Formally Secure Compilation
of Unsafe Low-level Components

Cătălin Hrițcu

Inria Paris

https://secure-compilation.github.io

Parcurs profesional

- 2001 - 2005 - **Infoiași** - student la licență
- 2005 - 2011 - **Saarland University** - MSc & PhD
- 2011 - 2013 - **U. of Pennsylvania** - PostDoc cu Benjamin Pierce, DARPA CRASH/SAFE
- 2013 - acum - **Inria Paris** - Cercetător
- 2017 - 2021 - **ERC Starting Grant SECOMP** - PI
- 2017 - 2020 - **DARPA SSITH/HOPE** - coPI

Computers are insecure

- **devastating low-level vulnerabilities**
- **teasing out 2 important security problems:**

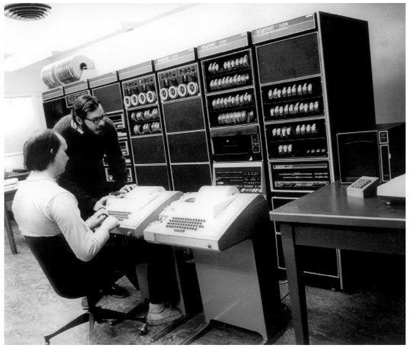
1. inherently insecure low-level languages

- **memory unsafe**: any buffer overflow can be catastrophic allowing remote attackers to gain complete control

2. unsafe interaction with unsafe code

- even code written in **safer languages** has to interoperate with **unsafe code**
- **unsafe interaction:** safety guarantees lost

How did we get here?



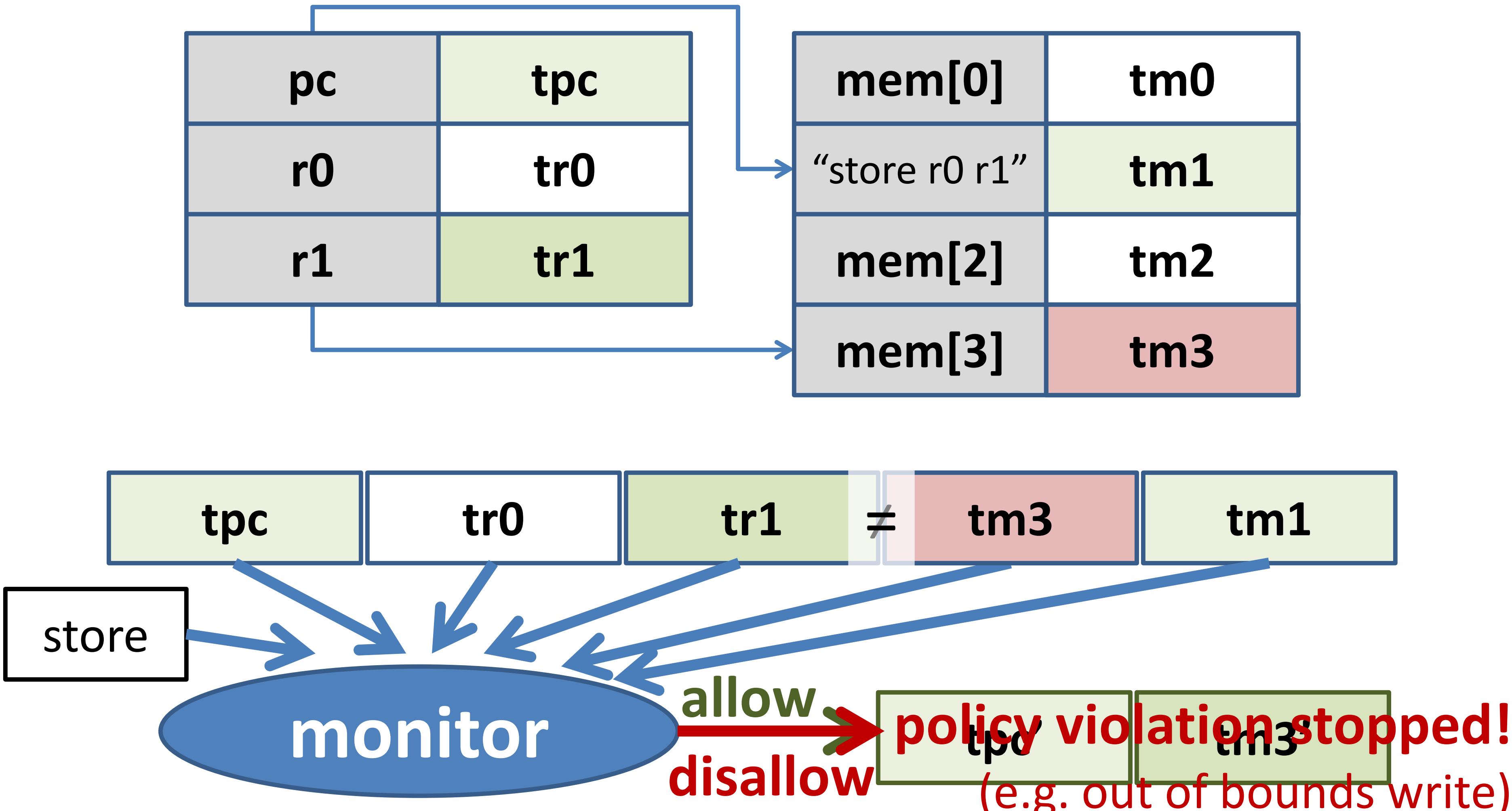
- **programming languages, compilers, and hardware architectures**
  - designed in an era of **scarce hardware resources**
  - too often **trade off security for efficiency**
- **the world has changed** (2017 vs 1972*)
  - security matters, hardware resources abundant
  - time to revisit some tradeoffs

