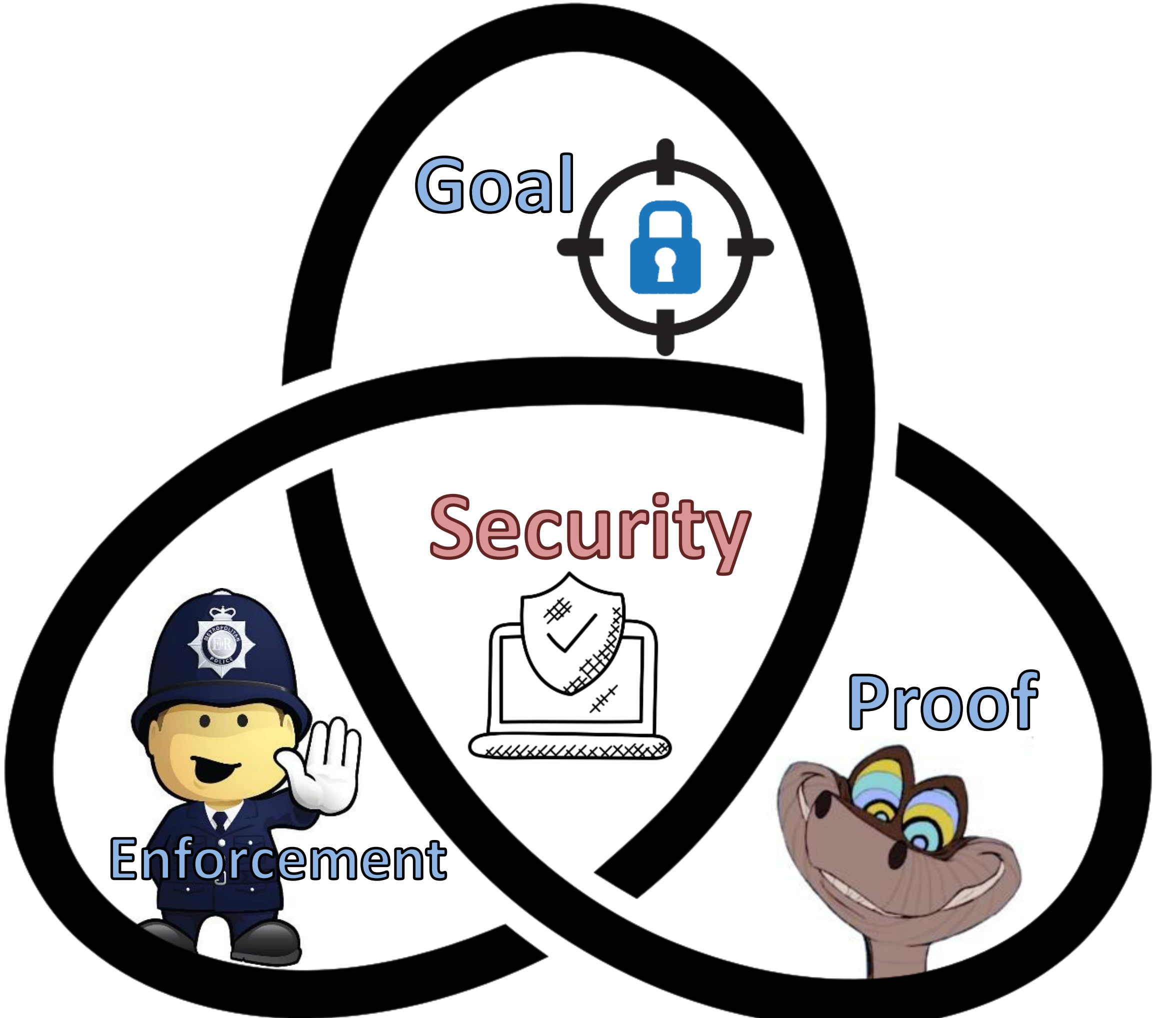
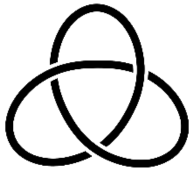
When Good Components Go Bad

Formally Secure Compilation Despite Dynamic Compromise

Cătălin Hrițcu

Inria Paris


Goal
Security
Proof
Enforcement



Security foundations research is about making this diagram mathematically formal

1. Security Goal [What are we trying to achieve?]

- **negative definition**: What (kind of) attacks are we trying to prevent?
- **positive definition**: What security property are we aiming for?

2. Security Enforcement [How can we effectively achieve it?]

- **static**: informal audit, program verification, type systems, ...
- **dynamic**: reference monitors, hardware mechanisms, crypto, ...
- **trade off security** vs. precision, efficiency, compatibility, ...

