SECOMP: Formally Secure Compilation of Compartmentalized C Programs

Cătălin Hrițcu, MPI-SP, Bochum

Hiring: PostDoc, interns, PhD students

Joint work with
Carmine Abate, Cezar-Constantin Andrici, Sven Argo, Arthur Azevedo de Amorim, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Akram El-Korashy, Boris Eng, Ana Nora Evans, Guglielmo Fachini, Deepak Garg, Aïna Linn Georges, Théo Laurent, Dongjae Lee, Guido Martínez, Marco Patrignani, Benjamin Pierce, Exequiel Rivas, Marco Stronati, Éric Tanter, Jérémy Thibault, Andrew Tolmach, Théo Winterhalter, ...

In part supported by ERC Starting Grant SECOMP

The C programming language is insecure

– any **buffer overflow** can be catastrophic

The C programming language is insecure

– any **buffer overflow** can be catastrophic

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

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

The C programming language is insecure

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

Mitigation: compartmentalization

Mitigation: compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures, pointers & shared memory

Mitigation: compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures, pointers & shared memory
  - used in most programs, **but not enforced at all during compilation**

Mitigation: compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures, pointers & shared memory
  - used in most programs, **but not enforced at all during compilation**
  - **add fine-grained compartments to C which can naturally interact**

Mitigation: compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures, pointers & shared memory
  - used in most programs, **but not enforced at all during compilation**
  - **add fine-grained compartments to C which can naturally interact**
- **Secure compilation chain that protects these abstractions**
  - all the way down, at compartments boundaries (so hopefully more efficient)

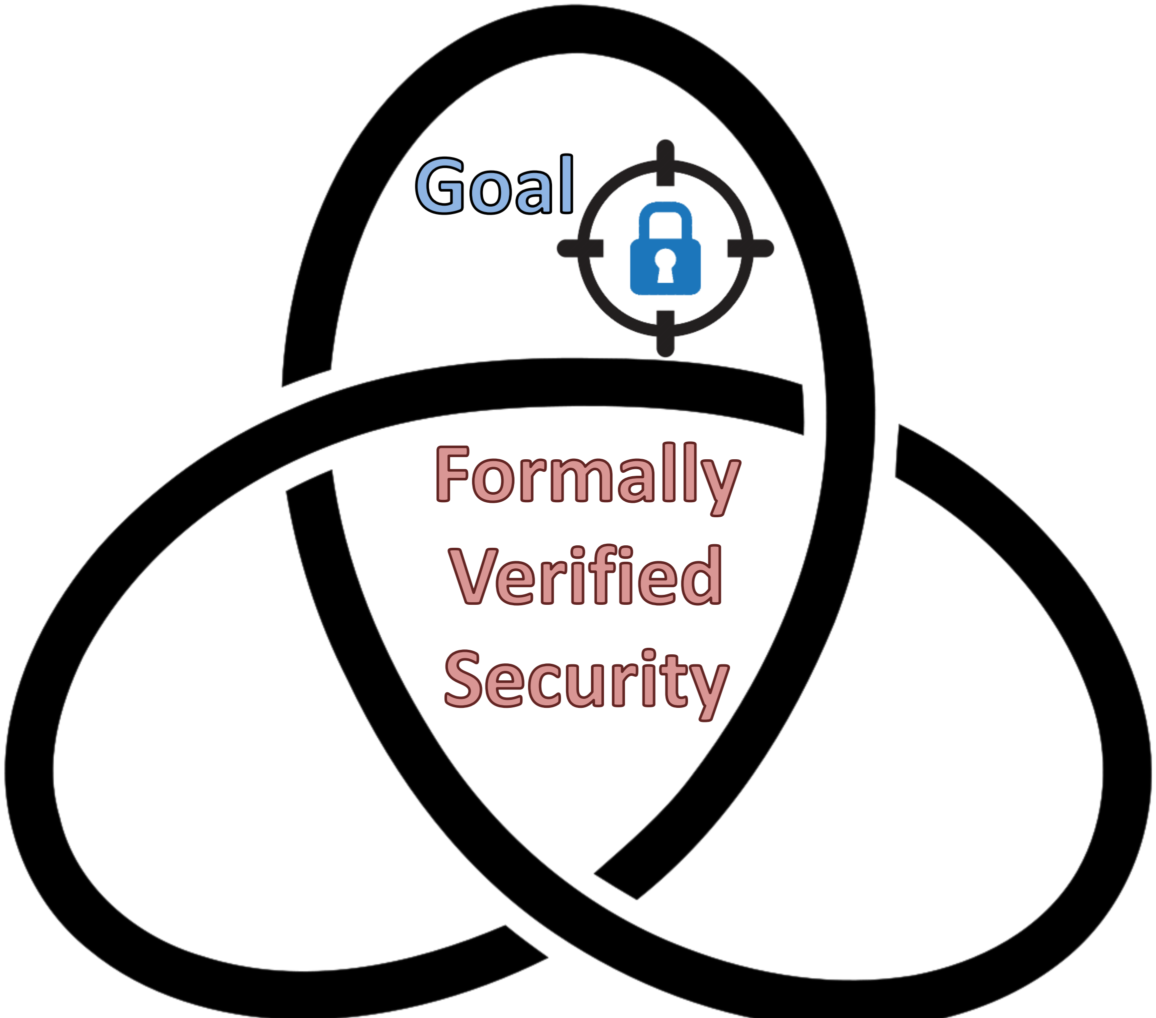
Mitigation: compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures, pointers & shared memory
  - used in most programs, **but not enforced at all during compilation**
  - **add fine-grained compartments to C which can naturally interact**
- **Secure compilation chain that protects these abstractions**
  - all the way down, at compartments boundaries (so hopefully more efficient)
  - against compartments dynamically compromised by undefined behavior

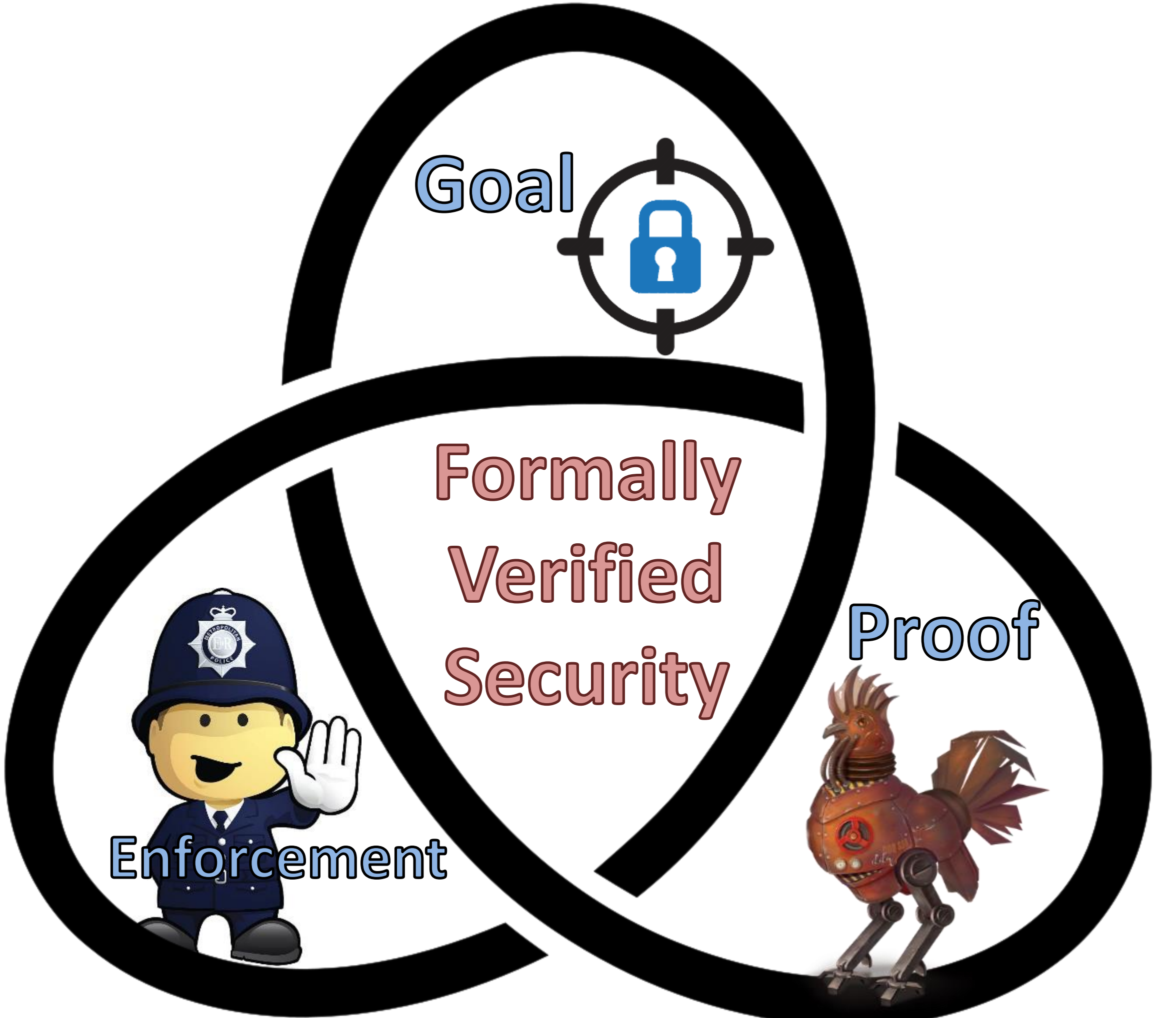
Mitigation: compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures, pointers & shared memory
  - used in most programs, **but not enforced at all during compilation**
  - **add fine-grained compartments to C which can naturally interact**
- **Secure compilation chain that protects these abstractions**
  - all the way down, at compartments boundaries (so hopefully more efficient)
  - against compartments dynamically compromised by undefined behavior
- **Targeting various enforcement mechanisms**
  - software-fault isolation (SFI), capability machines, ...

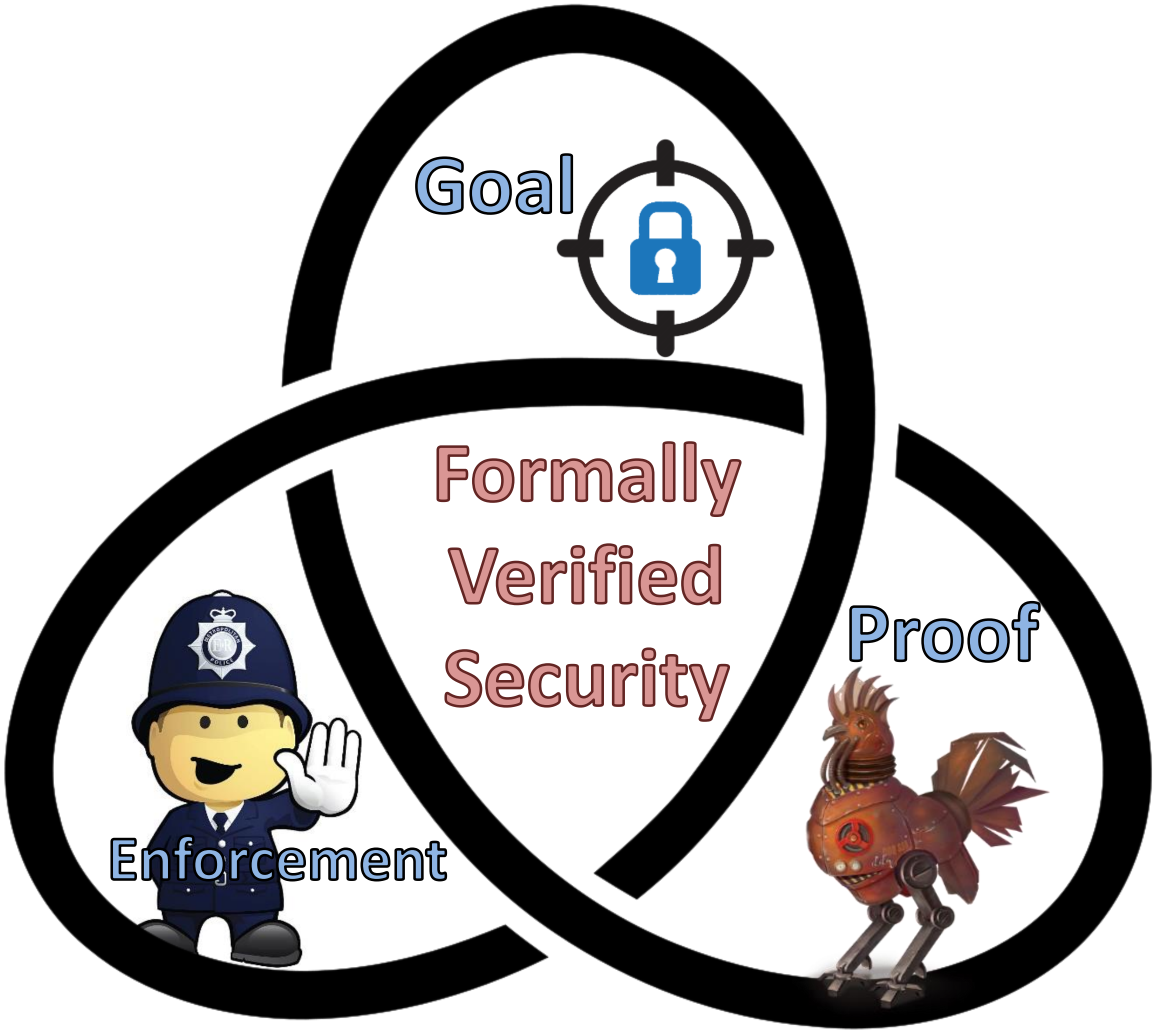
Formally Verified Security


Goal
Formally
Verified
Security


Goal
Formally
Verified
Security
Enforcement


Goal
Formally
Verified
Security
Proof
Enforcement

Formally Secure Compilation of C Compartments



1. Security Goal

1. Security Goal

- **What does it mean for a compilation chain for vulnerable C compartments to be secure?**

1. Security Goal

- **What does it mean for a compilation chain for vulnerable C compartments to be secure?**
- **As a warmup, I will first show an easier definition**
  - **protecting 1 trusted compartment from 1 untrusted one (arbitrary ASM)**
  - **trusted compartment has no vulnerabilities, e.g. formally verified**
    - e.g. EverCrypt verified crypto library, shipping in Firefox, Linux Kernel, ...
    - e.g. simple verified web server, linked with unverified libraries [POPL'24]
- **What does it mean to securely compile such a verified compartment against linked adversarial target-level code?**

Preserving security against adversarial contexts

Preserving security against adversarial contexts

∀ security property π

Preserving security against adversarial contexts

∀ security property π

verified program

Preserving security against adversarial contexts

∀ security property π

verified program

satisfies π

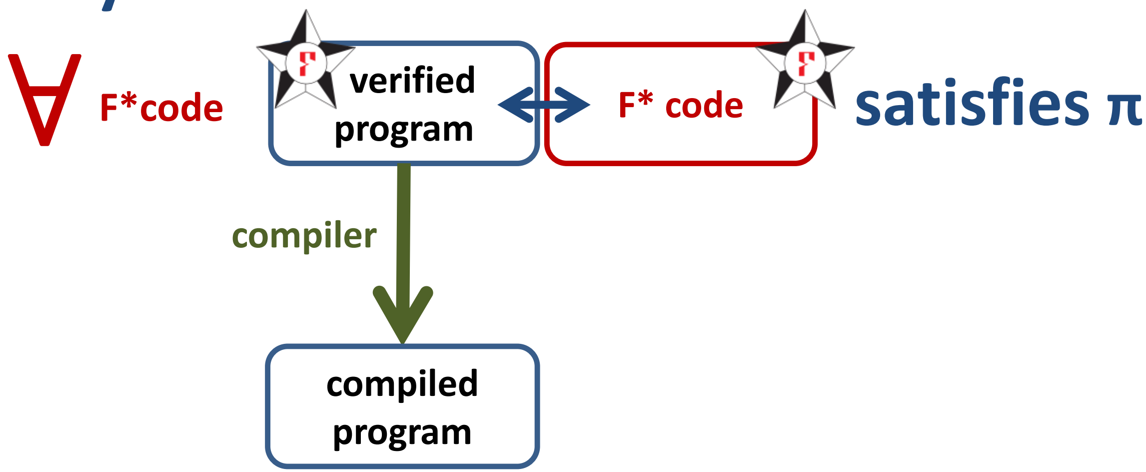
Preserving security against adversarial contexts

∀ security property π

∀ F*code verified program ⟷ F* code satisfies π

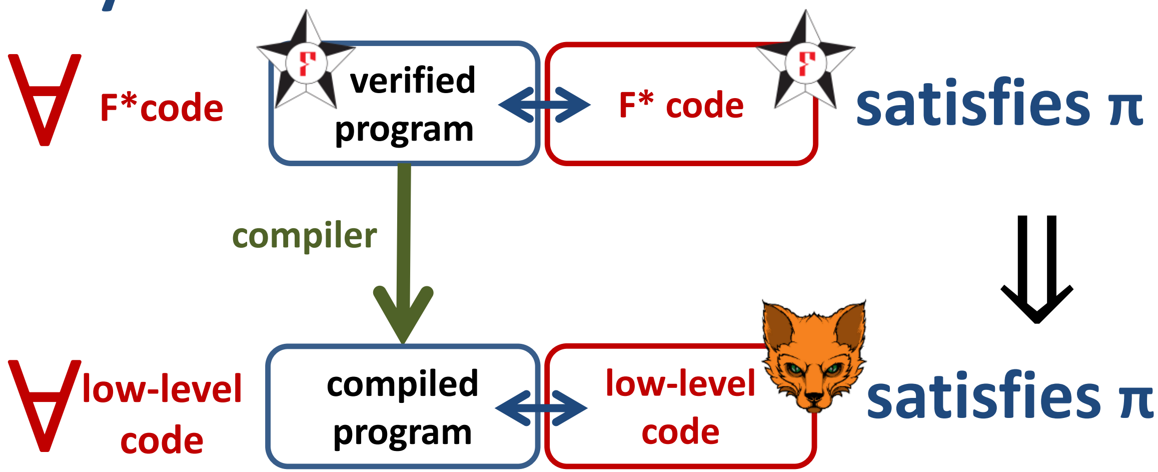
Preserving security against adversarial contexts

∀ security property π



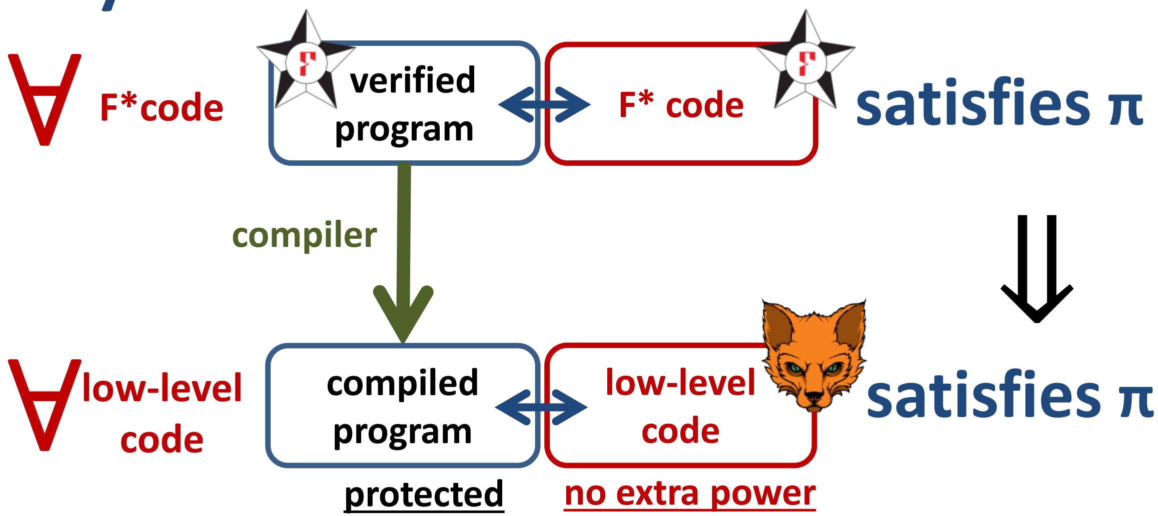
Preserving security against adversarial contexts