* "...the number of UNIX installations has grown to 10, with more expected..."

-- *Dennis Ritchie and Ken Thompson, June 1972*

Key enabler: Micro-Policies

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



software monitor’s decision is hardware cached

Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V

spec

DRAPER DOVER MICROSYSTEMS

Expressiveness

Way beyond MPX, SGX, SSM, etc

- information flow control (IFC) [POPL'14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

Verified
(in Coq)
[Oakland'15]

Evaluated
(<10% runtime overhead)
[ASPLOS'15]

spec

Micro-Policies Project

- **Formal methods** & **architecture** & **systems**
- **Previous:** DARPA CRASH/SAFE (2011-2014)
- **Current:** DARPA SSITH/HOPE (2017-2020)
- **PIs**:
  - *Draper Labs:* **Arun Thomas, Chris Casinghino**
  - *Dover Microsystems:* **Greg Sullivan**
  - *DornerWorks:* **Nathan Studer, David Johnson**
  - *UPenn*: **André DeHon**, **Benjamin Pierce**
  - *Inria Paris*: **Cătălin Hrițcu**
  - *Portland State*: **Andrew Tolmach**
  - *MIT:* **Howie Shrobe**

DRAPER

DOVER MICROSYSTEMS

ERC SECOMP Grand Challenge (2017-2021)

Use micro-policies to build **the first efficient formally secure compilers** for **realistic programming languages**

1. **Provide secure semantics for low-level languages**
   - C with protected components and memory safety
2. **Enforce secure interoperability with unsafe code**
   - ASM, C, and Low*

