Formally verifying a secure compilation chain for unsafe C components

Cătălin Hrițcu
MPI-SP, Bochum

Joint work with Carmine Abate, **Arthur Azevedo de Amorim**, **Roberto Blanco**, Ştefan Ciobâcă, Adrien Durier, **Akram El-Korashy**, Boris Eng, Ana Nora Evans, Guglielmo Fachini, **Deepak Garg**, Théo Laurent, **Marco Patrignani**, Benjamin Pierce, Marco Stronati, Éric Tanter, **Jérémy Thibault**, and Andrew Tolmach

Supported by ERC Starting Grant SECOMP.

Huge security problem:

The C programming language is unsafe

– 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, ...**

– **root cause**, but very challenging to fix:

  - **efficiency**, precision, scalability, backwards compatibility, deployment

Mitigation: fine-grained compartmentalization

- **The C programming language does provide useful abstractions**
  - structured control flow, procedures & interfaces, pointers & shared memory
  - used in most programs, **but not enforced at all during compilation**
  - **add fine-grained components to C: easy to define** and **can naturally interact**
- **Build secure compilation chain that protects these abstractions**
  - all the way down, at component boundaries (so hopefully more efficient)
  - against components dynamically compromised by undefined behavior
- **Target different enforcement mechanisms**
  - SFI, programmable tagged architecture, capability machines, ...

- **Formally verify the security of this compilation chain**

Formally verifying a secure compilation chain for unsafe C components

We've been working on this project for the last 5+ years

This talk

- **how far did we get?**
- **what were the main challenges we had to overcome?**
  - security definitions, enforcement, proof techniques
- **what's left for us to do?** (in the following 5 years?)
- **what are some more general open problems?**

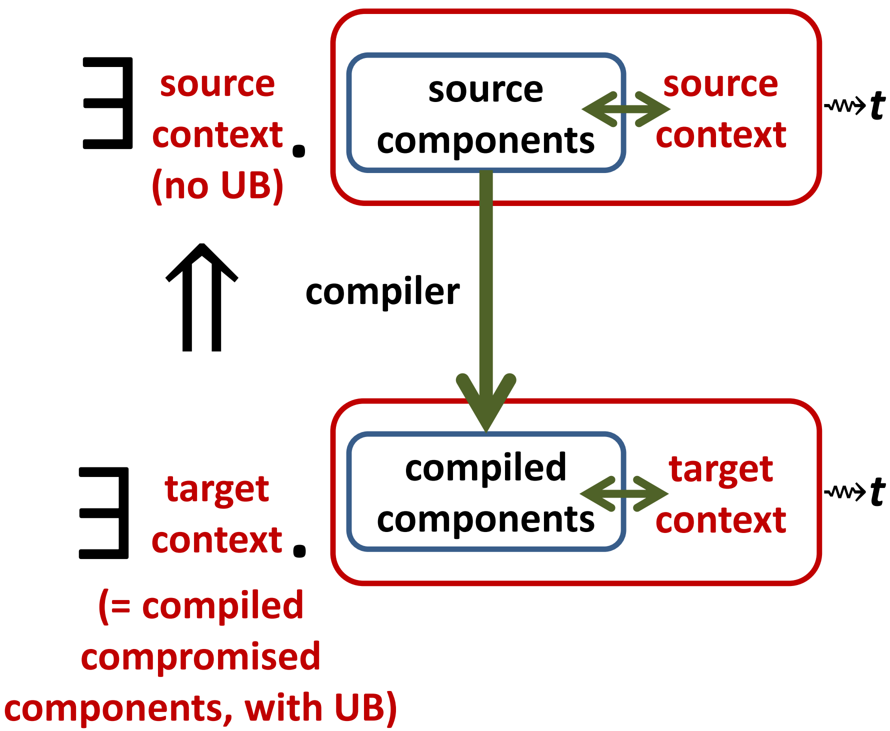
Defining Security Goal

- **Formal definition expressing end-to-end guarantees of secure compilation chain [CCS'18]**
- **Restrict spatial scope** of undefined behavior
  - **mutually-distrustful components**
    - **each component protected from all the others**
- **Restrict temporal scope** of undefined behavior
  - **dynamic compromise**
    - **each component gets guarantees**
      as long as it has not encountered undefined behavior

We reduce this security goal to a variant of:

Robust Safety Preservation

∀(not yet compromised) source components.
∀(bad/attack) finite trace *t*.