∀ security property π



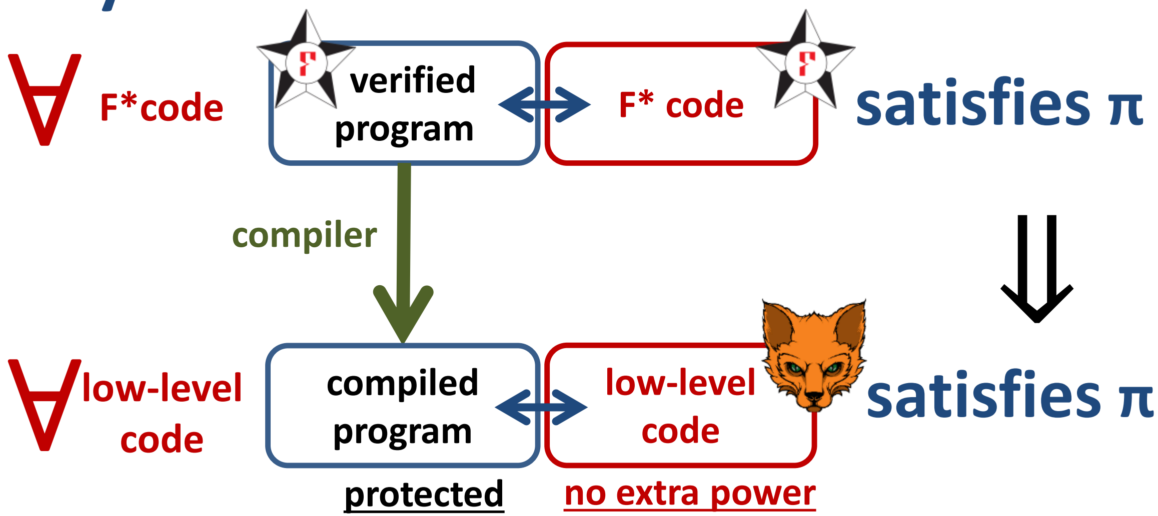
Preserving security against adversarial contexts

∀ security property π



Preserving security against adversarial contexts

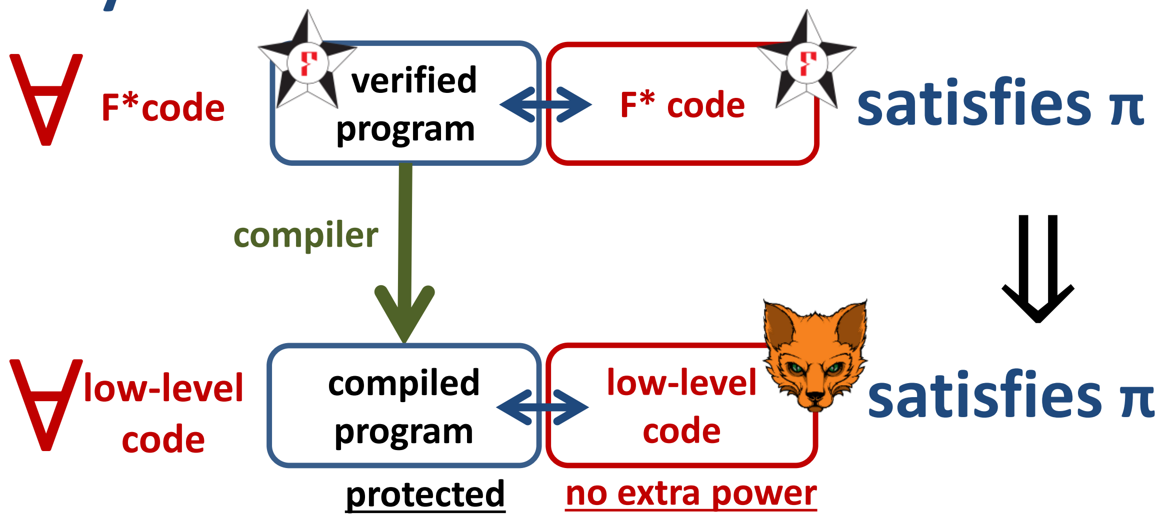
∀ security property π



Where π can e.g. be "the web server's private key is not leaked"

Preserving security against adversarial contexts

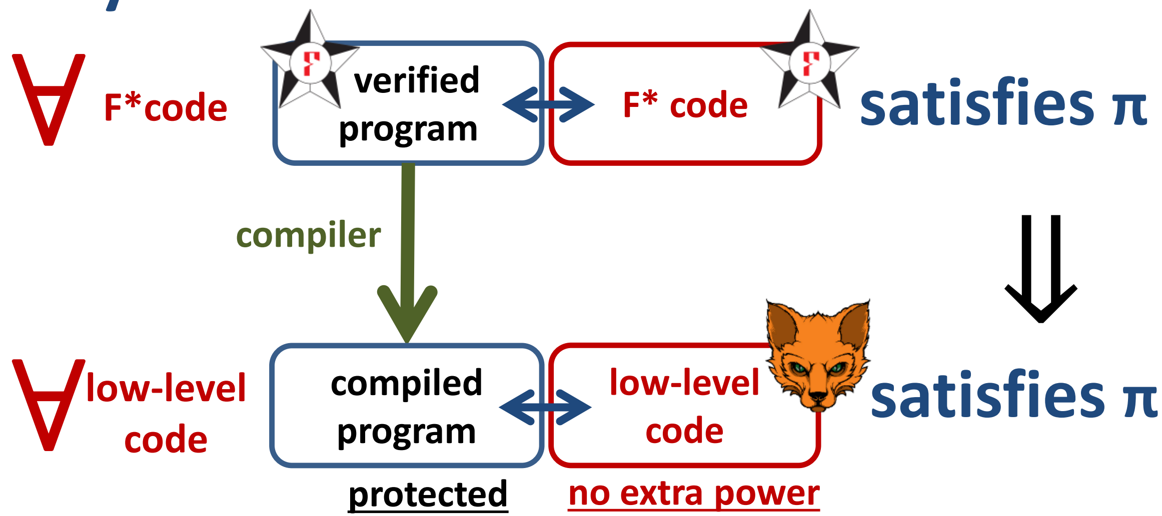
∀ security property π



Where π can e.g. be "the web server's private key is not leaked"

Preserving security against adversarial contexts

∀ security property π



Where π can e.g. be "the web server's private key is not leaked"

We explored many classes of properties one can preserve this way ...

Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

trace properties
(safety & liveness)

Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

hyperproperties
(noninterference)

trace properties
(safety & liveness)

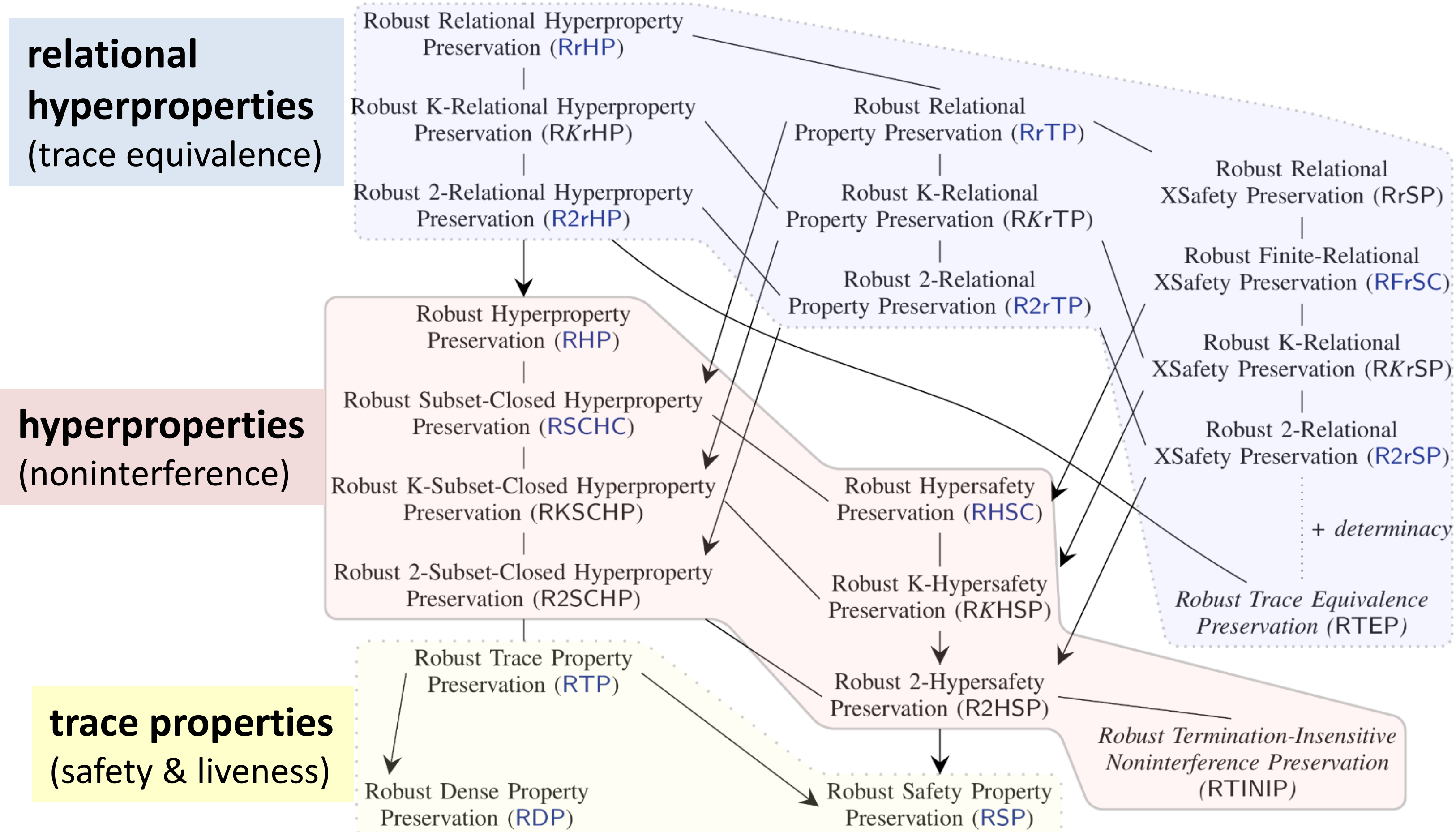
Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

relational hyperproperties
(trace equivalence)

hyperproperties
(noninterference)

trace properties
(safety & liveness)

Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

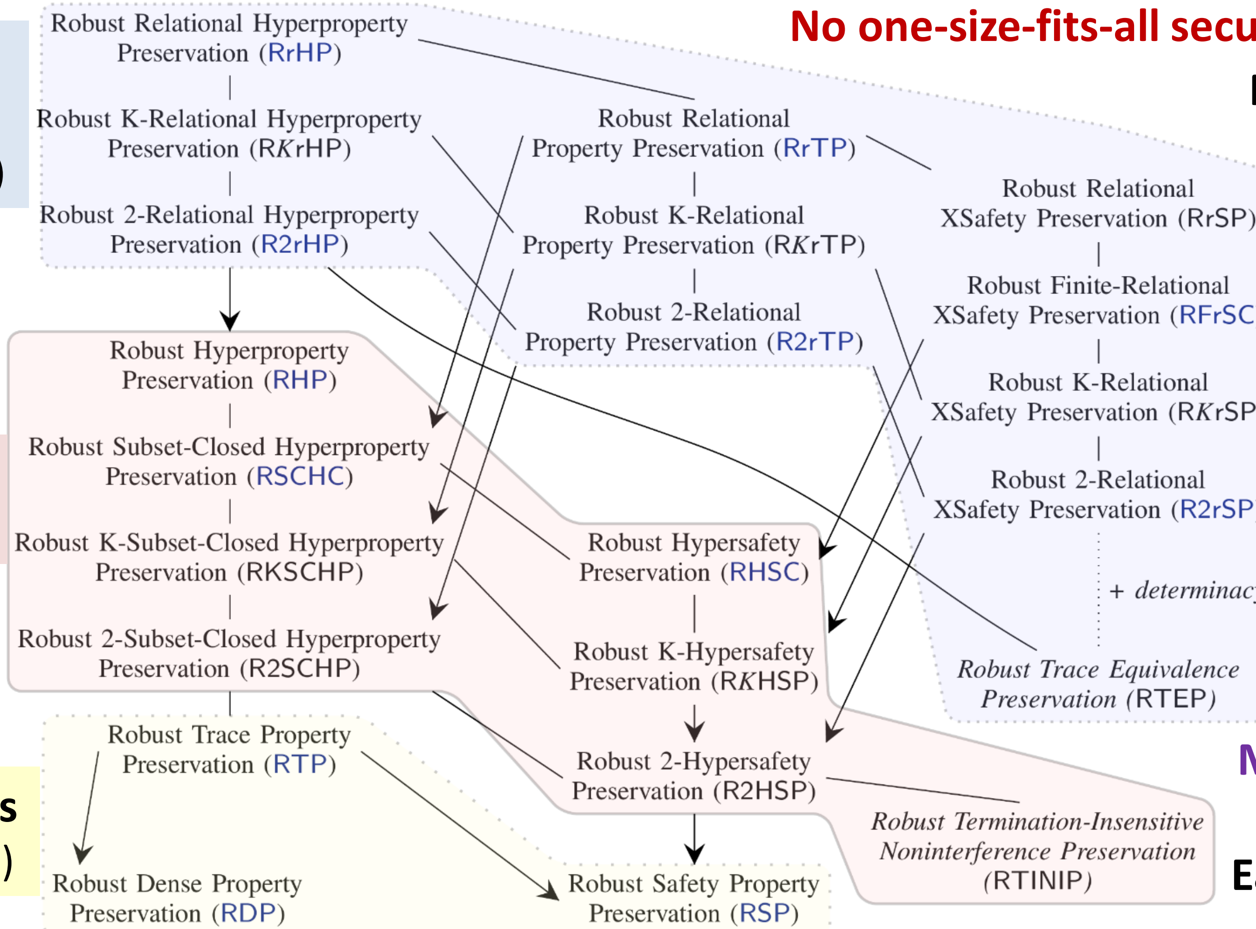


Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

relational hyperproperties
(trace equivalence)

hyperproperties
(noninterference)

trace properties
(safety & liveness)