3. Security Proof [How can we make sure we achieved it?]


TRUST ME. OUR CLOUD SECURITY IS SO GOOD EVEN YOU WON'T BE ABLE TO ACCESS YOUR DATA!
© D.Fletcher for CloudTweaks.com

Security proof

Easier and more scalable

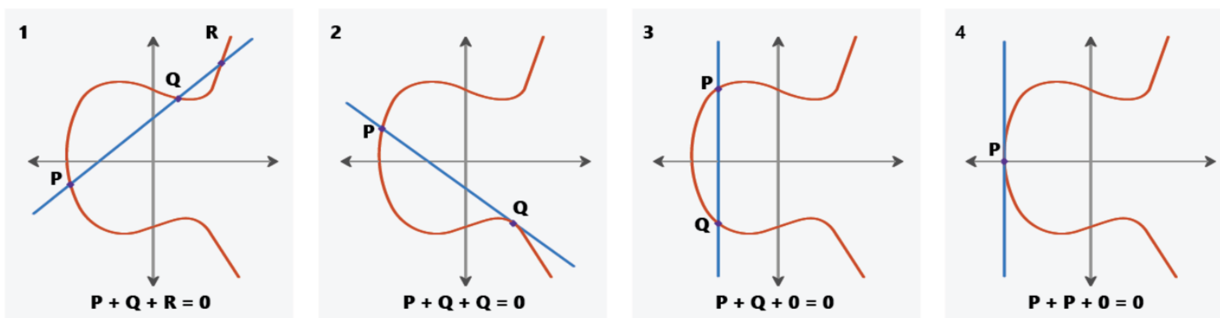
- **Marketing snake oil**: trussst me, it isss very sssecure
- ...
- **Security experts, metrics, standards**
- **Security testing**, red teaming, bounty programs
- ...
- **Mathematical proofs** with various levels of rigor
- **Formal, machine-checked proofs**
  - in a proof assistant like Coq, Isabelle, HOL, F*, EasyCrypt, ...
  - about **abstract models** or **concrete implementations**
  - under various **assumptions** and **trusted computing base**

Better assurance

EasyCrypt

EverCrypt cryptographic provider offers developers greater security assurances

April 2, 2019 | By Jonathan Protzenko, Researcher; Bryan Parno, Associate Professor, Carnegie Mellon University



Project Everest is a multiyear collaborative effort focused on building a verified, secure communications stack designed to improve the security of HTTPS, a key internet safeguard. This post, about the high-performance industrial-grade EverCrypt cryptographic provider, is the second in a series exploring the groundbreaking work, which is available on GitHub now.

EverCrypt: Verified Crypto Provider

- **Verified C (HACL*)**: ChachaPoly, SHA2+3, Blake2, Curve25519, ...
- **Verified X64 ASM (Vale)**: AES-GCM, Poly1305, Curve25519, ...
- **Good efficiency**, comparable to libcrypto or libsodium
- **Readable** C and ASM code
- **Deployed in production**
  - Mozilla Firefox (NSS)
  - Microsoft WinQUIC

- **Project Everest**, extending this to:
  - verified TLS implementation
  - verified HTTPS stack

EverCrypt formally

1. Security Goals

- **Memory safety** (no buffer overflows, use-after-frees, double-frees, ...)
- **Functional correctness** (code implements a simpler math function)
- **Side-channel resistance** (secret independent control & mem accesses)
- **Cryptographic security** (e.g. auth, int, and conf of AEAD constructions)

2. Security Enforcement

- **static**: program verification in F* for safety and correctness
- side-channel resistance and crypto security involve paper proofs

3. Security Proof

- **milestone**: 100.000+ lines of verifiably correct code, shipping
- **still**: big trusted computing base, some interesting proofs on paper

Formally Secure Compartmentalization

When Good Components Go Bad (CCS 2018)

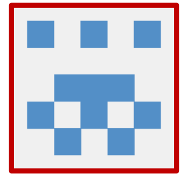
Beyond Good and Evil (CSF 2016)

Micro-Policies (IEEE S&P 2015)

Core team at Inria Paris

Carmine Abate

Rob Blanco



Florian Groult

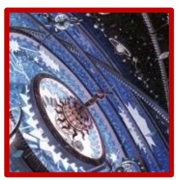
Cătălin Hrițcu

Théo Laurent

Jérémy Thibault

Collaborators

Arthur Azevedo de Amorim
CMU (ex Inria)



Boris Eng
Paris 7
(ex Inria)

Ana Nora Evans
U. Virginia
(ex Inria)

Guglielmo Fachini
Nozomi
(ex Inria)

Yannis Juglaret
DGA-MI
(ex Inria)

Benjamin Pierce
UPenn

Marco Stronati
Tezos
(ex Inria)

Andrew Tolmach
Portland State

Inherently insecure languages like C

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

Compartmentalization mitigation

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

1. Security Goal

[What are we trying to achieve?]

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

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

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

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

Can we formalize this intuition?

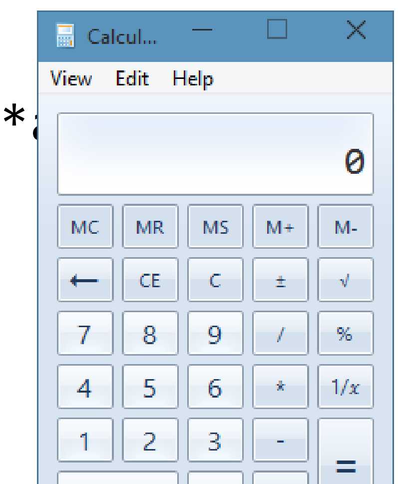
What is a compartmentalizing compilation chain supposed to enforce precisely?