OR prefix of t + UB in not yet compromised source component

Intuition: by repeating this game we explain longer and longer prefixes of t in terms of source semantics + component compromise

[When Good Components Go Bad, CCS'18]

Security Enforcement

(prototype secure compilation chain)

component C_2 **{**
private var counter;
private var password;
public procedure get_counter() {
counter := counter + 1;
return counter;
}
}

Compartmentalized unsafe source

Buffers, procedures, **components**

Compartmentalized intermediate machine

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

Programmable tagged architecture

new

Hardware-accelerated enforcement

[POPL'14, Oakland'15, ASPLOS'15, POST'18, CCS'18]

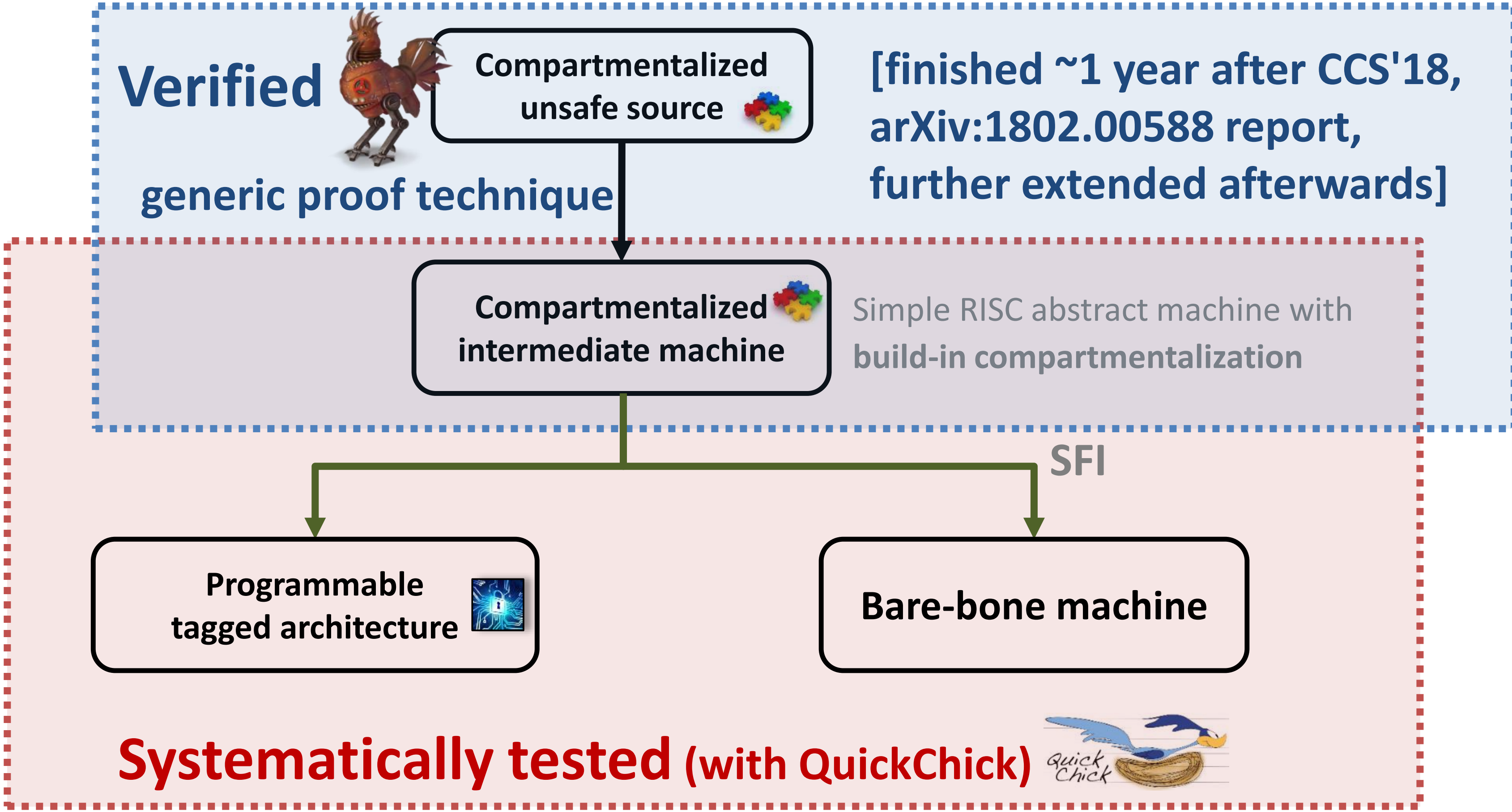
SFI

Bare-bone machine

Proving secure compilation

- formally verifying security of the whole compilation chain
- such proofs **very difficult and tedious**
  - wrong conjectures survived for decades
  - 250 pages of proof on paper for toy compiler
