Formally Secure Compilation
of Unsafe Low-level Components

Cătălin Hrițcu

Inria Paris

https://secure-compilation.github.io

Collaborators

Cătălin Hrițcu

Marco Stronati

Guglielmo Fachini

Arthur Azevedo de Amorim

Ana Nora Evans

le plus petit cirque du monde

Théo Laurent

Deepak Garg

Marco Patrignani

Carmine Abate

Andrew Tolmach

Benjamin Pierce

Yannis Juglaret

Computers are insecure

- **devastating low-level vulnerabilities**
- **inherently insecure low-level languages**
  - **memory unsafe**: any buffer overflow is catastrophic
  - **root cause**, but challenging to fix: efficiency, precision, scalability, backwards compatibility, deployment
- **compartmentalization, a strong practical defense**
  - **practically deployed low-level protection mechanisms**
    - process-level privilege separation (all web browsers)
    - software fault isolation (SFI, Google Native Client)
    - hardware enclaves (Intel SGX, ARM TrustZone)

Zoo

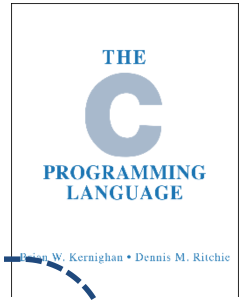
Zoo ... with very dangerous beasts

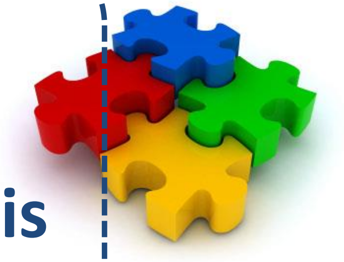
Zoo … with very dangerous beasts

(source: Jurassic Island: The Dinosaur Zoo)