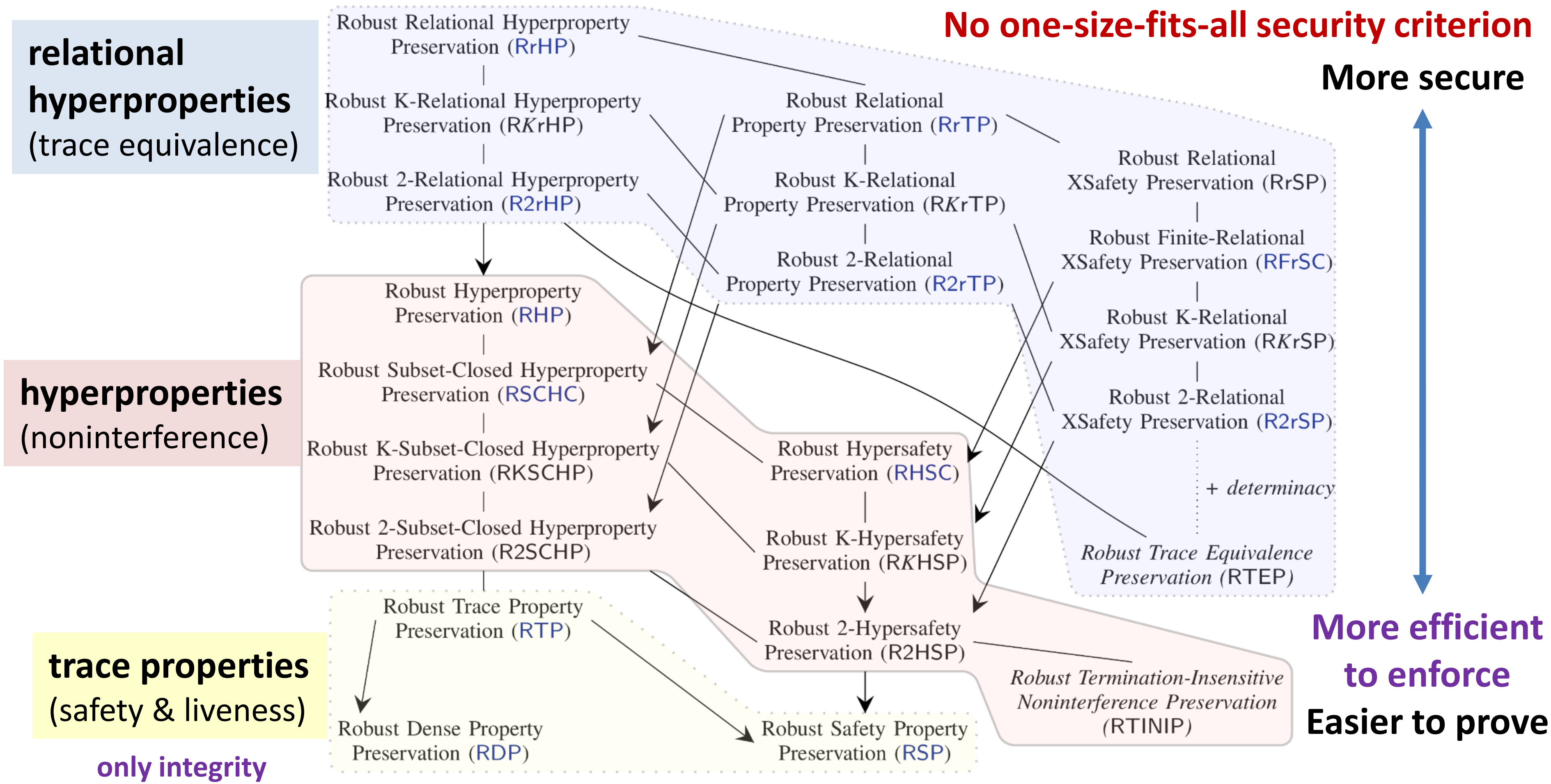
No one-size-fits-all security criterion

More secure

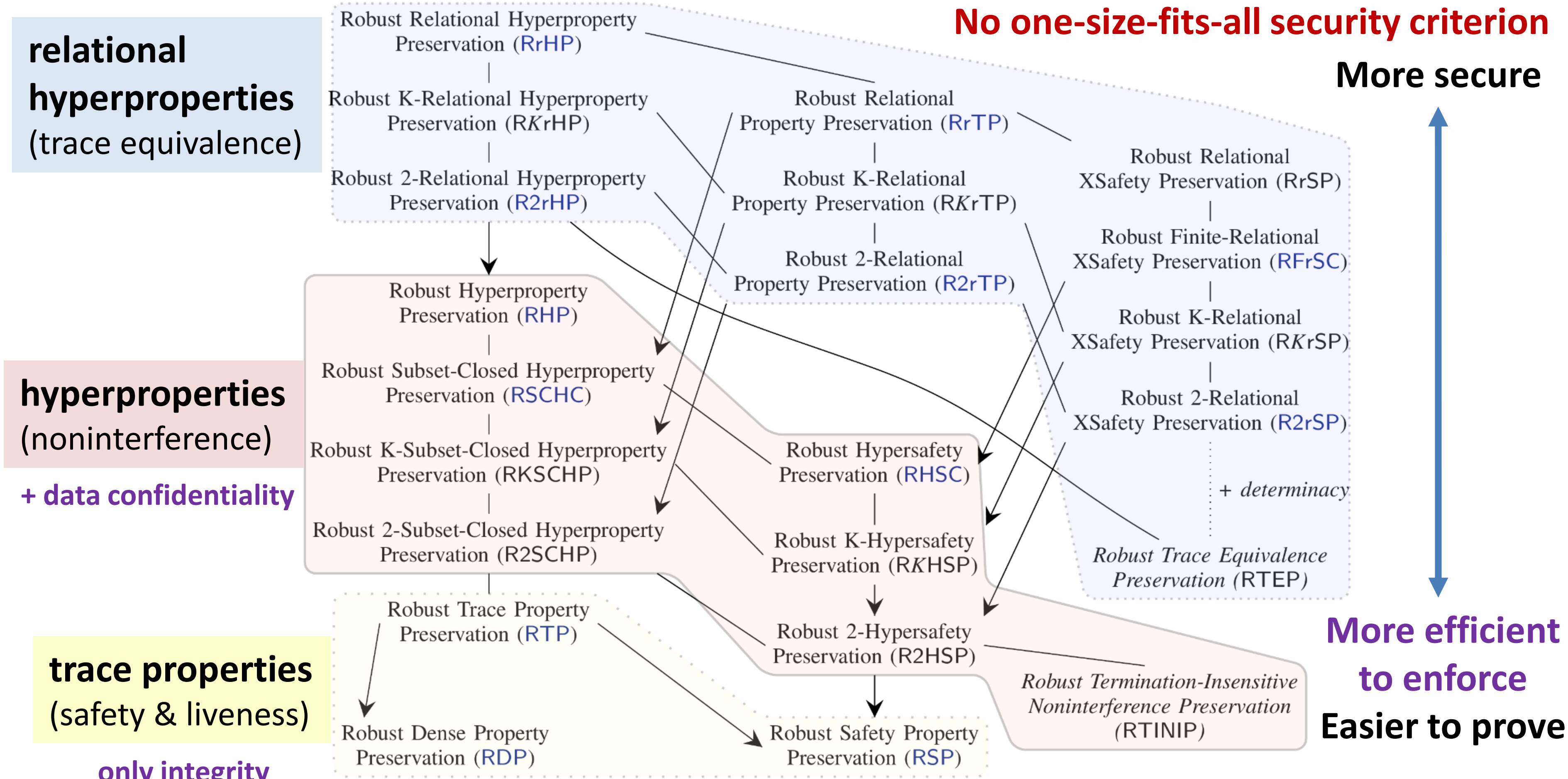
More efficient to enforce

Easier to prove

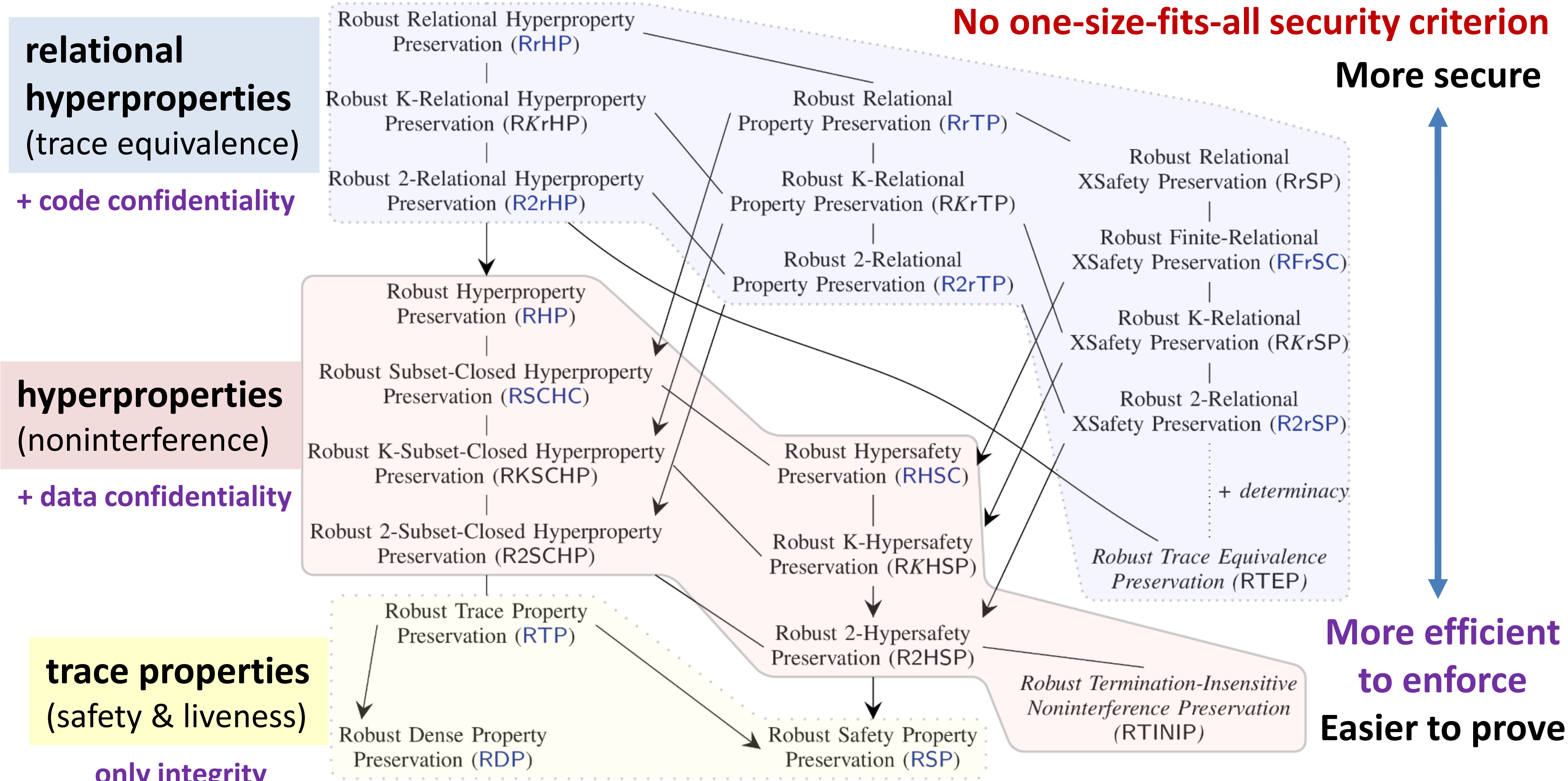
Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]



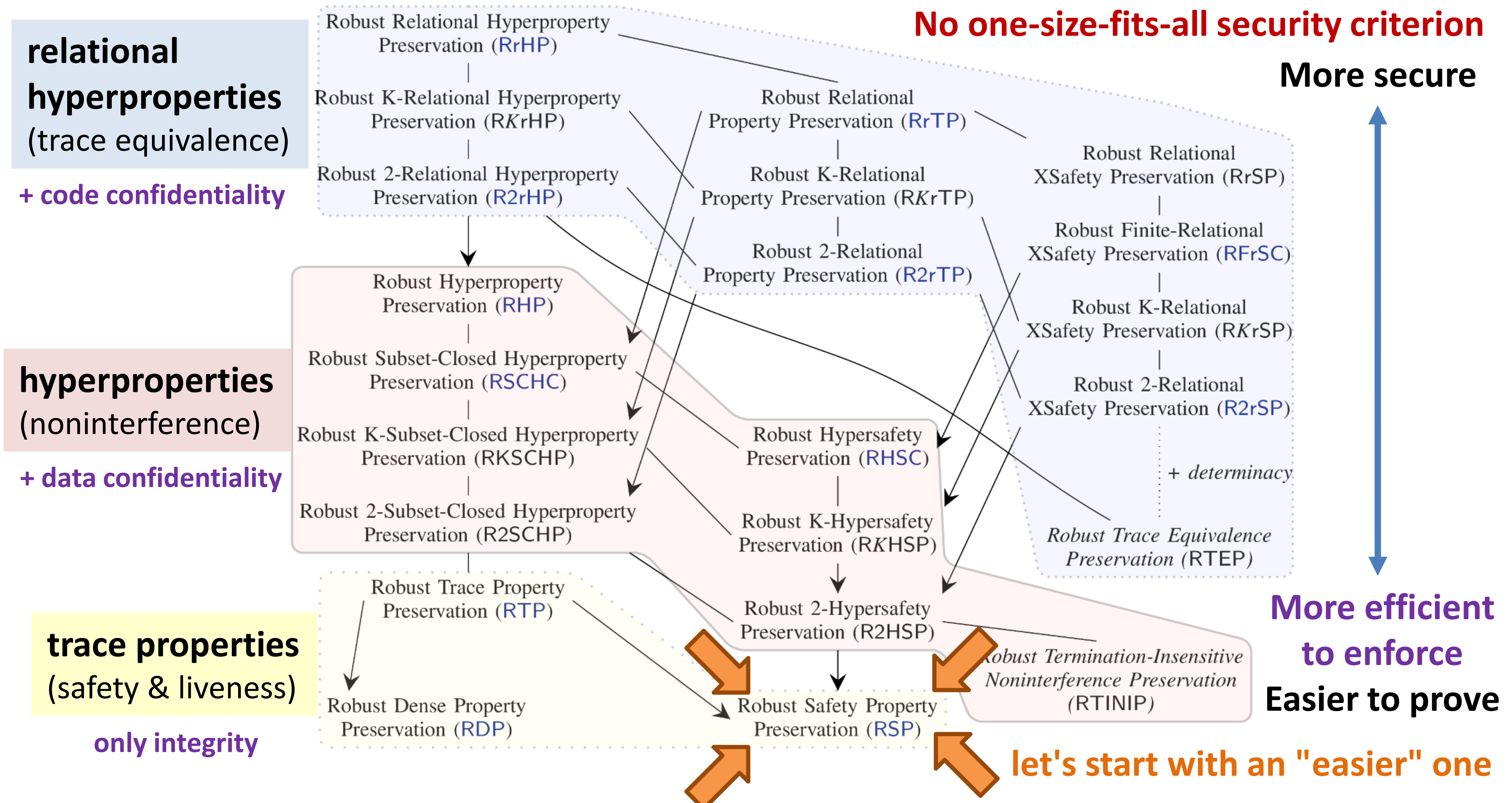
Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]



Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]



Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]



Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

No one-size-fits-all security criterion

More secure

relational hyperproperties (trace equivalence)

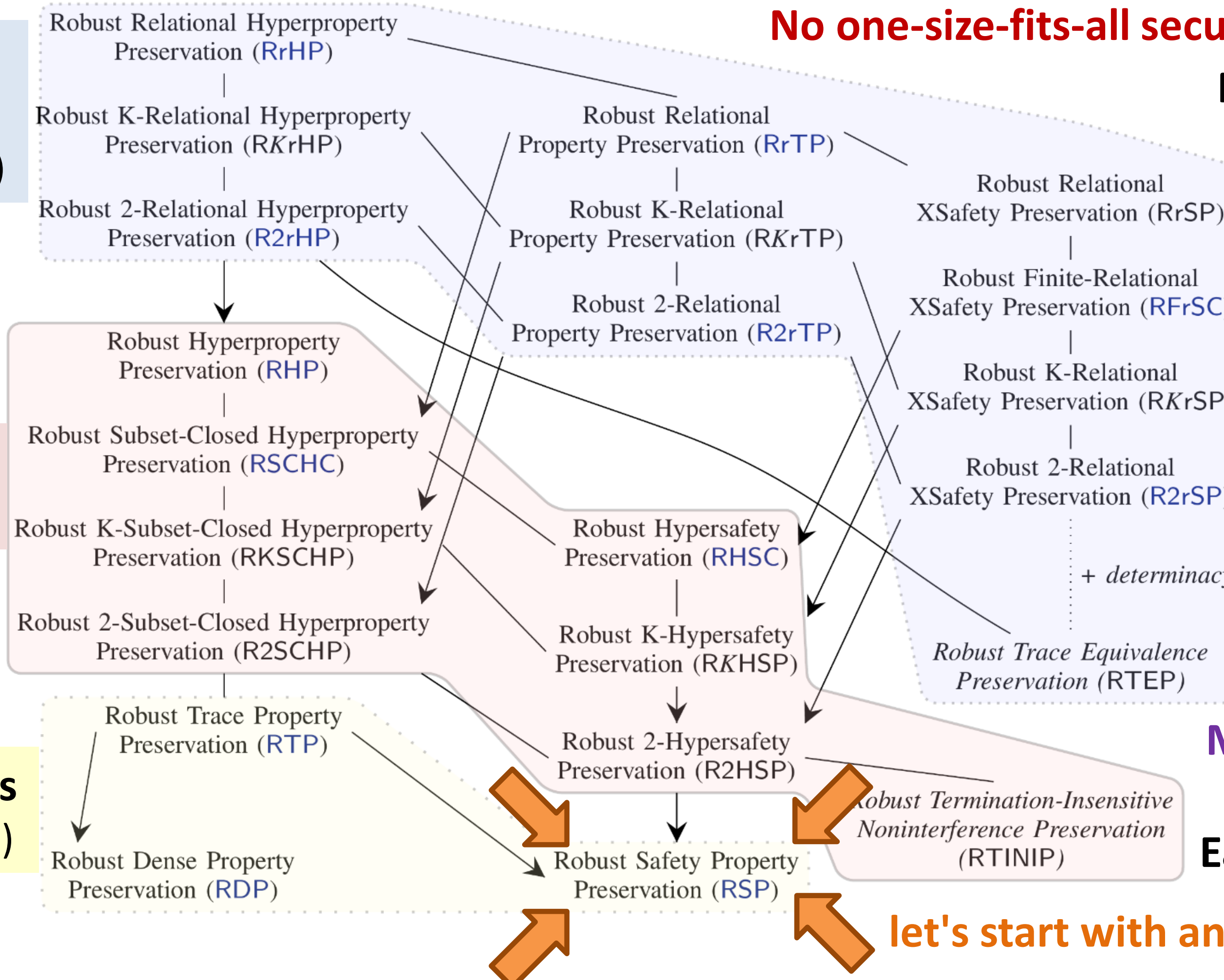
+ code confidentiality

hyperproperties (noninterference)

+ data confidentiality

trace properties (safety & liveness)

only integrity



More efficient to enforce

Easier to prove

let's start with an "easier" one

fine for code without vulnerabilities (e.g. verified) , but ...

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**
  - **every compartment should be protected from all the others**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**
  - **every compartment should be protected from all the others**
- **We don't know when a compartment will be compromised**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

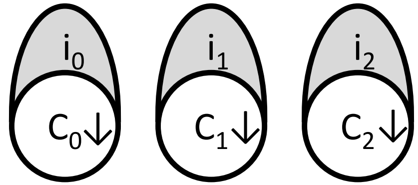
- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**
  - **every compartment should be protected from all the others**
- **We don't know when a compartment will be compromised**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

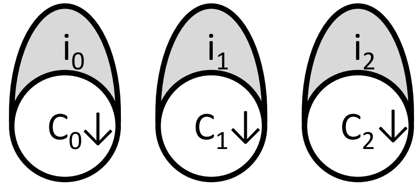
- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**
  - **every compartment should be protected from all the others**
- **We don't know when a compartment will be compromised**

Extra challenges in defining secure compilation for vulnerable C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**
  - **every compartment should be protected from all the others**
- **We don't know when a compartment will be compromised**
  - **every compartment should receive protection until compromised**

Security definition: If 

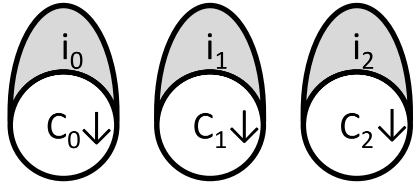
$\rightsquigarrow_{\text{machine}}$ m then

Security definition: If 

$\rightsquigarrow_{\text{machine}}$ m then

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

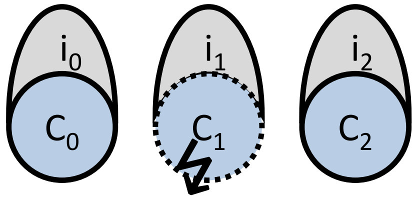
Security definition: If



$\rightsquigarrow_{\text{machine}}$ m then

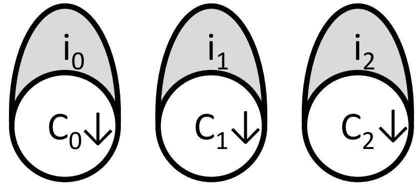
∃ a sequence of compartment compromises explaining the finite trace m in the source language, for instance $m=m_1 \cdot m_2 \cdot m_3$ and

(1)



$\rightsquigarrow_{\text{source}}$ $m_1 \cdot \text{Undef}(C_1)$

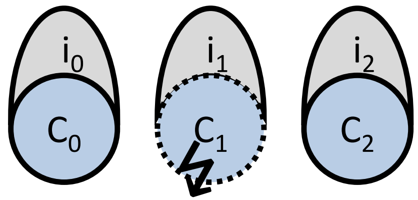
Security definition: If



$\rightsquigarrow_{\text{machine}}$ m then

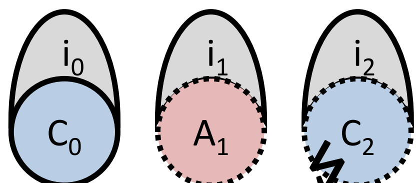
∃ a sequence of compartment compromises explaining the finite trace m in the source language, for instance $m = m_1 \cdot m_2 \cdot m_3$ and

(1)



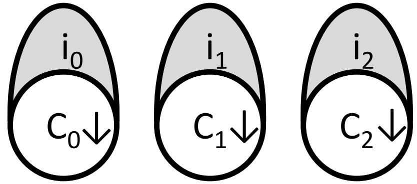
$\rightsquigarrow_{\text{source}}$ $m_1 \cdot \text{Undef}(C_1)$

(2) $\exists A_1$.



$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

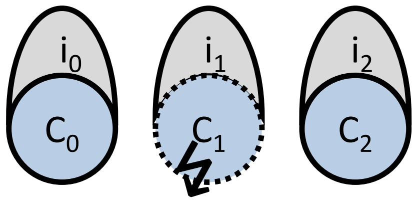
Security definition: If



$\rightsquigarrow_{\text{machine}}$ m then

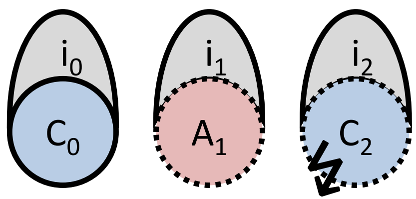
∃ a sequence of compartment compromises explaining the finite trace *m* in the source language, for instance $m = m_1 \cdot m_2 \cdot m_3$ and

(1)



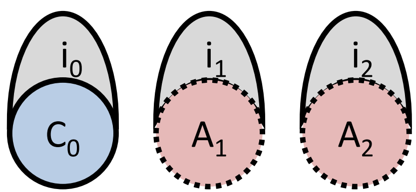
$\rightsquigarrow_{\text{source}}$ $m_1 \cdot \text{Undef}(C_1)$

(2) $\exists A_1$.



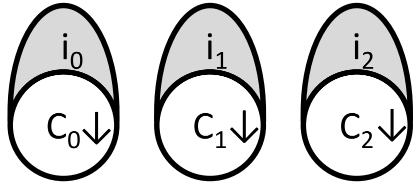
$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

(3) $\exists A_2$.