- we propose **more scalable proof techniques**
- **machine-checked proofs** in the Coq proof assistant
  - with **property-based testing** stopgap to find bugs early

Proving and testing our prototype



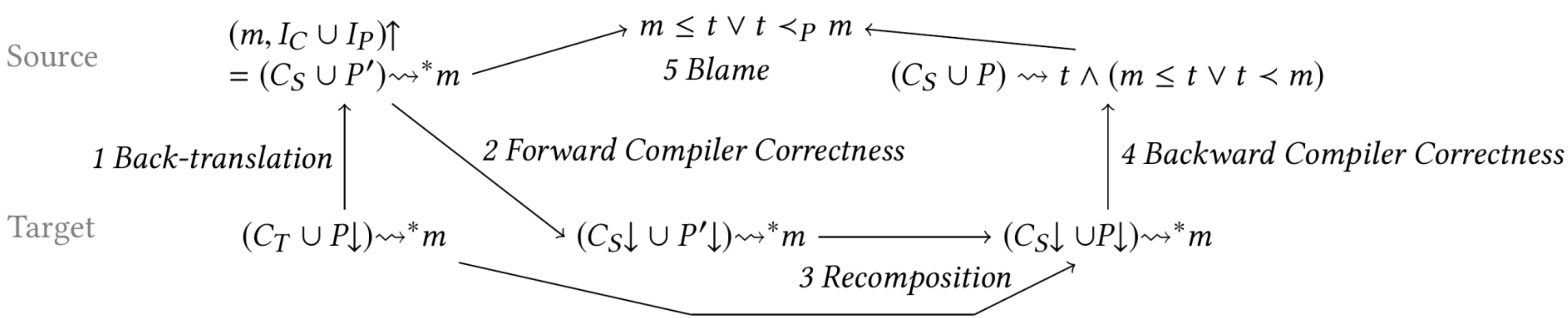
Scalable proof technique

(for our variant of Robust Safety Preservation)

1. back-translating **finite trace prefix** to **whole source program**

2+4. compiler correctness proof (à la CompCert) **used as a black-box**

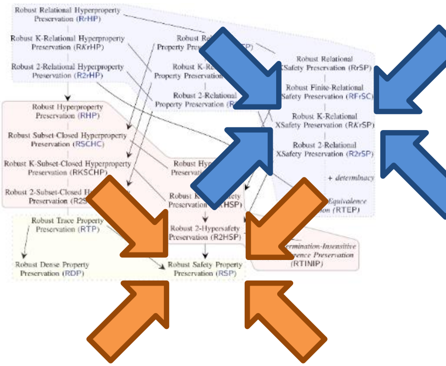
3+5. also simulation proofs



Extending proof technique

- **Recent**: From memory isolated components [CCS'18] to **fine-grained dynamic memory sharing** by **passing safe pointers** (e.g. capabilities)
  - [SecurePtrs, Akram El-Korashy et al, arXiv:2110.01439]

- **Ongoing: beyond robust preservation of safety**
  - Back-translating finite sets of finite traces [Jérémy Thibault et al, CSF'19]
  - Nanopass Back-Translation of Call-Return Trees [Jérémy Thibault, upcoming]

