My Journey in Secure Compilation

Cătălin Hrițcu, MPI-SP, Bochum, Germany

My companions on this journey:

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, ...

Good programming languages provide helpful abstractions for writing more secure code

Good programming languages provide helpful abstractions for writing more secure code

- structured control flow, procedures, modules, types, interfaces, correctness and security specifications, ...

Good programming languages provide helpful abstractions for writing more secure code

- structured control flow, procedures, modules, types, interfaces, correctness and security specifications, ...
- **suppose we have a secure source program ...**
  - For instance formally verified in F* [POPL'16,'17,'18,'20,'24, ICFP'17,'19, ...]
    - e.g. EverCrypt verified crypto library, shipping in Firefox, Linux Kernel, ...

Good programming languages provide helpful abstractions for writing more secure code

- structured control flow, procedures, modules, types, interfaces, correctness and security specifications, ...

- **suppose we have a <u>secure</u> source program ...**
  - For instance formally verified in F* [POPL'16,'17,'18,'20,'24, ICFP'17,'19, ...]
    - e.g. EverCrypt verified crypto library, shipping in Firefox, Linux Kernel, ...
  - Or a program written entirely in safe OCaml or Rust

- **What happens when we compile such a secure source program and link it with adversarial target code?**

- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**

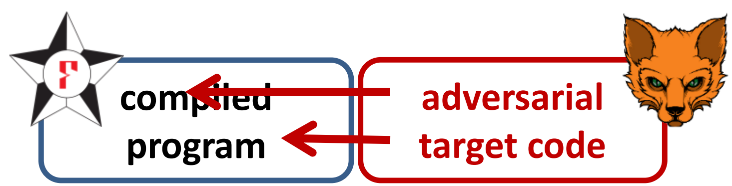
- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**

- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**
  - **not just hypothetical**: verified code often linked with unverified code, safe OCaml and Rust often linked with C/C++/ASM code (e.g. libraries)

- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**
  - **not just hypothetical**: verified code often linked with unverified code, safe OCaml and Rust often linked with C/C++/ASM code (e.g. libraries)
  - target-level code can be buggy, vulnerable, compromised, malicious

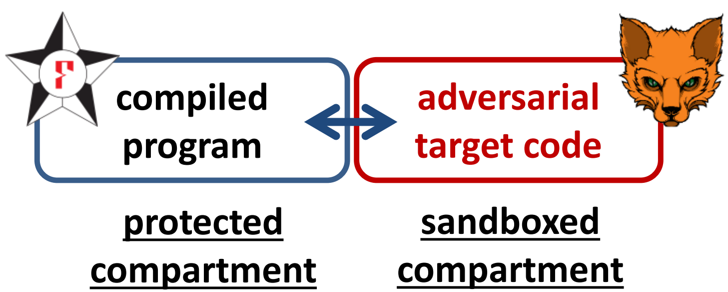
- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**
  - **not just hypothetical**: verified code often linked with unverified code, safe OCaml and Rust often linked with C/C++/ASM code (e.g. libraries)
  - target-level code can be buggy, vulnerable, compromised, malicious
  - **currently: all abstractions and source-level guarantees are lost**

- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**
  - **not just hypothetical**: verified code often linked with unverified code, safe OCaml and Rust often linked with C/C++/ASM code (e.g. libraries)
  - target-level code can be buggy, vulnerable, compromised, malicious
  - **currently: all abstractions and source-level guarantees are lost**
    - **lower-level attacks become possible: break control flow, memory safety, etc.**



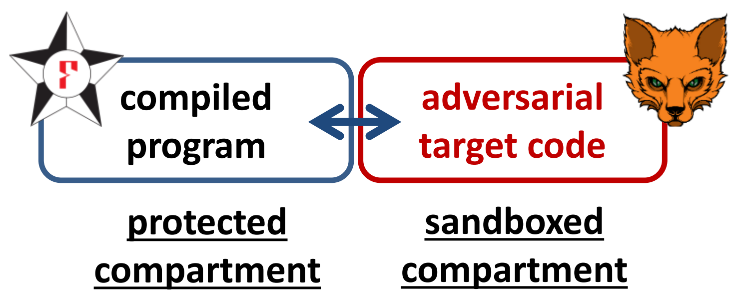
Secure Compilation

- **What happens when we compile such a <u>secure</u> source program and link it with adversarial target code?**
  - **not just hypothetical**: verified code often linked with unverified code, safe OCaml and Rust often linked with C/C++/ASM code (e.g. libraries)
  - target-level code can be buggy, vulnerable, compromised, malicious
  - **currently: all abstractions and source-level guarantees are lost**
    - **lower-level attacks become possible: break control flow, memory safety, etc.**



Secure Compilation of Secure Source Programs

- **What happens when we compile such a secure source program and link it with adversarial target code?**
  - **not just hypothetical**: verified code often linked with unverified code, safe OCaml and Rust often linked with C/C++/ASM code (e.g. libraries)
  - target-level code can be buggy, vulnerable, compromised, malicious
  - **currently: all abstractions and source-level guarantees are lost**
    - **lower-level attacks become possible: break control flow, memory safety, etc.**



Secure Compilation of Secure Source Programs

- **Protect source-level abstractions all the way down**
  even against linked adversarial target code

Secure Compilation of Secure Source Programs

- **Protect source-level abstractions all the way down even against linked adversarial target code**
  - **various enforcement mechanisms for sandboxing untrusted code**: software-fault isolation (SFI), capability machines, tagged architectures, ...
  - shared responsibility: compiler, linker, loader, OS, HW

Secure Compilation of Secure Source Programs

- **Protect source-level abstractions all the way down even against linked adversarial target code**
  - **various enforcement mechanisms for sandboxing untrusted code**: software-fault isolation (SFI), capability machines, tagged architectures, ...
  - shared responsibility: compiler, linker, loader, OS, HW
- **This is very challenging:**
  - **the originally proposed formal criterion was fully abstract compilation** [Abadi, Protection in programming-language translations. 1999]

Secure Compilation of Vulnerable Source Programs

Secure Compilation of Vulnerable Source Programs

- **Insecure languages like C enable devastating vulnerabilities**

Secure Compilation of Vulnerable Source Programs

- **Insecure languages like C enable devastating vulnerabilities**
  - **undefined behavior** pervasive in C: buffer overflows, use after frees, double frees, invalid type casts, various concurrency bugs, ...

Secure Compilation of Vulnerable Source Programs

- **Insecure languages like C enable devastating vulnerabilities**
  - **undefined behavior** pervasive in C: buffer overflows, use after frees, double frees, invalid type casts, various concurrency bugs, ...
  - **undefined behavior** also present in unsafe Rust, OCaml, ...

Secure Compilation of Vulnerable Source Programs

- **Insecure languages like C enable devastating vulnerabilities**
  - **undefined behavior** pervasive in C: buffer overflows, use after fr double frees, invalid type casts, various concurrency bugs, ...
  - **undefined behavior** also present in unsafe Rust, OCaml, ...
- **Yet even the C language does provide some useful abstractions:**
  - structured control flow, procedures, pointers to shared memory

Secure Compilation of <u>Vulnerable</u> Source Programs

- **Insecure languages like C enable devastating vulnerabilities**
  - **undefined behavior** pervasive in C: buffer overflows, use after frees, double frees, invalid type casts, various concurrency bugs, ...
  - **undefined behavior** also present in <u>unsafe</u> Rust, OCaml, ...
- **Yet even the C language does provide some useful abstractions:**
  - structured control flow, procedures, pointers to shared memory
  - **but not enforced during compilation for programs with UB: all guarantees are lost!**

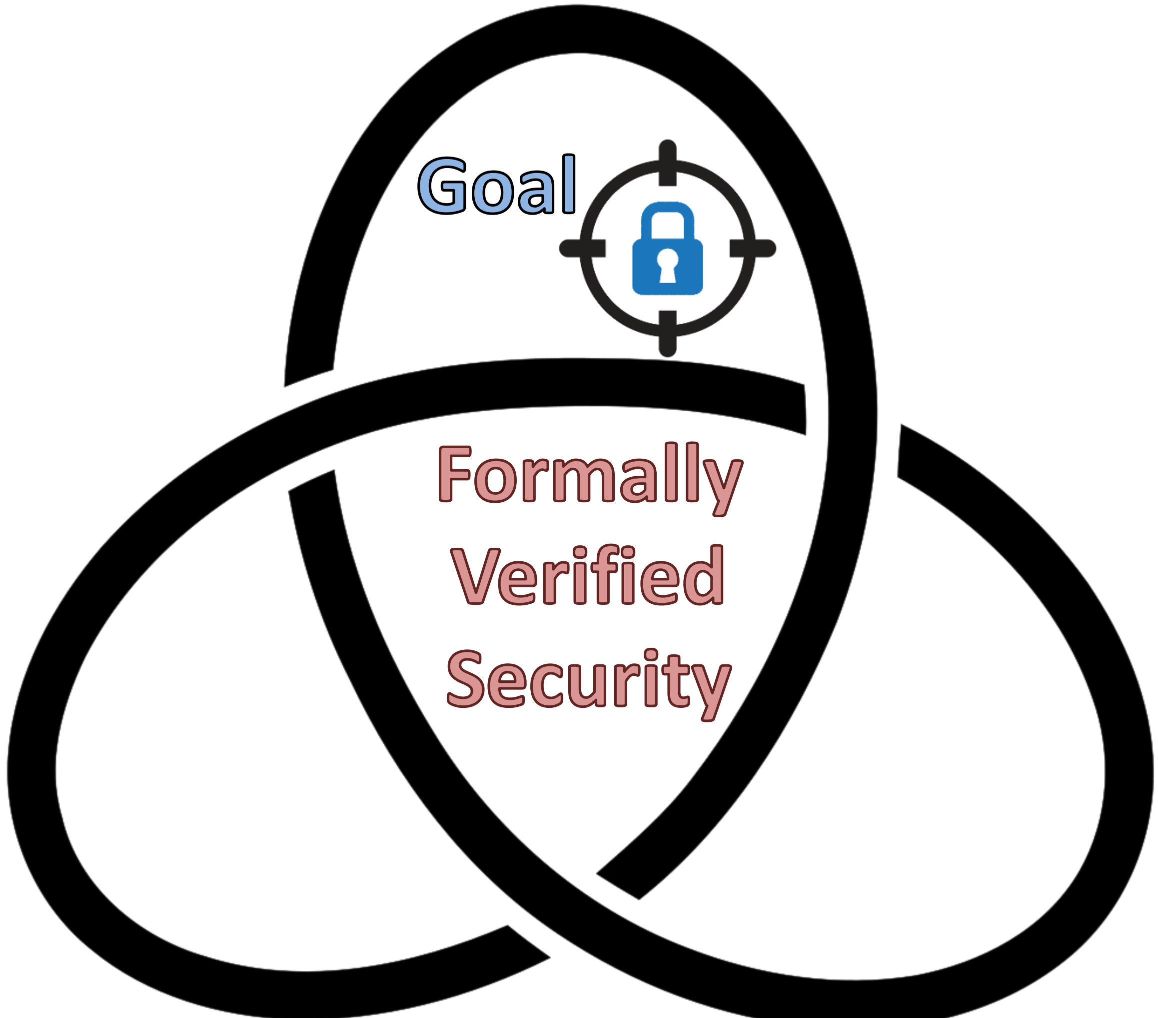
Secure Compilation of Vulnerable Source Programs

- **Insecure languages like C enable devastating vulnerabilities**
  - **undefined behavior** pervasive in C: buffer overflows, use after frees, double frees, invalid type casts, various concurrency bugs, ...
  - **undefined behavior** also present in unsafe Rust, OCaml, ...
- **Yet even the C language does provide some useful abstractions:**
  - structured control flow, procedures, pointers to shared memory
  - **but not enforced during compilation for programs with UB: all guarantees are lost!**
  - **we add one more abstraction to C: fine-grained compartments that can naturally interact**

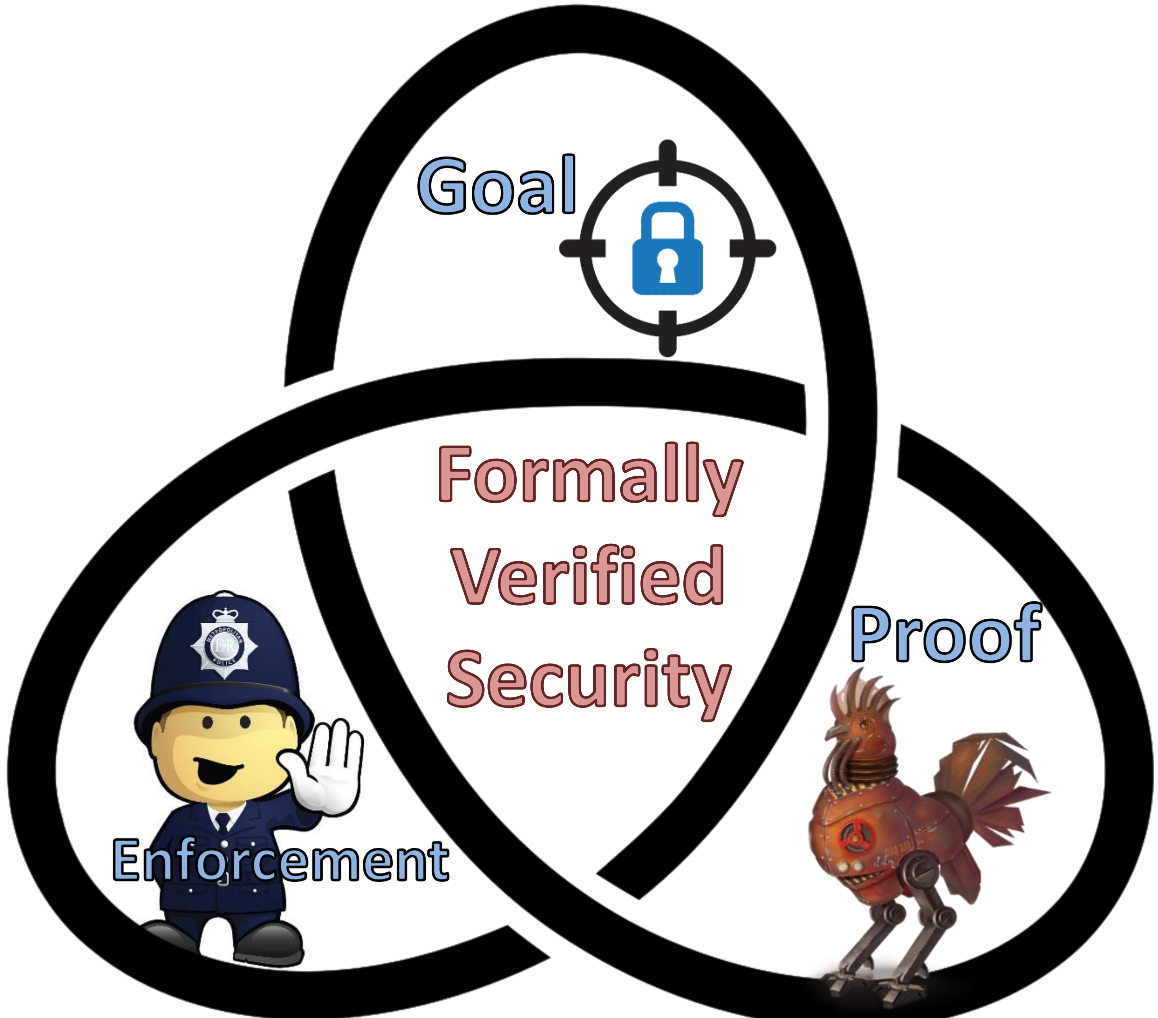
Secure Compilation of Vulnerable Source Programs

- **Insecure languages like C enable devastating vulnerabilities**
  - **undefined behavior** pervasive in C: buffer overflows, use after frees, double frees, invalid type casts, various concurrency bugs, ...
  - **undefined behavior** also present in unsafe Rust, OCaml, ...
- **Yet even the C language does provide some useful abstractions:**
  - structured control flow, procedures, pointers to shared memory
  - **but not enforced during compilation for programs with UB: all guarantees are lost!**
  - **we add one more abstraction to C: fine-grained compartments that can naturally interact**
- **Secure compilation chain that protects these abstractions**
  - all the way down, at compartment boundaries (hopefully more efficient than removing UB)
  - against compartments dynamically compromised by undefined behavior
  - using the same kind of enforcement mechanisms for **compartmentalization**