Compartmentalization

for unsafe, low-level languages

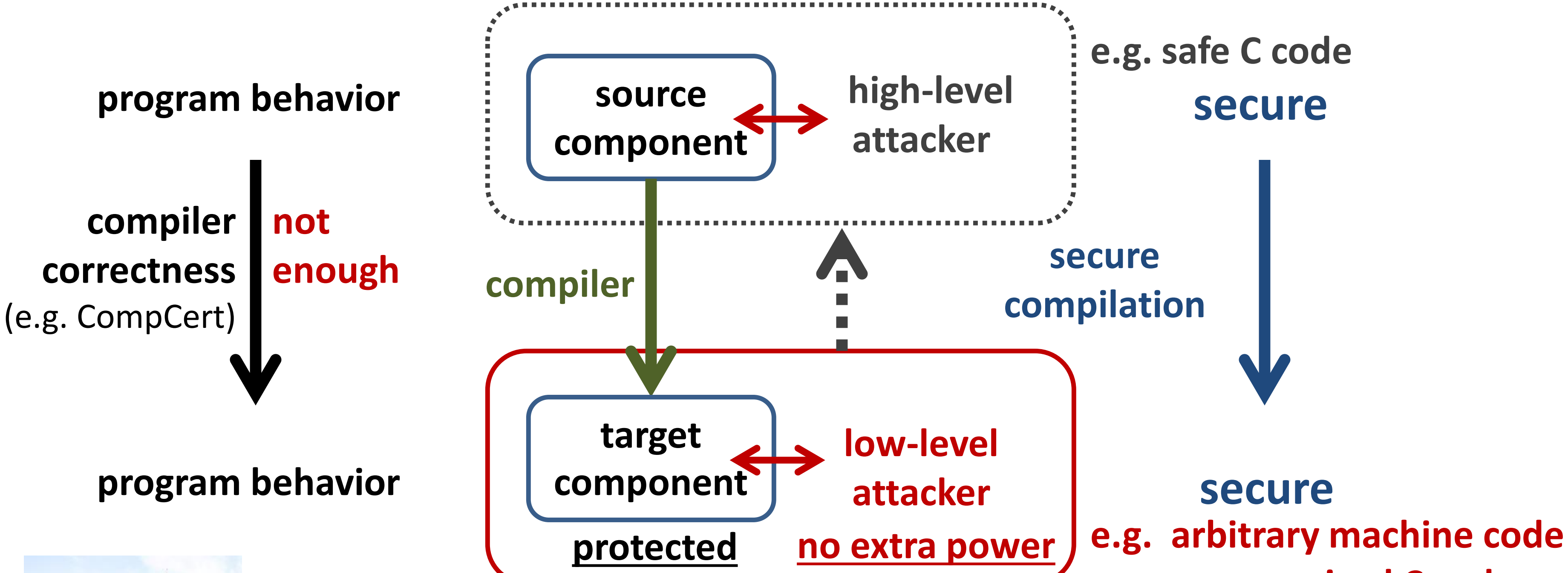




- **Add components to C-like language**
  - interacting only via **strictly enforced interfaces**
- **Secure compilation chain** **Goal: Build this**
  - use compartmentalization to **efficiently enforce**: component separation, call and return discipline, ...
- **Interesting attacker model** **Goal: Formalize this**
  - **mutual distrust, dynamic compromise, least privilege**
    - each component should be protected from all the others until it becomes compromised (by exhibiting undefined behavior) and starts attacking the remaining uncompromised components

Formally secure compilation