$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot m_3$

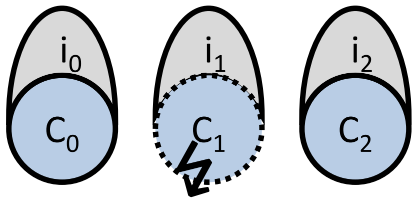
Security definition: If



$\rightsquigarrow_{\text{machine}}$ m then

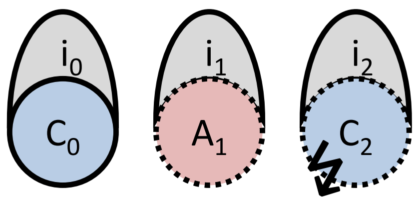
∃ a sequence of compartment compromises explaining the finite trace m in the source language, for instance $m=m_1 \cdot m_2 \cdot m_3$ and

(1)



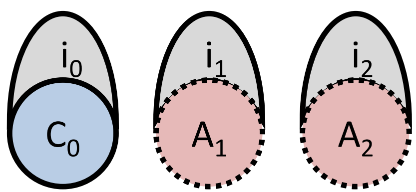
$\rightsquigarrow_{\text{source}}$ $m_1 \cdot \text{Undef}(C_1)$

(2) $\exists A_1.$



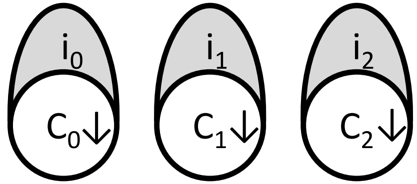
$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

(3) $\exists A_2.$



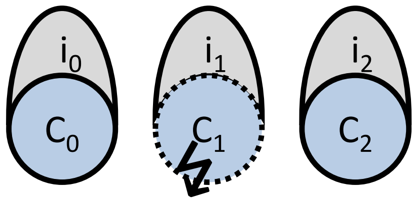
$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot m_3$

Finite trace *m* records which compartment encountered undefined behavior and allows us to rewind execution

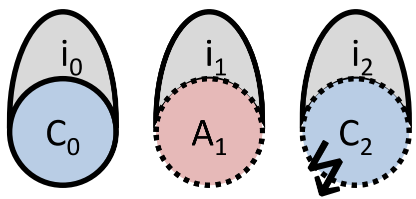
Security definition: If 

$\rightsquigarrow_{\text{machine}}$ m then

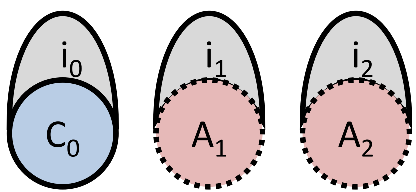
∃ a sequence of compartment compromises explaining the finite trace m in the source language, for instance $m=m_1 \cdot m_2 \cdot m_3$ and

(1) 

$\rightsquigarrow_{\text{source}}$ $m_1 \cdot \text{Undef}(C_1)$

(2) $\exists A_1.$ 

$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

(3) $\exists A_2.$ 

$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot m_3$

Finite trace *m* records which compartment encountered undefined behavior and allows us to rewind execution

We can reduce this to a **variant of robust safety preservation** [CCS'18]

We reduce our security goal to a variant of:

Robust Safety Preservation

We reduce our security goal to a variant of:

Robust Safety Preservation

∀source compartments.

∀π safety property.

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

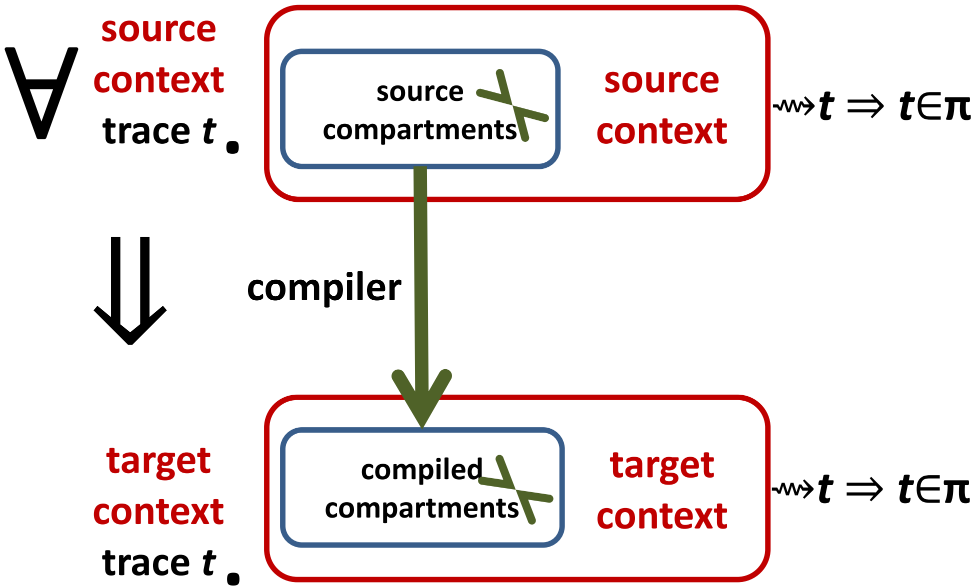
⇓

We reduce our security goal to a variant of:

Robust Safety Preservation

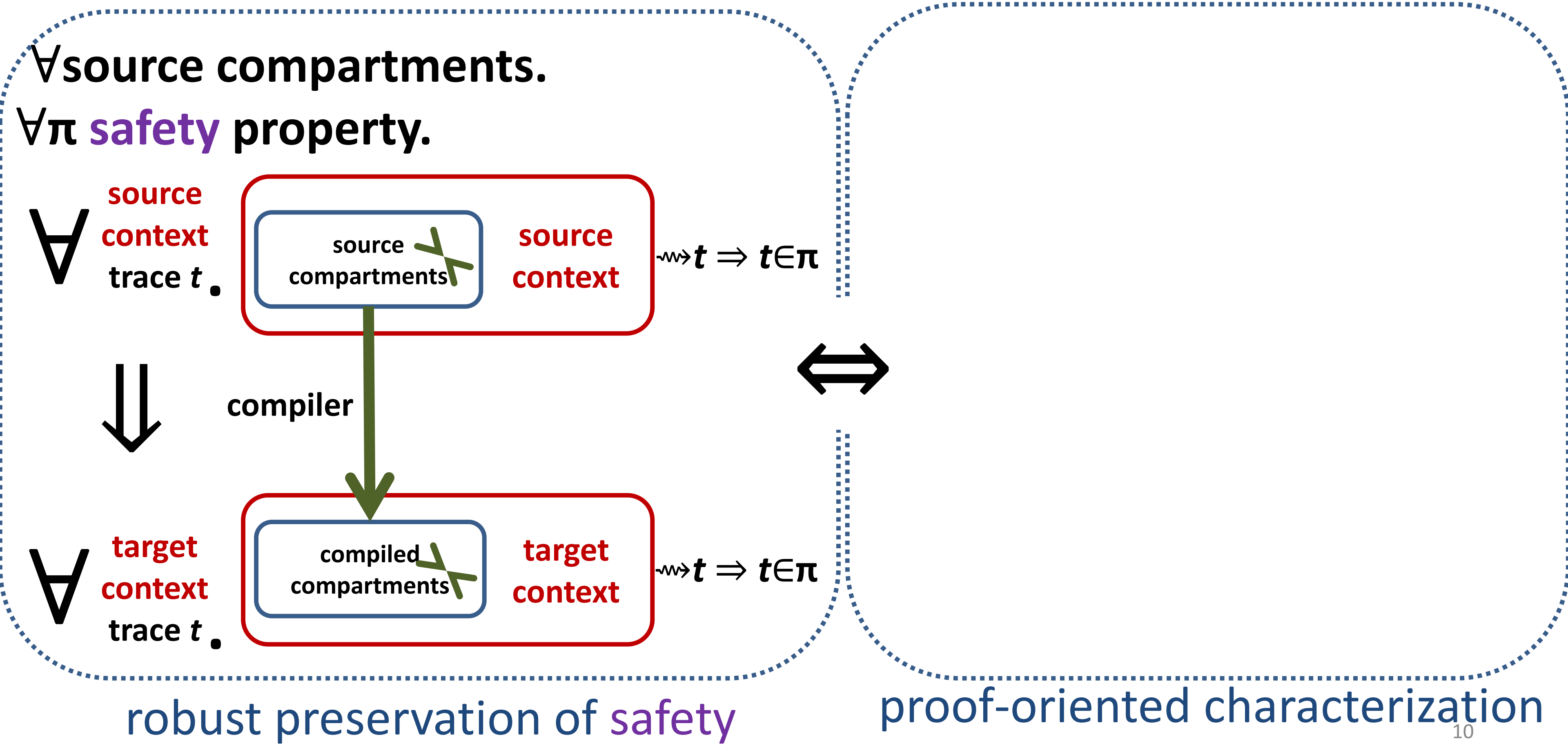
∀source compartments.

∀π safety property.



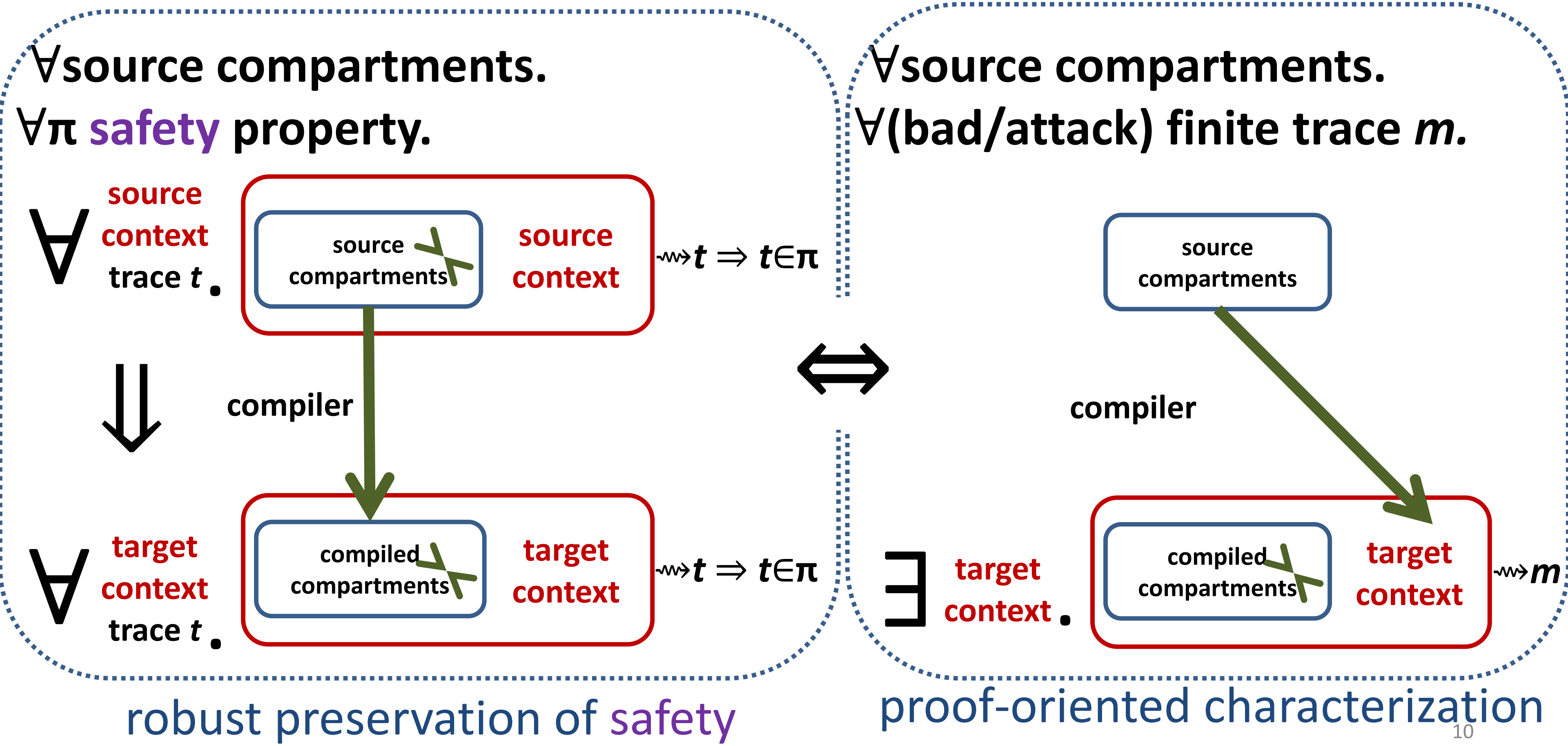
We reduce our security goal to a variant of:

Robust Safety Preservation



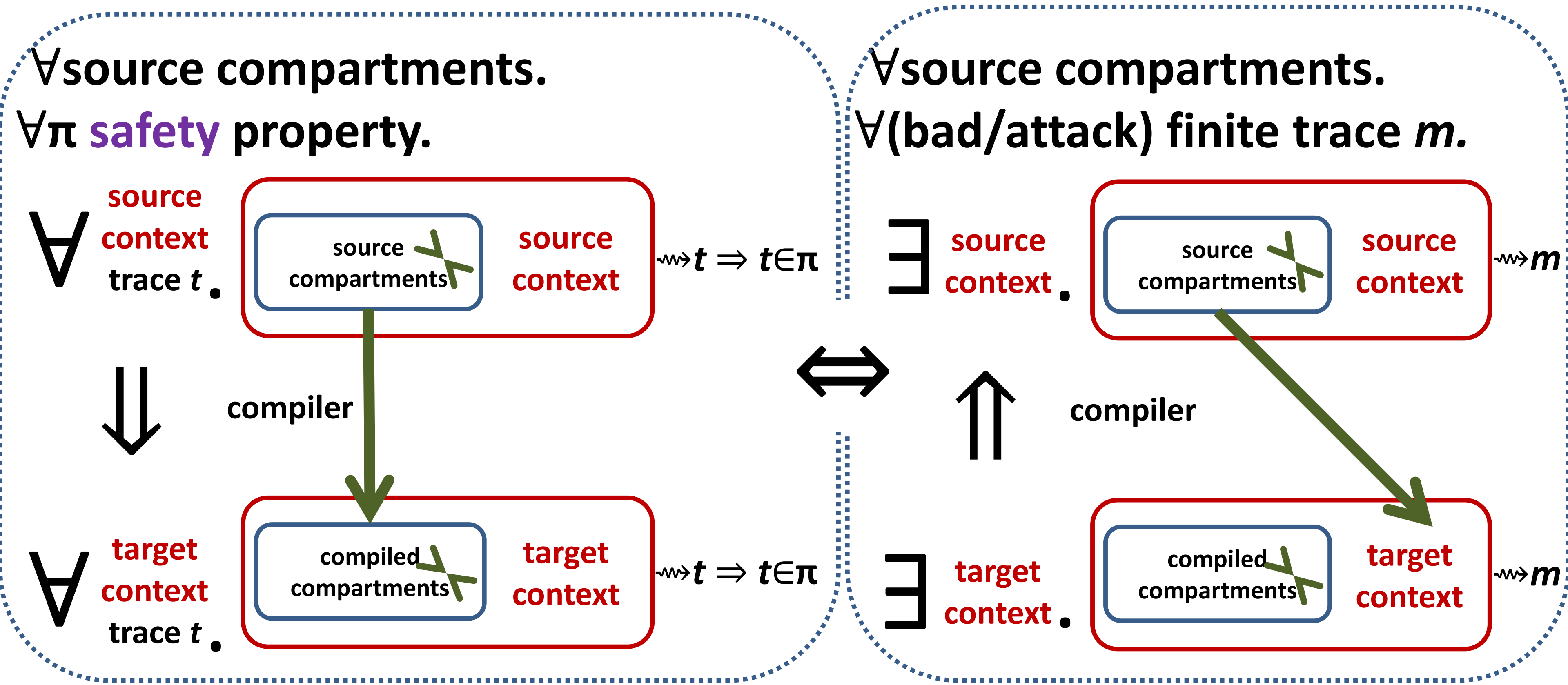
We reduce our security goal to a variant of:

Robust Safety Preservation



We reduce our security goal to a variant of:

Robust Safety Preservation

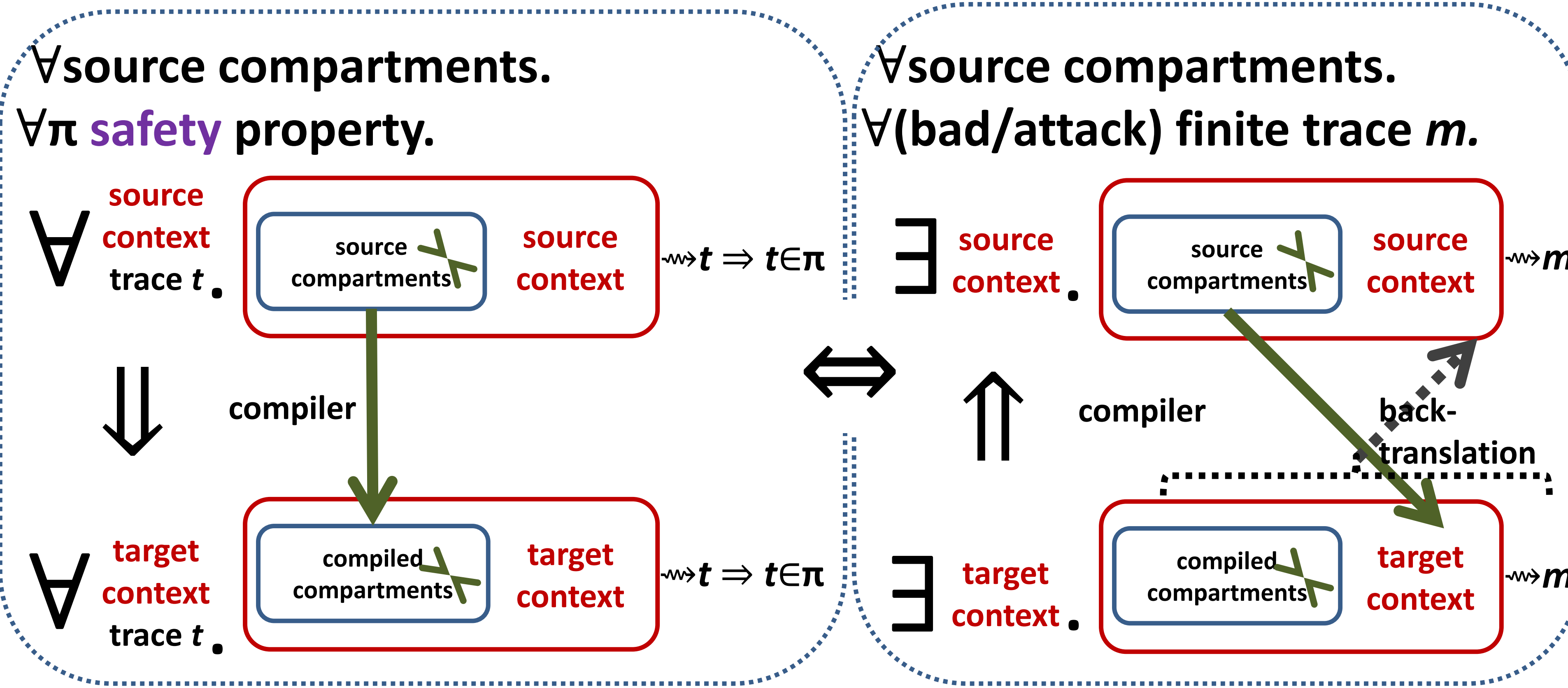


robust preservation of safety

proof-oriented characterization

We reduce our security goal to a variant of:

Robust Safety Preservation



robust preservation of safety

proof-oriented characterization

2. Security Enforcement

CompCert C
with compartments

2. Security Enforcement

CompCert C
with compartments

SECOMP: CompCert extended with secure compartments

2. Security Enforcement

CompCert C with compartments

SECOMP: CompCert extended with secure compartments

CompCert RISC-V ASM with compartments

magically secure semantics

2. Security Enforcement

CompCert C
with compartments

SECOMP: CompCert extended with secure compartments

CompCert RISC-V ASM
with compartments

magically secure semantics

Software-Fault Isolation

vanilla ASM

2. Security Enforcement

CompCert C with compartments

SECOMP: CompCert extended with secure compartments

CompCert RISC-V ASM with compartments

magically secure semantics

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

[POPL'14, S&P'15, ASPLOS'15, POST'18, CCS'18, CSF'23]