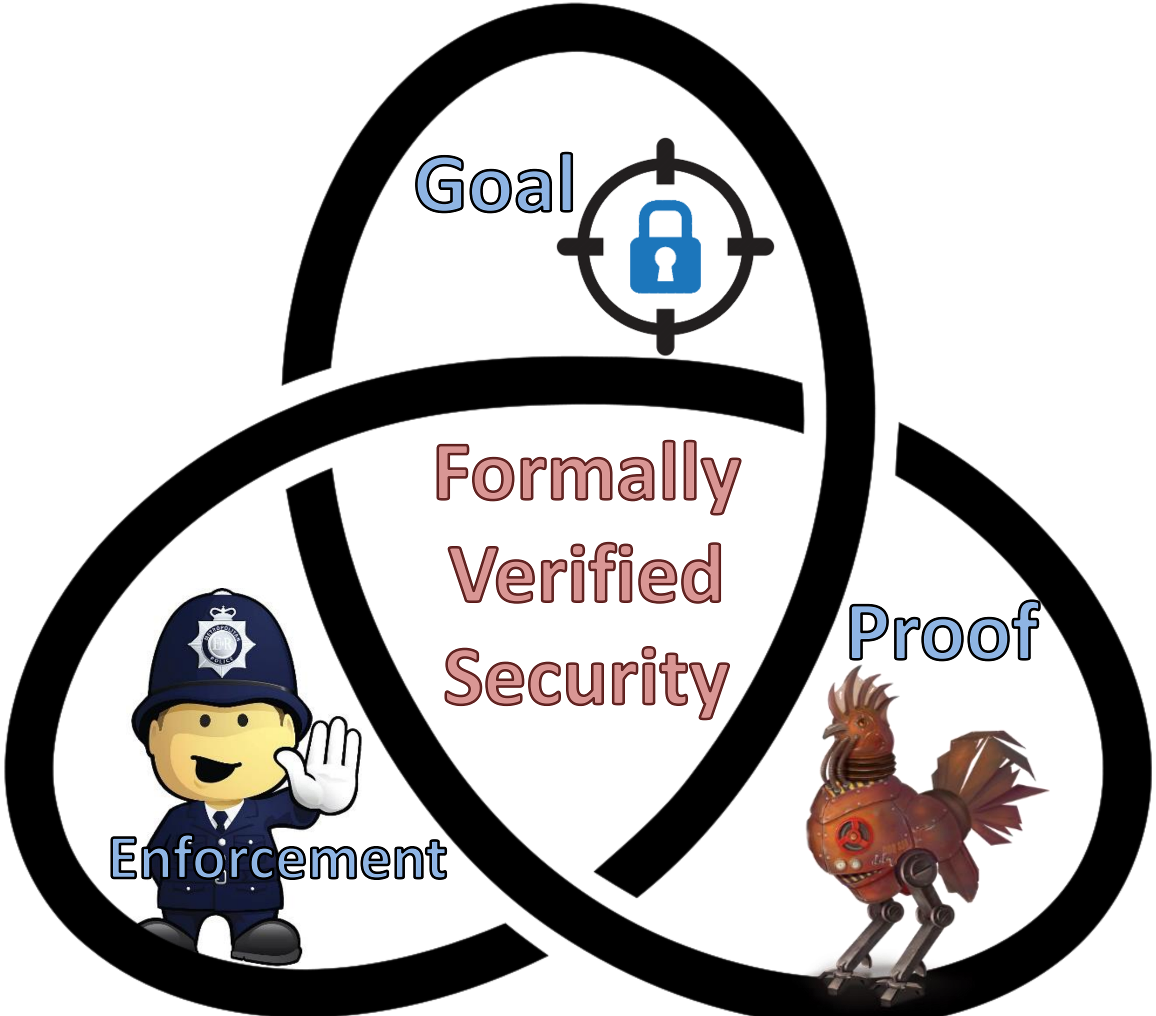
Formally Verified Security


Goal
Formally
Verified
Security


Goal
Formally
Verified
Security
Enforcement


Goal
Formally
Verified
Security
Enforcement
Proof

Secure Compilation



1. Security Goal

1. Security Goal

- **Question A:**
  What does it mean to securely compile a secure source program against linked adversarial target-level code?

1. Security Goal

- **Question A:**
  What does it mean to securely compile a secure source program against linked adversarial target-level code?
  - e.g. simple verified web server, linked with unverified libraries [POPL'24]

1. Security Goal

- **Question A:**
  What does it mean to securely compile a secure source program against linked adversarial target-level code?
  - e.g. simple verified web server, linked with unverified libraries [POPL'24]

- **We want to enable source-level security reasoning**

1. Security Goal

- **Question A:**
  What does it mean to securely compile a secure source program against linked adversarial target-level code?
  – e.g. simple verified web server, linked with unverified libraries [POPL'24]

- **We want to enable source-level security reasoning**
  – **linked adversarial target code cannot break the security of compiled program,** any more than some linked source code already could

1. Security Goal

- **Question A:**
  What does it mean to securely compile a secure source program against linked adversarial target-level code?
  - e.g. simple verified web server, linked with unverified libraries [POPL'24]

- **We want to enable source-level security reasoning**
  - **linked adversarial target code cannot break the security of compiled program,** any more than some linked source code already could
  - **no "low-level" attacks introduced by compilation and linking**

Preserving security against adversarial contexts

Preserving security against adversarial contexts

∀ security property π

Preserving security against adversarial contexts

∀ security property π

Preserving security against adversarial contexts

∀ security property π

source program

satisfies π

Preserving security against adversarial contexts

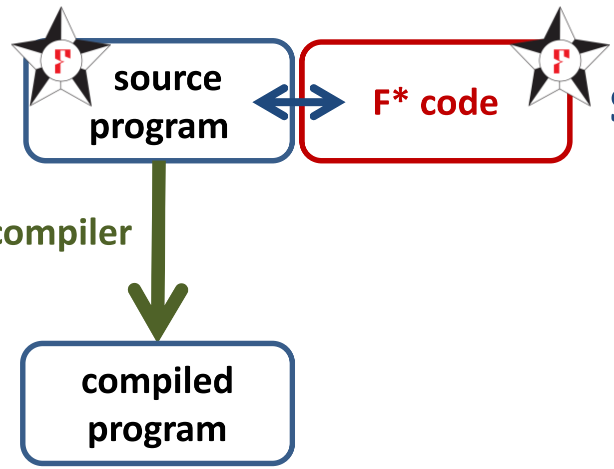
∀ security property π

∀ F*code [source program ↔ F* code] satisfies π

Preserving security against adversarial contexts

∀ security property π

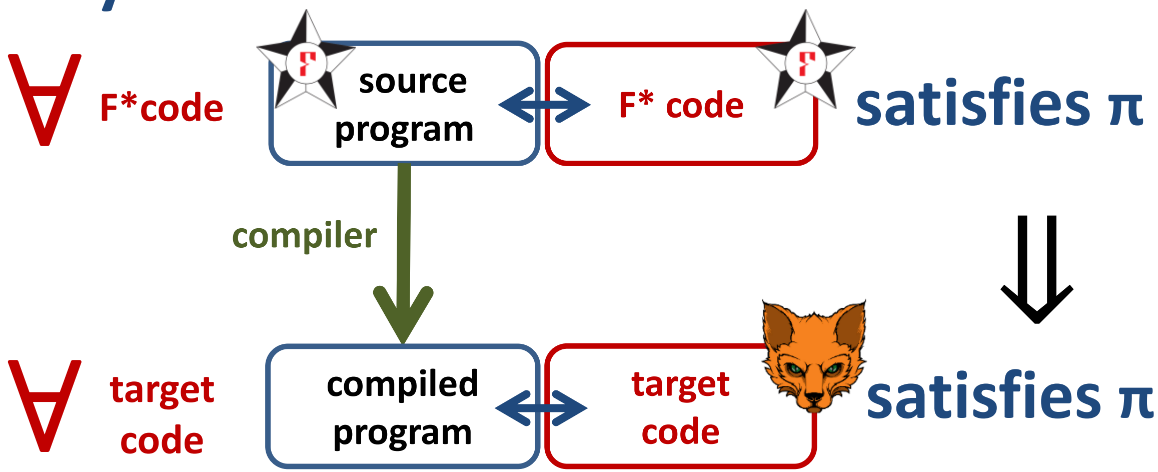
∀ F*code



satisfies π

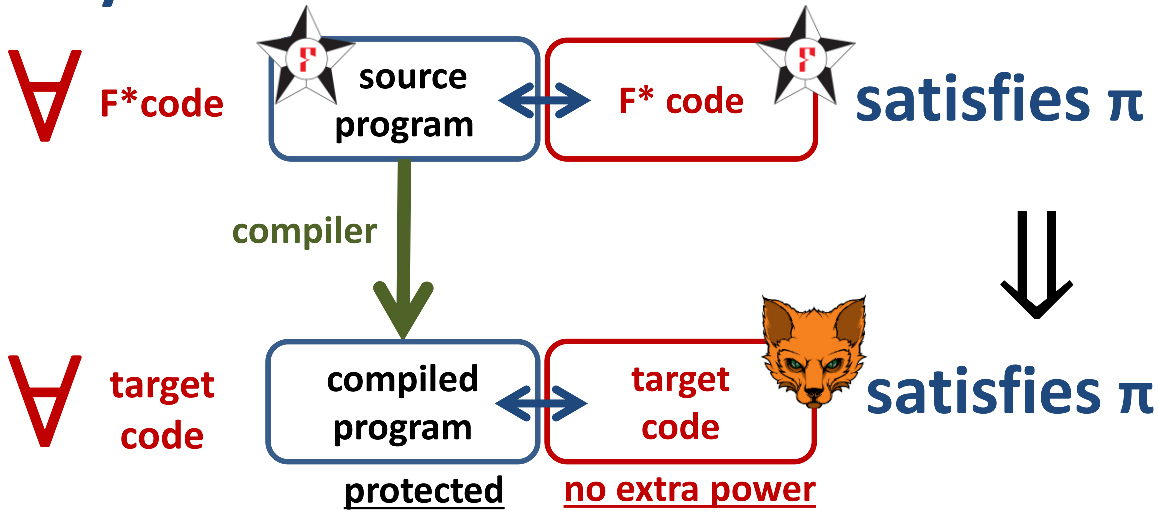
Preserving security against adversarial contexts

∀ security property π



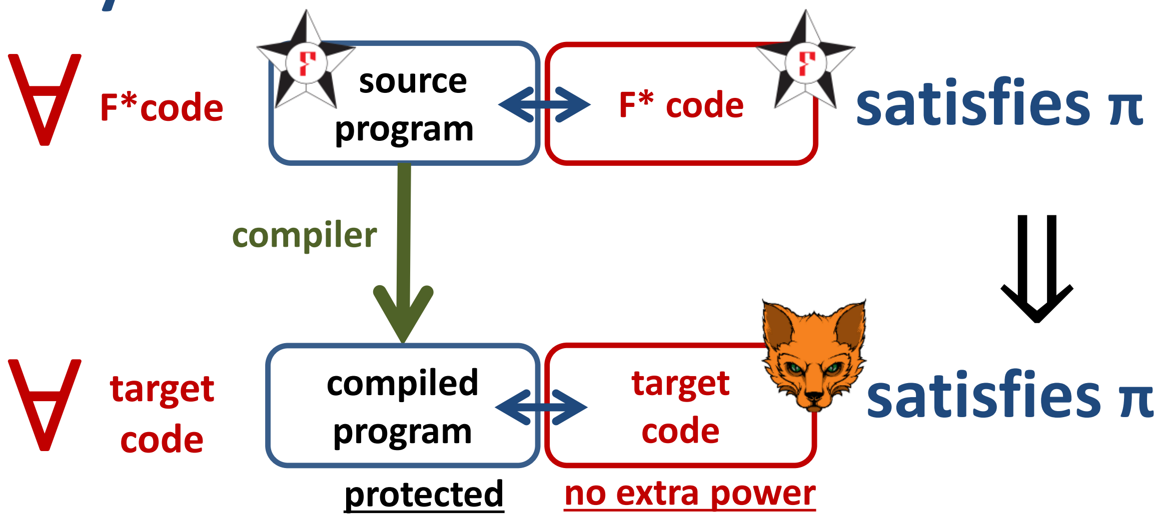
Preserving security against adversarial contexts

∀ security property π



Preserving security against adversarial contexts

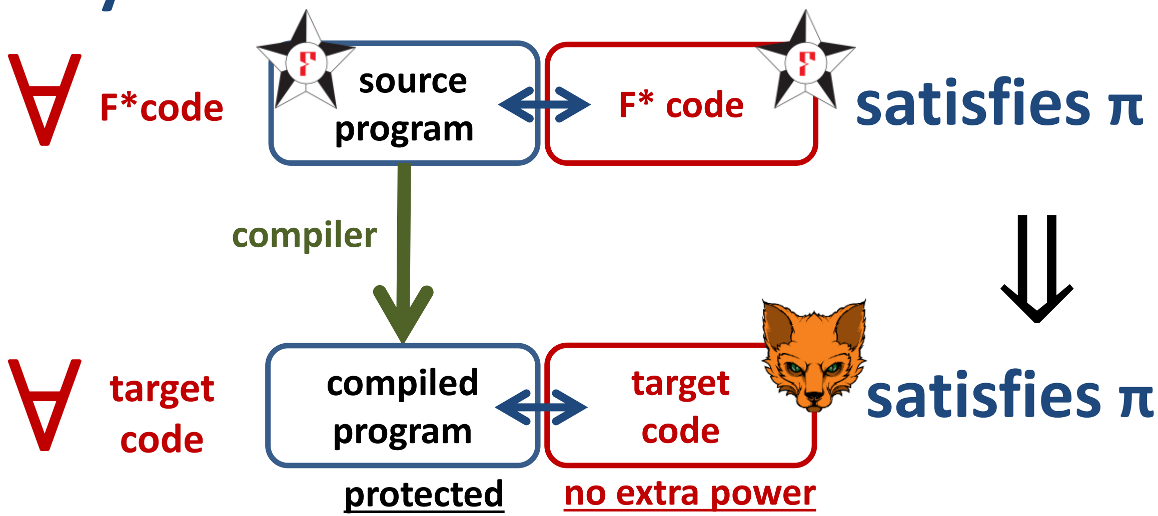
∀ security property π



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

Preserving security against adversarial contexts

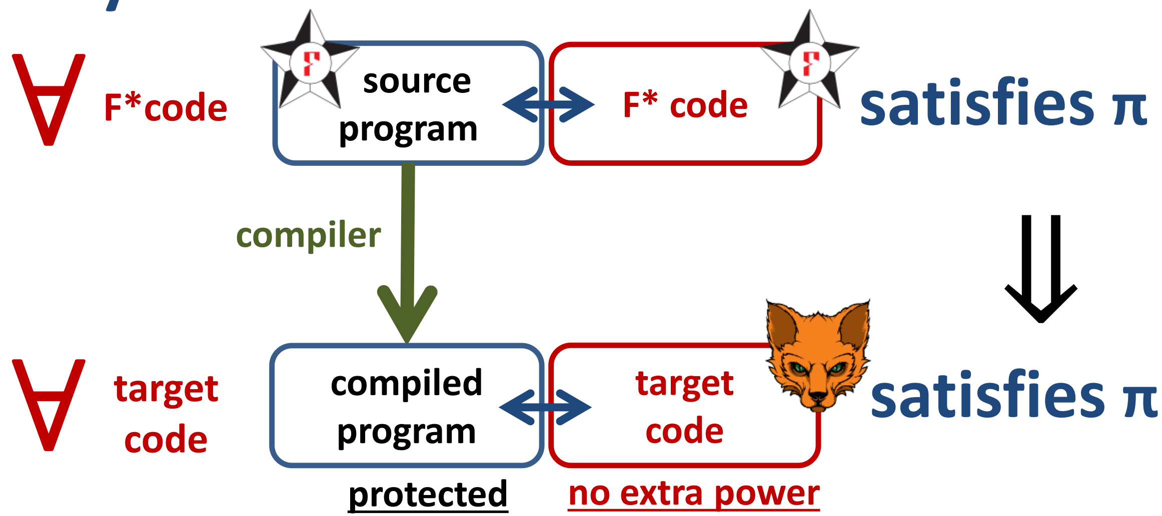
∀ security property π



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)