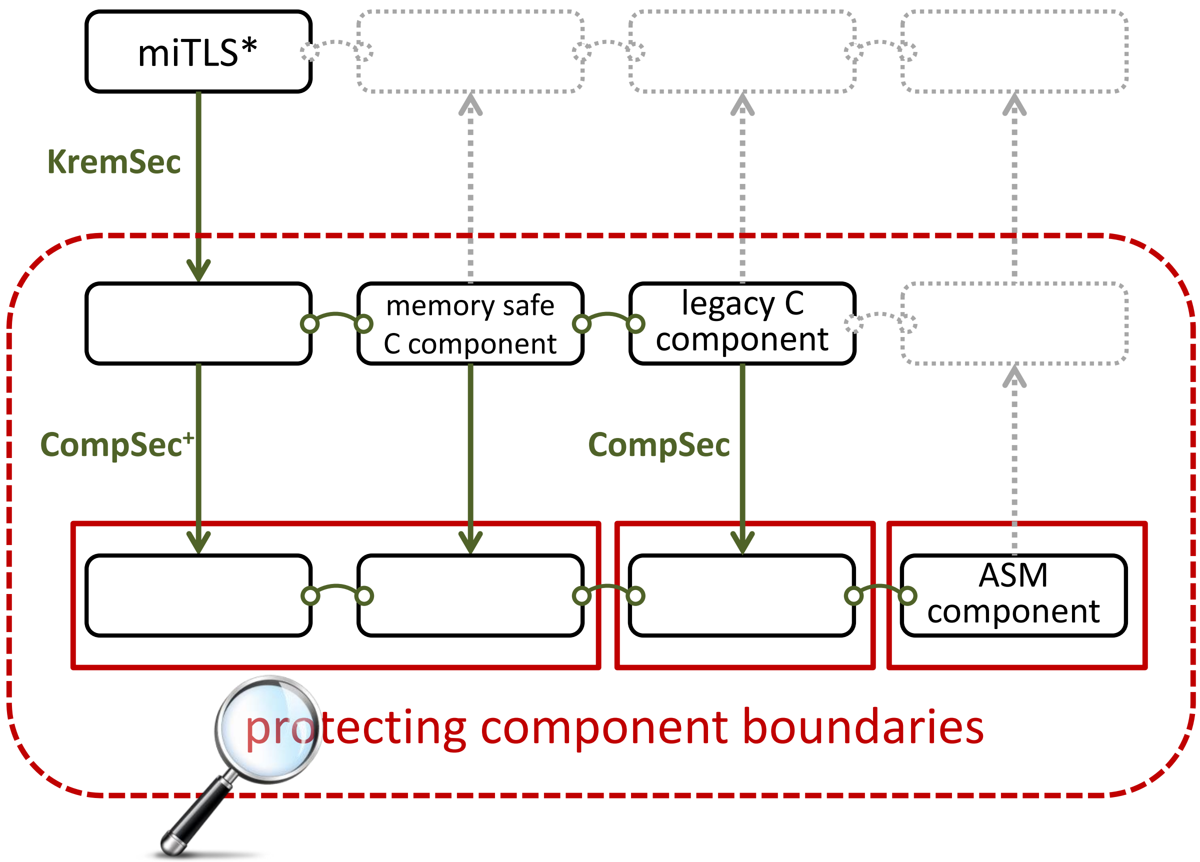
[= safe C subset embedded in F* for verification]

Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)