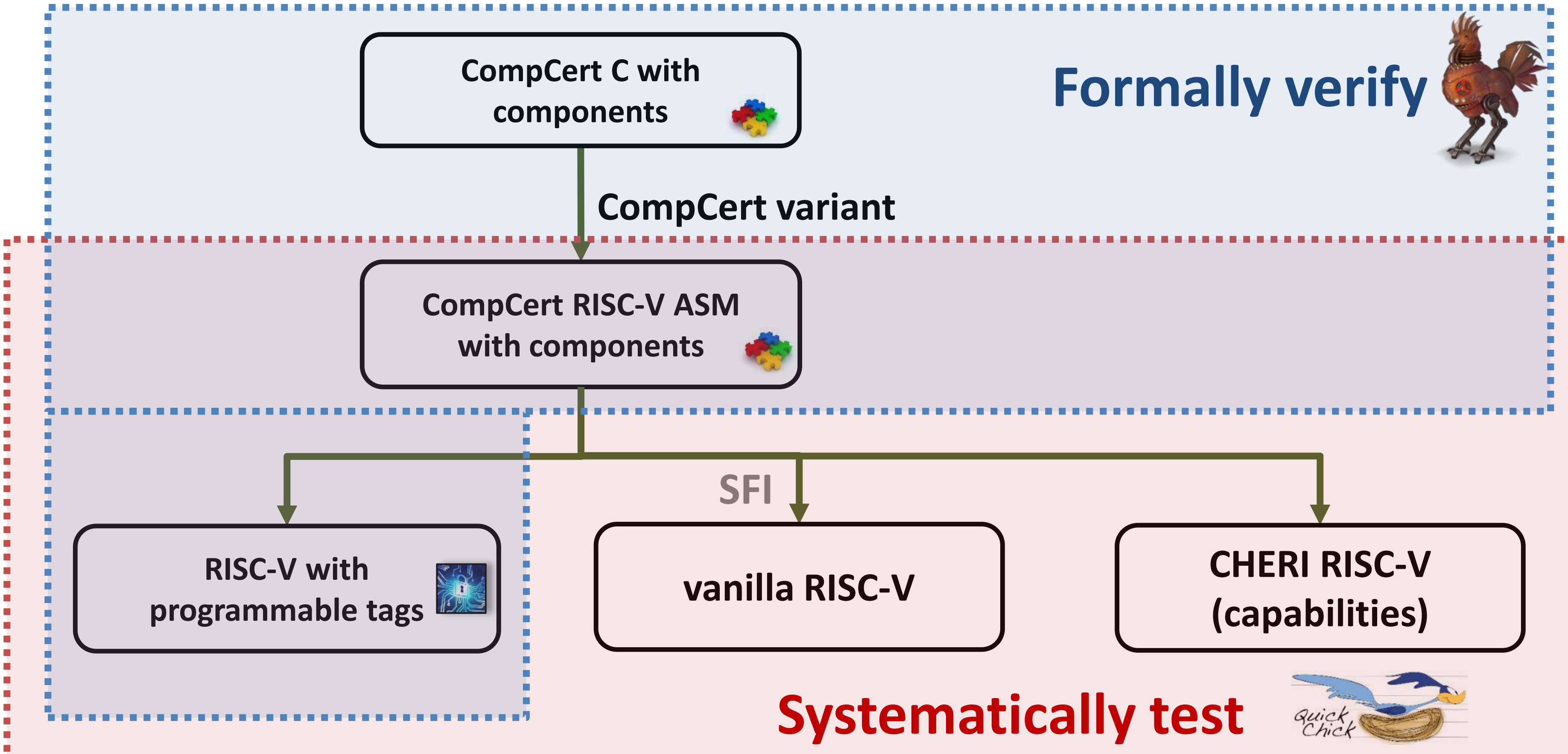


[CSF'19, ESOP'20]

Ongoing: applying this to CompCert

- CompCert already temporally restricts UB
- **Added spatial UB restrictions**:
  - **extended CompCert with components and interfaces**
- **Mostly done: extending <u>correct</u> compilation proofs**
  - proof technique uses correct compilation "as black box", mostly
  - but adding components to all CompCert levels still required some work
- **Coming soon**: secure compilation proofs for CompCert
  - need to port back-translation and recomposition proofs
  - first time this kind of secure compilation proofs would be done at this scale