holy grail of preserving security all the way down



program behavior

compiler correctness
(e.g. CompCert)

not enough

program behavior

source component

high-level attacker

compiler

target component

low-level attacker

protected

no extra power

e.g. safe C code

secure

secure compilation

secure

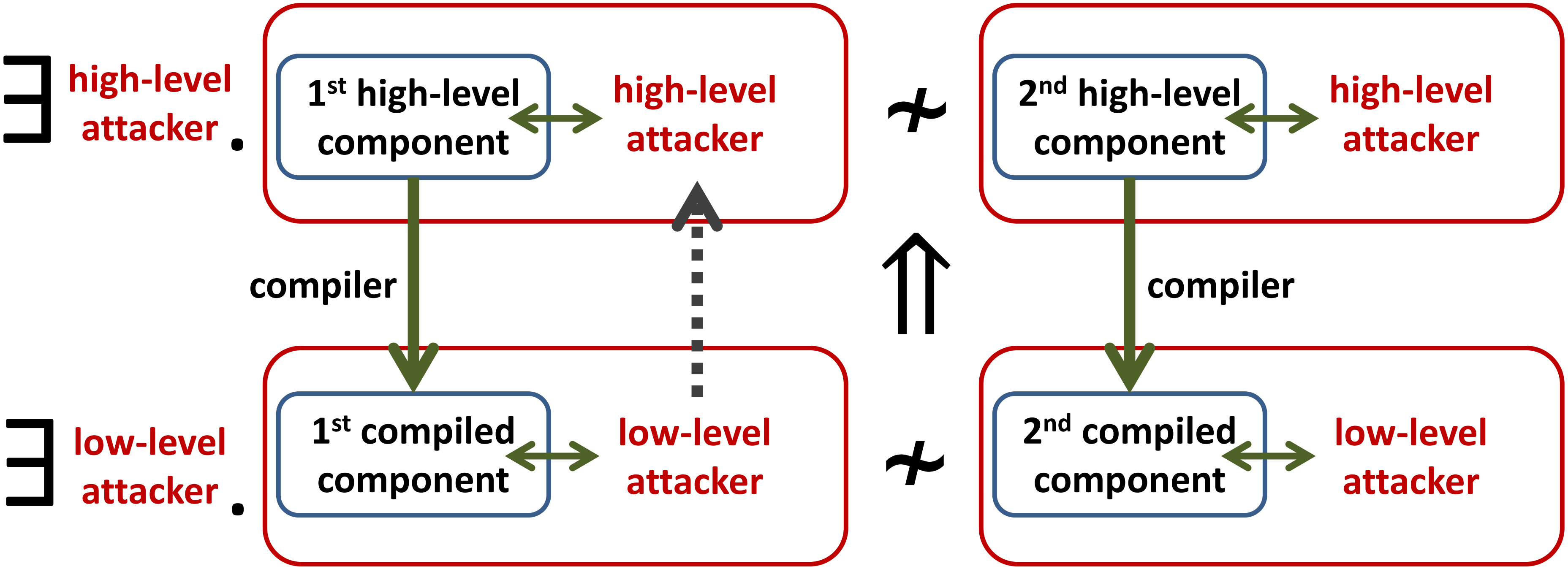
e.g. arbitrary machine code
e.g. compromised C code

Benefit: sound security reasoning in the source language

forget about compilation chain (linker, loader, runtime)
forget that libraries are written in a lower-level language

Fully abstract compilation

(preservation of observational equivalence)



$\exists$ high-level attacker.