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

Challenge formalizing security of mitigations

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

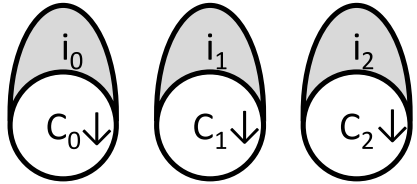
```
#include <string.h>
int main (int argc, char **
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```



Compartmentalizing compilation should ...

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

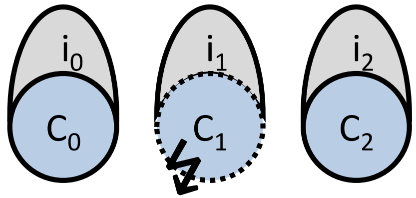
Security definition: If



$\rightsquigarrow t$ then

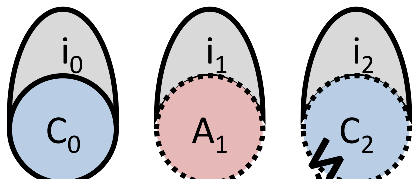
∃ a sequence of component compromises explaining the finite trace t in the source language, for instance $t = m_1 \cdot m_2 \cdot m_3$ and

(1)



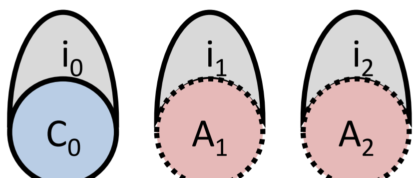