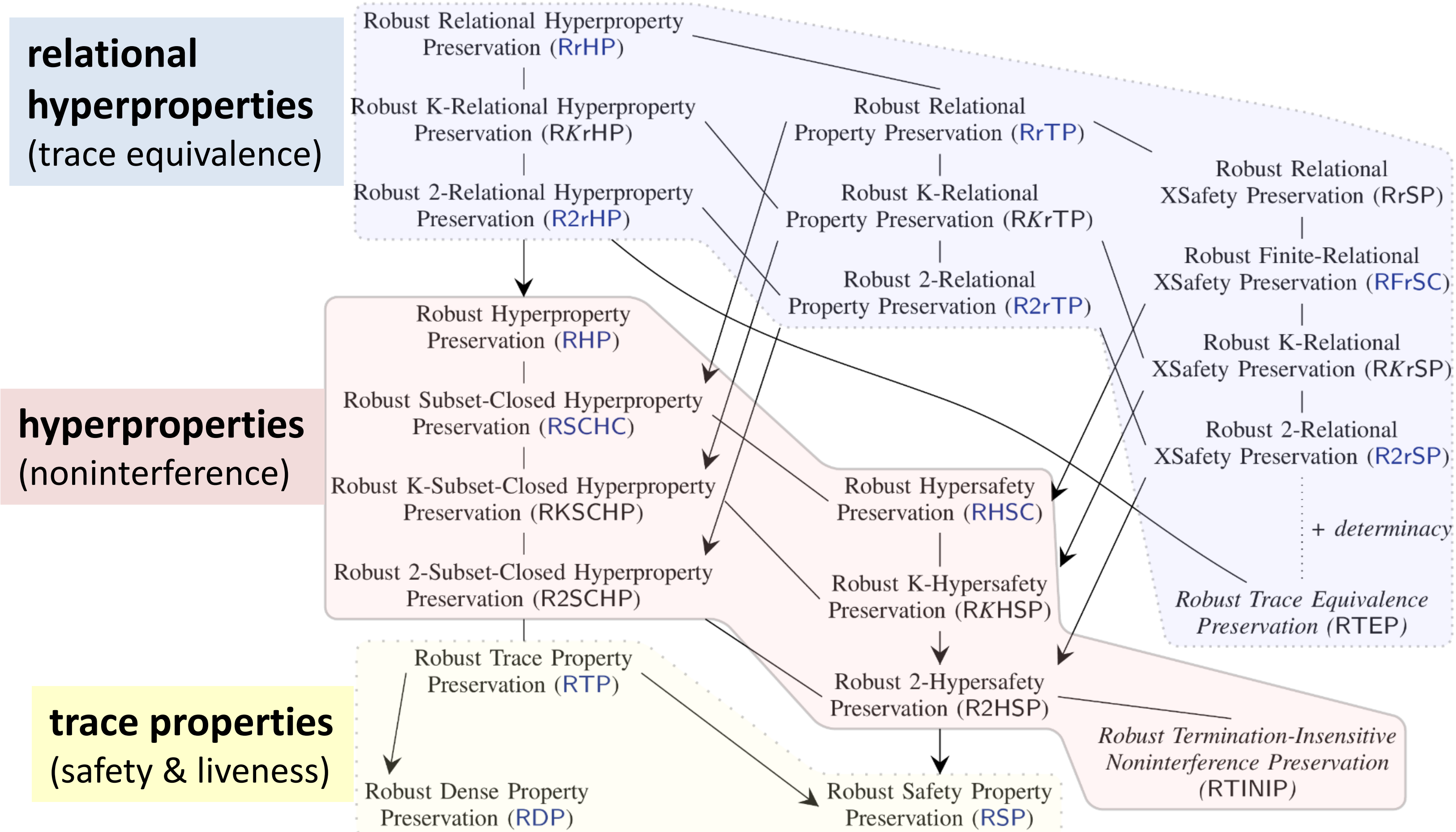
Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

relational hyperproperties
(trace equivalence)

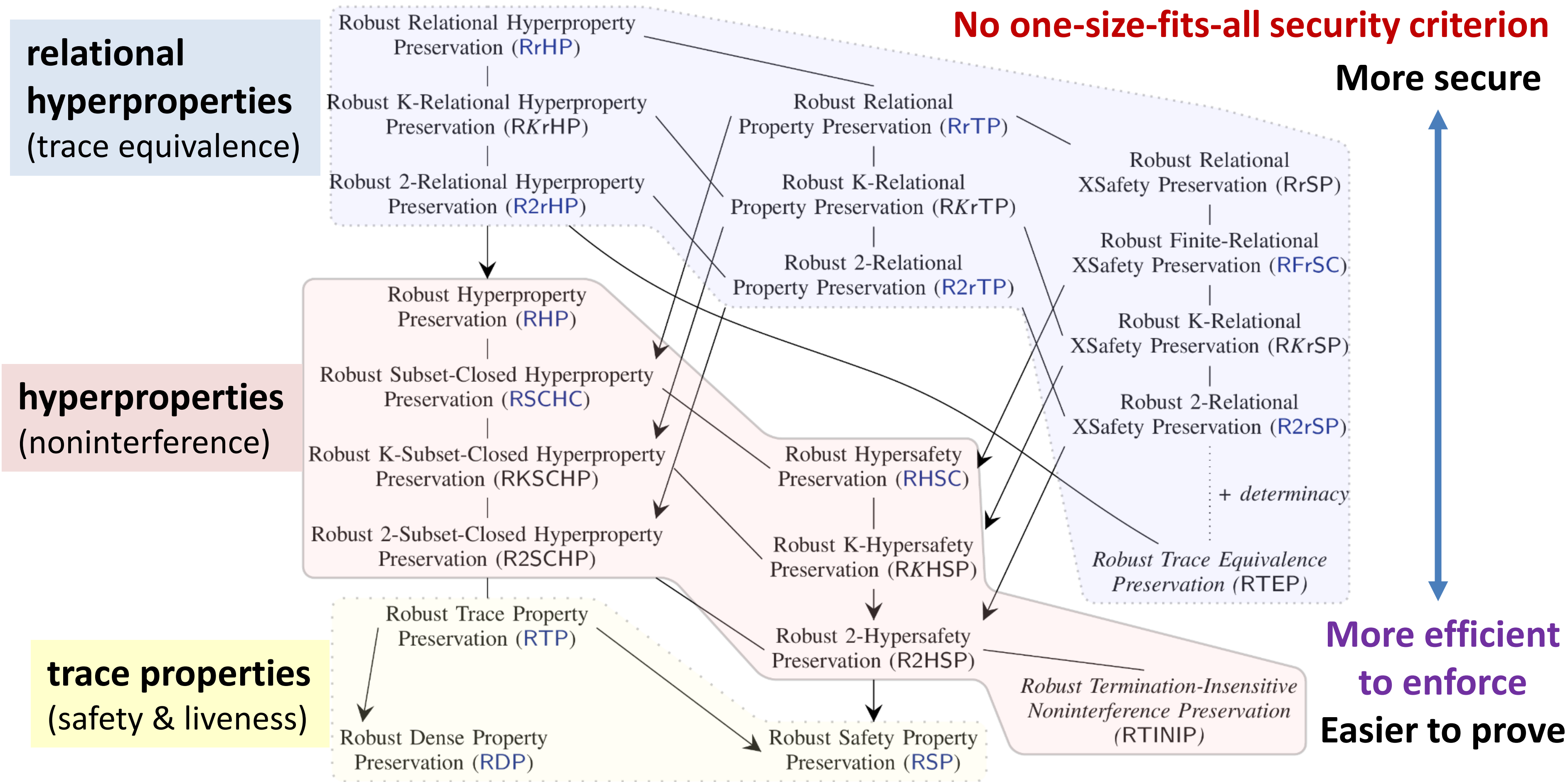
hyperproperties
(noninterference)

trace properties
(safety & liveness)

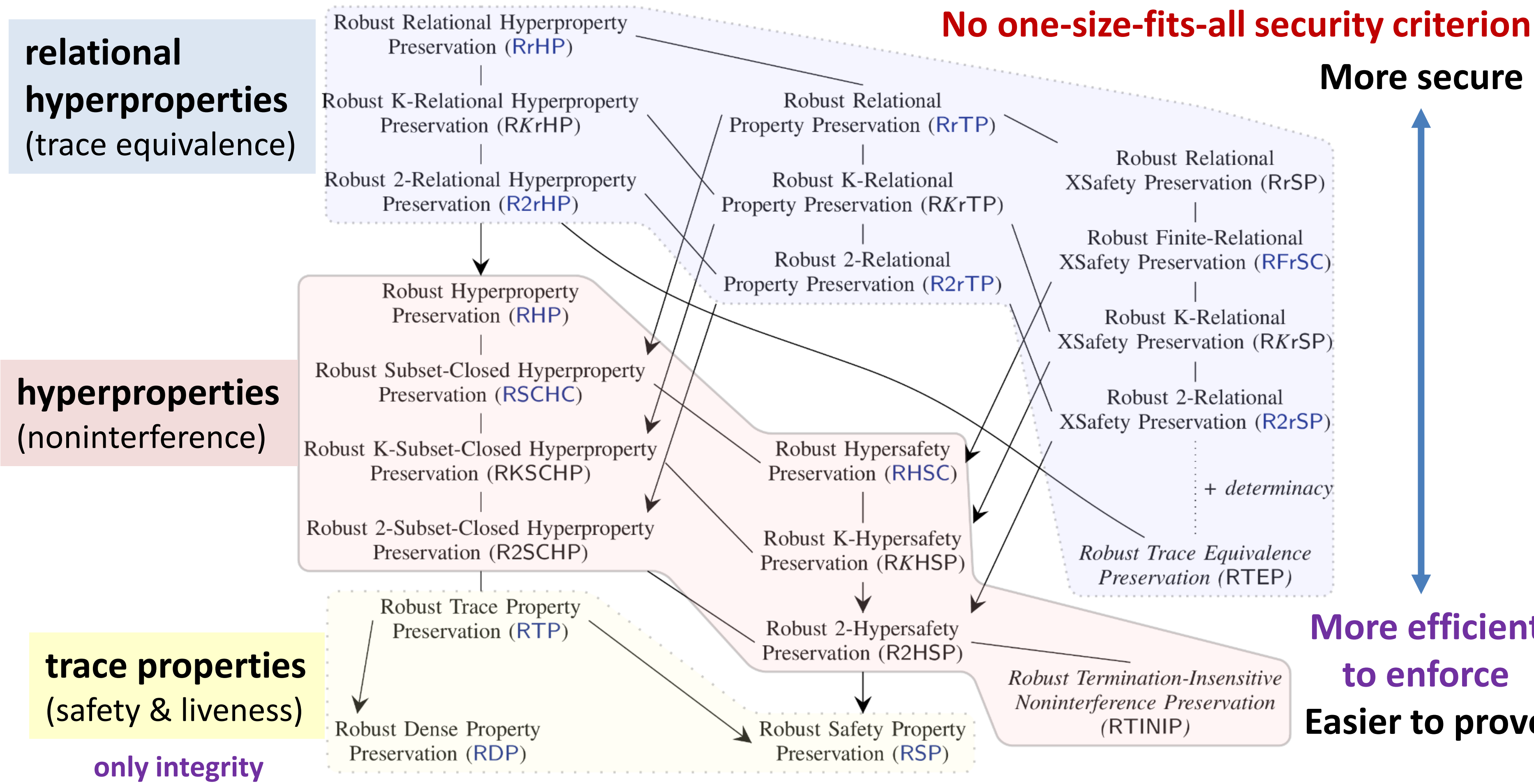
Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]



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



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

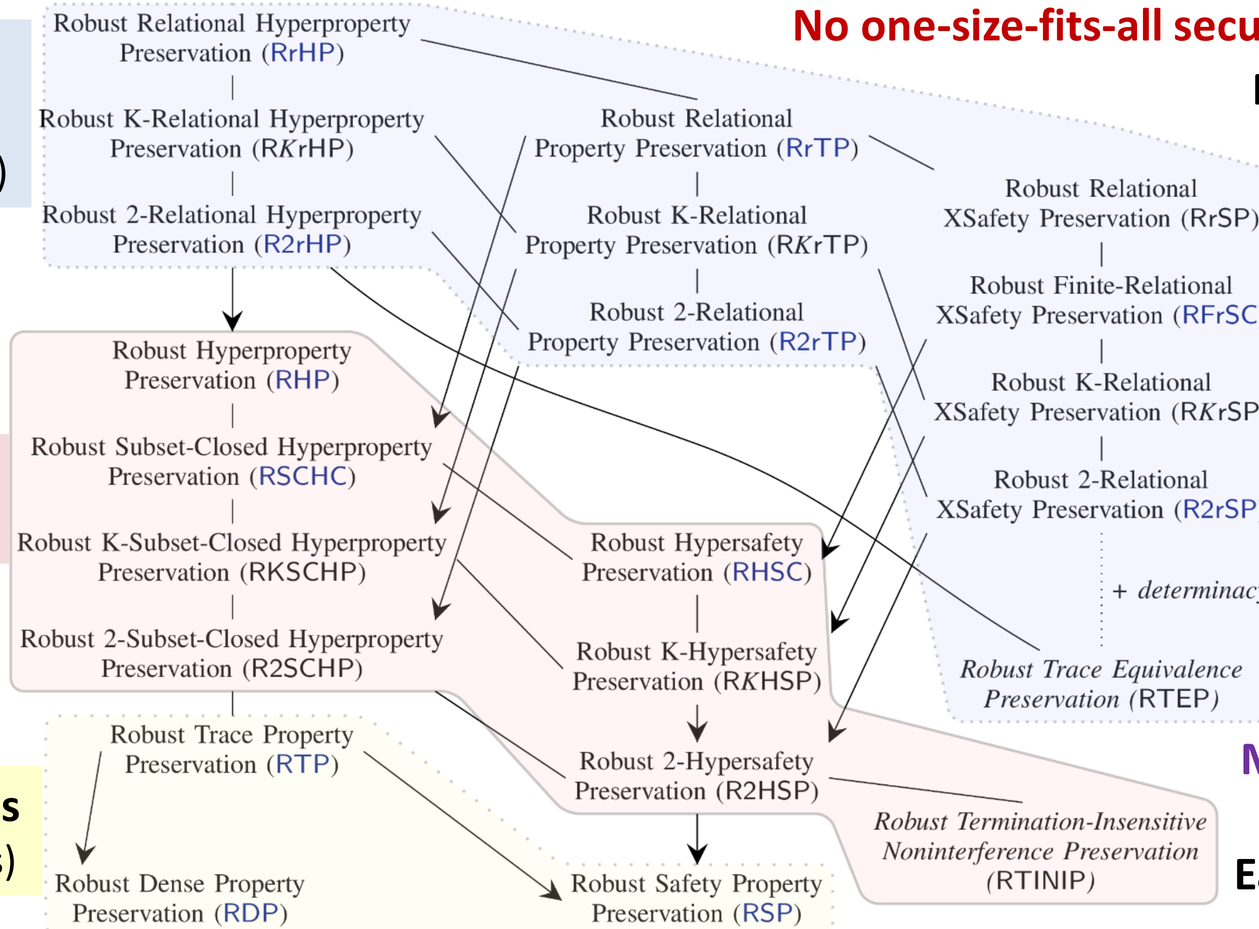
relational hyperproperties
(trace equivalence)

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



More secure

More efficient to enforce

Easier to prove

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

No one-size-fits-all security criterion

relational hyperproperties (trace equivalence)

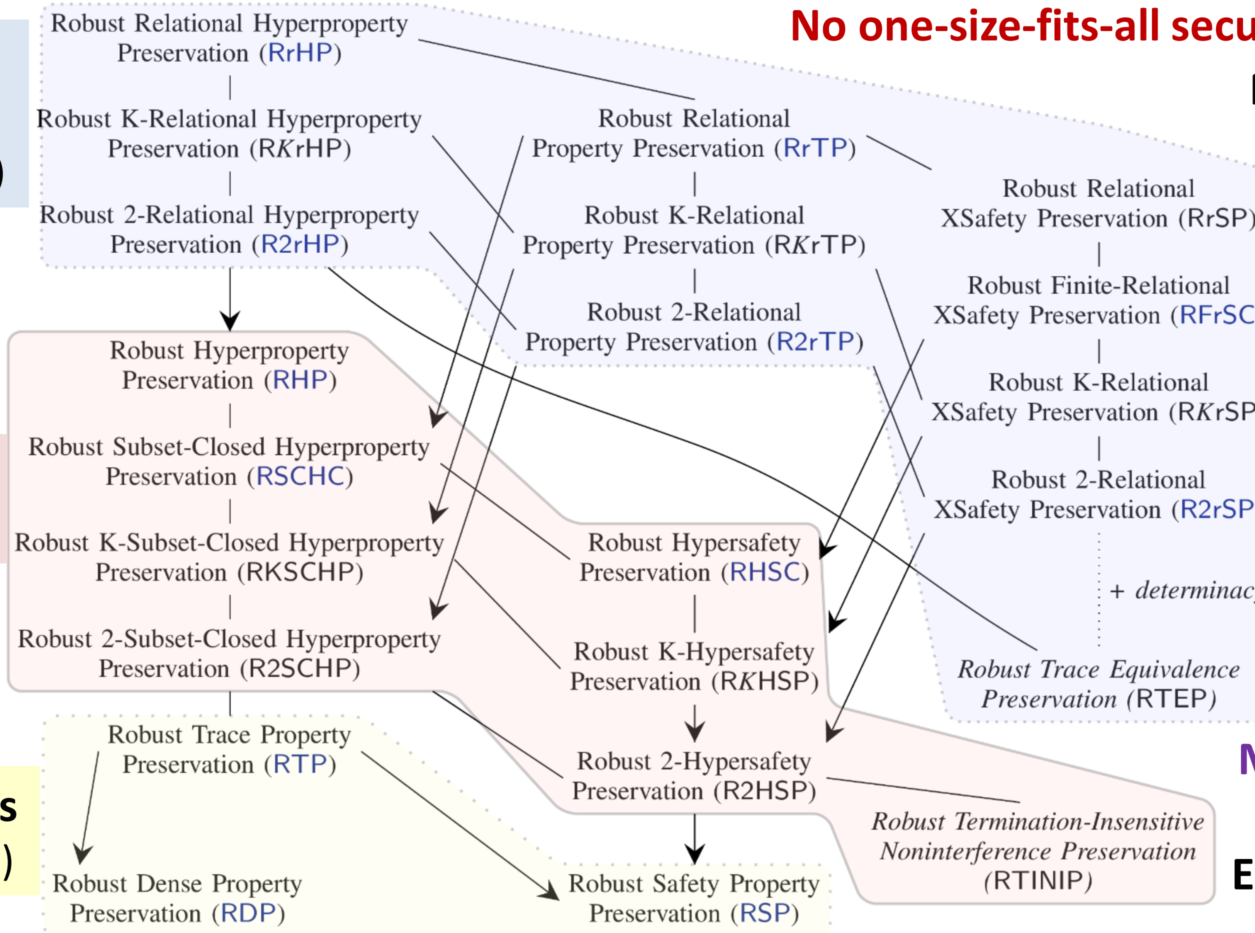
+ code confidentiality

hyperproperties (noninterference)