1st high-level component ↔ high-level attacker $\nsim$ 2nd high-level component ↔ high-level attacker

compiler

$\Uparrow$

compiler

$\exists$ low-level attacker.

1st compiled component ↔ low-level attacker $\nsim$ 2nd compiled component ↔ low-level attacker

Issues: (1) hard to realistically and efficiently achieve

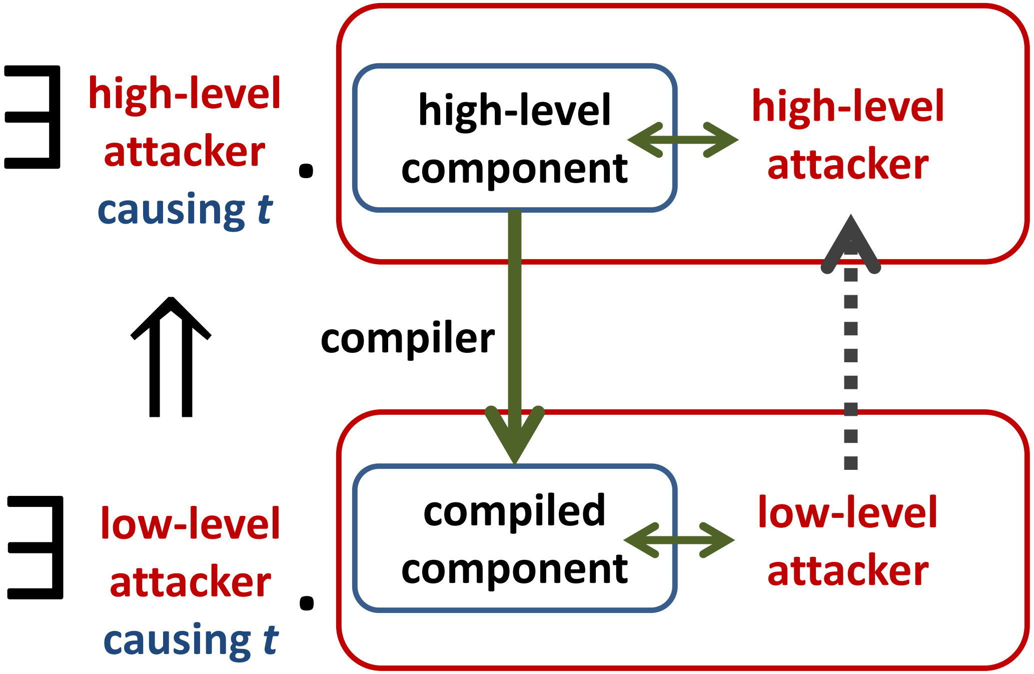
(2) challenging to prove at scale

(3) not intuitive to most security people

(4) doesn't quite work for unsafe languages

Our **new target**: **Robust compilation**

$\forall$ (bad, attack) trace *t*



$\exists$ **high-level attacker causing *t*** .

high-level component

high-level attacker

$\Uparrow$

compiler

$\exists$ **low-level attacker causing *t*** .

compiled component

low-level attacker

robust trace property preservation
(robust = in adversarial context)

gives up on confidentiality
(relational/hyper properties)

intuition:

- **stronger** than compiler correctness
- seems **weaker** than full abstraction
  + compiler correctness