$\rightsquigarrow^* m_1 \cdot \mathrm{Undef}(C_1)$

(2) $\exists A_1.$



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

(3) $\exists A_2.$



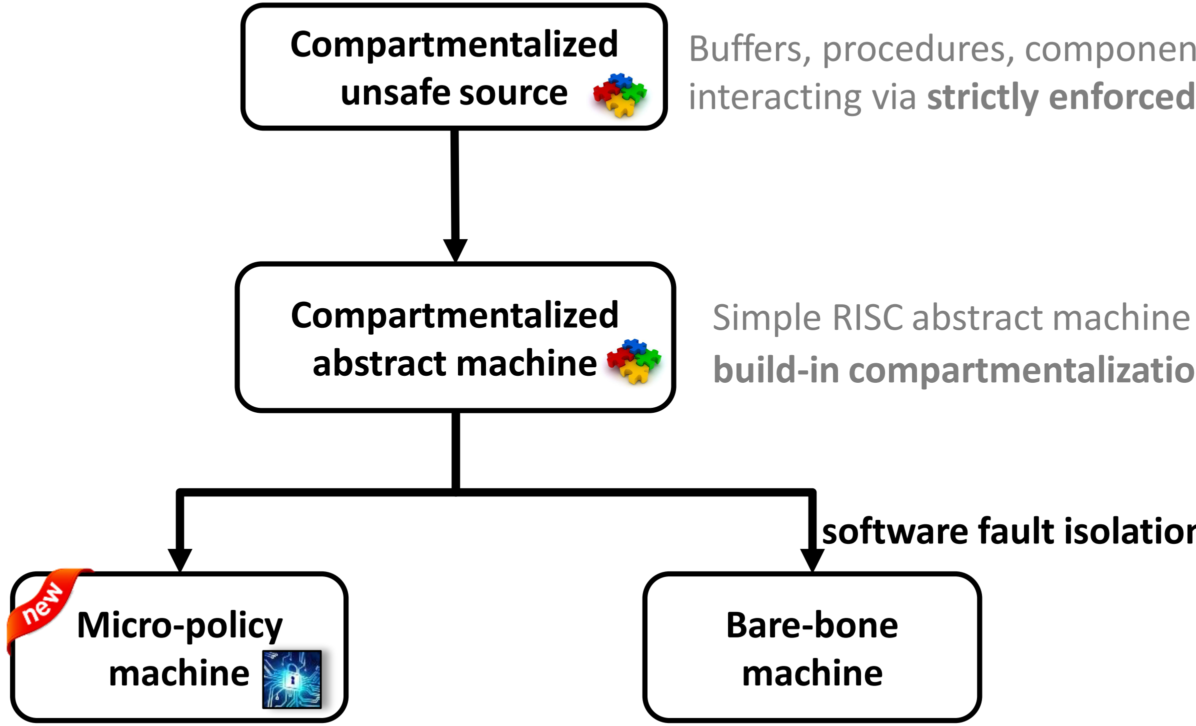
$\rightsquigarrow m_1 \cdot m_2 \cdot m_3$

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

2. Security Enforcement

[How can we effectively enforce this?]

Proof-of-concept secure compilation chain



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

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

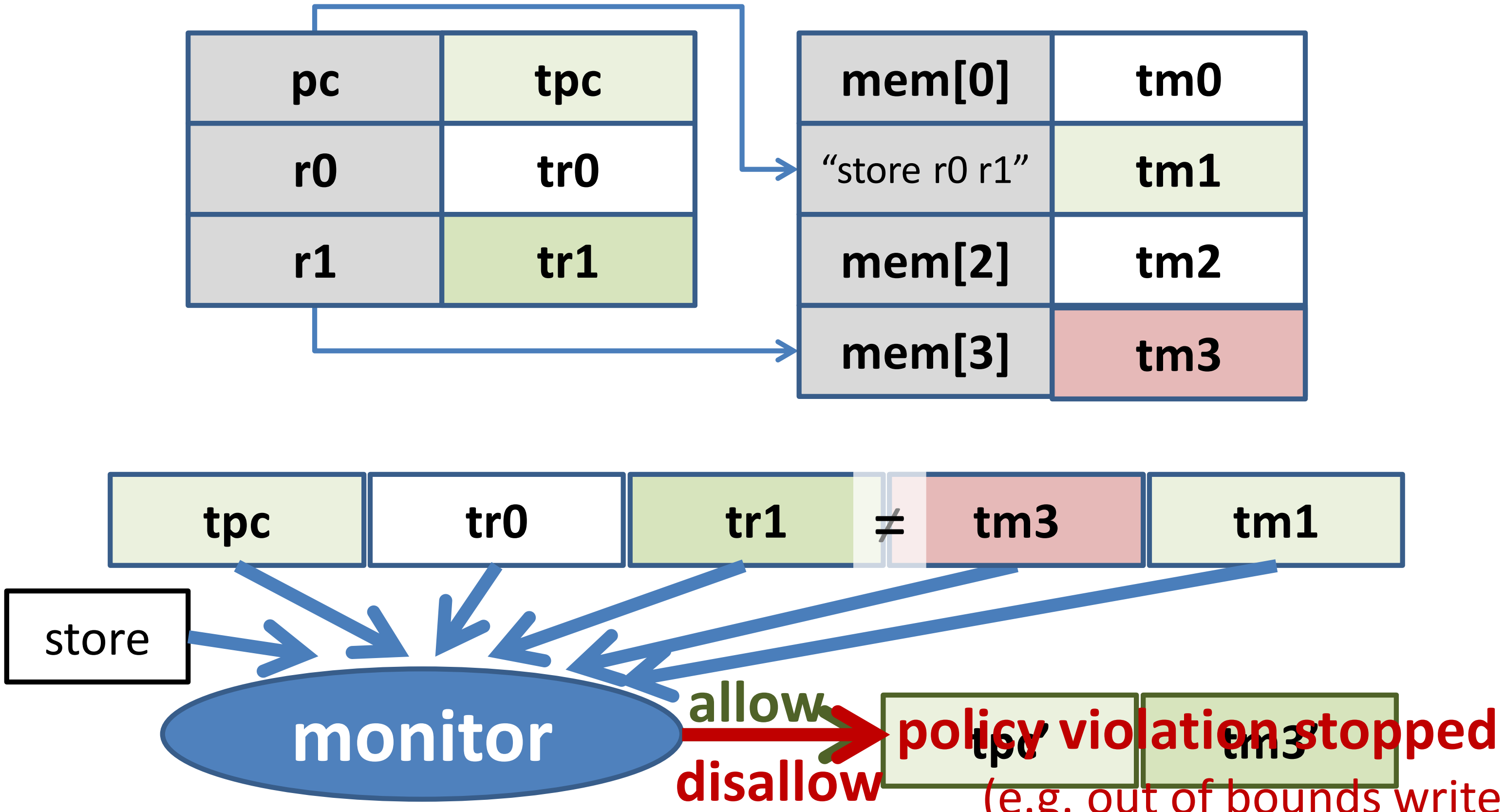
Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

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

Expectation: other enforcement mechanisms should work as well

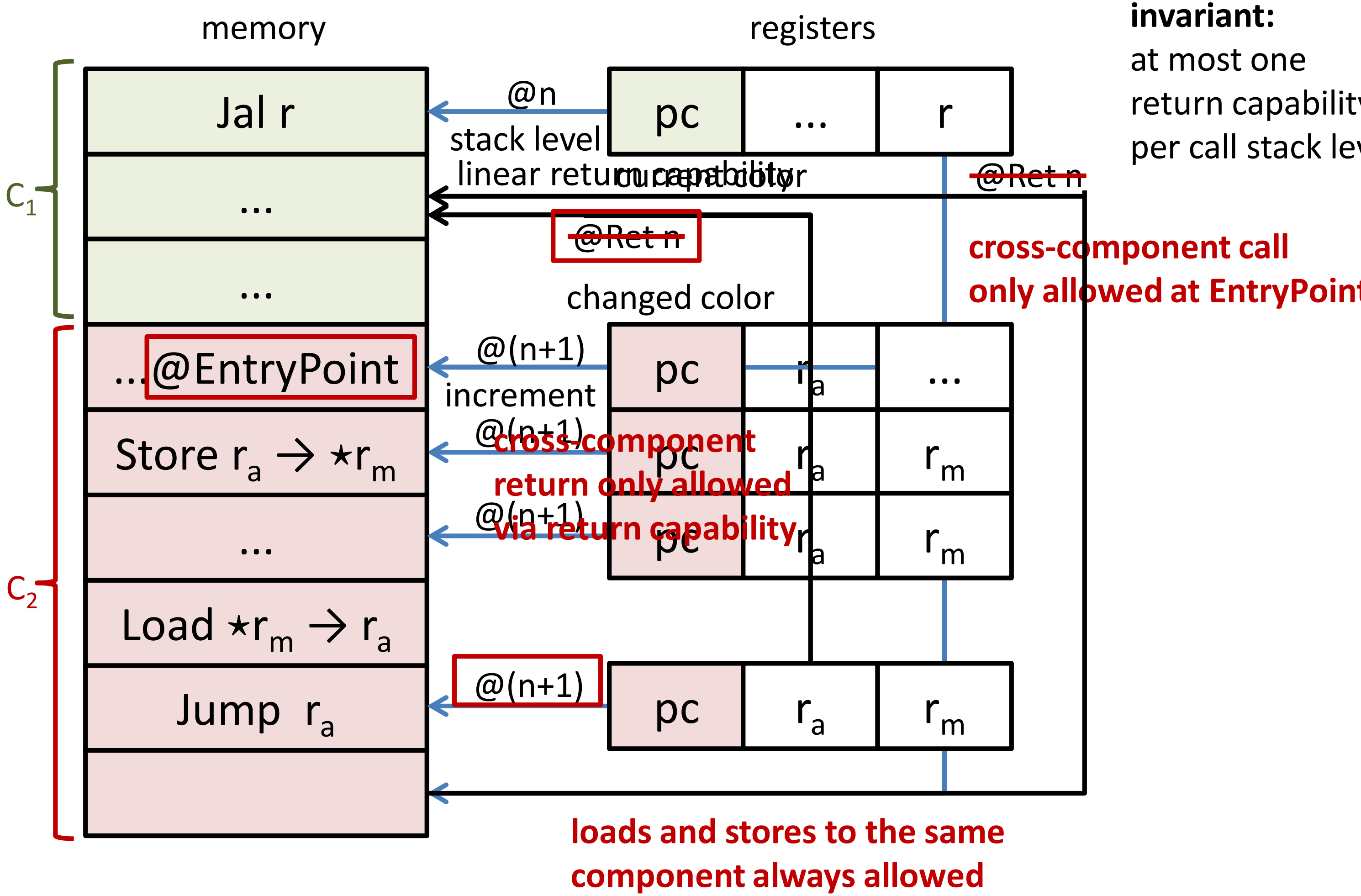