+ data confidentiality

trace properties (safety & liveness)

only integrity



More secure

More efficient to enforce

Easier to prove

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

relational hyperproperties
(trace equivalence)

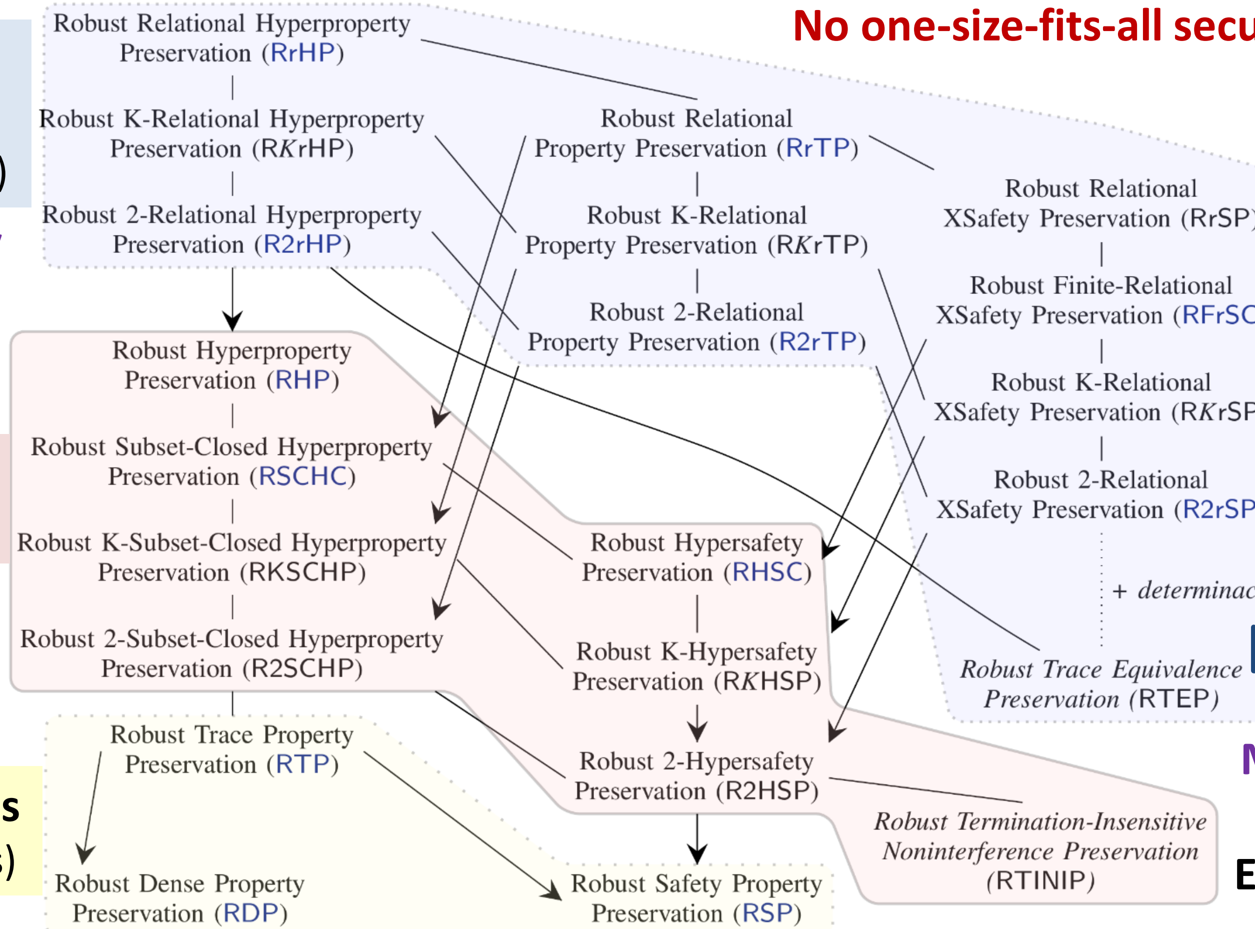
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



No one-size-fits-all security criterion

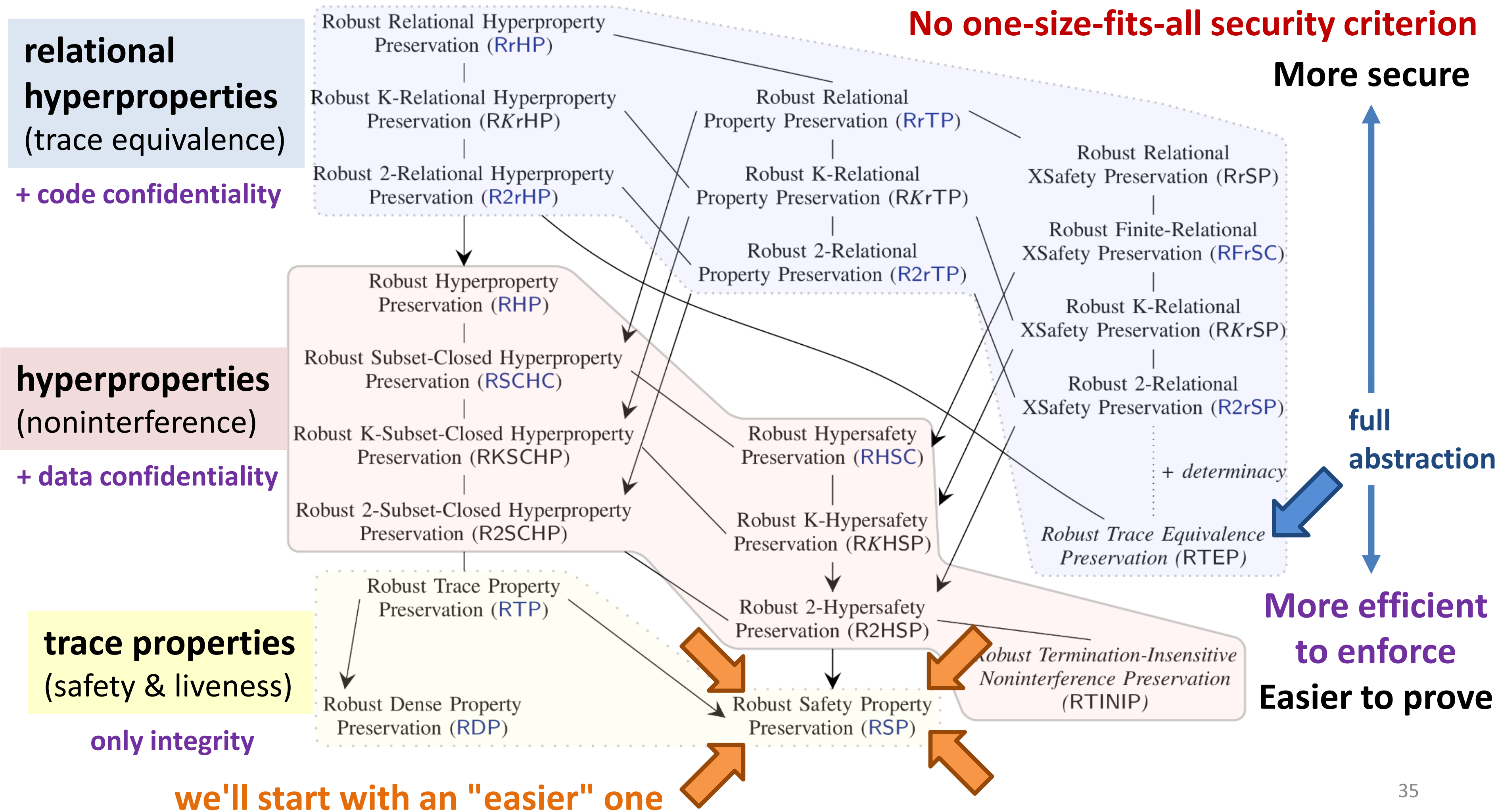
More secure

full abstraction

More efficient to enforce

Easier to prove

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



Robust Safety Preservation [CSF'19]

Robust Safety Preservation [CSF'19]

∀source program.

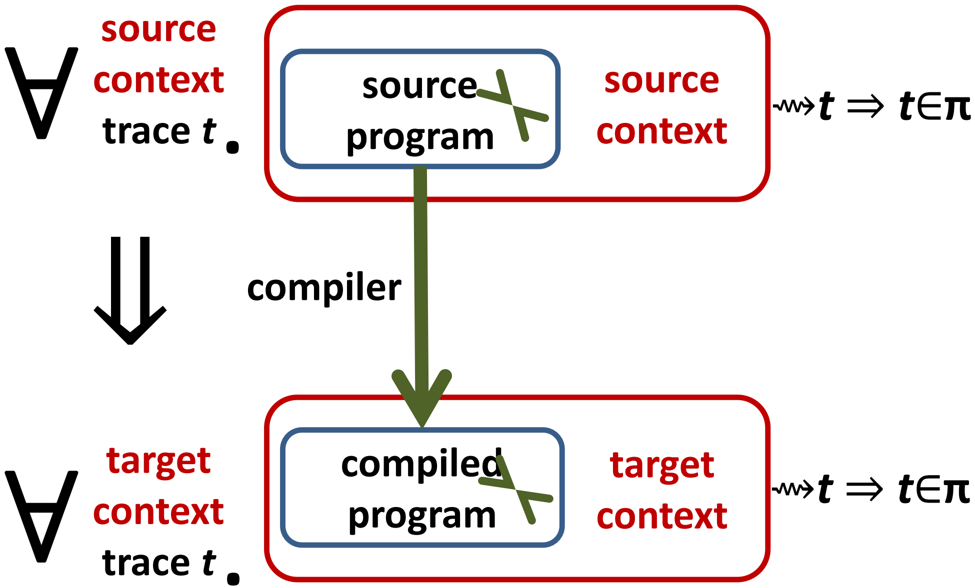
∀π safety property.

∀ source context trace t. (source program ⋈ source context) ⇝ t ⇒ t∈π

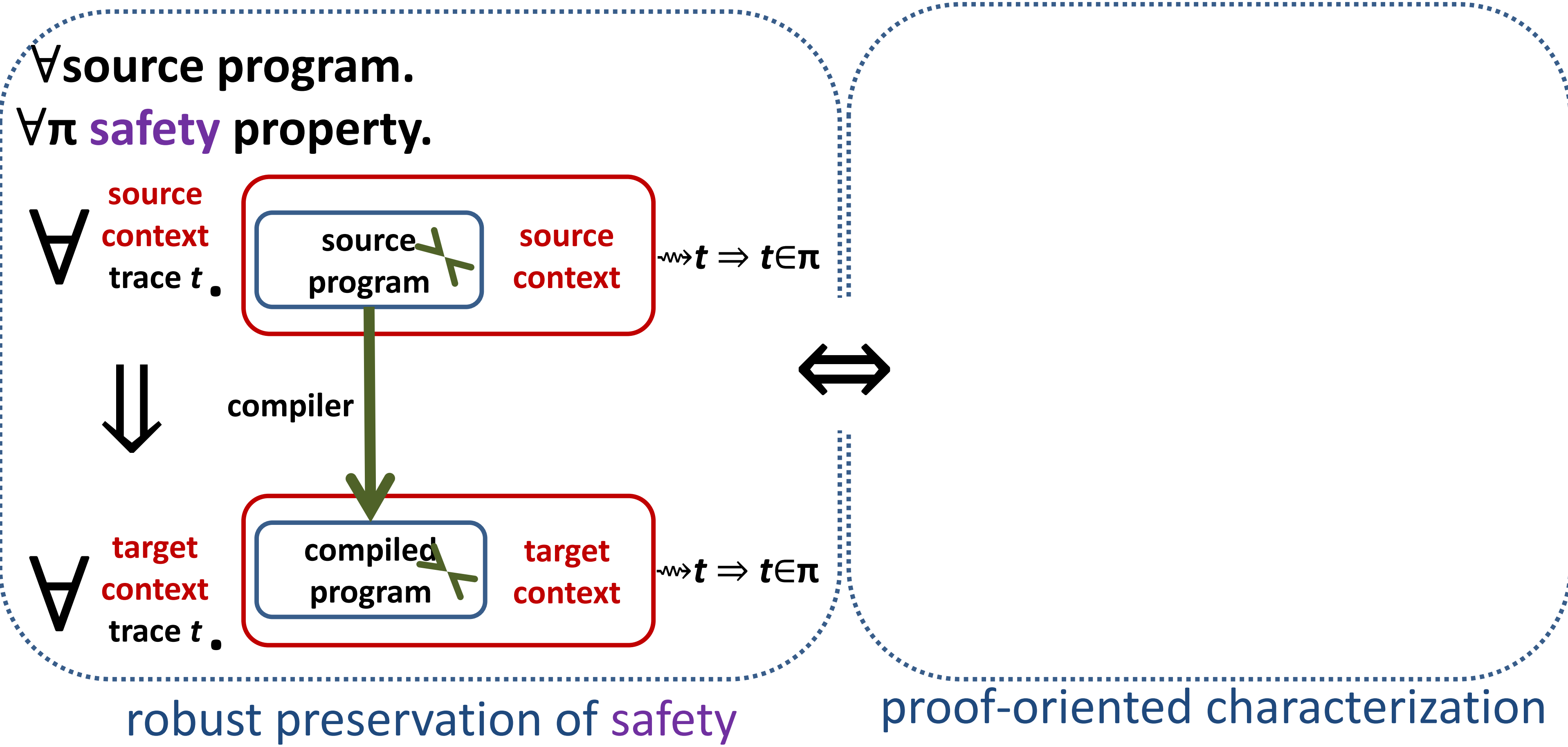
⇓

Robust Safety Preservation [CSF'19]

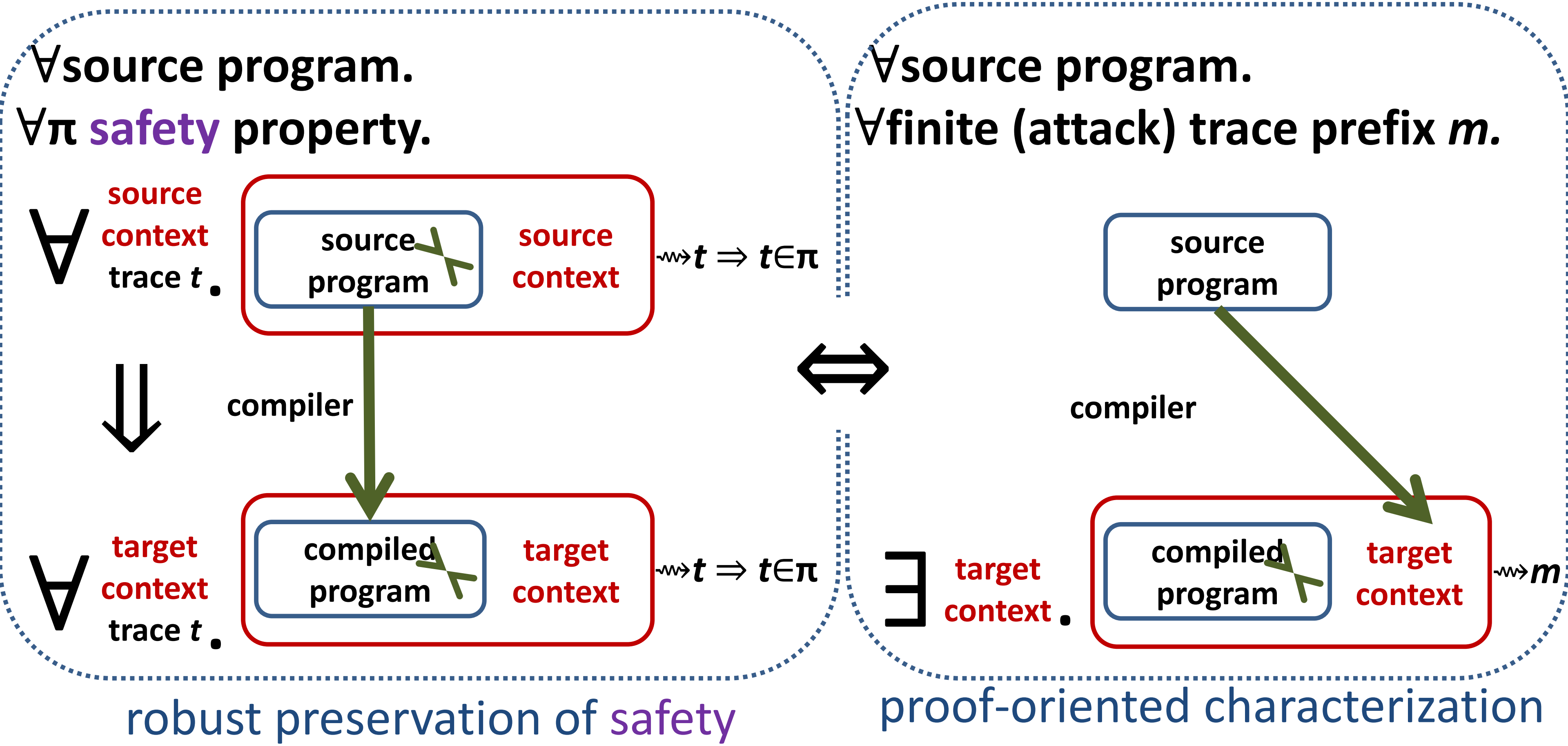
∀source program.
∀π safety property.