less extensional than FA

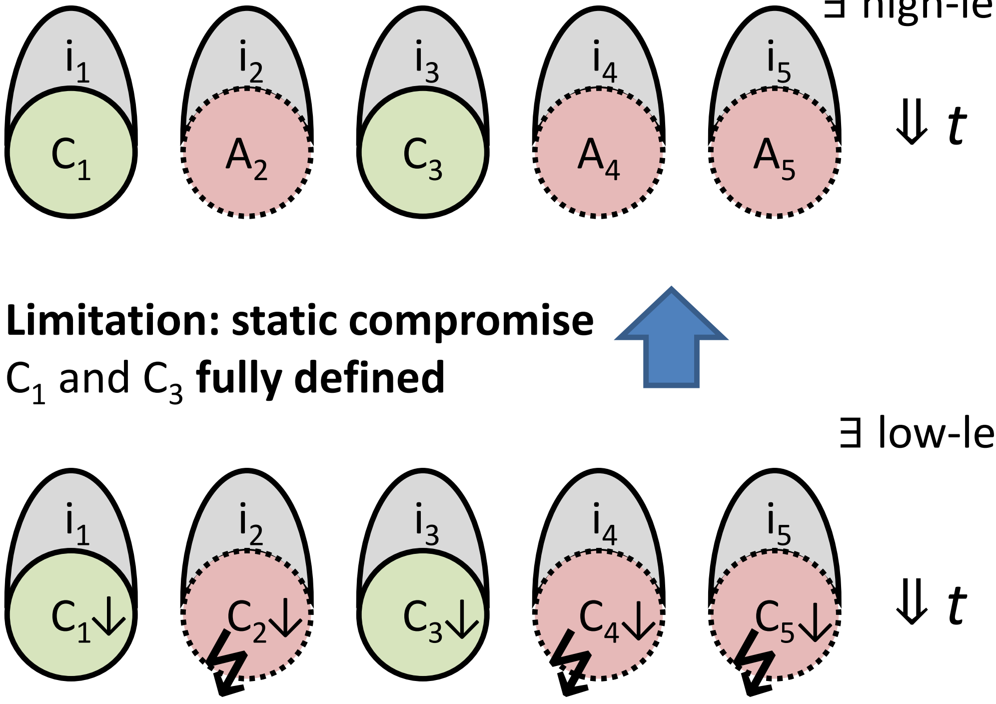
Advantages: **easier to realistically achieve and prove**

useful: preservation of **invariants** and other **integrity properties**

works for unsafe languages (supporting dynamic compromise)

Mutually distrustful components

∀compromise scenarios. ∀ (bad, attack) traces *t*.



∃ high-level attack from some **fully defined** A_2, A_4, A_5

Limitation: static compromise

C_1 and C_3 **fully defined**

∃ low-level attack from compromised $C_2\downarrow$, $C_4\downarrow$, $C_5\downarrow$

C_1 and C_3 can get guarantees only if they are perfectly secure
(i.e. fully defined = do not exhibit undefined behavior in **any** context)

This is the most we were able to do for full abstraction!

[Beyond Good and Evil - Juglaret, Hrițcu, et al, CSF'16]

Static compromise not good enough

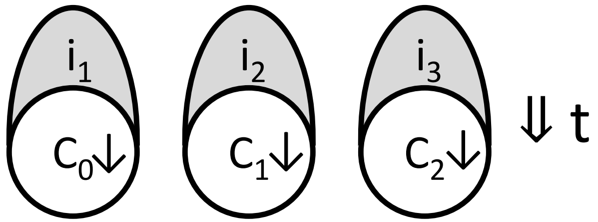
neither C_1 not C_2 are fully defined

yet C_1 is protected until calling C_1.parse

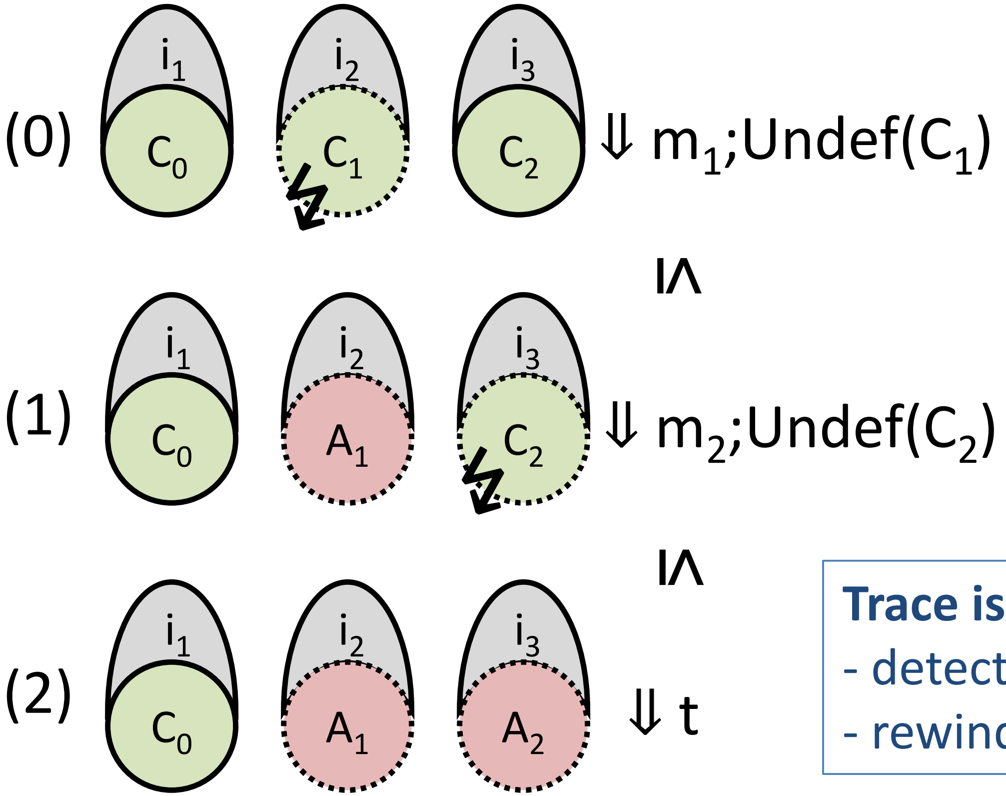
and C_2 can't actually be compromised

```
component C0 {
  export valid;
  valid(data) { ... }
}
component C1 {
  import E.read, C2.init, C2.process;
  main() {
    C2.init();
    x := E.read();
    y := C1.parse(x);     //(V1) can UNDEF if x is malformed
    C2.process(x,y);
  }
  parse(x) { ... }
}
component C2 {
  import E.write, C0.valid;
  export init, process;
  init() { ... }
  process(x,y) { ... }  //(V2) can UNDEF if not initialized
}
```