Hardware-accelerated enforcement

2. Security Enforcement

CompCert C with compartments

SECOMP: CompCert extended with secure compartments

CompCert RISC-V ASM with compartments

magically secure semantics

Software-Fault Isolation

vanilla ASM

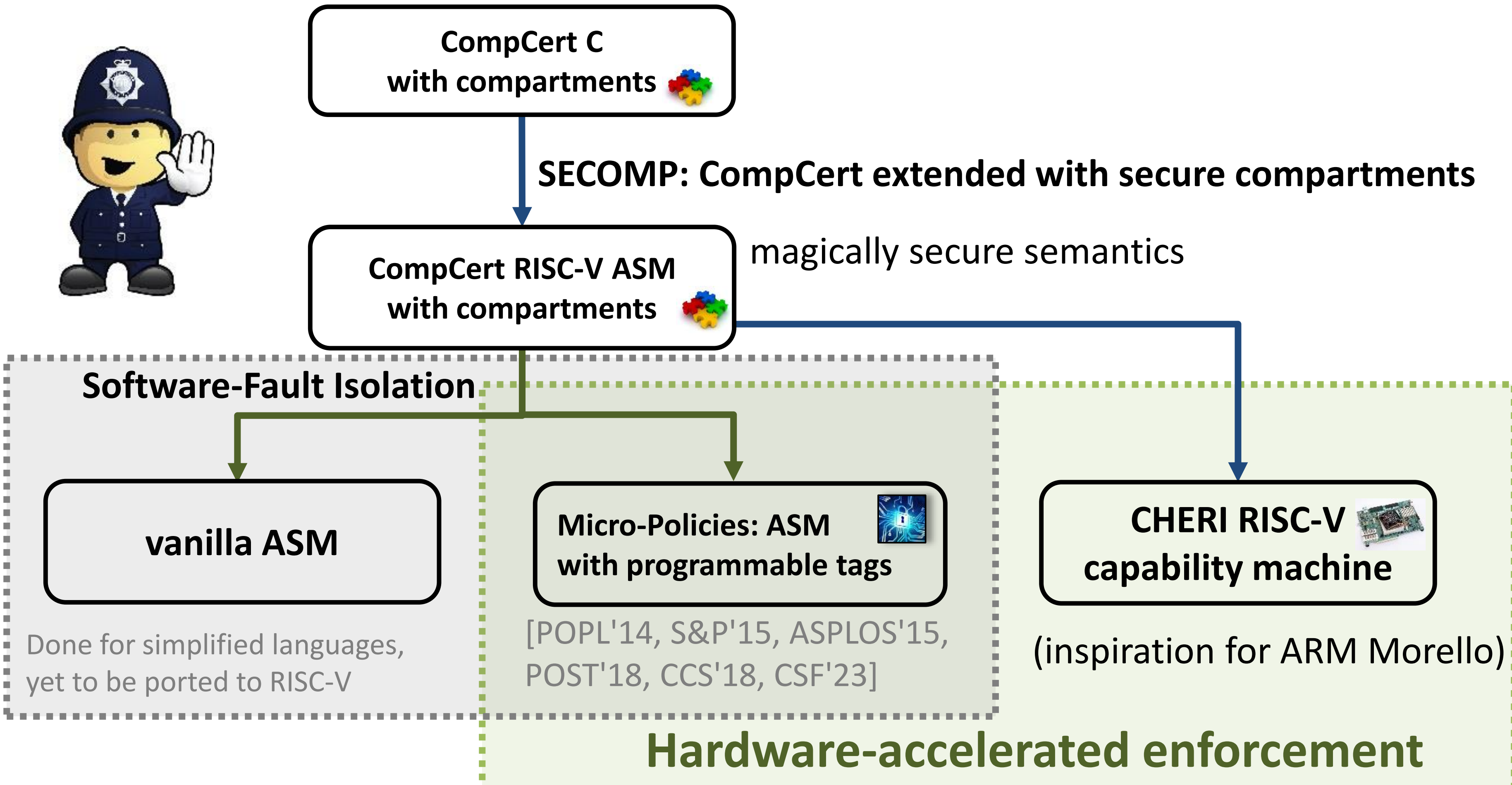
Done for simplified languages, yet to be ported to RISC-V

Micro-Policies: ASM with programmable tags

[POPL'14, S&P'15, ASPLOS'15, POST'18, CCS'18, CSF'23]

Hardware-accelerated enforcement

2. Security Enforcement



CompCert C with Compartments

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - interacting via clearly specified interfaces (simple ones for now)
  - procedure calls and returns, no shared memory (for now)

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - interacting via clearly specified interfaces (simple ones for now)
  - procedure calls and returns, no shared memory (for now)

```
comp_fib exports fib

comp_fib int fib(int n) {

  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - – interacting via clearly specified interfaces (simple ones for now)
  - – procedure calls and returns, no shared memory (for now)

Export →

```
comp_fib exports fib

comp_fib int fib(int n) {


  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - – interacting via clearly specified interfaces (simple ones for now)
  - – procedure calls and returns, no shared memory (for now)

Export →

```
comp_fib exports fib

comp_fib int fib(int n) {


  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

```
comp_main imports comp_fib[fib]
comp_main imports_syscall printf scanf

comp_main int input;

comp_main int main() {
  scanf("%d", &input);
  int r = fib(input);
  printf("fib(%d) = %d\n", n, r);
  return 0;
}
```

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - interacting via clearly specified interfaces (simple ones for now)
  - procedure calls and returns, no shared memory (for now)

Export →

```
comp_fib exports fib

comp_fib int fib(int n) {


  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

← Imports

```
comp_main imports comp_fib[fib]
comp_main imports_syscall printf scanf

comp_main int input;

comp_main int main() {
  scanf("%d", &input);
  int r = fib(input);
  printf("fib(%d) = %d\n", n, r);
  return 0;
}
```

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - interacting via clearly specified interfaces (simple ones for now)
  - procedure calls and returns, no shared memory (for now)

Export

```
comp_fib exports fib

comp_fib int fib(int n) {

  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Imports

```
comp_main imports comp_fib[fib]
comp_main imports_syscall printf scanf

comp_main int input;

comp_main int main() {
  scanf("%d", &input);
  int r = fib(input);
  printf("fib(%d) = %d\n", n, r);
  return 0;
}
```

Calls allowed: respect the interface

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - interacting via clearly specified interfaces (simple ones for now)
  - procedure calls and returns, no shared memory (for now)

Export

Prevented: memory is private

```
comp_fib exports fib

comp_fib int fib(int n) {
  input++;

  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Imports

Calls allowed: respect the interface

```
comp_main imports comp_fib[fib]
comp_main imports_syscall printf scanf

comp_main int input;

comp_main int main() {
  scanf(“%d”, &input);
  int r = fib(input);
  printf("fib(%d) = %d\n", n, r);
  return 0;
}
```

CompCert C with Compartments

- **Various abstractions already there** (e.g. procedures)
- **Added mutually distrustful compartments**
  - interacting via clearly specified interfaces (simple ones for now)
  - procedure calls and returns, no shared memory (for now)

Export

Prevented: does not respect the interface

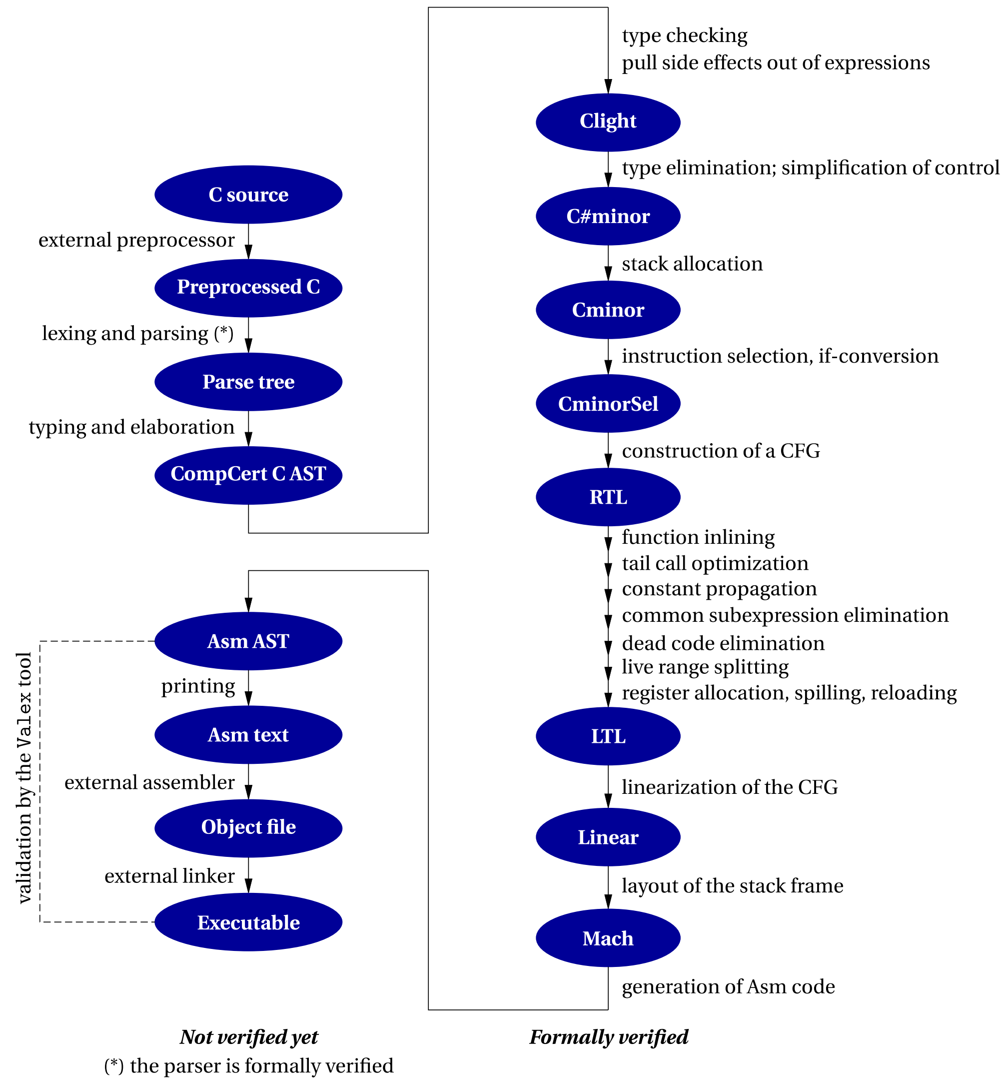
```
comp_fib exports fib

comp_fib int fib(int n) {
  main();
  if (n < 2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Imports

Calls allowed: respect the interface

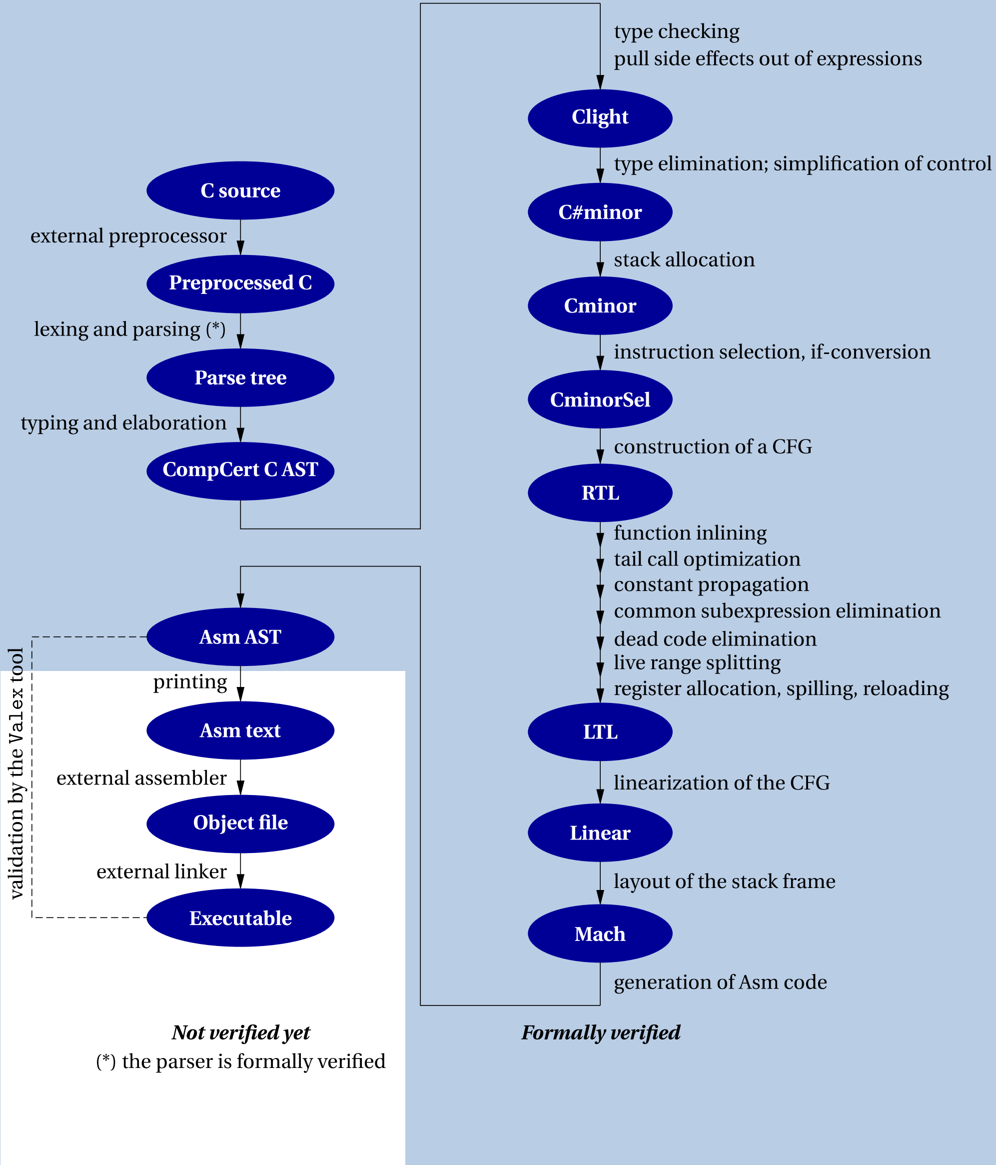
```
comp_main imports comp_fib[fib]
comp_main imports_syscall printf scanf

comp_main int input;