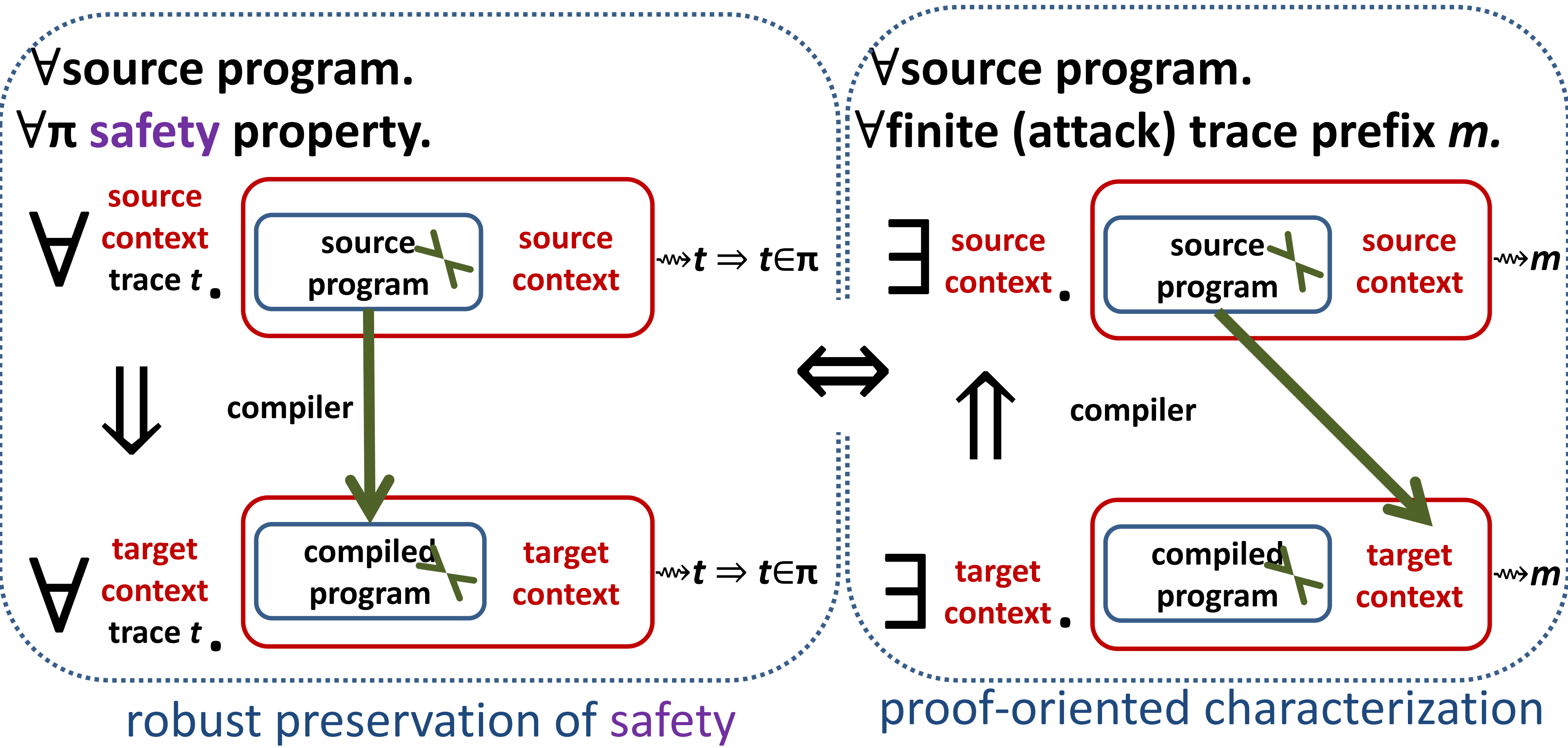
Robust Safety Preservation [CSF'19]



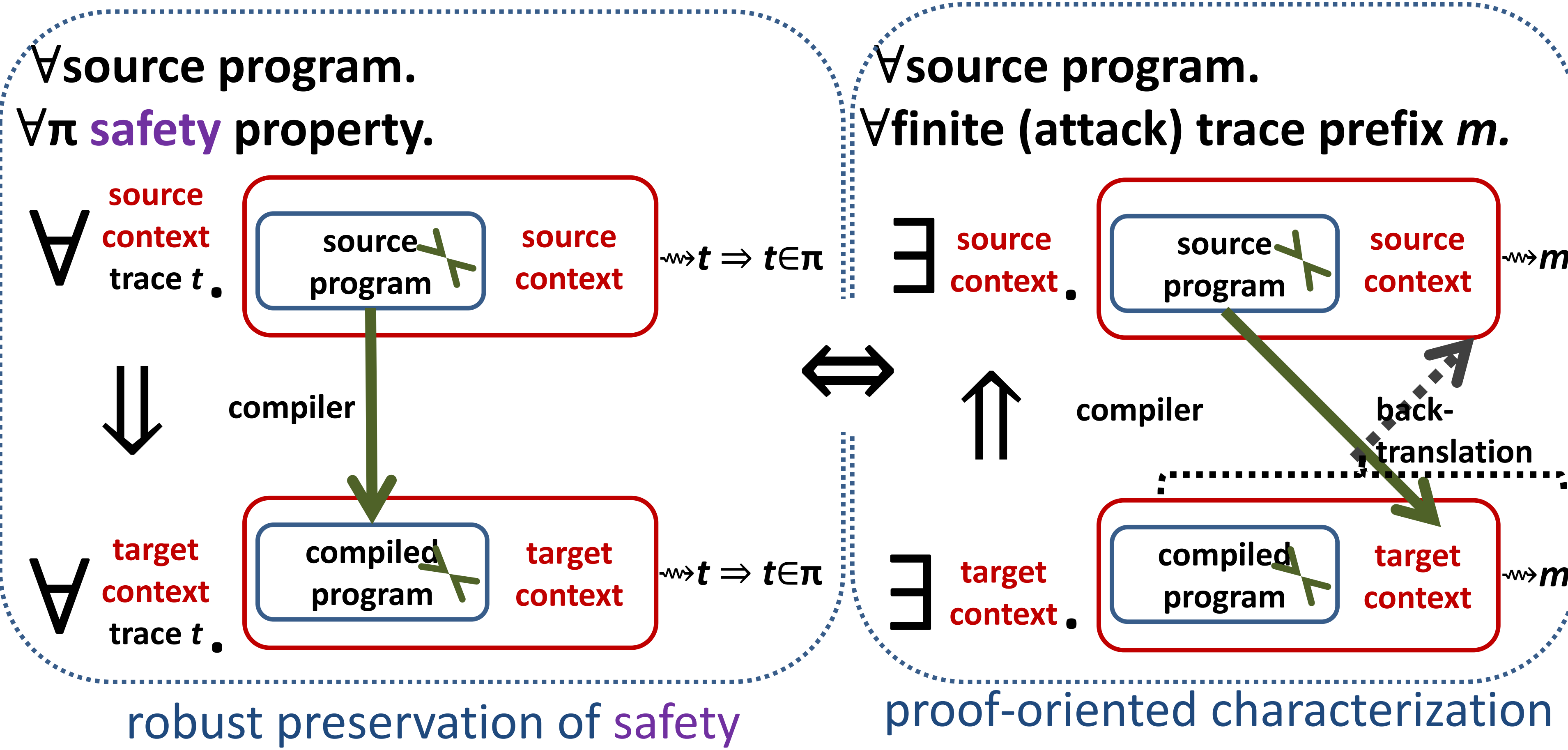
Robust Safety Preservation [CSF'19]



Robust Safety Preservation [CSF'19]



Robust Safety Preservation [CSF'19]



1. Security Goal

- **Question A:**
  What does it mean to securely compile a secure source program against linked adversarial target-level code?

robust safety preservation

1. Security Goal

- **Question A:**
  What does it mean to securely compile a <u>secure</u> source program against linked adversarial target-level code?

  robust safety preservation

- **Question B:**
  What does it mean for a compilation chain for <u>vulnerable</u> C compartments to be secure?

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

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

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

Extra challenges in defining secure compilation for <u>vulnerable</u> 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 <u>vulnerable</u> 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 <u>vulnerable</u> 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**

Why is formalizing security for this hard?

- **We want source-level security reasoning principles**
  - easier to **reason about security of the C source language** if an application is compartmentalized

Why is formalizing security for this hard?

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

Why is formalizing security for this hard?

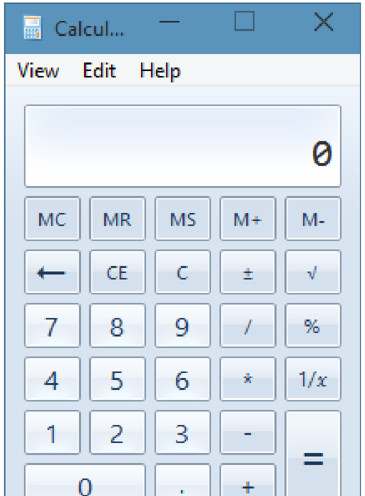
- **We want source-level security reasoning principles**
  - easier to **reason about security of the C source language** if an application is compartmentalized
- **... even in the presence of undefined behavior**
  - can't be expressed at all by source language semantics!
  - **what does the following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

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

Why is formalizing security for this hard?

- **We want source-level security reasoning principles**
  - easier to **reason about security of the C source language** if an 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;
}
```

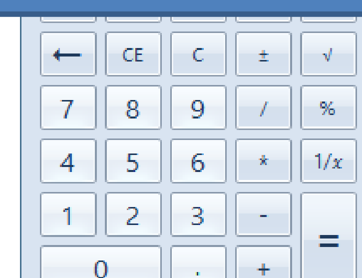


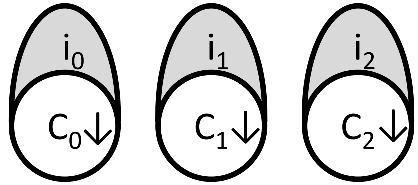
Why is formalizing security for this hard?

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

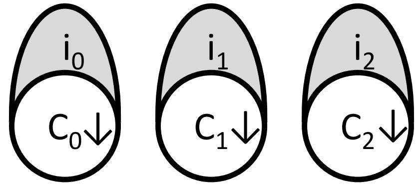
Key idea: secure compartmentalization <u>restricts the scope</u> of undefined behavior:
(1) **spatially**, to only the compartment encountering it
(2) **temporally**, only give up on a compartment once compromised