Dynamic compromise



∃ a **dynamic compromise scenario** explaining t in source language
for instance ∃[A_1,A_2] leading to the following compromise sequence:



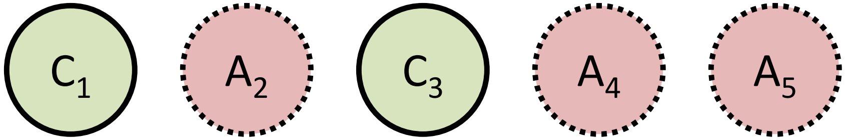
Trace is very helpful
- detect undefined behavior
- rewind execution

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]

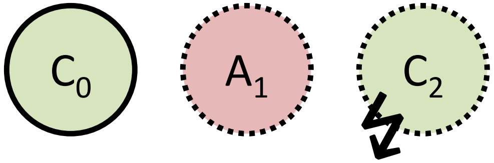
Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe, low-level languages)

Mutual distrust

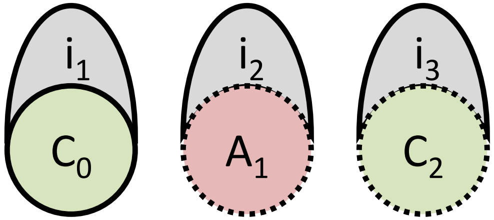


Dynamic compromise



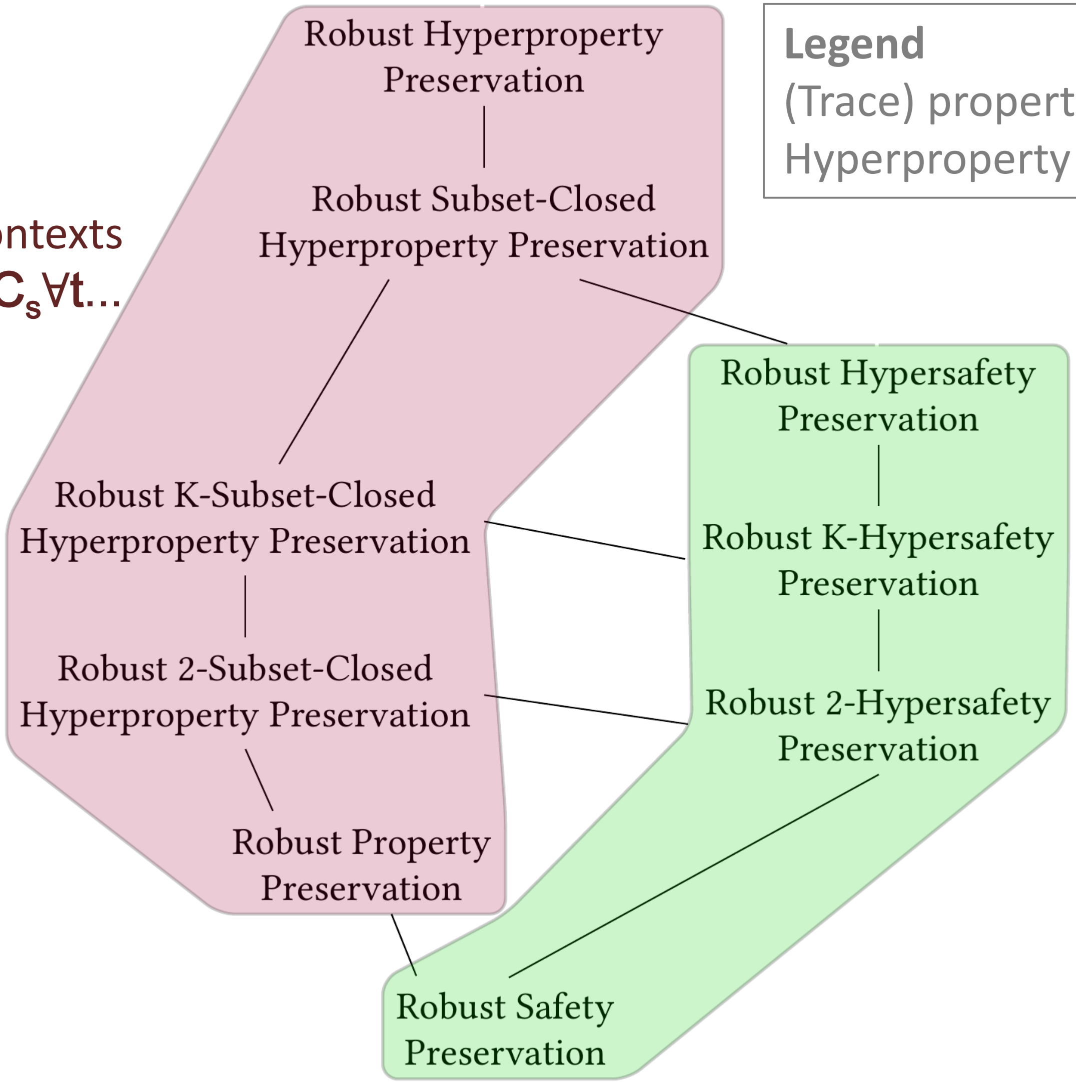
$\Downarrow m_2; \mathrm{Undef}(C_2)$

Least privilege



Beyond trace properties

Legend
(Trace) property = set of traces
Hyperproperty = set of sets of traces



Robust Hyperproperty Preservation

Robust Subset-Closed Hyperproperty Preservation

Robust K-Subset-Closed Hyperproperty Preservation

Robust 2-Subset-Closed Hyperproperty Preservation

Robust Property Preservation

Robust Hypersafety Preservation

Robust K-Hypersafety Preservation

Robust 2-Hypersafety Preservation

Robust Safety Preservation

back-translating contexts
$\forall P \forall C_t \exists \mathbf{C_s} \forall \mathbf{t} \ldots$

back-translating finite trace prefixes
$\forall P \forall C_t \forall \mathbf{t} \exists \mathbf{C_s} \ldots$

[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, et al]

Vision for ...