comp_main int main() {
  scanf("%d", &input);
  int r = fib(input);
  printf("fib(%d) = %d\n", n, r);
  return 0;
}
```

CompCert extended with compartments



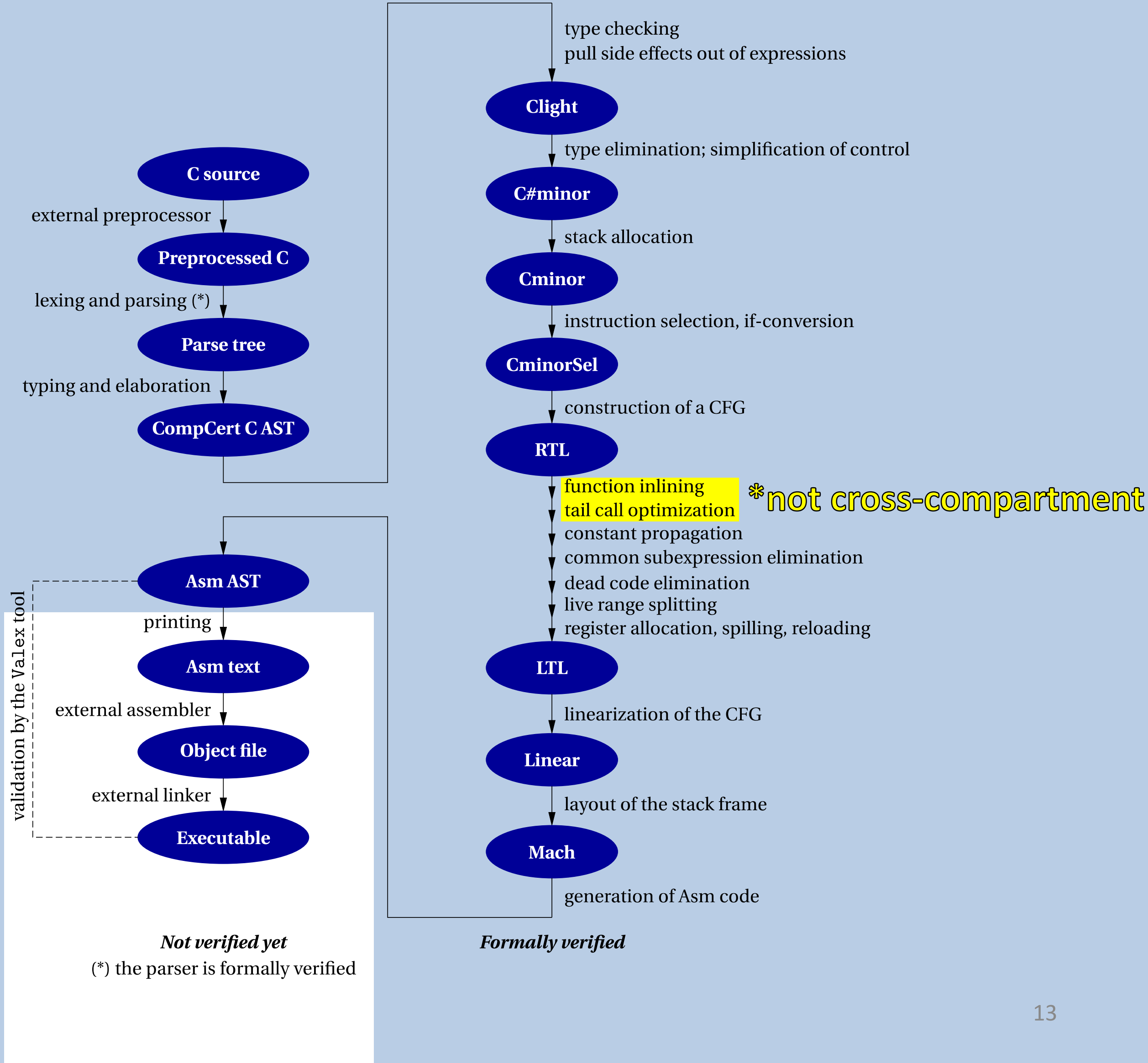
CompCert extended with compartments

all 19 verified compilation passes* from Clight to RISC-V ASM (magically secure semantics)



CompCert extended with compartments

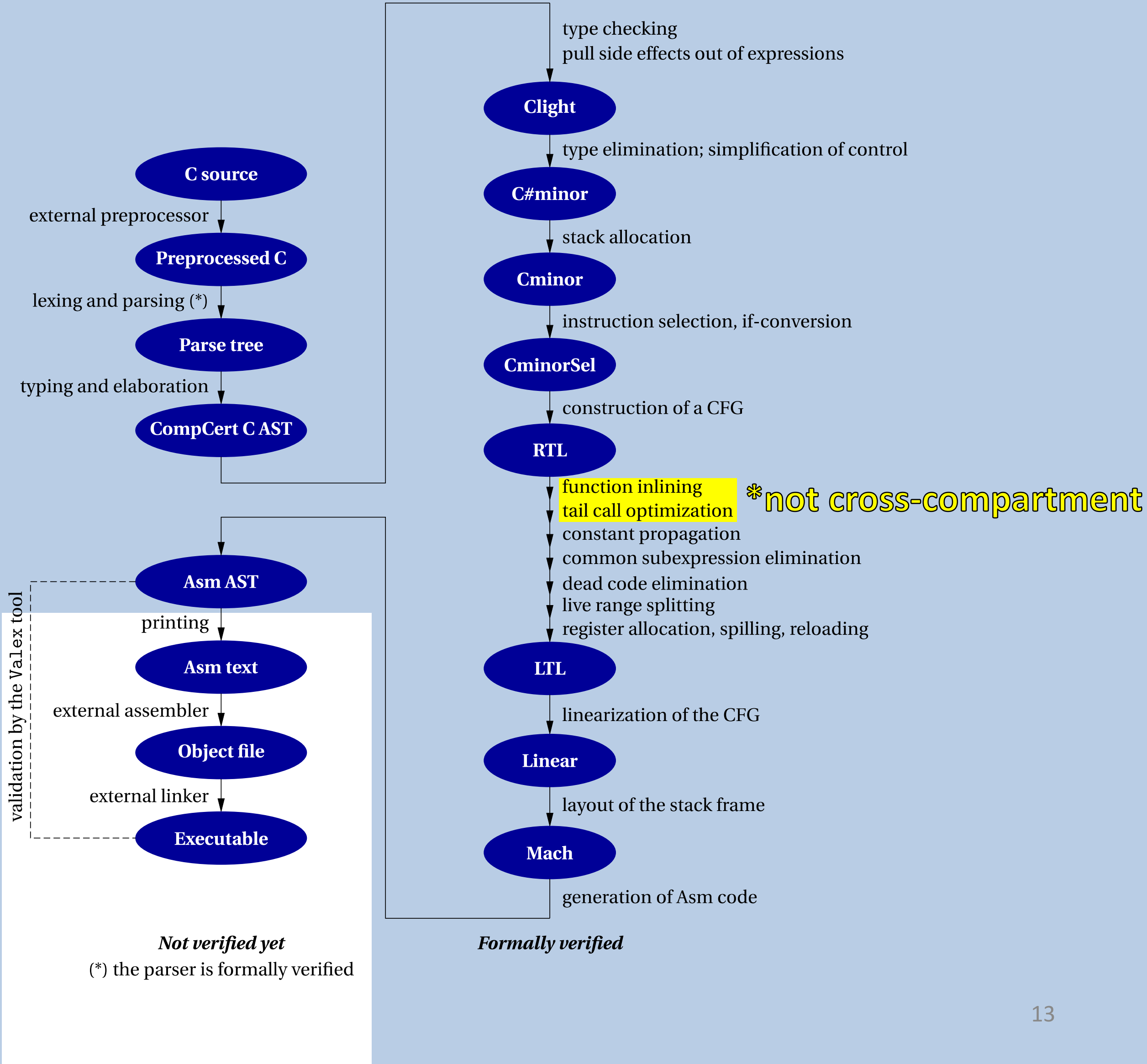
all 19 verified compilation passes*
from Clight to RISC-V ASM
(magically secure semantics)



CompCert extended with compartments

all 19 verified compilation passes*
from Clight to RISC-V ASM
(magically secure semantics)

extended compiler correctness
12+ KLoC, only 9.4% change
reused for security



CompCert RISC-V with Compartments

- **Added compartments with interfaces** (like for all languages)

CompCert RISC-V with Compartments

- **Added compartments with interfaces** (like for all languages)
- **New shadow stack**
  - ensures well-bracketedness of cross-compartment control flow

CompCert RISC-V with Compartments

- **Added compartments with interfaces** (like for all languages)
- **New shadow stack**
  - ensures well-bracketedness of cross-compartment control flow
- **Need to protect stack-spilled call arguments**
  - so that malicious caller cannot exploit callbacks
    to covertly change arguments of a previous call
  - discovered during one of security proof steps (recomposition)

CompCert RISC-V with Compartments

- **Added compartments with interfaces** (like for all languages)
- **New shadow stack**
  - ensures well-bracketedness of cross-compartment control flow
- **Need to protect stack-spilled call arguments**
  - so that malicious caller cannot exploit callbacks
    to covertly change arguments of a previous call
  - discovered during one of security proof steps (recomposition)
- **Abstract machine with magically secure semantics**
  - independent of actual enforcement (lower-level backends)

Capabilities Backend

- **Targeting the CHERI RISC-V capability machine**

Capabilities Backend

- **Targeting the CHERI RISC-V capability machine**
- **Secure and efficient calling convention enforcing stack safety**
  [Aïna Linn Georges et al, Le temps de cerises, OOPSLA 2022]

Capabilities Backend

- **Targeting the CHERI RISC-V capability machine**
- **Secure and efficient calling convention enforcing stack safety**
  [Aïna Linn Georges et al, Le temps de cerises, OOPSLA 2022]
  - **Uninitialized capabilities**: cannot read memory before initializing
  - **Directed capabilities**: cannot access old stack frames

Capabilities Backend

- **Targeting the CHERI RISC-V capability machine**
- **Secure and efficient calling convention enforcing stack safety**
  [Aïna Linn Georges et al, Le temps de cerises, OOPSLA 2022]
  - **Uninitialized capabilities**: cannot read memory before initializing
  - **Directed capabilities**: cannot access old stack frames
- Mutual distrustful compartments: **capability-protected wrappers**
  - on calls and returns clear registers and
    prevent passing capabilities between compartments

Capabilities Backend

- **Targeting the CHERI RISC-V capability machine**
- **Secure and efficient calling convention enforcing stack safety**
  [Aïna Linn Georges et al, Le temps de cerises, OOPSLA 2022]
  - **Uninitialized capabilities**: cannot read memory before initializing
  - **Directed capabilities**: cannot access old stack frames
- Mutual distrustful compartments: **capability-protected wrappers**
  - on calls and returns clear registers and
    prevent passing capabilities between compartments
- Also investigating **calling convention based solely on wrappers**
  - no new kind of capability over what CHERI already provides
  - but more interesting stack layout (not a single contiguous block)

3. Security Proof

3. Security Proof

3. Security Proof

Proving that our compilation chain
for C compartments achieves secure compilation

3. Security Proof

Proving that our compilation chain for C compartments achieves secure compilation

- such proofs generally **very difficult and tedious**
  - wrong full abstraction conjecture survived for decades
  - 250 pages of proof on paper even for toy compilers

3. Security Proof

Proving that our compilation chain for C compartments achieves secure compilation

- such proofs generally **very difficult and tedious**
  - wrong full abstraction conjecture survived for decades
  - 250 pages of proof on paper even for toy compilers
- we propose a **more scalable proof technique**

3. Security Proof

Proving that our compilation chain for C compartments achieves secure compilation

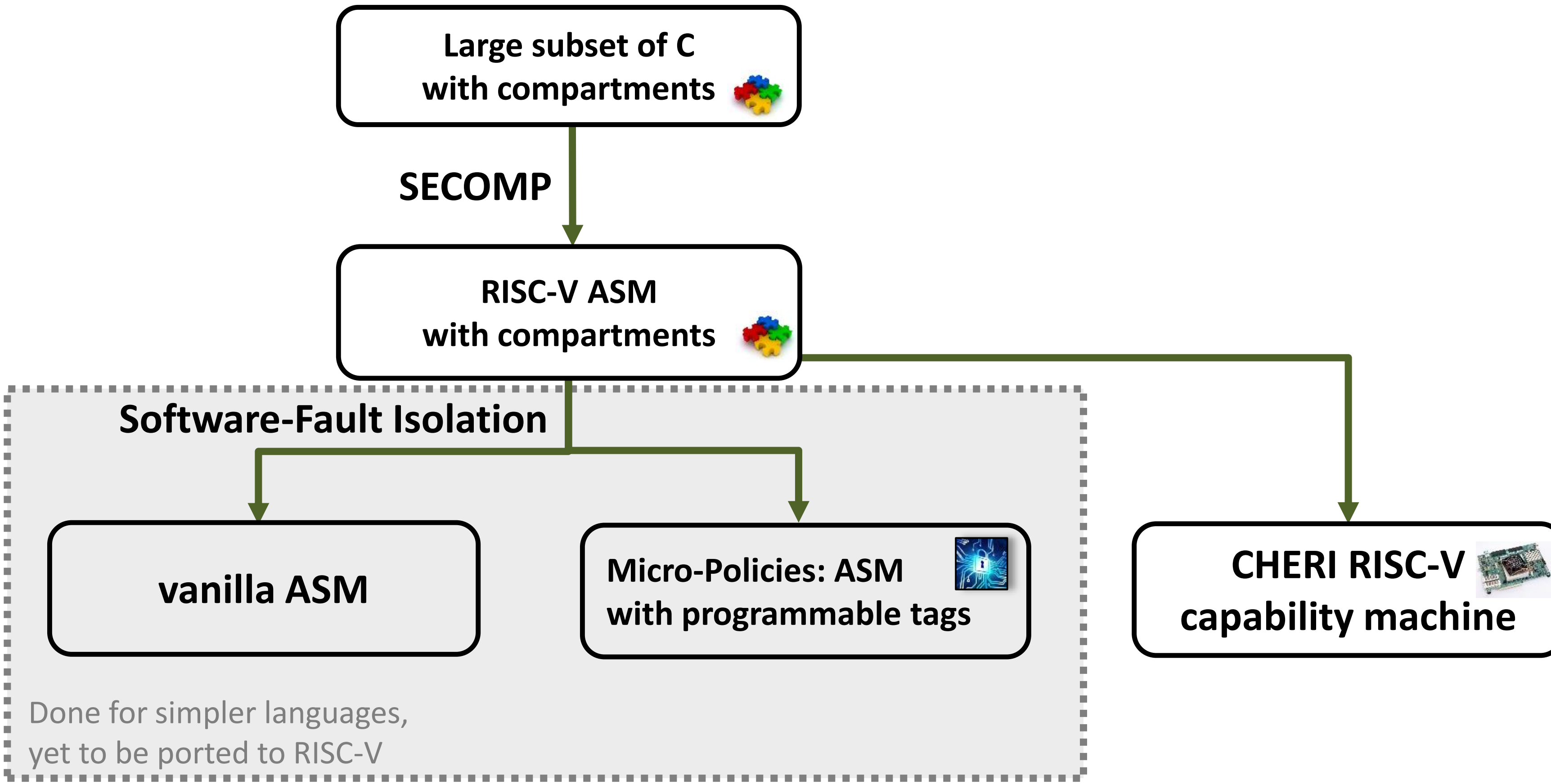
- such proofs generally **very difficult and tedious**
  - wrong full abstraction conjecture survived for decades
  - 250 pages of proof on paper even for toy compilers
- we propose a **more scalable proof technique**
- we focus on **machine-checked proofs** in the Coq proof assistant

3. Security Proof

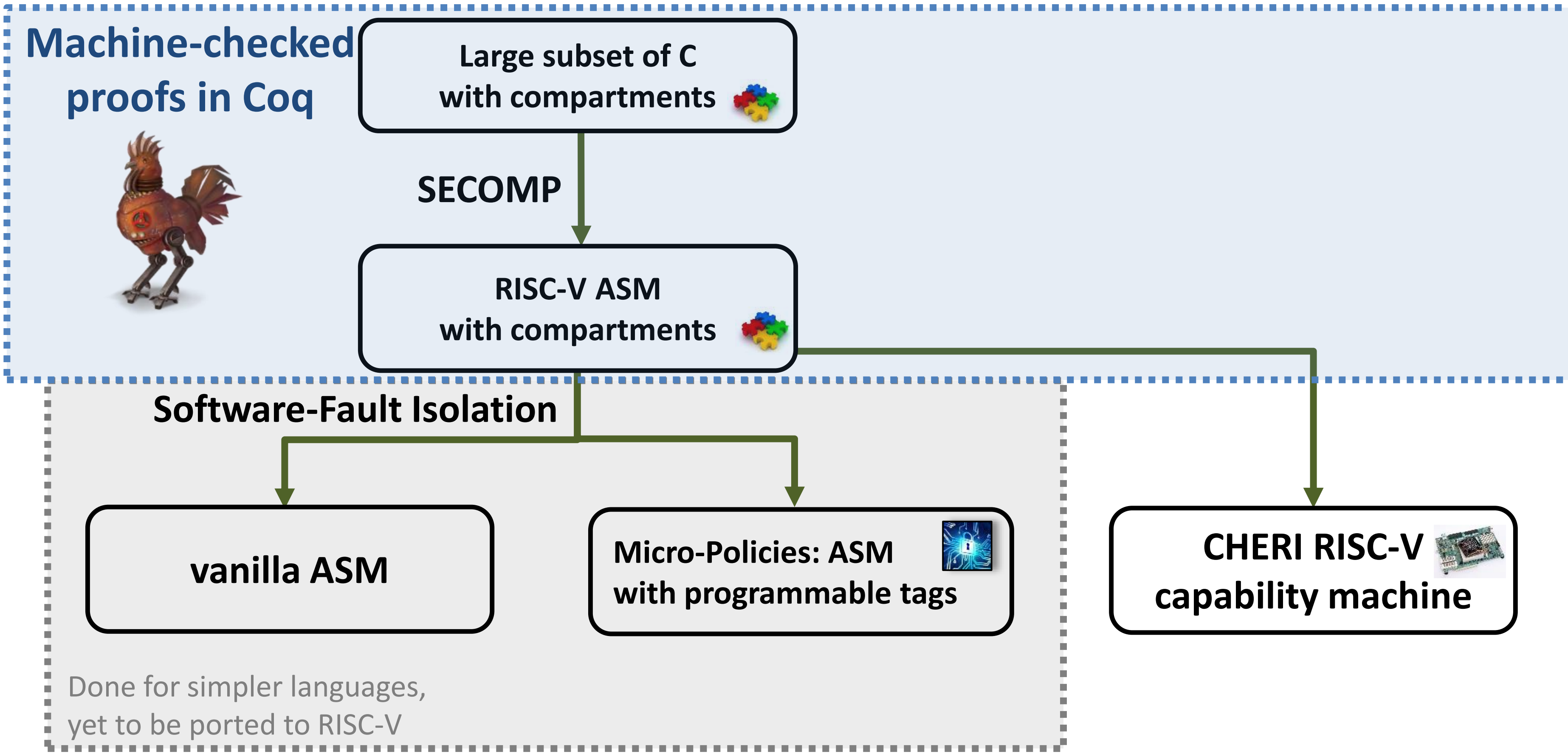
Proving that our compilation chain for C compartments achieves secure compilation

- such proofs generally **very difficult and tedious**
  - wrong full abstraction conjecture survived for decades
  - 250 pages of proof on paper even for toy compilers
- we propose a **more scalable proof technique**
- we focus on **machine-checked proofs** in the Coq proof assistant
  - with **property-based testing** stopgap [POPL'17, ICFP'13, ITP'15, JFP'16]
    - to find wrong conjectures early
    - to deal with the parts we couldn't (yet) verify

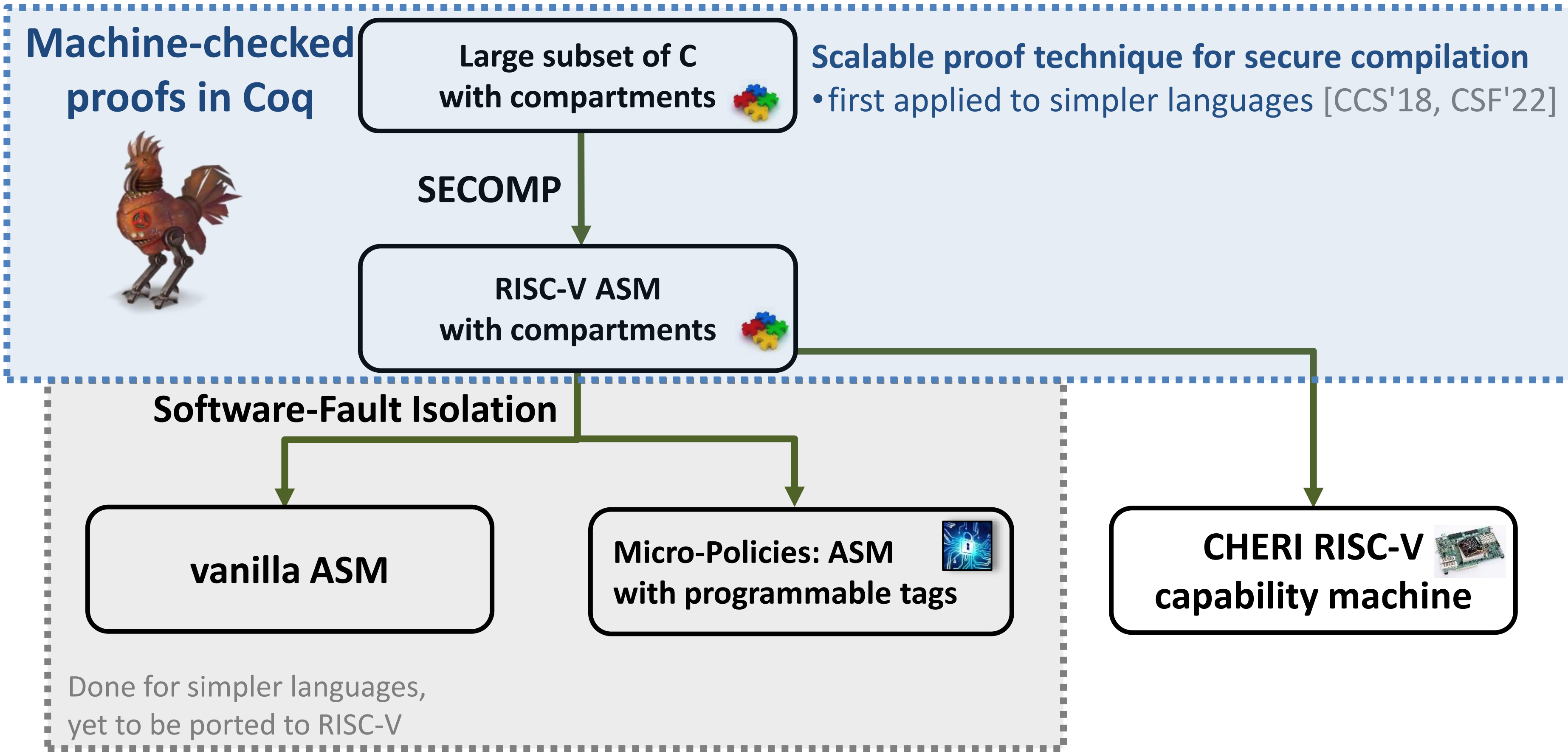
Secure Compilation Proofs in Coq



Secure Compilation Proofs in Coq



Secure Compilation Proofs in Coq



Secure Compilation Proofs in Coq

Machine-checked proofs in Coq

Large subset of C with compartments

SECOMP

RISC-V ASM with compartments

Scalable proof technique for secure compilation
- first applied to simpler languages [CCS'18, CSF'22]
- then scaled up to C compartments [CCS'24]
  - this reuses extended CompCert correctness proof
  - verified strong full-abstraction-like property (~38K LoC)

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

CHERI RISC-V capability machine

Done for simpler languages, yet to be ported to RISC-V

Secure Compilation Proofs in Coq

Machine-checked proofs in Coq

Large subset of C with compartments

SECOMP

RISC-V ASM with compartments

Scalable proof technique for secure compilation

- first applied to simpler languages [CCS'18, CSF'22]
- then scaled up to C compartments [CCS'24]
  - this reuses extended CompCert correctness proof
  - verified strong full-abstraction-like property (~38K LoC)
- milestone in terms of realism!

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

CHERI RISC-V capability machine

Done for simpler languages, yet to be ported to RISC-V

Secure Compilation Proofs in Coq

Machine-checked proofs in Coq

Large subset of C with compartments

SECOMP

RISC-V ASM with compartments

Scalable proof technique for secure compilation

- first applied to simpler languages [CCS'18, CSF'22]
- then scaled up to C compartments [CCS'24]
  - this reuses extended CompCert correctness proof
  - verified strong full-abstraction-like property (~38K LoC)