Micro-Policies [Oakland'15, ASPLOS '15,...]

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



software monitor's decision is hardware cached

Compartmentalization micro-policy

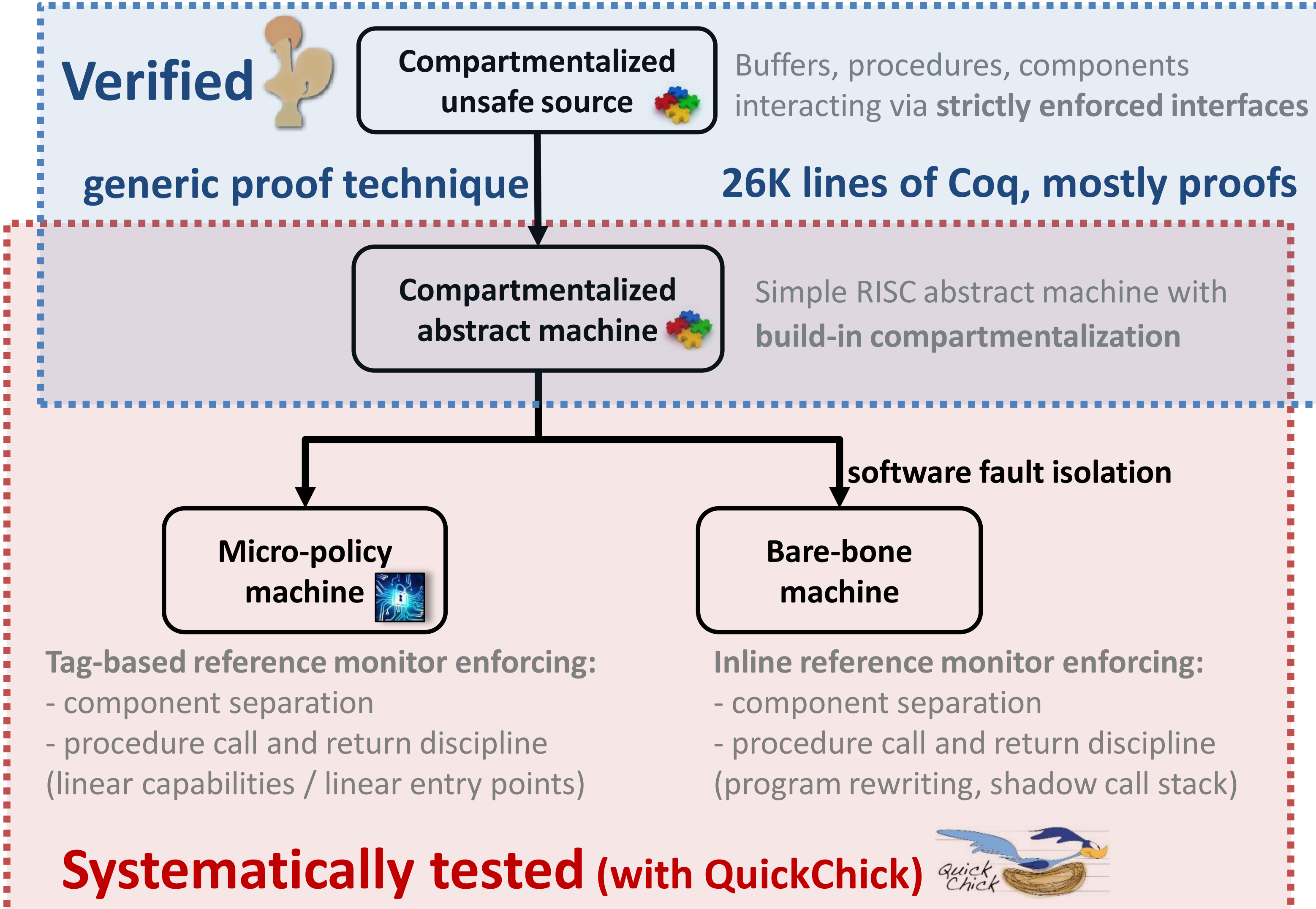


invariant:
at most one
return capability
per call stack level

3. Security Proof

[How can we make sure we achieved our goal?]

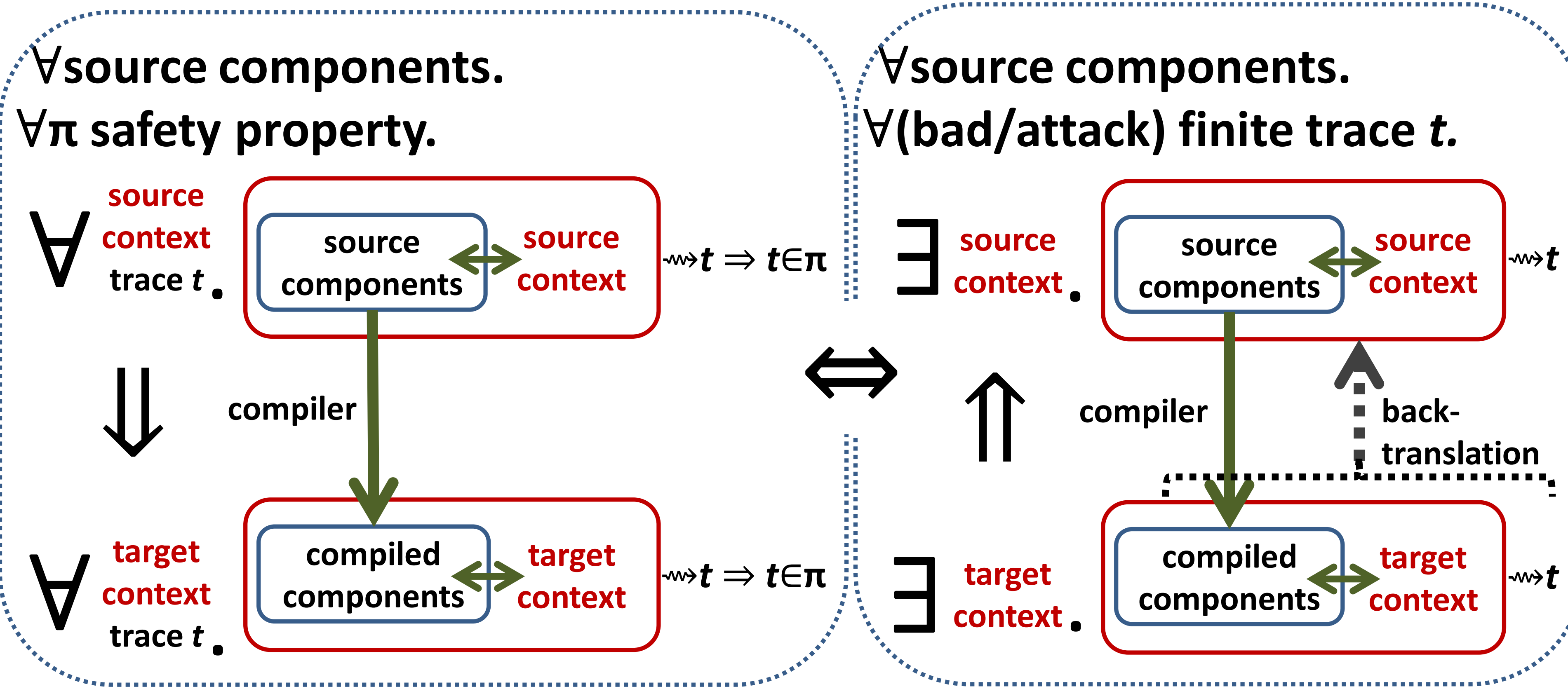
Proof-of-concept formally secure compilation chain in Coq



https://secure-compilation.github.io

We reduce our proof goal to a variant of:

Robust Safety Preservation



robust preservation of safety

proof-oriented characterization

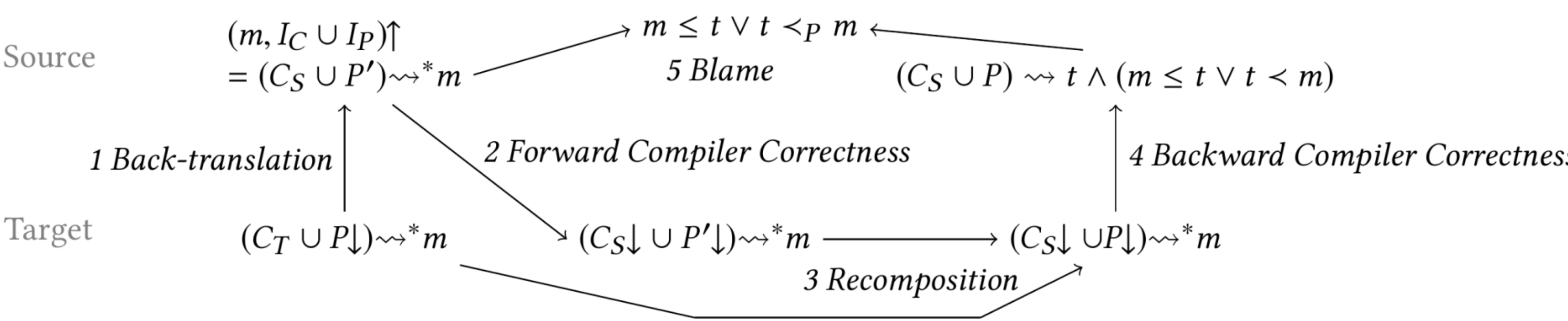
Simple and scalable proof technique