```
    strcpy(c, argv[1]);
    return 0;
}
```



Security definition: If 

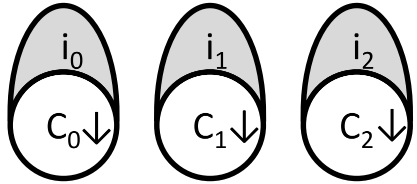
$\rightsquigarrow_{\text{machine}}$ m then

Security definition: If 

$\rightsquigarrow_{machine}$ m then

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

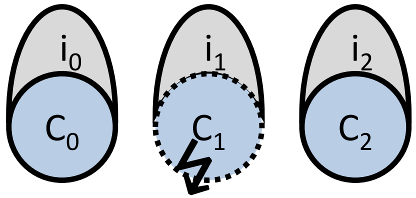
Security definition: If



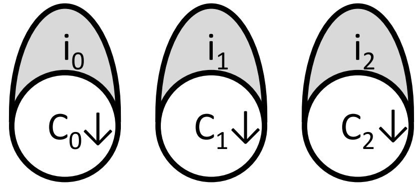
$\rightsquigarrow_{machine}$ *m* then

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

(1)

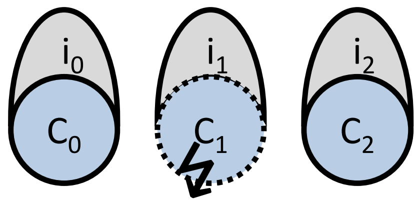


$\rightsquigarrow_{source}$ m_1·Undef(C_1)

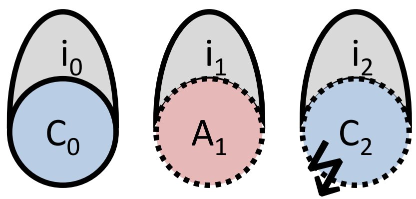
Security definition: If 

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

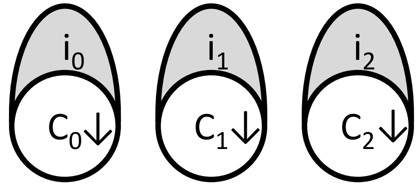
∃ a sequence of compartment compromises explaining finite IO trace prefix m in the source language, for instance $m = m_1 \cdot m_2 \cdot m_3$ and

(1) 

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

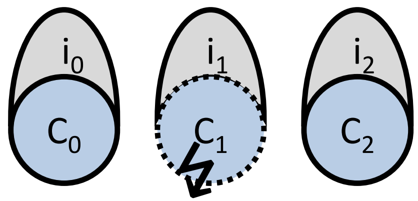
(2) $\exists A_1.$ 

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

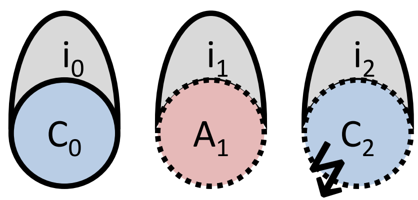
Security definition: If 

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

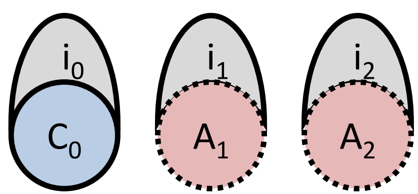
∃ a sequence of compartment compromises explaining finite IO trace prefix m in the source language, for instance $m=m_1{\cdot}m_2{\cdot}m_3$ and

(1) 

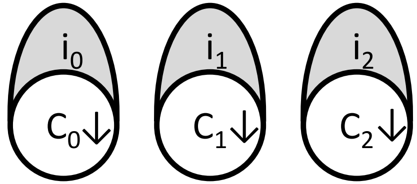
$\rightsquigarrow_{\text{source}}$ $m_1{\cdot}\text{Undef}(C_1)$

(2) $\exists A_1$. 

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

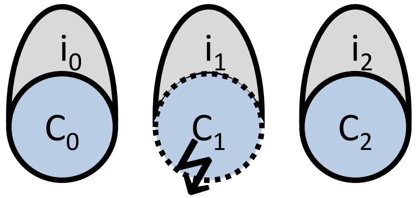
(3) $\exists A_2$. 

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

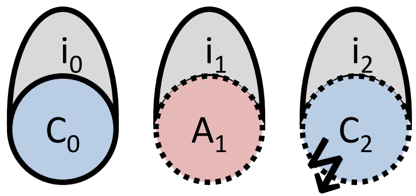
Security definition: If 

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

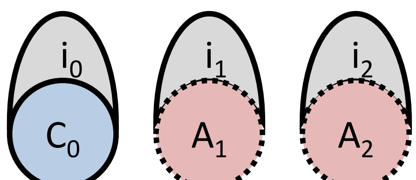
∃ a sequence of compartment compromises explaining finite IO trace prefix m in the source language, for instance $m=m_1{\cdot}m_2{\cdot}m_3$ and

(1) 

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

(2) $\exists A_1.$ 

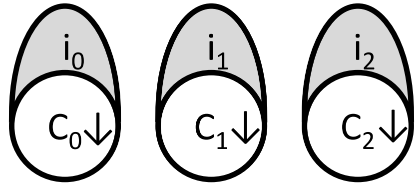
$\rightsquigarrow_{\text{source}}$ $m_1{\cdot}m_2{\cdot}\text{Undef}(C_2)$

(3) $\exists A_2.$ 

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

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

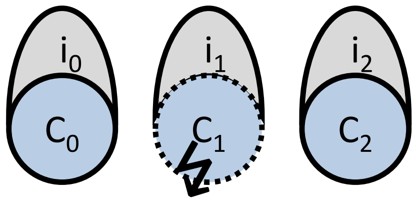
Security definition: If



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

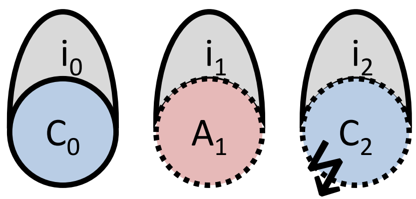
∃ a sequence of compartment compromises explaining finite IO trace prefix *m* in the source language, for instance $m = m_1 \cdot m_2 \cdot m_3$ and

(1)



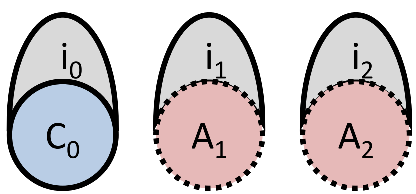
$\rightsquigarrow_{\text{source}}\ m_1 \cdot \text{Undef}(C_1)$

(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 prefix *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]

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

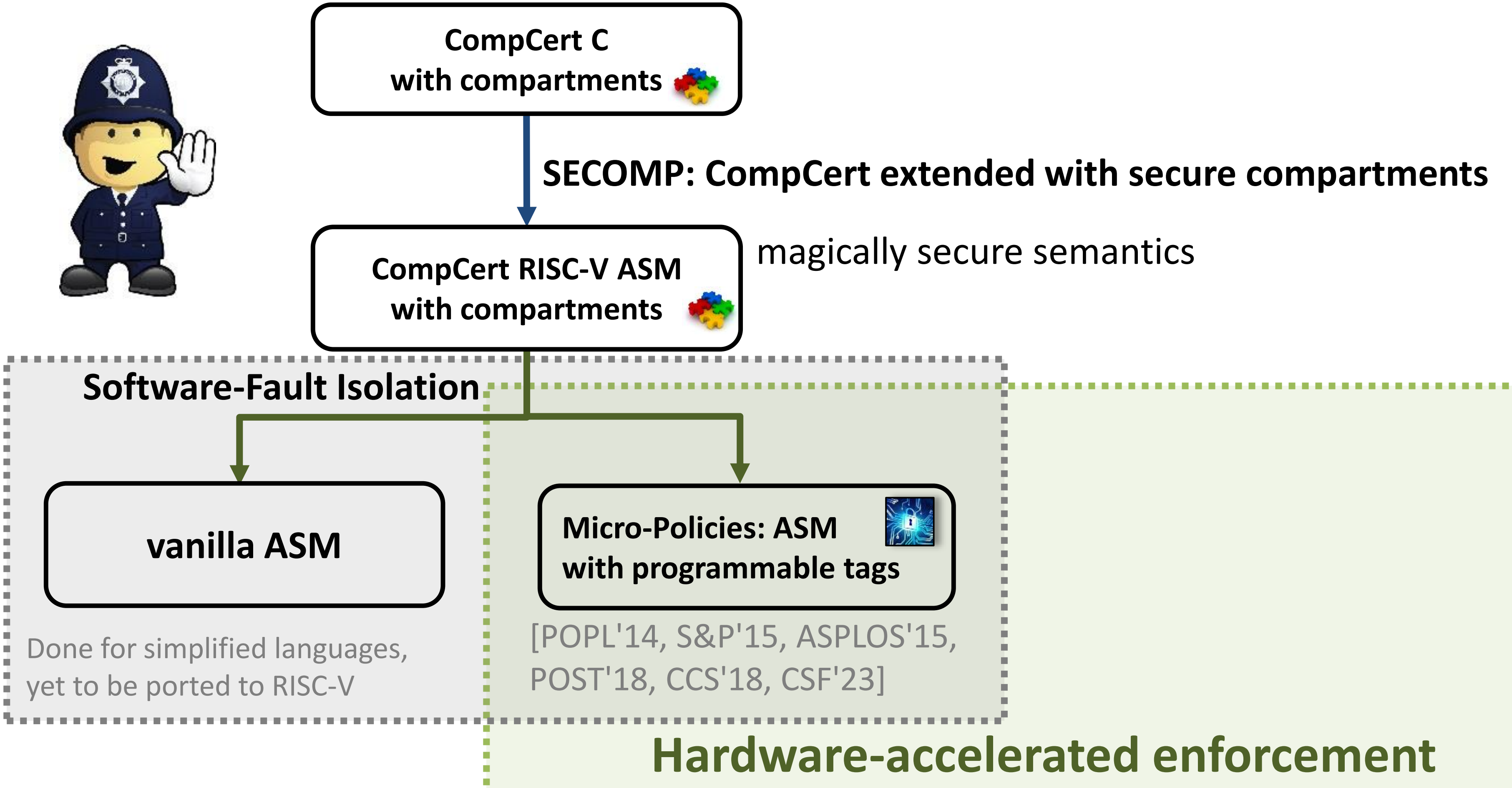
vanilla ASM

Micro-Policies: ASM with programmable tags

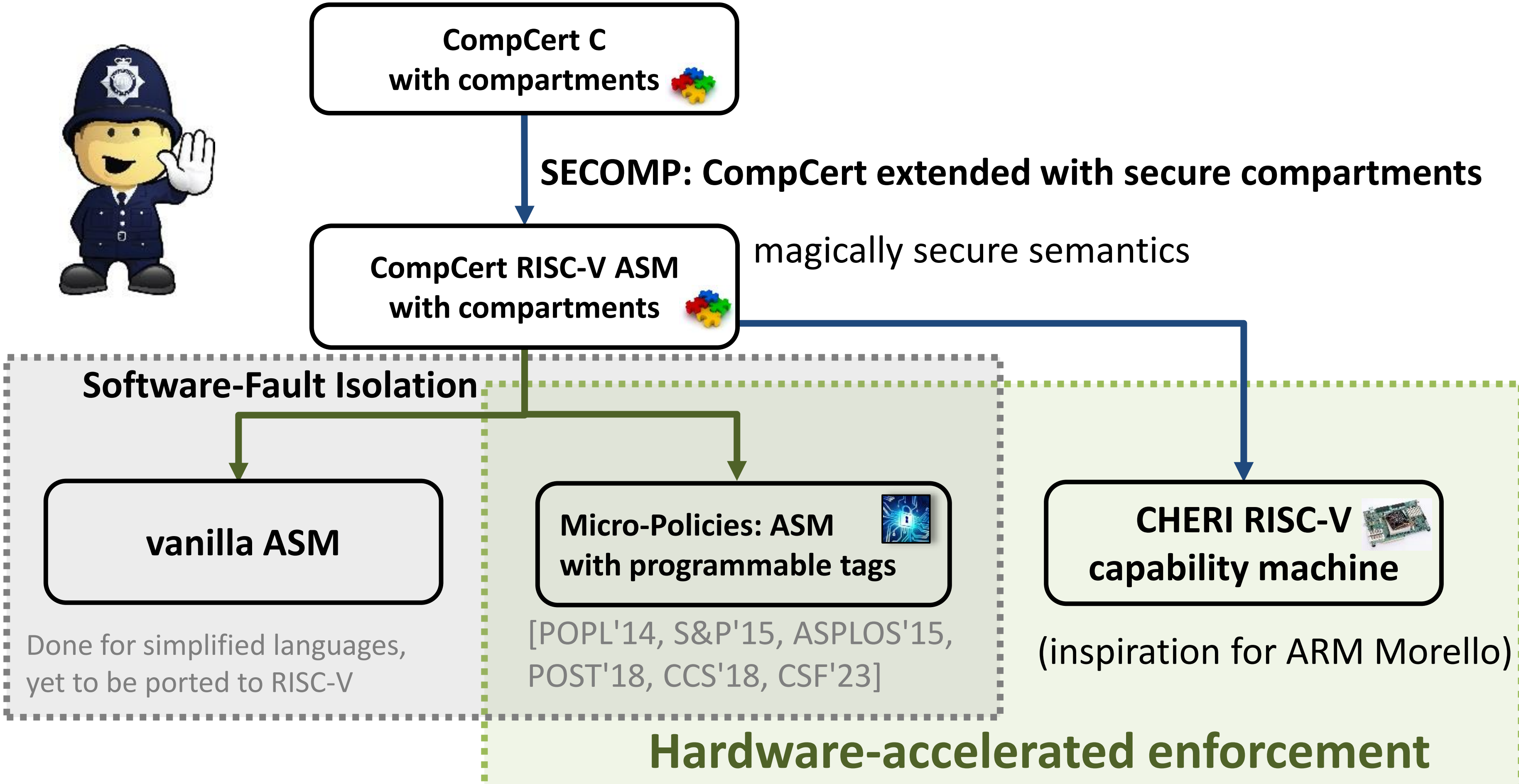
[POPL'14, S&P'15, ASPLOS'15, POST'18, CCS'18, CSF'23]

Hardware-accelerated enforcement

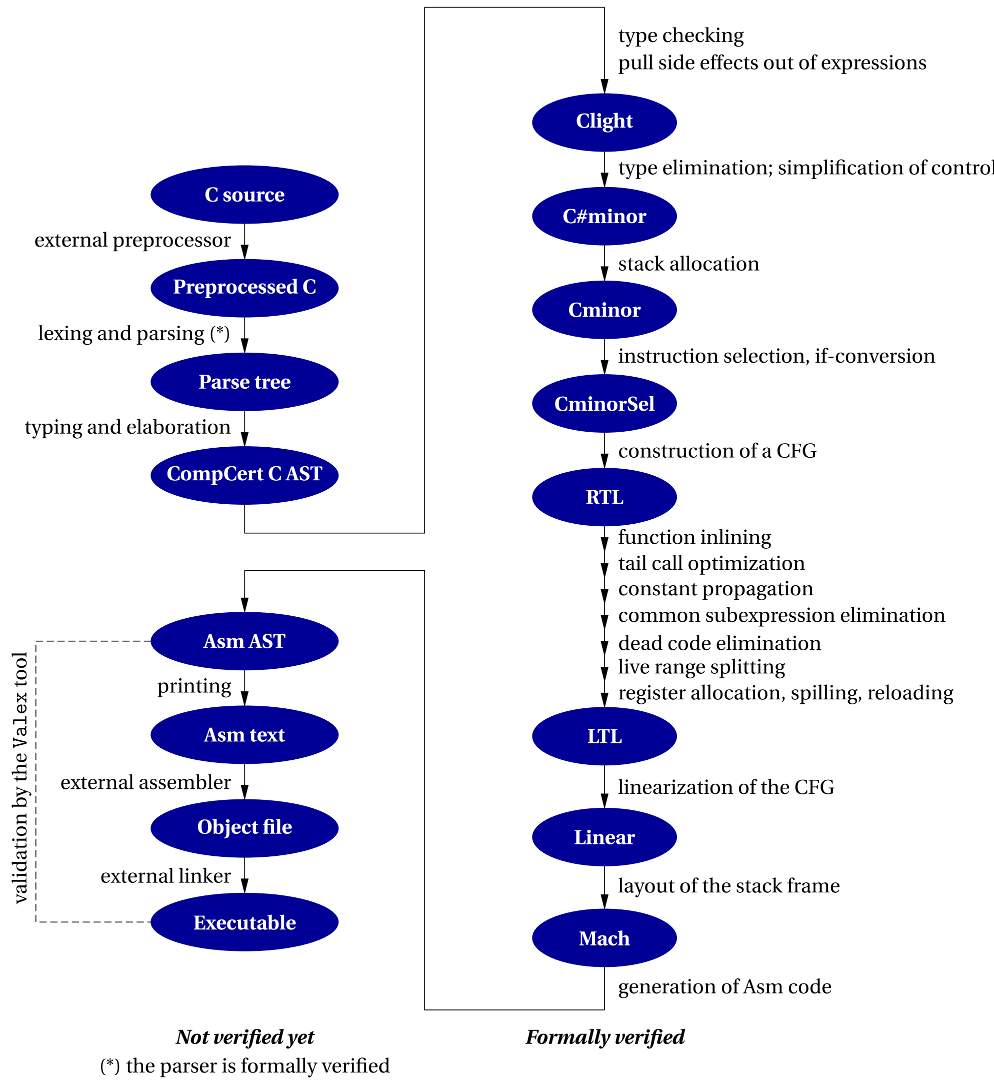
2. Security Enforcement



2. Security Enforcement

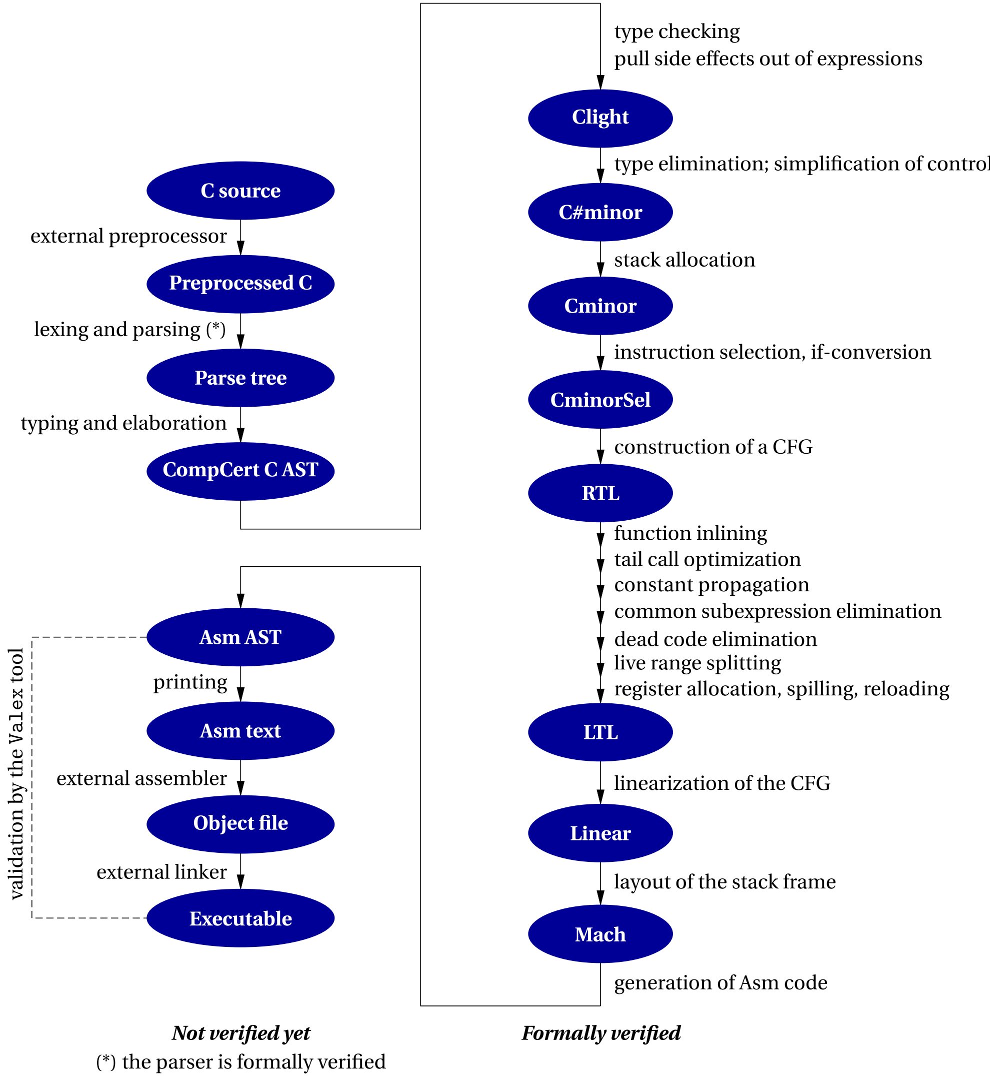


CompCert extended with compartments



CompCert extended with compartments

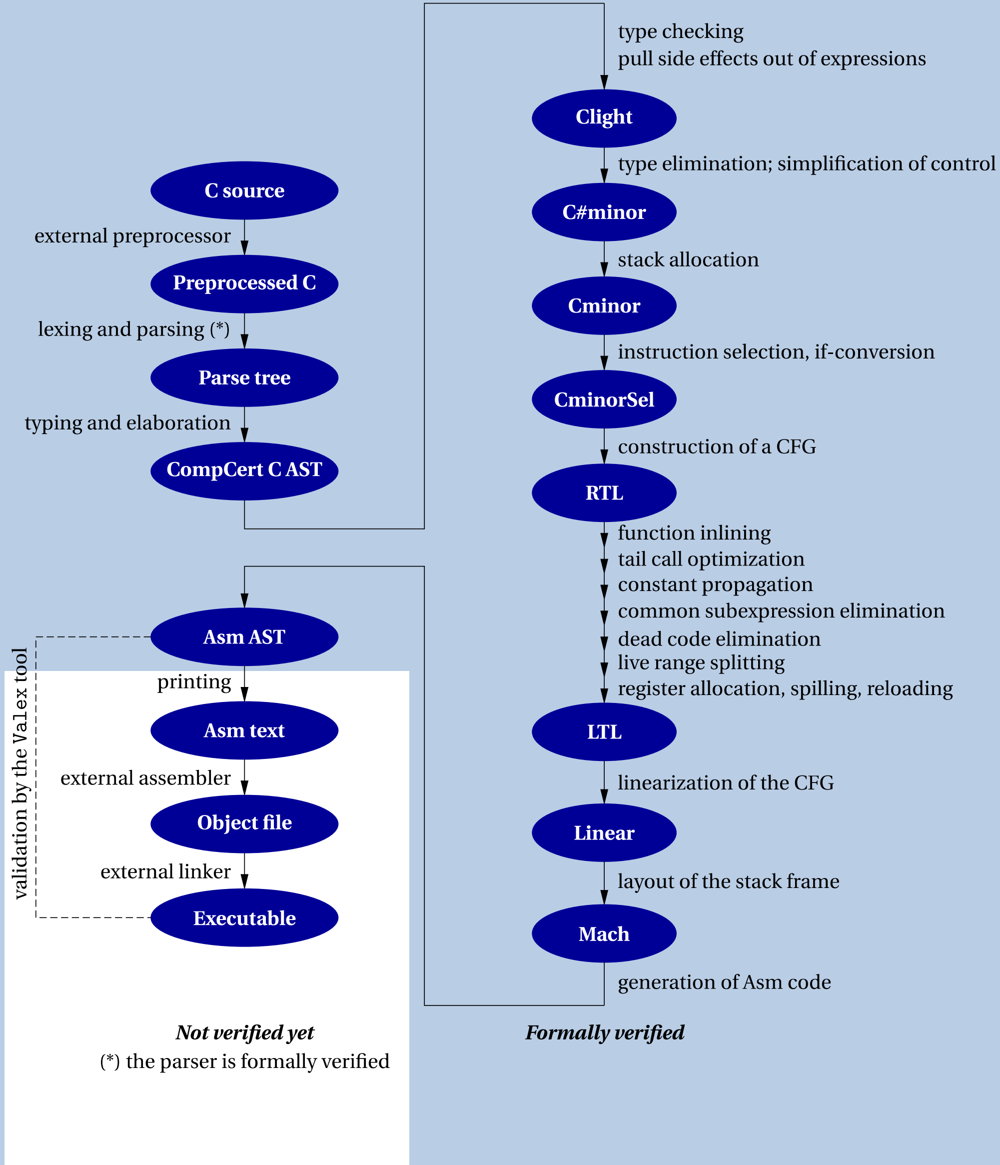
mutually distrustful,
with clearly specified interfaces,
interacting via procedure calls



CompCert extended with compartments

mutually distrustful,
with clearly specified interfaces,
interacting via procedure calls

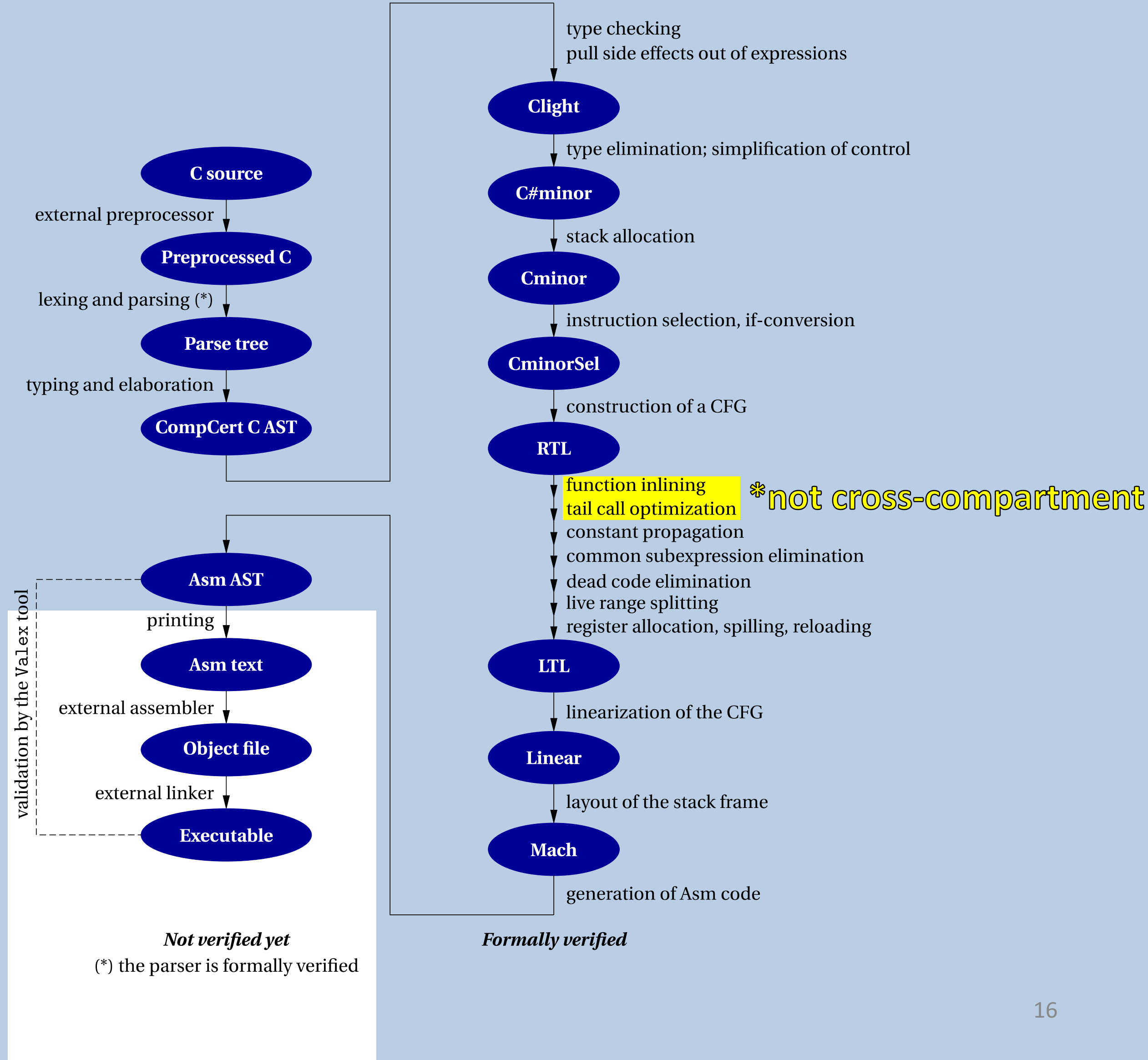
all 19 verified compilation passes*
from Clight to RISC-V ASM
(magically secure semantics)



CompCert extended with compartments

mutually distrustful,
with clearly specified interfaces,
interacting via procedure calls

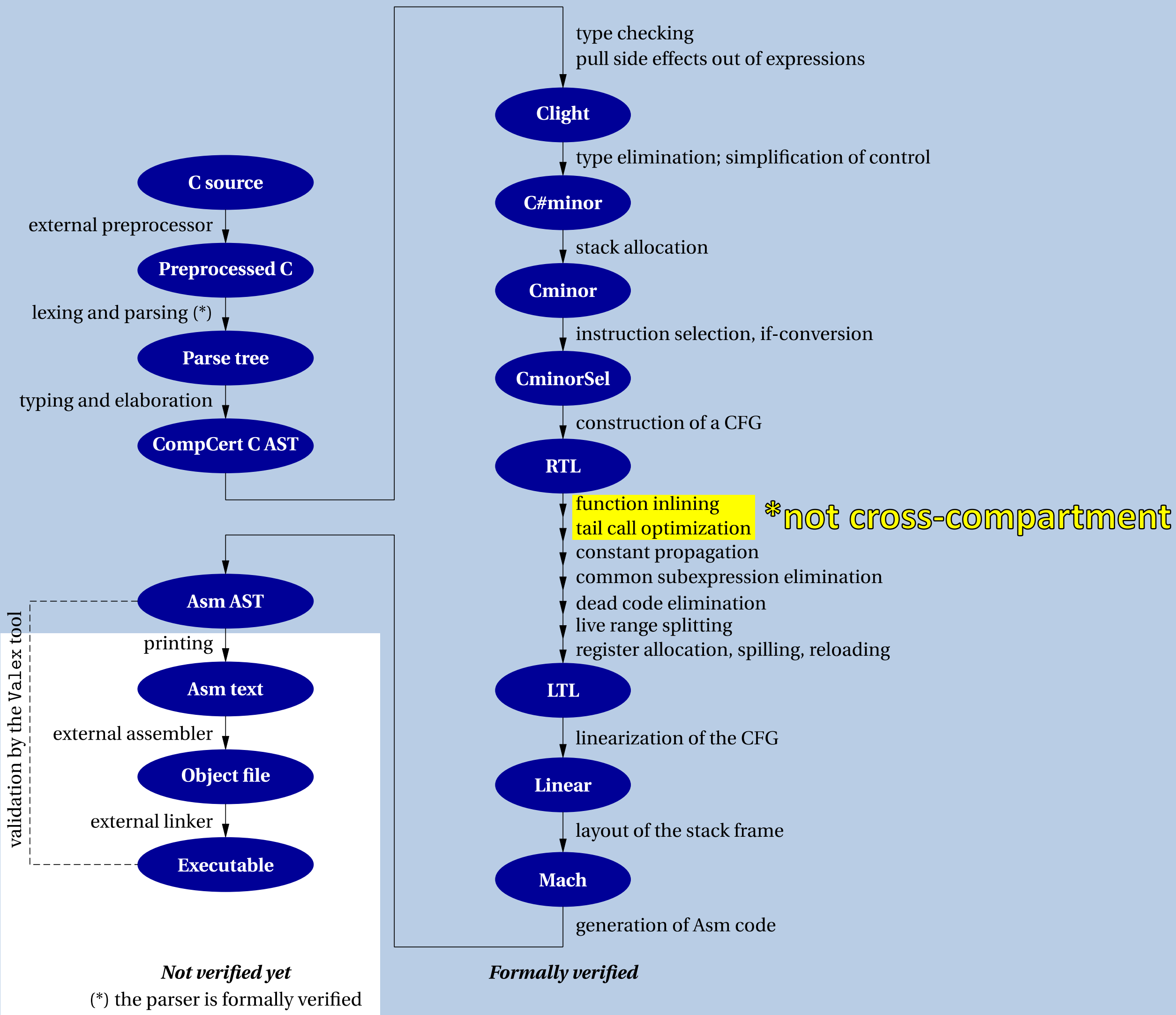
all 19 verified compilation passes*
from Clight to RISC-V ASM
(magically secure semantics)



CompCert extended with compartments

mutually distrustful,
with clearly specified interfaces,
interacting via procedure calls

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