Formally Secure Compilation
of Unsafe Low-level Components

Collaborators


Cătălin Hrițcu


Marco Stronati

Guglielmo Fachini


Arthur Azevedo de Amorim


Ana Nora Evans

Théo Laurent


Deepak Garg


Marco Patrignani

Carmine Abate


Andrew Tolmach

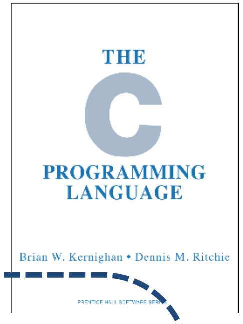

Benjamin Pierce


Yannis Juglaret

Inria Paris CMU U. Virginia ENS Paris UPenn Portland State MPI-SWS U. Trento

Compartmentalization

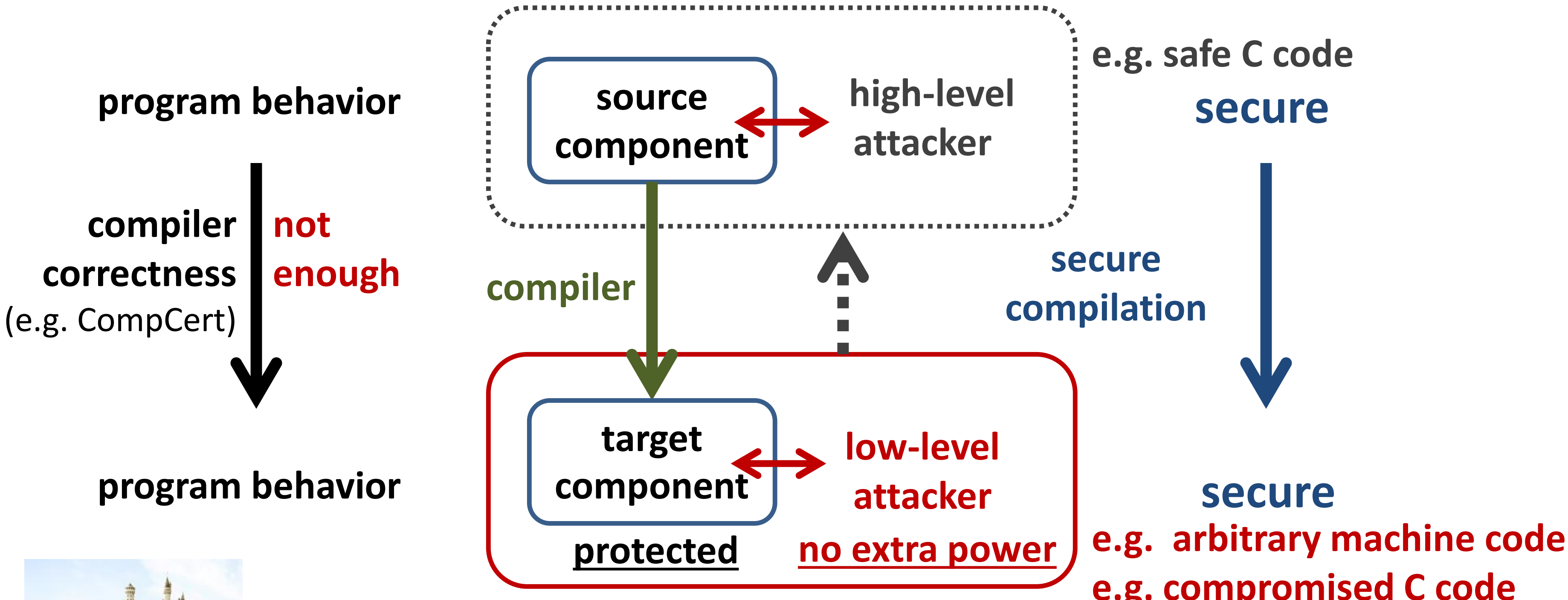
for unsafe, low-level languages



- **Add components to C-like language**
  - interacting only via **strictly enforced interfaces**
- **Secure compilation chain** **Goal: Build this**
  - use low-level security mechanisms to **efficiently enforce**: component separation, call and return discipline, ...
- **Interesting attacker model** **Goal: Formalize this**
  - **mutual distrust, dynamic compromise, least privilege**
    - e.g. dynamic compromise = "each component should be protected from all the others until it becomes compromised and starts attacking the remaining uncompromised components"