(for our variant of Robust Safety Preservation)

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

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

3+5. simulation proofs



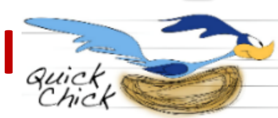
When Good Components Go Bad

1. **Goal: formally secure compartmentalization**

   – **first definition** supporting **mutually distrustful components** and **dynamic compromise**
   – **restricting undefined behavior spatially** and **temporally**

2. **Enforcement: proof-of-concept secure compilation chain**
   – **software fault isolation** or **tag-based reference monitor**

3. **Proof: combining formal proof and property-based testing**
   – **Generic proof technique** that **extends** and **scales well**



Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **shared memory** (ongoing) and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster
- **Extend all this to dynamic component creation**
  - rewind to when compromised component was created
- **... and dynamic privileges**
  - capabilities, dynamic interfaces, history-based access control, ...
- **From robust safety to hypersafety (confidentiality)** [CSF'19]
- **Secure compilation of EverCrypt, miTLS, ...**

My dream: secure compilation at scale

language

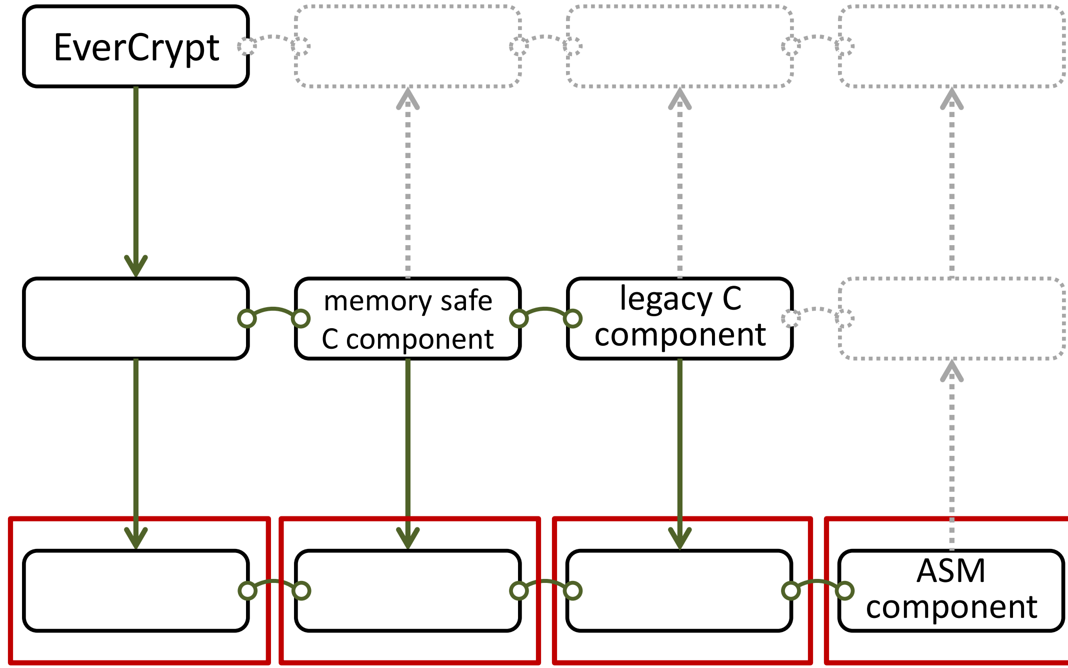
C language

+ components

+ memory safety

ASM language

(RISC-V + micro-policies)



Going beyond Robust Preservation of Safety [CSF'19]

relational hyperproperties
(trace equivalence)