Building and verifying realistic secure compartmentalizing compilation chains

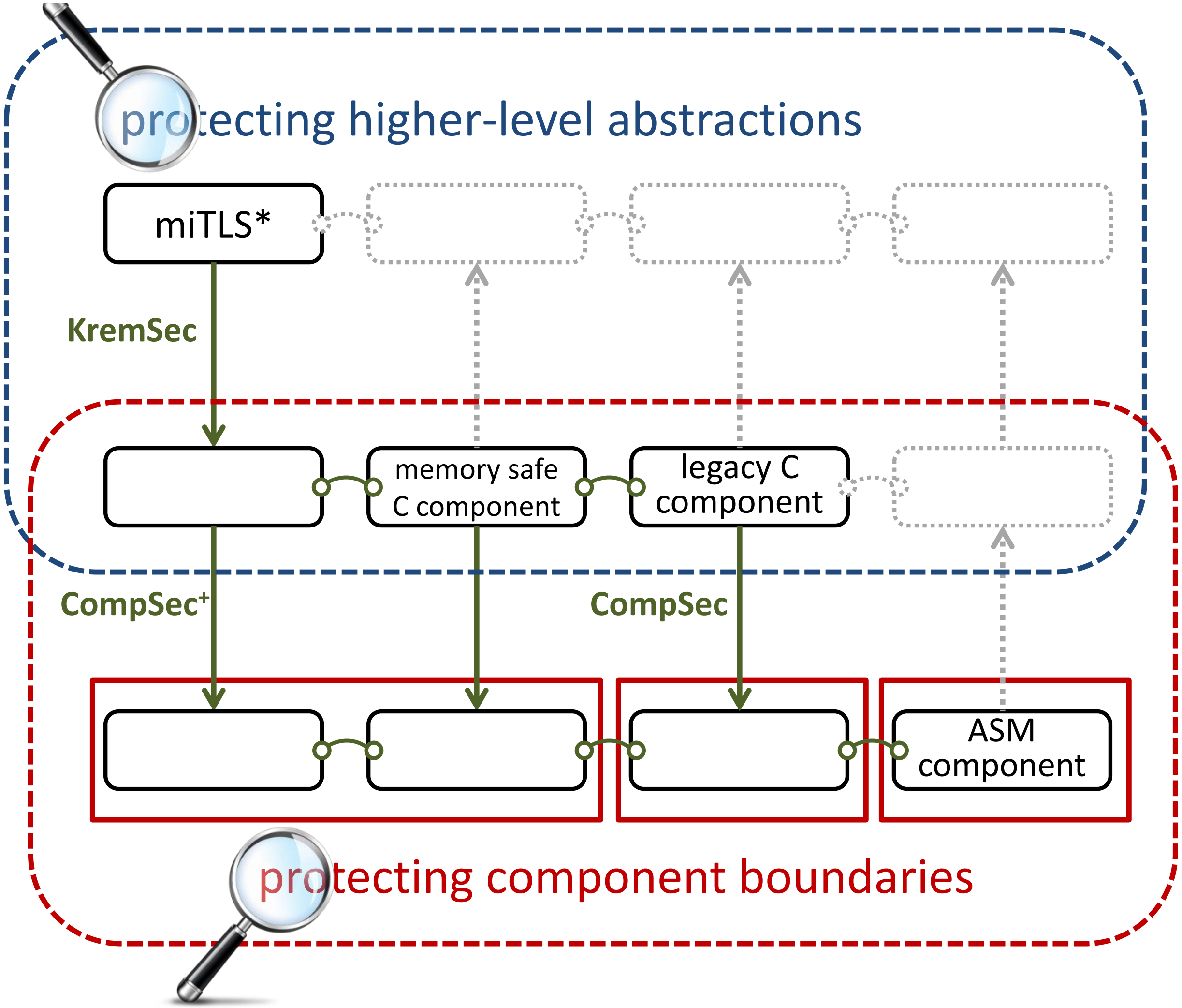
(i.e. mostly vaporware at this point)

Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)



protecting higher-level abstractions

miTLS*

KremSec

memory safe C component

legacy C component

CompSec$^+$

CompSec

ASM component

protecting component boundaries

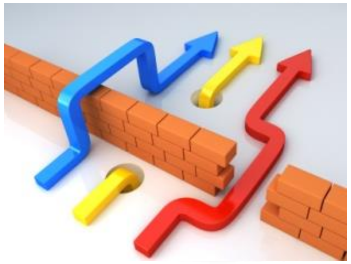
Protecting component boundaries

- **Add mutually distrustful components to C**
  - interacting only via **strictly enforced interfaces**
- **CompCert-based compilation chain**
  - propagate interface information to produced binary
- **Micro-policy simultaneously enforcing**
  - component separation
  - type-safe procedure call and return discipline

- **Software fault isolation fallback**
  - when tagged hardware support not available
- **Good progress on this but in much simplified setting**

Protecting higher-level abstractions

- **Low*: enforcing specifications in C**
  - some can be turned into **contracts,** checked dynamically; **micro-policies can speed this up too**
- **Limits of purely-dynamic enforcement**
  - functional purity, termination, relational reasoning
  - **push these limits further and combine with static analysis**



BACKUP SLIDES

Broad view on secure compilation

- Different security goals / attacker models
  - Fully abstract compilation and variants, **robust compilation**, noninterference preservation, ...
- Different enforcement mechanisms
  - **reference monitors**, secure hardware, static analysis, software rewriting, randomization, ...
- Different proof techniques
  - (bi)**simulation**, logical relations, multi-language semantics, embedded interpreters, ...