Formally secure compilation

holy grail of preserving security all the way down



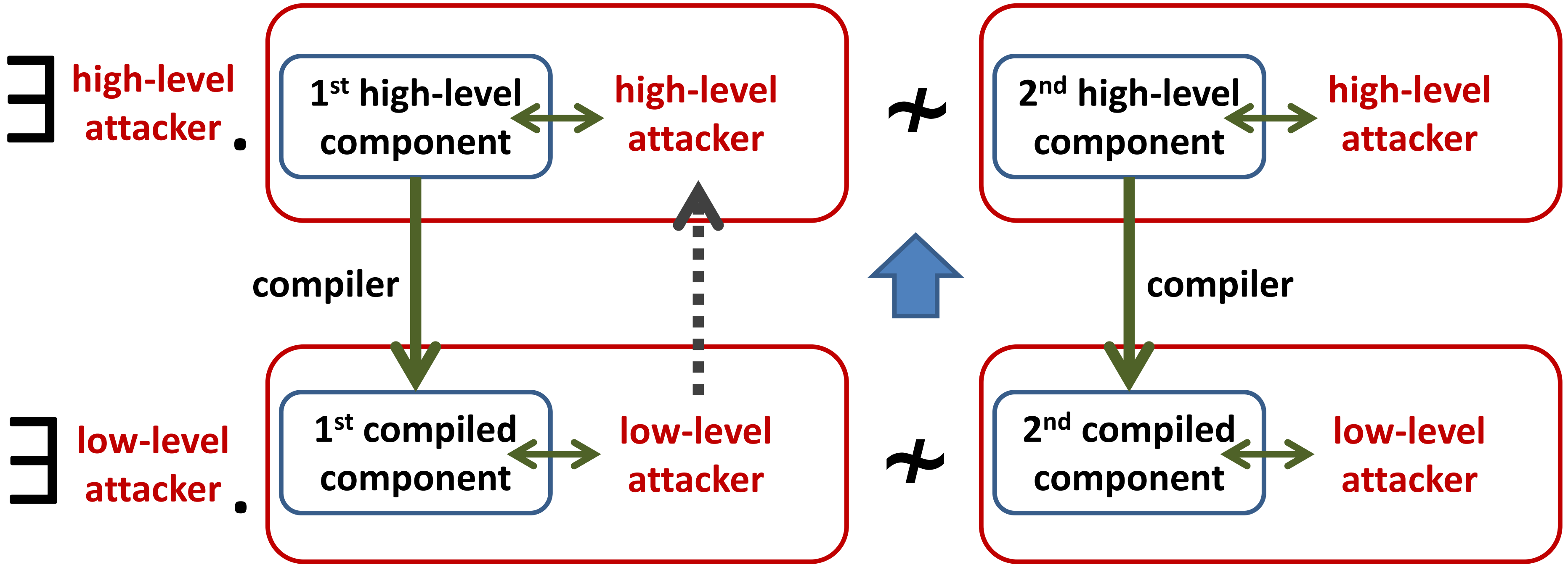
Benefit: sound security reasoning in the source language

forget about compilation chain (linker, loader, runtime)

forget that libraries are written in a lower-level language

Fully abstract compilation

preservation of observational equivalence



Undefined behavior

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

Buffer overflow

```
$ gcc target.c -fno-stack-protector
$ ./a.out haha
$ ./a.out hahahahahahahahahaha
zsh: segmentation fault (core dumped)
```

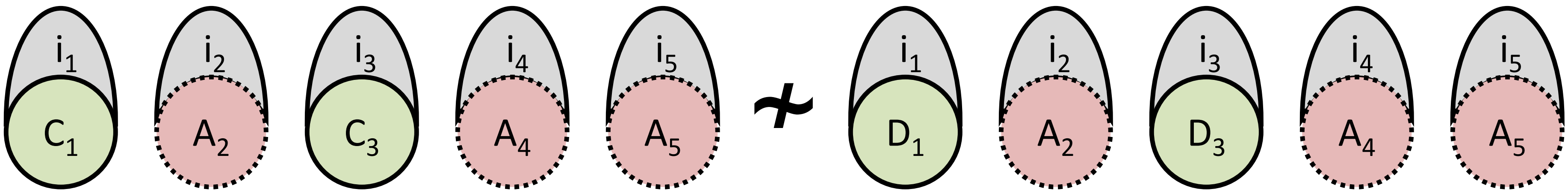
Source reasoning vs undefined behavior

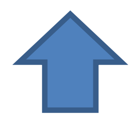
- **Source reasoning**
  = We want to reason formally about security
  with respect to source language semantics
- **Undefined behavior**
  = can't be expressed at all by source language semantics!
- **Observational equivalence doesn't work with undefined behavior!?**
  – `int buf[5]; buf[42] ~ int buf[5]; buf[43]?`
- **Can we somehow avoid undefined behavior?**

Full abstraction
with mutually distrustful components

∀compromise scenarios.