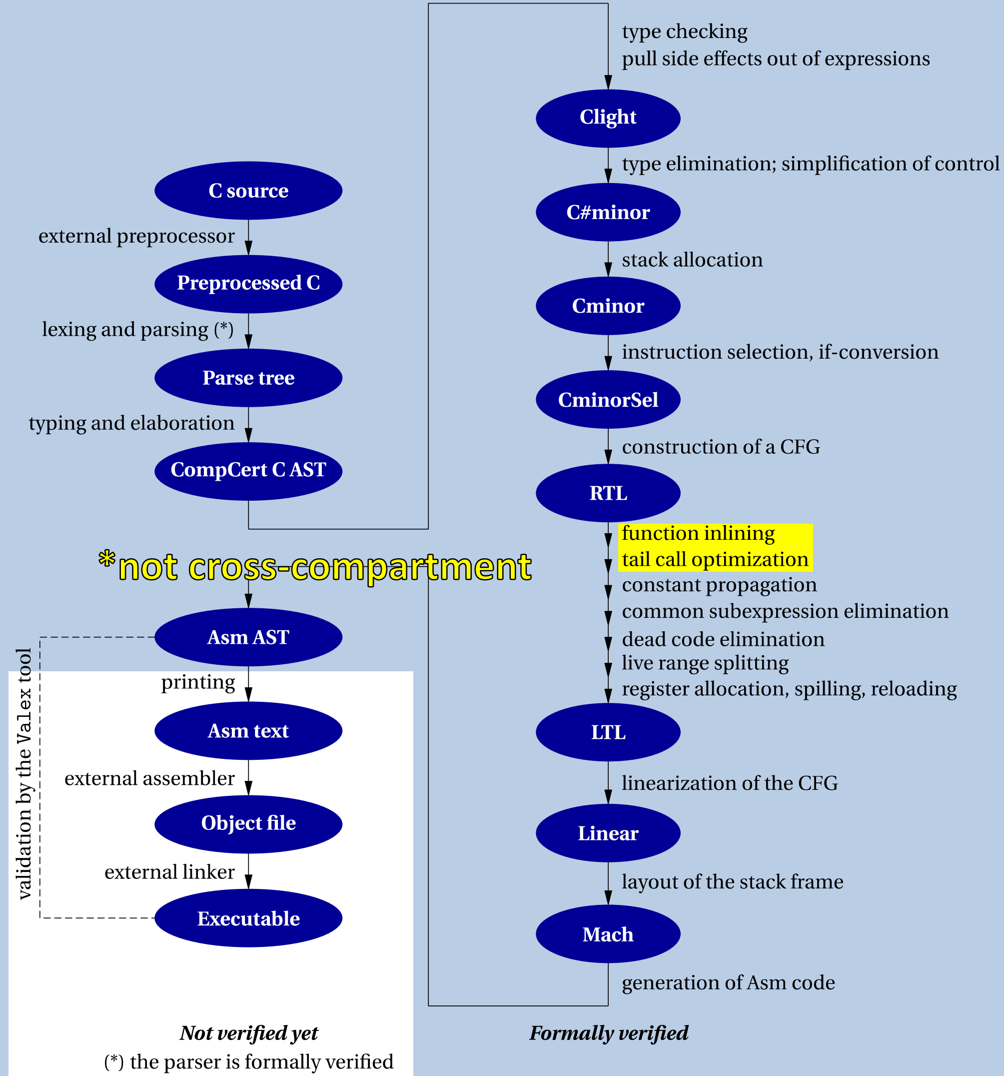


CompCert extended with compartments

mutually distrustful,
with clearly specified interfaces,
interacting via procedure calls

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

extended compiler correctness
18K LoC, only 13.6% change,
reused for security



Capabilities Backend

- **Targeting variant of CHERI RISC-V capability machine**
  - capabilities = unforgeable pointers with base and bounds

Capabilities Backend

- **Targeting variant of CHERI RISC-V capability machine**
  - capabilities = unforgeable pointers with base and bounds
  - we only enforce **compartment isolation**, not memory safety

Capabilities Backend

- **Targeting variant of CHERI RISC-V capability machine**
  - capabilities = unforgeable pointers with base and bounds
  - we only enforce **compartment isolation**, not memory safety
- **Secure and efficient calling convention enforcing stack safety**
  [Aïna Linn Georges et al, Le temps de cerises, OOPSLA 2022]

Capabilities Backend

- **Targeting variant of CHERI RISC-V capability machine**
  - capabilities = unforgeable pointers with base and bounds
  - we only enforce **compartment isolation**, not memory safety
- **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 variant of CHERI RISC-V capability machine**
  – capabilities = unforgeable pointers with base and bounds
  – we only enforce **compartment isolation**, not memory safety
- **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

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 decades [Devriese et al. POPL'18]
  - 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 decades [Devriese et al. POPL'18]
  - 250 pages of proof on paper even for toy compilers
- we work on **more scalable proof techniques**

3. Security Proof

Proving that our compilation chain for C compartments achieves secure compilation

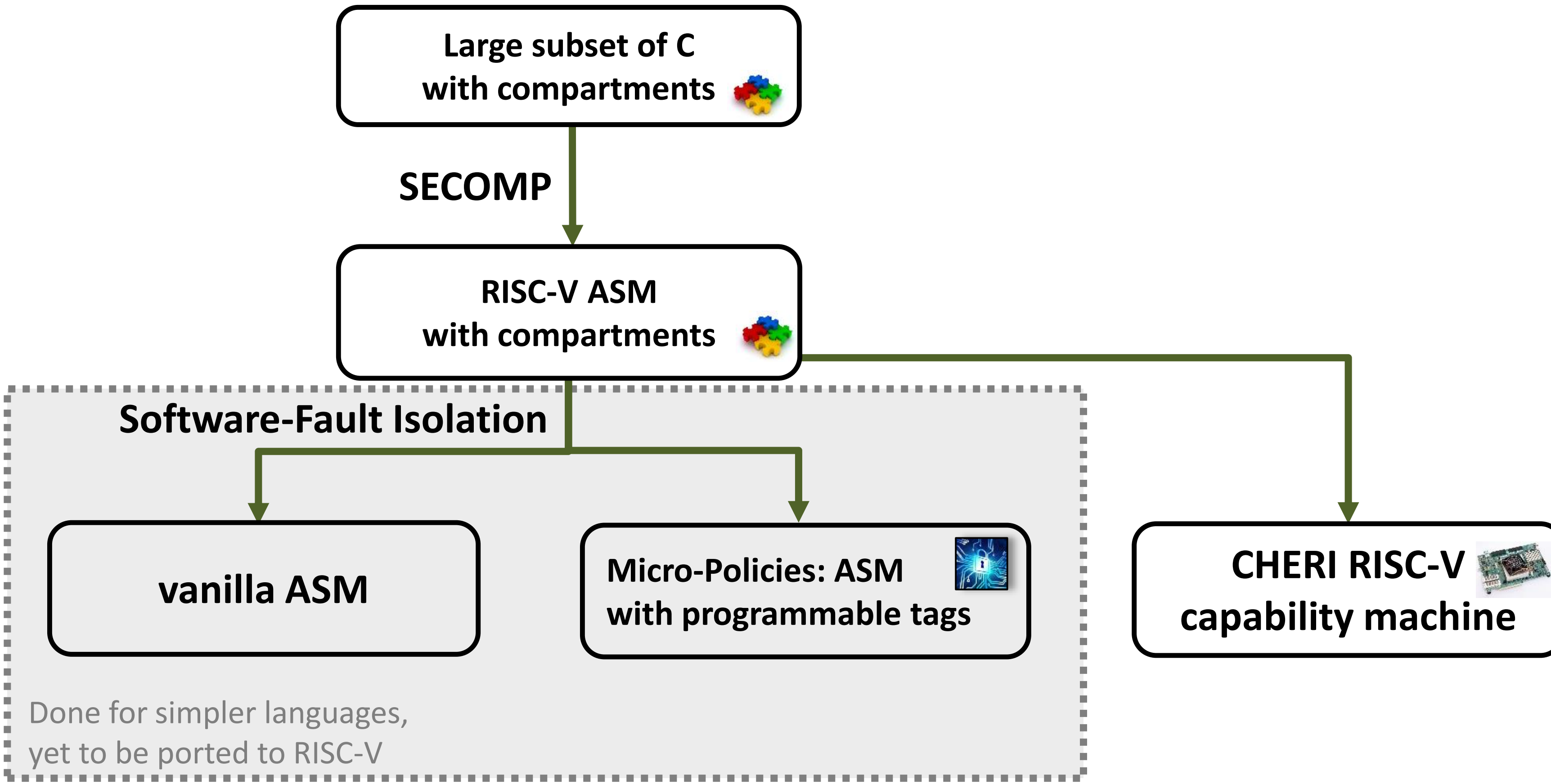
- such proofs generally **very difficult and tedious**
  - wrong full abstraction conjecture survived decades [Devriese et al. POPL'18]
  - 250 pages of proof on paper even for toy compilers
- we work on **more scalable proof techniques**
- we do **machine-checked proofs** in the Coq proof assistant

3. Security Proof

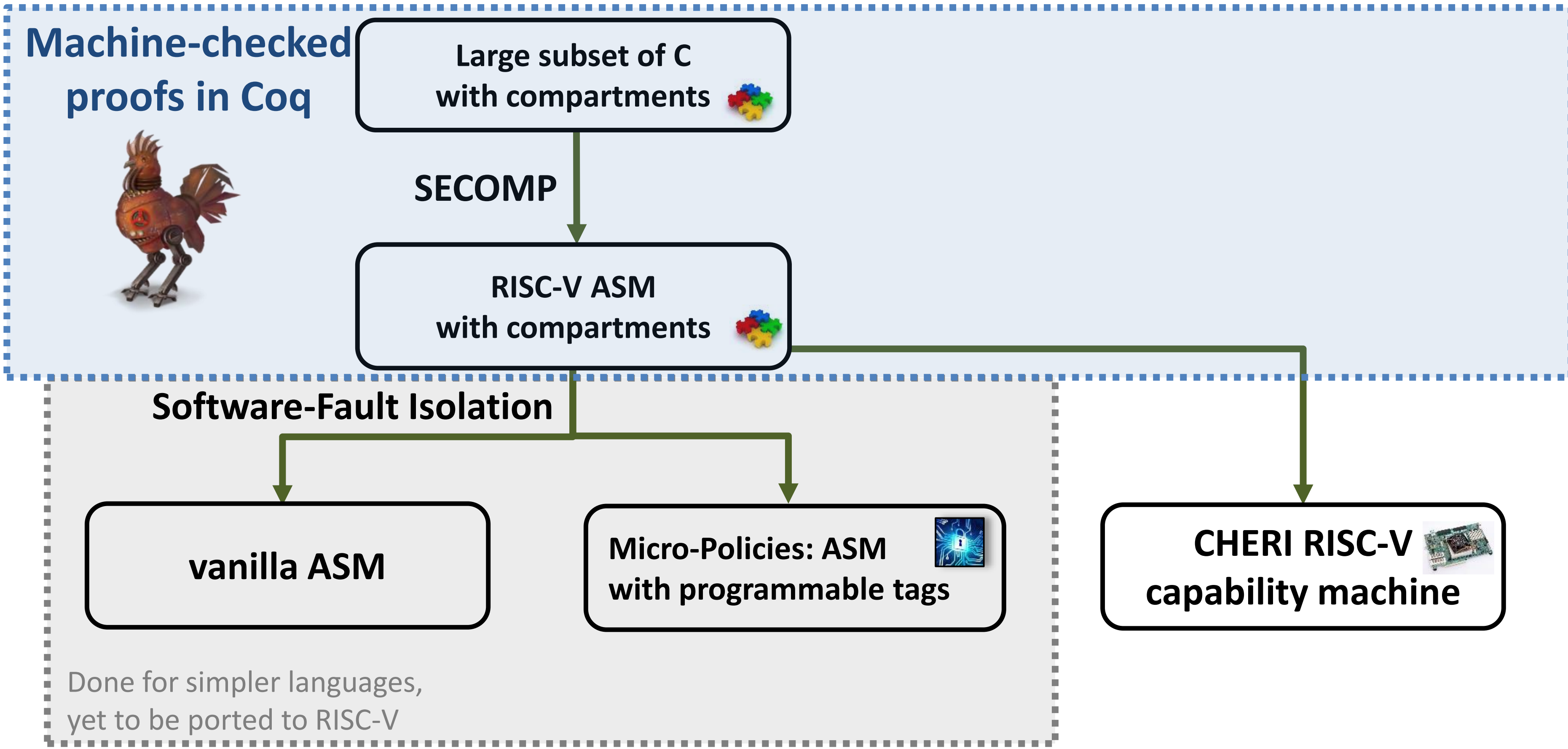
Proving that our compilation chain for C compartments achieves secure compilation

- such proofs generally **very difficult and tedious**
  - wrong full abstraction conjecture survived decades [Devriese et al. POPL'18]
  - 250 pages of proof on paper even for toy compilers
- we work on **more scalable proof techniques**
- we do **machine-checked proofs** in the Coq proof assistant
- as stopgap we use **property-based testing** [POPL'17, ICFP'13, ITP'15, JFP'16]
    - to find wrong conjectures early
    - to deal with the parts we couldn't (yet) verify

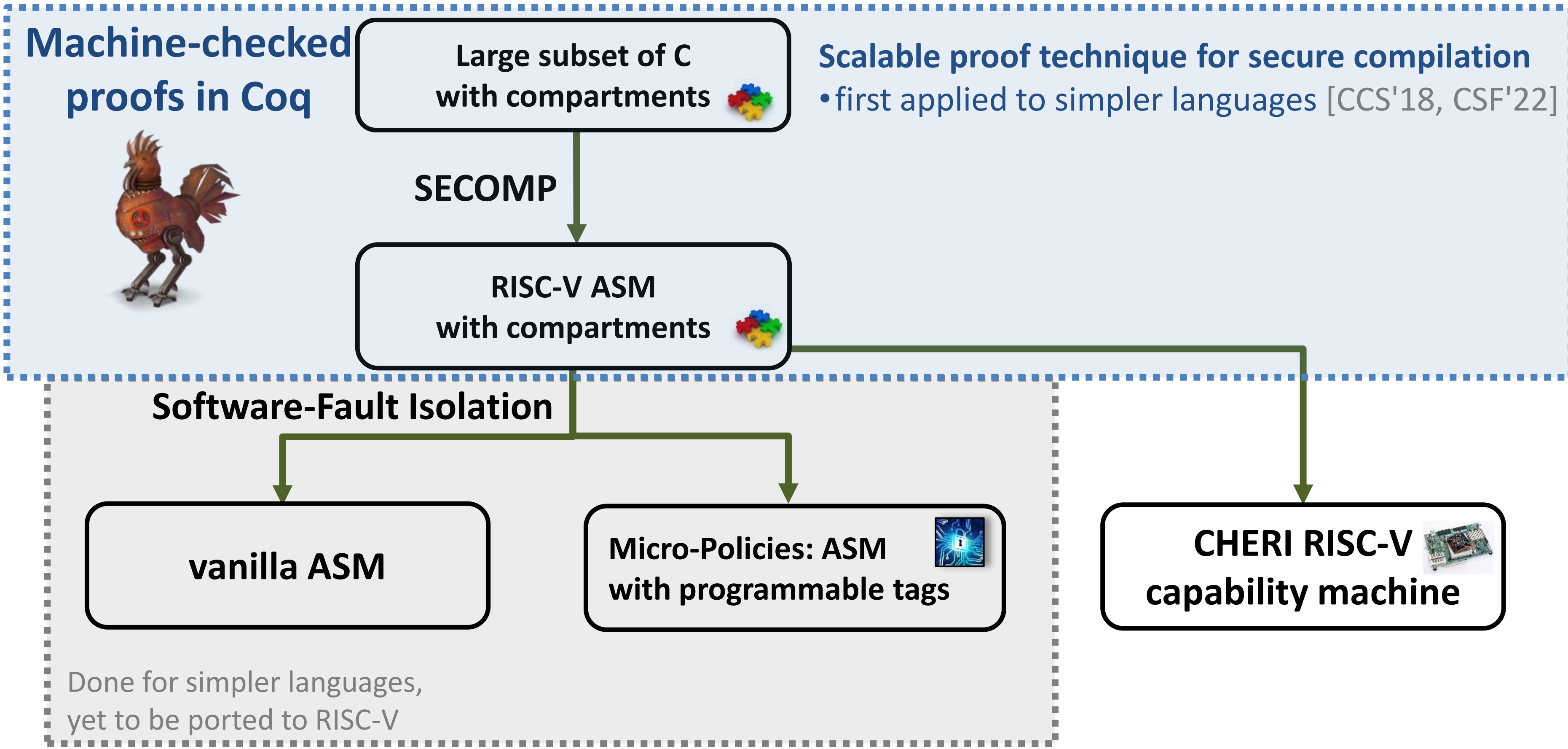
Secure Compilation Proofs in Coq



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]
  - reuses extended CompCert correctness proof (~130K LoC)
  - verified strong secure compilation property (+ ~43K 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]
  - reuses extended CompCert correctness proof (~130K LoC)
  - verified strong secure compilation property (+ ~43K 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]
  - reuses extended CompCert correctness proof (~130K LoC)
  - verified strong secure compilation property (+ ~43K 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]
  - reuses extended CompCert correctness proof (~130K LoC)
  - verified strong secure compilation property (+ ~43K 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]
  - reuses extended CompCert correctness proof (~130K LoC)
  - verified strong secure compilation property (+ ~43K 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

Big verification challenge for the future

Open problem: verified backends

Open problem: verified backends

- **Currently we only implemented a 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 a 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 a 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**
  - e.g. 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

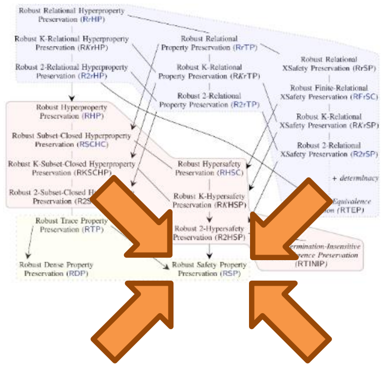