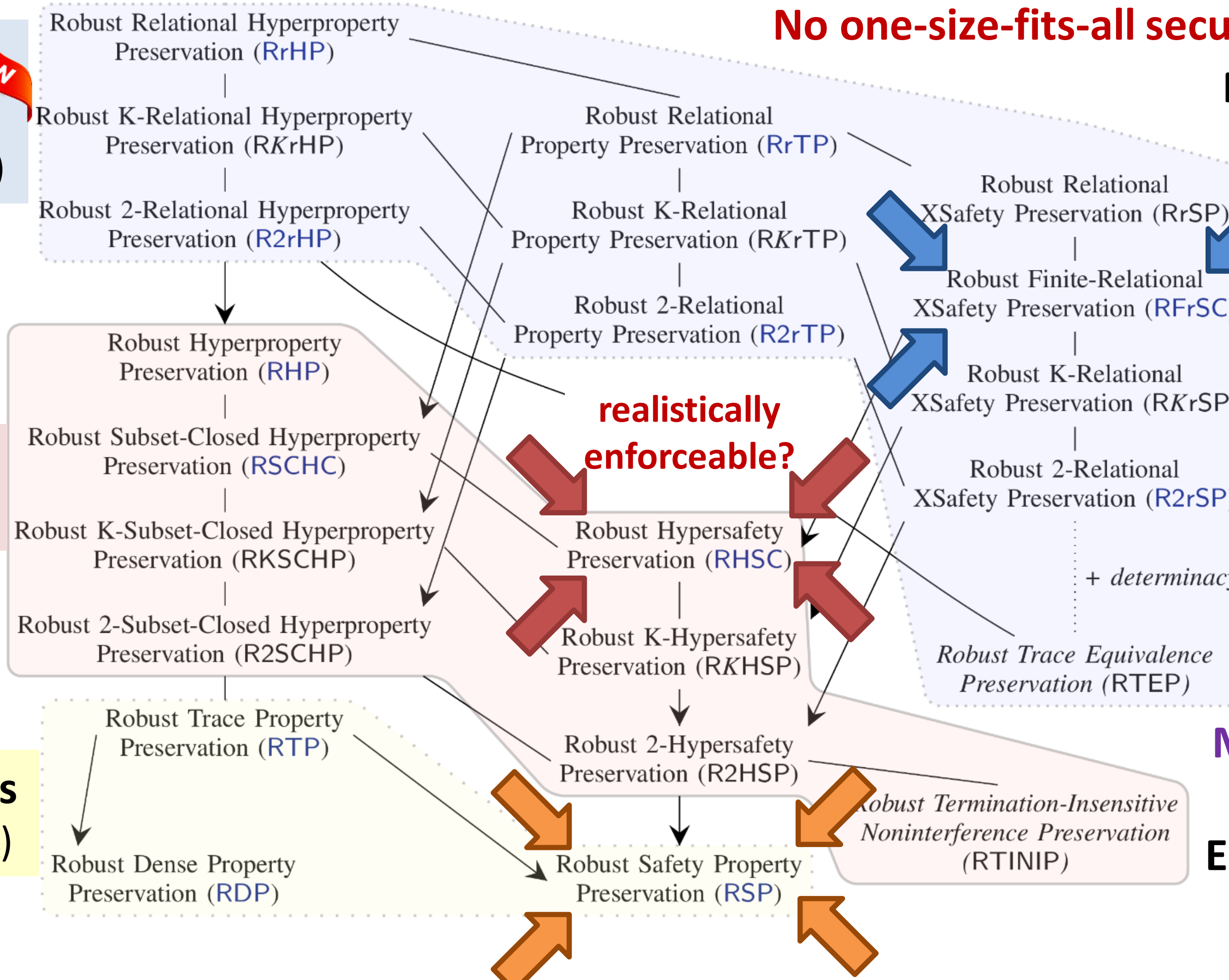
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



No one-size-fits-all security criterion

More secure

current proof technique

More efficient to enforce

Easier to prove

When Good Components Go Bad

1. **Goal: formally secure compartmentalization**
   - **first definition** supporting **mutually distrustful components** and **dynamic compromise**
   - **restricting undefined behavior spatially** and **temporally**

2. **Enforcement: proof-of-concept secure compilation chain**
   - **software fault isolation** or **tag-based reference monitor**
3. **Proof: combining formal proof and property-based testing**
   - **Generic proof technique** that **extends** and **scales well**