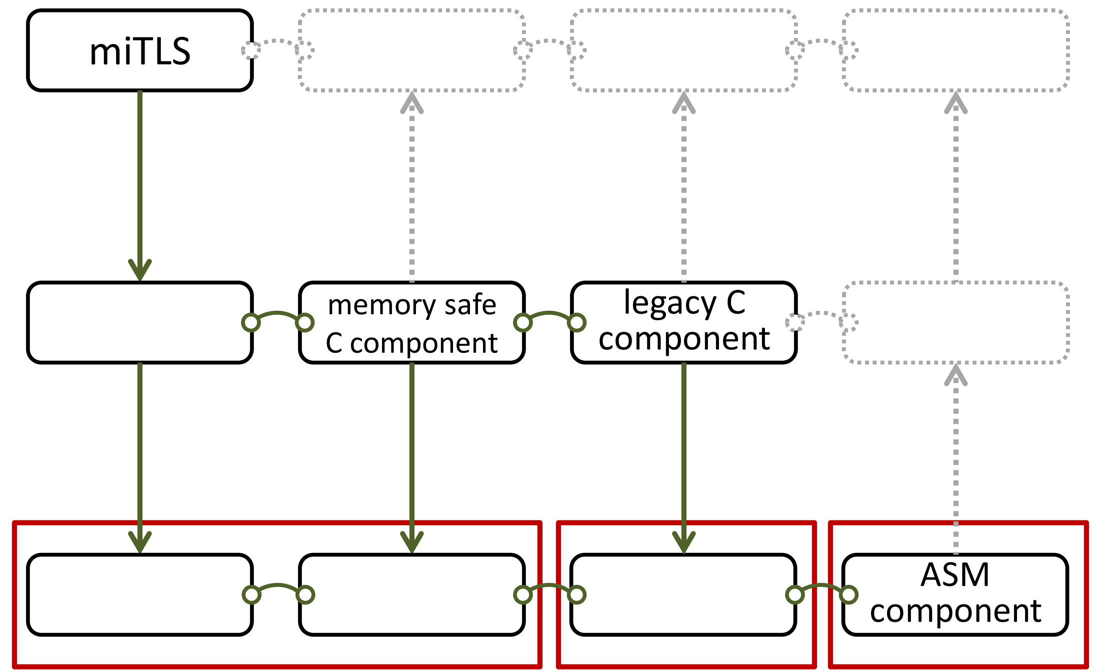
My dream: secure compilation at scale

We're HIRING **PostDocs & Researchers**

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)



Thank you

Past group members:

Alejandro Aguirre
Ana Nora Evans
Anna Bednarik
Arthur Azevedo de Amorim
Clément Pit-Claudel
Danel Ahman
Diane Gallois-Wong
Guglielmo Fachini
Li-yao Xia
Marco Stronati
Nick Giannarakis
Simon Forest
Tomer Libal
Victor Dumitrescu
Yannis Juglaret
Zoe Paraskevopoulou

Current group:

Carmine Abate
Exe Rivas
Florian Groult
Guido Martínez
Jérémy Thibault
Kenji Maillard
Rob Blanco
Théo Laurent

Jury:

David Pointcheval
Frank Piessens
Gilles Barthe
Thomas Jensen
David Pichardie
Karthik Bhargavan
Tamara Rezk
Xavier Leroy

Prosecco team:

Benjamin Beurdouche
Benjamin Lipp
Bruno Blanchet
Denis Merigoux
Elizabeth Labrada
Éric Tanter
Graham Steel
Harry Halpin
Karthik Bhargavan
Marina Polubelova
Mathieu Mourey
Prasad Naldurg

Family:

Beate Brockmann
Gabriela Merticariu
Ioan Hrițcu
Stela Hrițcu