Extending proof technique in other ways

- **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]
  - A Semantic Approach to Robust Property Preservation [Niklas Mück et al, PriSC'25]
    - solves this (and previous) problem using DimSum multi-language semantics framework based on Iris

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]
  - A Semantic Approach to Robust Property Preservation [Niklas Mück et al, PriSC'25]
    - solves this (and previous) problem using DimSum multi-language semantics framework based on Iris

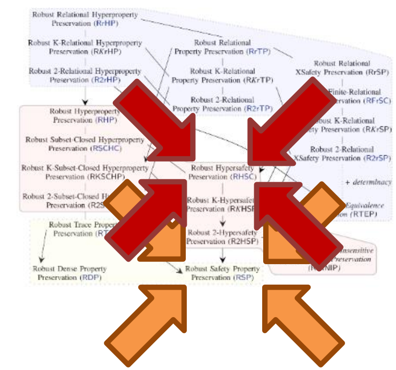
- **Beyond preserving safety against adversarial contexts**



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]
  - A Semantic Approach to Robust Property Preservation [Niklas Mück et al, PriSC'25]
    - solves this (and previous) problem using DimSum multi-language semantics framework based on Iris

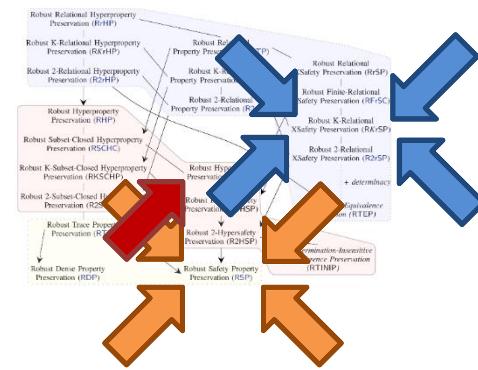
- **Beyond preserving safety against adversarial contexts**
  - towards preserving **hyperproperties** (data confidentiality)



Extending proof technique in other ways

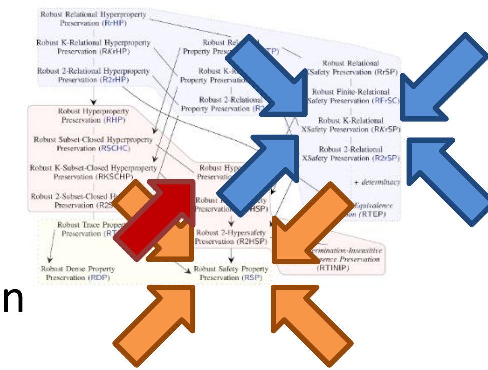
- **Fine-grained dynamic memory sharing** by **capability passing (on CHERI or Morello)**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]
  - A Semantic Approach to Robust Property Preservation [Niklas Mück et al, PriSC'25]
    - solves this (and previous) problem using DimSum multi-language semantics framework based on Iris

- **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]
  - A Semantic Approach to Robust Property Preservation [Niklas Mück et al, PriSC'25]
    - solves this (and previous) problem using DimSum multi-language semantics framework based on Iris



- **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 + ongoing work first presented at PriSC'21]

Enforcement tricky beyond safety

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

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)
    - **speculative constant time (chapter in new Security Foundations draft volume)**

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**
- **Started looking into Spectre defenses compilers can insert**
  - **Speculative Load Hardening** (implemented in LLVM + selective variant in Jasmin DSL)
    - **<u>speculative constant time</u> (chapter in new Security Foundations draft volume)**
  - **Strong/Ultimate SLH and New Flexible SLH variant enforce <u>relative security</u> (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)
    - **speculative constant time (chapter in new Security Foundations draft volume)**
  - **Strong/Ultimate SLH and New Flexible SLH variant enforce relative security (paper soon)**
  - **Future work: property-based testing for scaling this up to LLVM and x86/ARM**

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**

SPECTRE

- **Started looking into Spectre defenses compilers can insert**
  - **Speculative Load Hardening** (implemented in LLVM + selective variant in Jasmin DSL)
    - **speculative constant time (chapter in new Security Foundations draft volume)**
  - **Strong/Ultimate SLH and New Flexible SLH variant enforce relative security (paper soon)**
  - **Future work: property-based testing for scaling this up to LLVM and x86/ARM**
- **Combining this with compartmentalization practically interesting**
  - Especially for languages like Wasm, which are used for same-process isolation

Protecting higher-level abstractions

(than those of the C programming language)

Protecting higher-level abstractions

(than those of the C programming language)

- **Securely Compiling Verified F* Programs With IO** [Cezar Andrici et al, POPL'24]
  - using reference monitoring and higher-order contracts
  - first step towards formally secure F*-OCaml interoperability? (lots of steps left though :)

Protecting higher-level abstractions

(than those of the C programming language)

- **Securely Compiling Verified F* Programs With IO** [Cezar Andrici et al, POPL'24]
  - using reference monitoring and higher-order contracts
  - first step towards formally secure F*-OCaml interoperability? (lots of steps left though :)
  - preserving **all relational hyperproperties** against adversarial contexts