- milestone in terms of realism!
  - optimizing C compiler with 19 passes

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

CHERI RISC-V capability machine

Done for simpler languages, yet to be ported to RISC-V

Secure Compilation Proofs in Coq

Machine-checked proofs in Coq

Large subset of C with compartments

SECOMP

RISC-V ASM with compartments

Scalable proof technique for secure compilation

- first applied to simpler languages [CCS'18, CSF'22]
- then scaled up to C compartments [CCS'24]
  - this reuses extended CompCert correctness proof
  - verified strong full-abstraction-like property (~38K LoC)
- milestone in terms of realism!
  - optimizing C compiler with 19 passes

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

CHERI RISC-V capability machine

Done for simpler languages, yet to be ported to RISC-V

Quick Chick

Systematic testing

Secure Compilation Proofs in Coq

Machine-checked proofs in Coq

Large subset of C with compartments

SECOMP

RISC-V ASM with compartments

Scalable proof technique for secure compilation

- first applied to simpler languages [CCS'18, CSF'22]
- then scaled up to C compartments [CCS'24]
  - this reuses extended CompCert correctness proof
  - verified strong full-abstraction-like property (~38K LoC)
- milestone in terms of realism!
  - optimizing C compiler with 19 passes

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

Done for simpler languages, yet to be ported to RISC-V

CHERI RISC-V capability machine

Big verification challenge for the future

Systematic testing

More scalable proof technique

for our variant of Robust Safety Preservation [CCS'18,CSF'22]

More scalable proof technique

for our variant of Robust Safety Preservation [CCS'18,CSF'22]

back-translating **finite execution prefix** to **whole source program**

More scalable proof technique

for our variant of Robust Safety Preservation [CCS'18,CSF'22]

back-translating **finite execution prefix** to **whole source program**

compiler correctness (extended from CompCert and reused)

More scalable proof technique

for our variant of Robust Safety Preservation [CCS'18,CSF'22]

back-translating **finite execution prefix** to **whole source program**

compiler correctness (extended from CompCert and reused)

recomposition and *blame* steps also simulation proofs

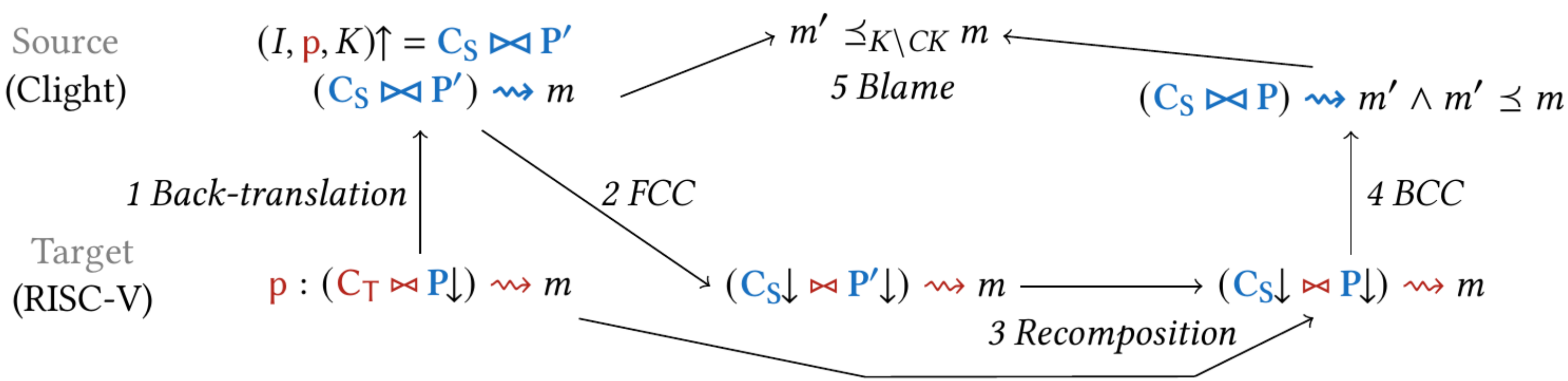
More scalable proof technique

for our variant of Robust Safety Preservation [CCS'18,CSF'22]

back-translating **finite execution prefix** to **whole source program**

compiler correctness (extended from CompCert and reused)

recomposition and *blame* steps also simulation proofs



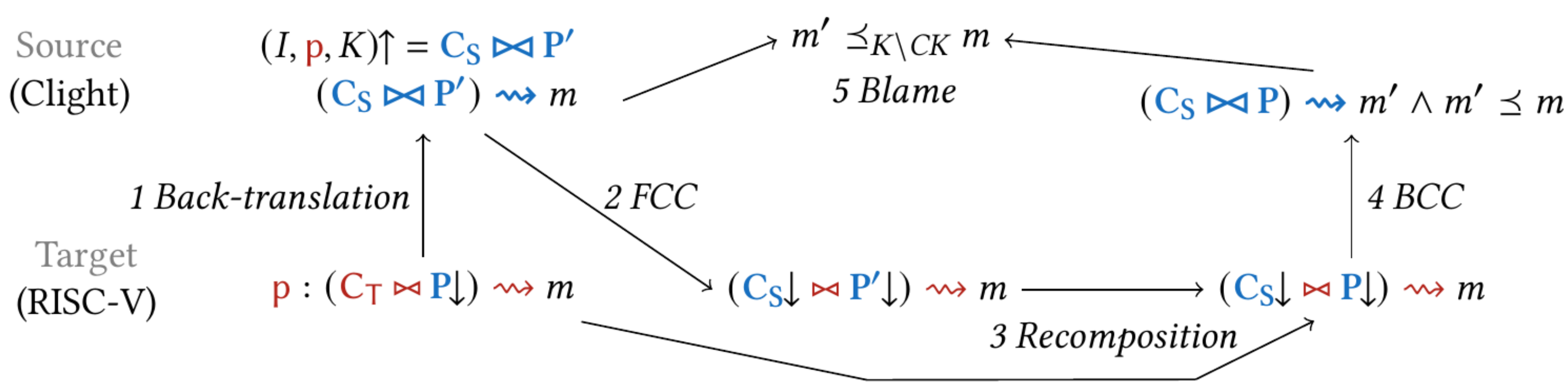
More scalable proof technique

for our variant of Robust Safety Preservation [CCS'18,CSF'22]

back-translating **finite execution prefix** to **whole source program**

compiler correctness (extended from CompCert and reused)

recomposition and *blame* steps also simulation proofs

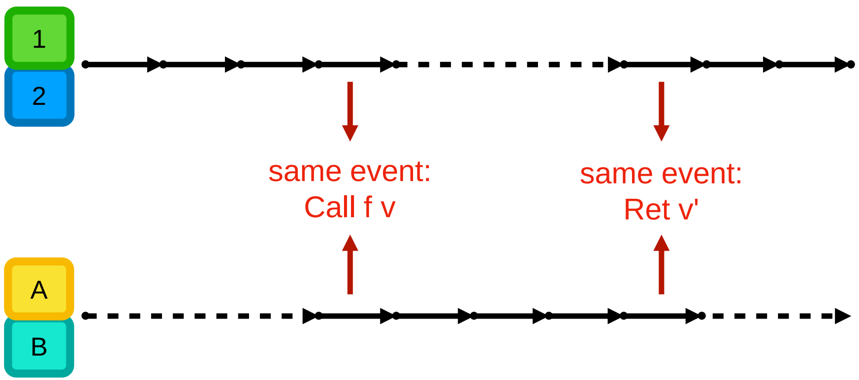


Challenging proof engineering for scaling this to CompCert [CCS'24]

Recomposition for SECOMP RISC-V

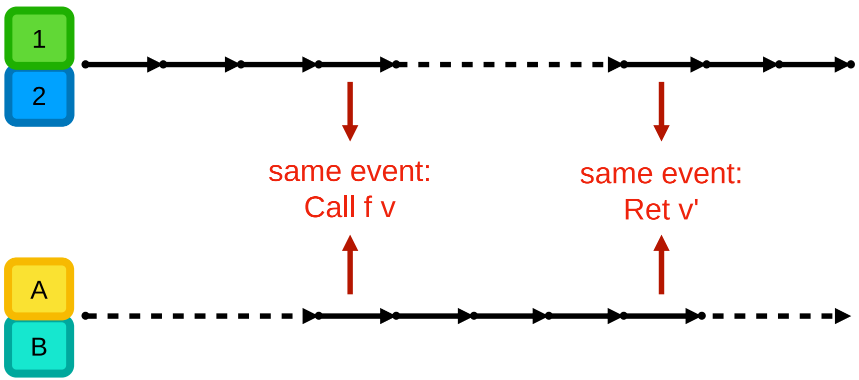
Recomposition for SECOMP RISC-V

From two synchronized RISC-V executions

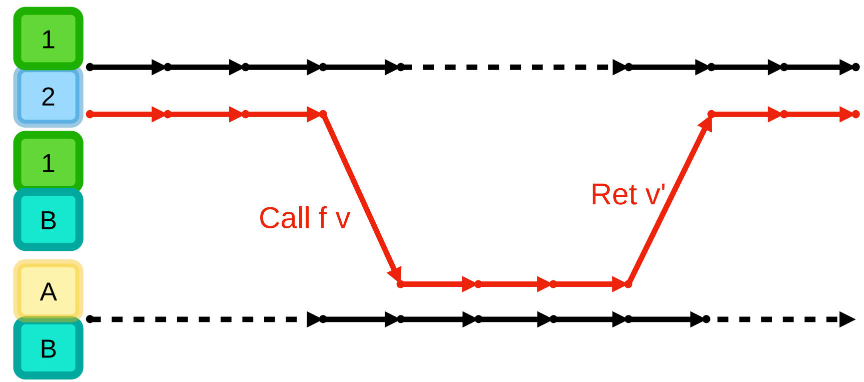


Recomposition for SECOMP RISC-V

From two synchronized RISC-V executions

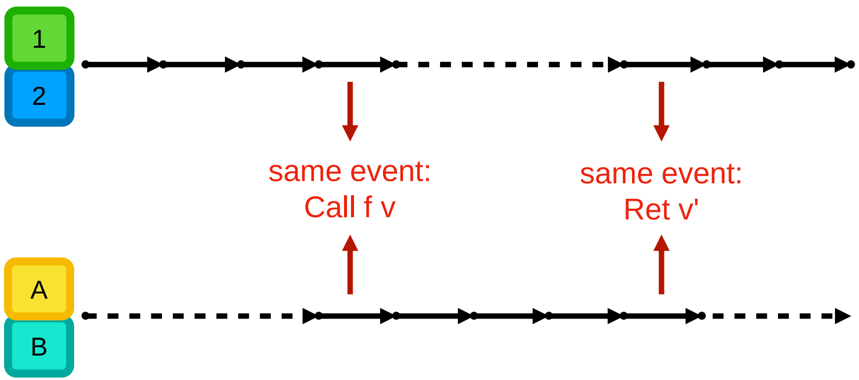


Obtain a “recomposed” execution:

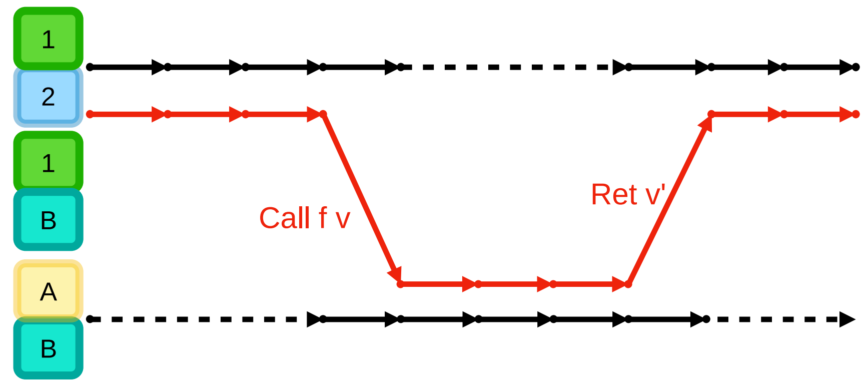


Recomposition for SECOMP RISC-V

From two synchronized RISC-V executions



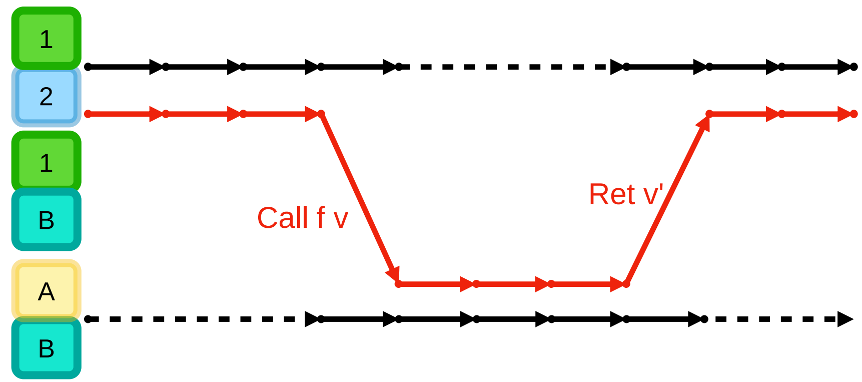
Obtain a “recomposed” execution:



Challenging 3-way simulation proof with subtle invariants

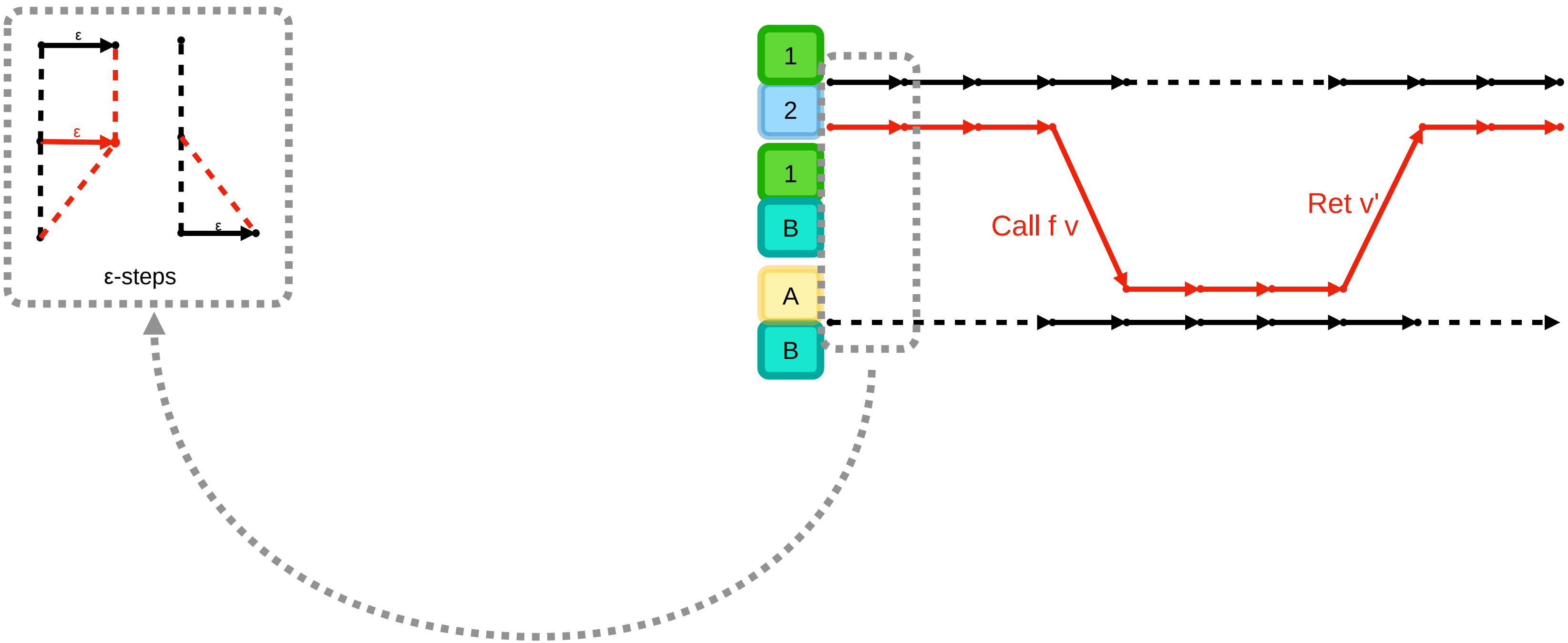
Generic diagrams for recomposition

8 generic 3-way simulation diagrams for proving recomposition



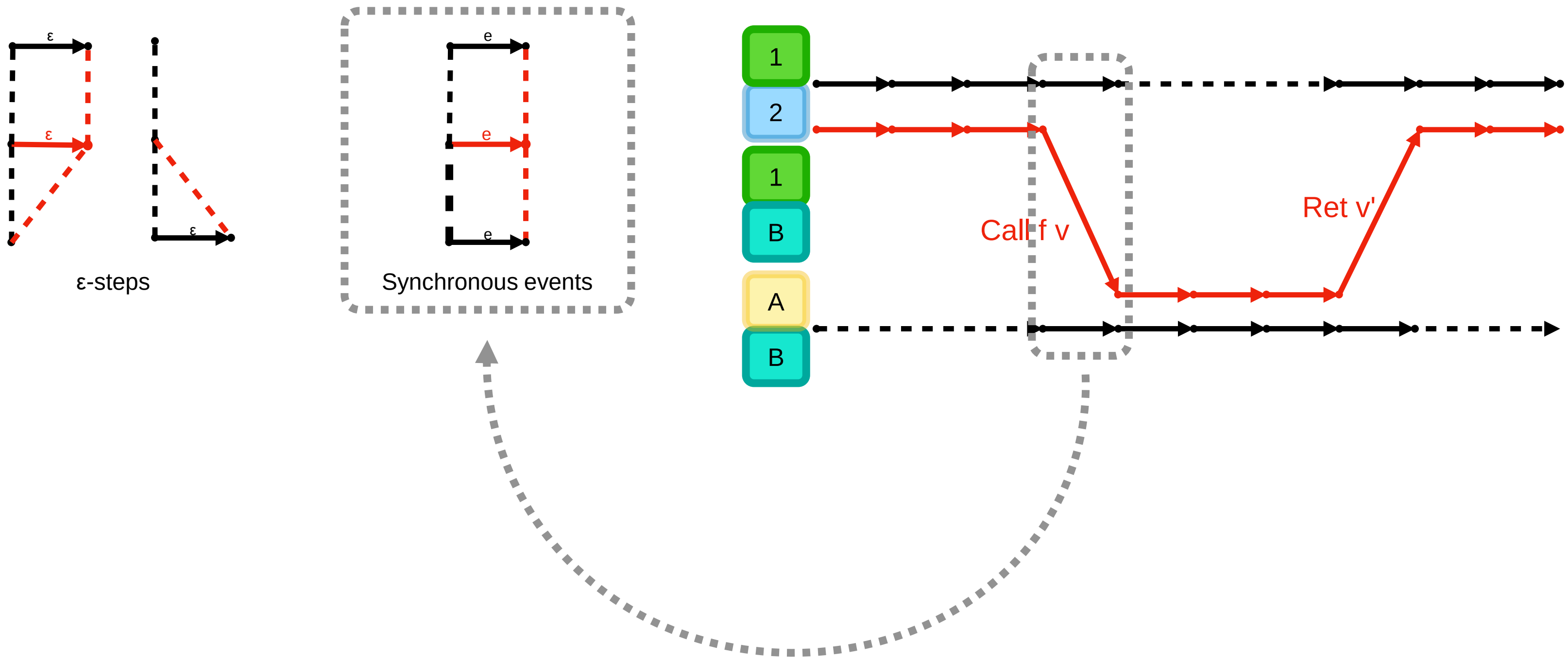
Generic diagrams for recomposition

8 generic 3-way simulation diagrams for proving recomposition



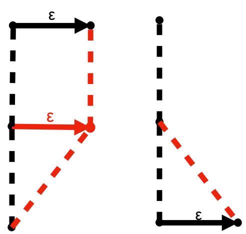
Generic diagrams for recomposition

8 generic 3-way simulation diagrams for proving recomposition

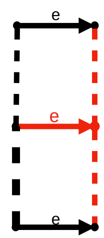


Generic diagrams for recomposition

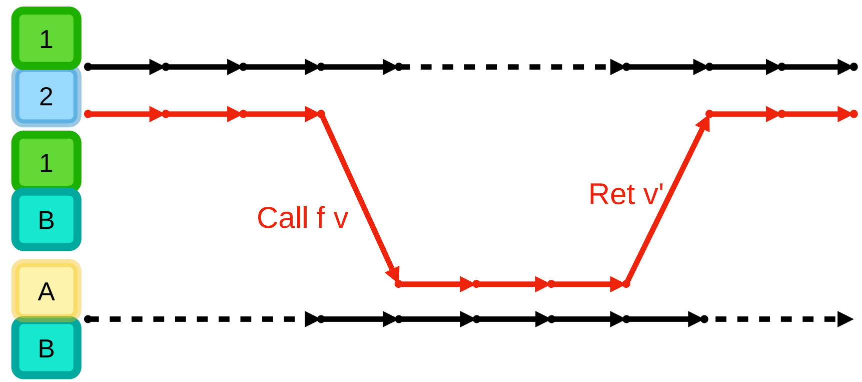
8 generic 3-way simulation diagrams for proving recomposition



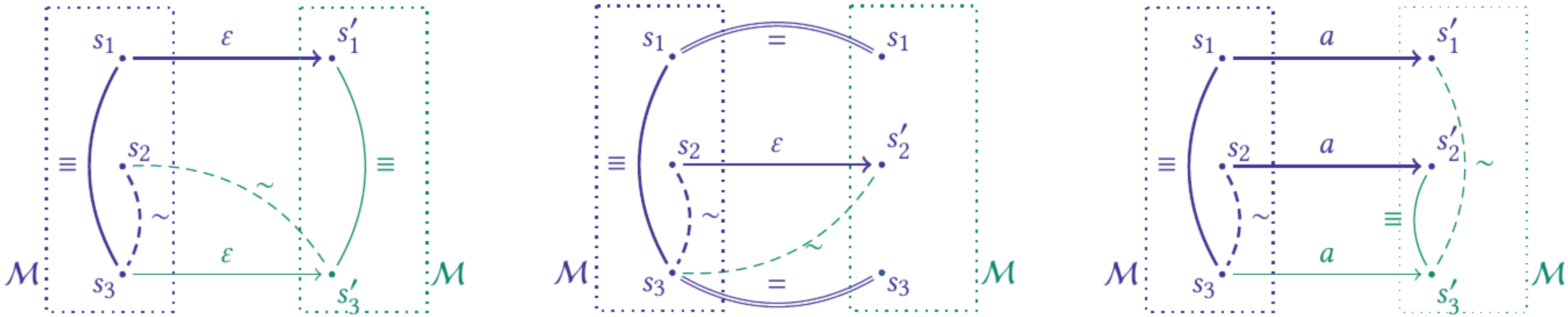
ε-steps



Synchronous events



Generic diagrams for recomposition

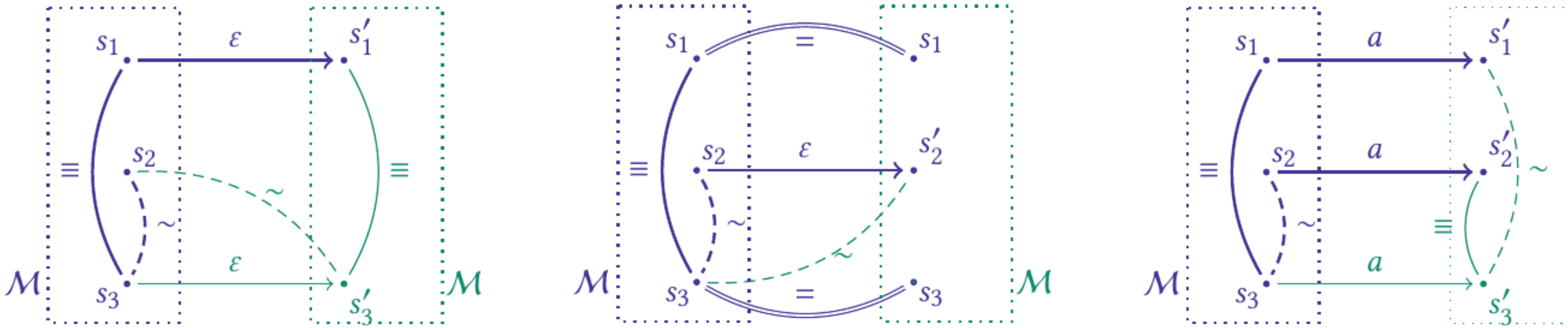


(a) Silent step in strongly related states **(b) Silent step in weakly related states** **(c) Non-silent step with swapping relations**

Figure 4: Recomposition diagrams

+ 5 more such diagrams

Generic diagrams for recomposition



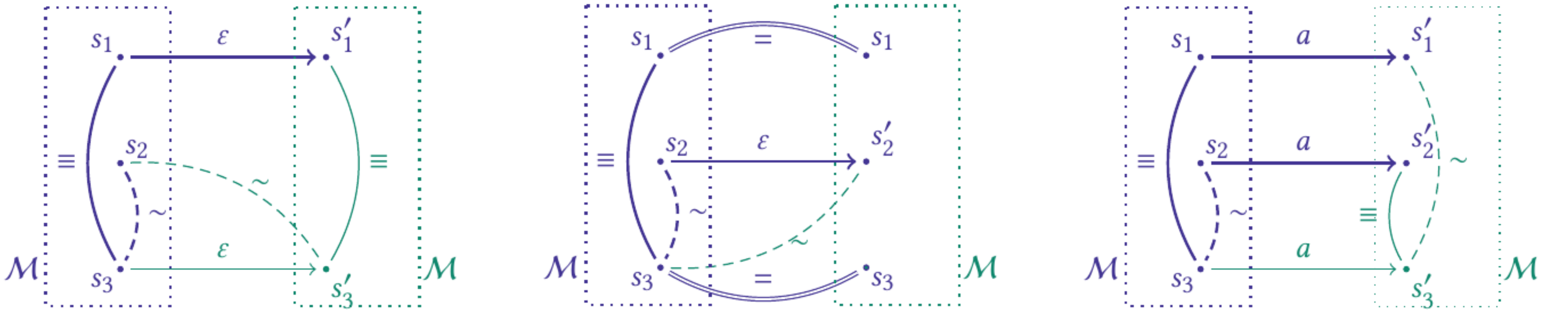
(a) Silent step in strongly related states **(b) Silent step in weakly related states** **(c) Non-silent step with swapping relations**

Figure 4: Recomposition diagrams

+ 5 more such diagrams

+ many more proof engineering novelties for secure completion proof [CCS'24]

Generic diagrams for recomposition



(a) Silent step in strongly related states **(b) Silent step in weakly related states** **(c) Non-silent step with swapping relations**

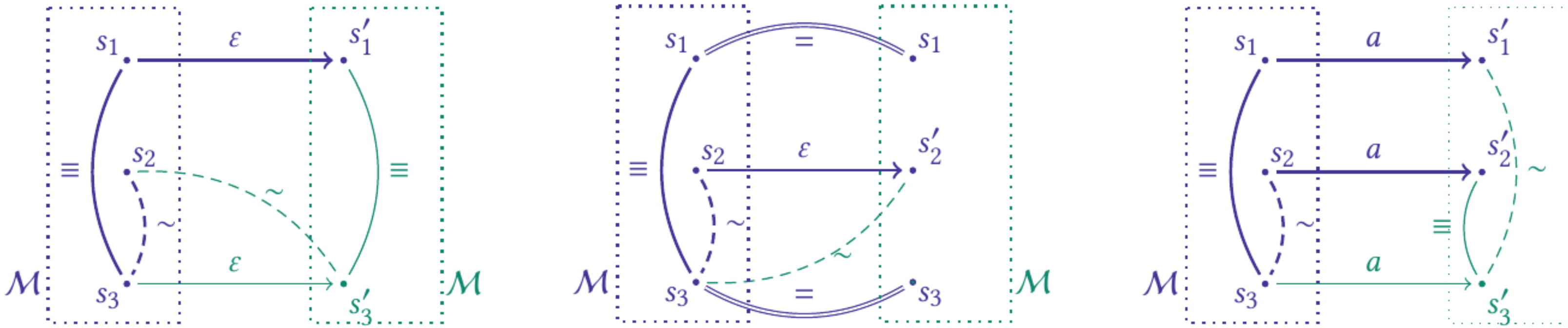
Figure 4: Recomposition diagrams

+ 5 more such diagrams

+ many more proof engineering novelties for secure completion proof [CCS'24]

not too terrible: 38 KLoC is only 30% of CompCert correctness proof

Generic diagrams for recomposition



(a) Silent step in strongly related states (b) Silent step in weakly related states (c) Non-silent step with swapping relations

Figure 4: Recomposition diagrams

+ 5 more such diagrams

+ many more proof engineering novelties for secure completion proof [CCS'24]

not too terrible: 38 KLoC is only 30% of CompCert correctness proof

first compiler for realistic language proved to offer strong security guarantees for compartmentalized code

Open problem: verified backends

Open problem: verified backends

- **Currently we only implemented the SECOMP backend based on CHERI RISC-V plus fancy capabilities**
  – would be nice to also have backends targeting vanilla CHERI RISC-V or Arm Morello
  – would be nice to also implement a Wasm backend (software fault isolation)

Open problem: verified backends

- **Currently we only implemented the SECOMP backend based on CHERI RISC-V plus fancy capabilities**
  – would be nice to also have backends targeting vanilla CHERI RISC-V or Arm Morello
  – would be nice to also implement a Wasm backend (software fault isolation)
- **These backends do the actual security enforcement**
  – so they would be great targets for formal verification

Open problem: verified backends

- **Currently we only implemented the SECOMP backend based on CHERI RISC-V plus fancy capabilities**
  - would be nice to also have backends targeting vanilla CHERI RISC-V or Arm Morello
  - would be nice to also implement a Wasm backend (software fault isolation)
- **These backends do the actual security enforcement**
  - so they would be great targets for formal verification
- **Verifying backends is challenging though**
  - more concrete view of memory as array of bytes (vs CompCert one)