Future: multiple enforcement mechanisms



Open problems

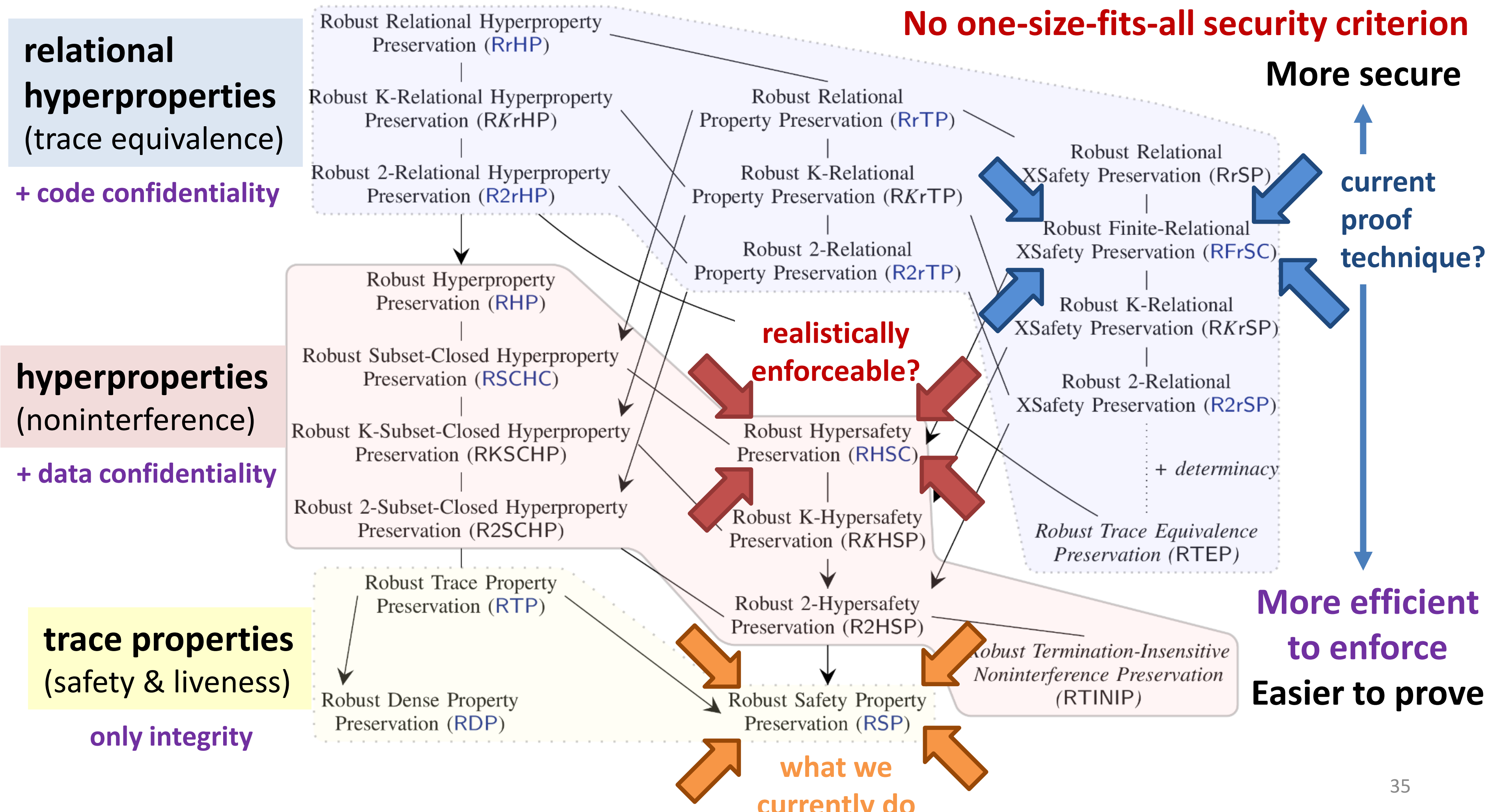
- **Dynamic component creation**
  - from code-based to data-based compartmentalization
  - criterion: rewind to when compromised component was created

- **Enforcement beyond robust preservation of safety**
  - **in the presence of side-channels or even micro-architectural attacks**

- **Protect abstractions of verification language like Low* (Everest)**
  - **Some related work in progress: safe F*-ML interop** by runtime monitoring and turning checkable F* specifications into dynamic contracts

BACKUP SLIDES

Going beyond Robust Preservation of Safety [CSF'19, ESOP'20]



Scalable proof technique

(for our variant of Robust Safety Preservation)

1. back-translating **finite trace prefix** to **whole source program**