  - once code stored in memory, can no longer hide all the information about compartment's code (code layout leaks)
    - proof step inspired by full abstraction doesn't work all the way down (recomposition)

Extending proof technique in other ways

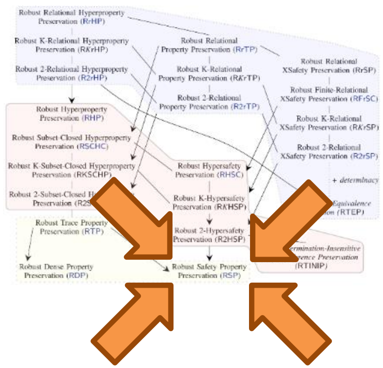
Extending proof technique in other ways

- **Fine-grained dynamic memory sharing** by **capability passing (on CHERI or Morello)**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]

Extending proof technique in other ways

- **Fine-grained dynamic memory sharing** by **capability passing (on CHERI or Morello)**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]

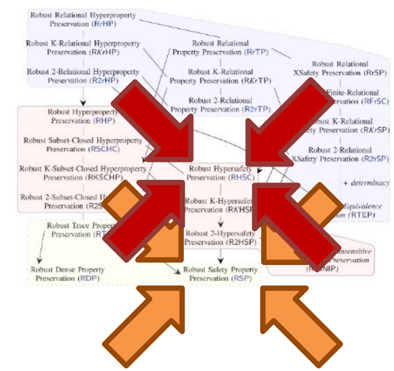
- **Beyond preserving safety against adversarial contexts**



Extending proof technique in other ways

- **Fine-grained dynamic memory sharing** by **capability passing (on CHERI or Morello)**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]

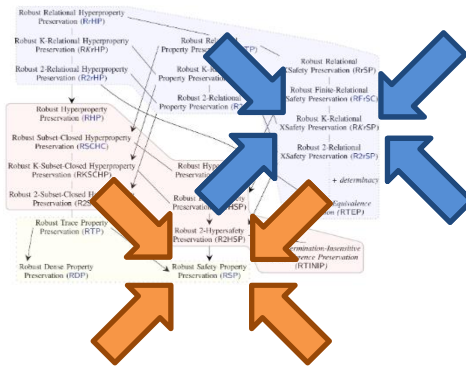
- **Beyond preserving safety against adversarial contexts**
  - towards preserving **hyperproperties** (data confidentiality)



Extending proof technique in other ways

- **Fine-grained dynamic memory sharing** by **capability passing (on CHERI or Morello)**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]

- **Beyond preserving safety against adversarial contexts**
  - towards preserving **hyperproperties** (data confidentiality)
  - even **relational hyperproperties** (observational equivalence)

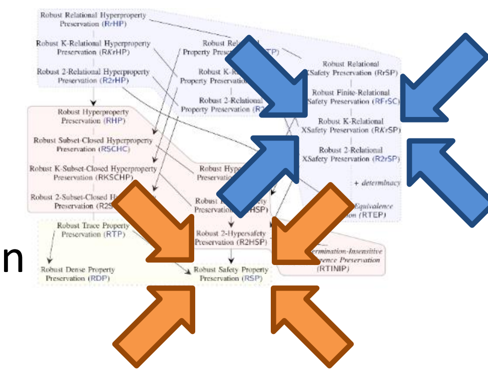


Extending proof technique in other ways

- **Fine-grained dynamic memory sharing** by **capability passing (on CHERI or Morello)**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]

- **Beyond preserving safety against adversarial contexts**
  - towards preserving **hyperproperties** (data confidentiality)
  - even **relational hyperproperties** (observational equivalence)
    - secure compilation criteria strictly stronger than full abstraction
    - can do this for CompCert, but won't hold for backends

  [Jérémy Thibault et al, CSF'19 + more ongoing work]



Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)

SPECTRE

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**

SPECTRE

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**
  - **compartments running in the same process, "universal read gadgets" easy**

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**
  - **compartments running in the same process, "universal read gadgets" easy**
- **Started looking into Spectre defenses compilers can insert**

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**
  - **compartments running in the same process, "universal read gadgets" easy**
- **Started looking into Spectre defenses compilers can insert**
  - **Speculative Load Hardening** (implemented in LLVM + selective variant in Jasmin DSL)
    - **enforces speculative constant time: chapter in new Security Foundations textbook**

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**
  - **compartments running in the same process, "universal read gadgets" easy**
- **Started looking into Spectre defenses compilers can insert**
  - **Speculative Load Hardening** (implemented in LLVM + selective variant in Jasmin DSL)
    - **enforces speculative constant time: chapter in new Security Foundations textbook**
  - **Ultimate SLH** [Zhang et al, USENIX SEC'23]**: enforces <u>relative security</u> (chapter soon)**

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**
  - **compartments running in the same process, "universal read gadgets" easy**
- **Started looking into Spectre defenses compilers can insert**
  - **Speculative Load Hardening** (implemented in LLVM + selective variant in Jasmin DSL)
    - **enforces speculative constant time: chapter in new Security Foundations textbook**
  - **Ultimate SLH** [Zhang et al, USENIX SEC'23]**: enforces <u>relative security</u> (chapter soon)**
  - **New "Flexible" SLH variant**: tested for relative security, hopefully proof and paper soon

Enforcement tricky beyond safety

- **Preserving hypersafety against adversarial contexts** (e.g. data confidentiality)
  - **challenging at the lowest level: micro-architectural side-channels attacks**
  - **compartments running in the same process, "universal read gadgets" easy**
- **Started looking into Spectre defenses compilers can insert**
  - **Speculative Load Hardening** (implemented in LLVM + selective variant in Jasmin DSL)
    - **enforces speculative constant time: chapter in new Security Foundations textbook**
  - **Ultimate SLH** [Zhang et al, USENIX SEC'23]**: enforces relative security (chapter soon)**
  - **New "Flexible" SLH variant**: tested for relative security, hopefully proof and paper soon
- **Combining this with compartmentalization practically interesting**
  - Especially for languages like Wasm, which are used for same-process isolation

Last slide on future work / open problems

Last slide on future work / open problems

- **Dynamic compartment creation**
  - from code-based to data-based compartmentalization (e.g. browser tabs)

Last slide on future work / open problems

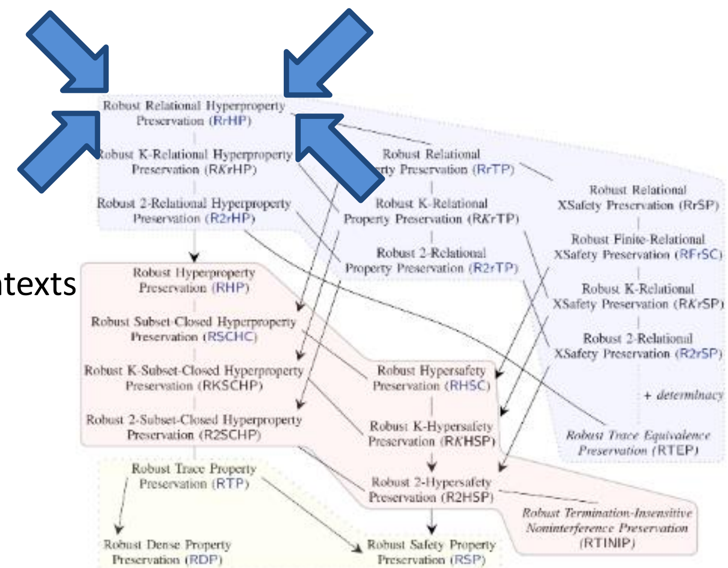
- **Dynamic compartment creation**
  – from code-based to data-based compartmentalization (e.g. browser tabs)
- **Dynamic privileges**
  – passing capabilities, dynamic interfaces, history-based access control, ...

Last slide on future work / open problems

- **Dynamic compartment creation**
  - from code-based to data-based compartmentalization (e.g. browser tabs)
- **Dynamic privileges**
  - passing capabilities, dynamic interfaces, history-based access control, ...
- **Protecting higher-level abstractions**
  (than those of the C language)
  - **Securely Compiling Verified F* Programs With IO**
    [Cezar-Constantin Andrici et al, POPL'24]
    - using reference monitoring and higher-order contracts

Last slide on future work / open problems

- **Dynamic compartment creation**
  - from code-based to data-based compartmentalization (e.g. browser tabs)
- **Dynamic privileges**
  - passing capabilities, dynamic interfaces, history-based access control, ...
- **Protecting higher-level abstractions** (than those of the C language)
  - **Securely Compiling Verified F* Programs With IO** [Cezar-Constantin Andrici et al, POPL'24]
    - using reference monitoring and higher-order contracts
    - preserving **all relational hyperproperties** against adversarial contexts
    - first step towards formally secure F*-OCaml interoperability



SECOMP: Formally Secure Compilation of Compartmentalized C Programs

1. Goal: formalized end-to-end security guarantees

- preserve properties **against adversarial contexts**
- we overcame additional challenges to support **mutually distrustful compartments** and **dynamic compromise**

2. Enforcement: protect abstractions all the way down

- **Extended CompCert languages with compartments**
- **Unverified backend targeting CHERI RISC-V capability machine**

3. Proof: verify security of our compilation chain

- **more scalable proof technique machine-checked in Coq**
- **first compiler for realistic language proved to offer strong security guarantees for compartmentalized code**