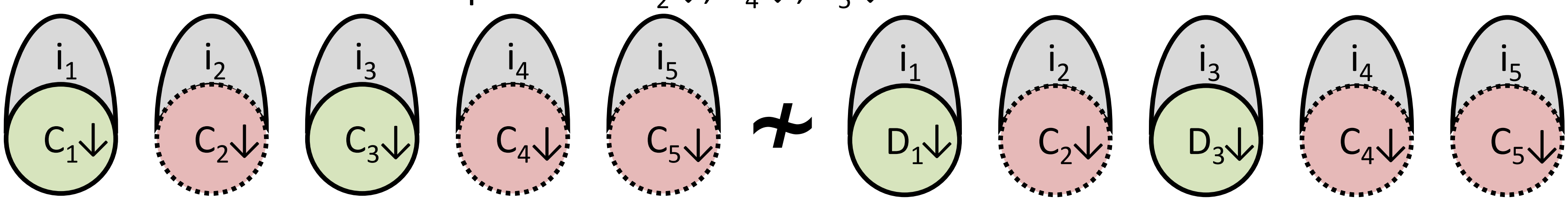
∃ high-level attack from some **fully defined** A_2, A_4, A_5





if C_1, C_3, D_1, D_3 **fully defined** and

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



Limitation: static compromise model: C_1, C_3, D_1, D_3 get guarantees only if perfectly safe
(i.e. fully defined = do not exhibit undefined behavior in **any** context)

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

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

Static compromise not good enough

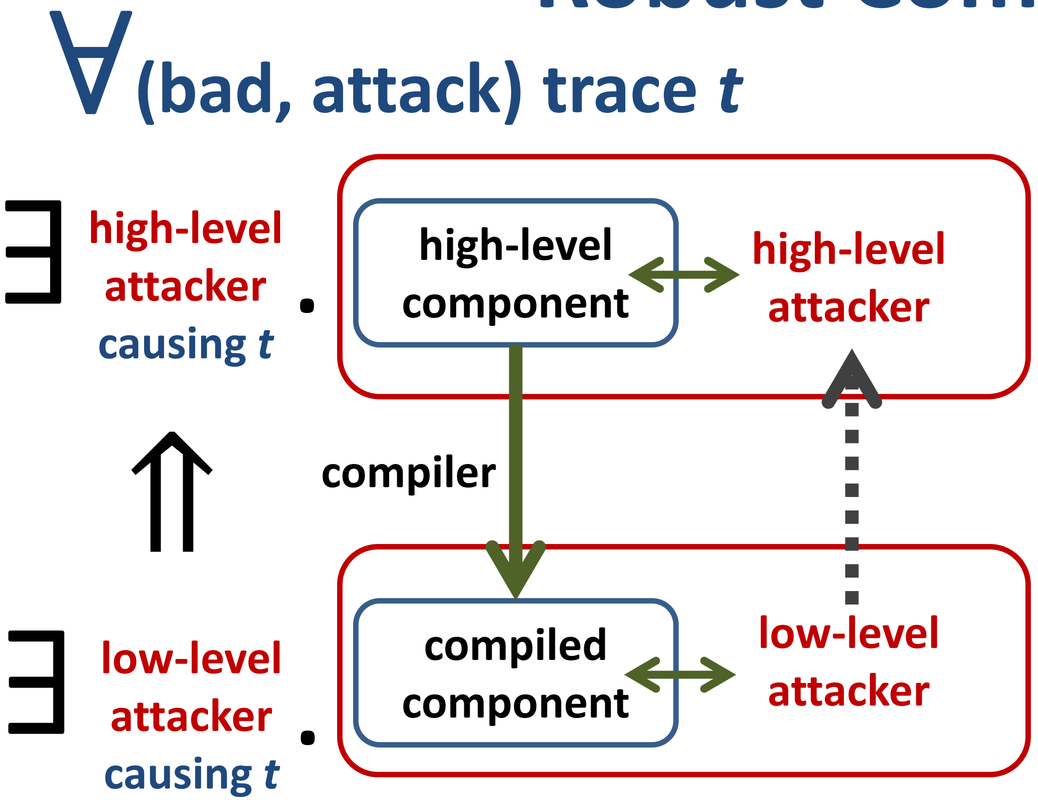
neither C_1 not C_2 are fully defined

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
}
```

New secure compilation criterion: Robust Compilation



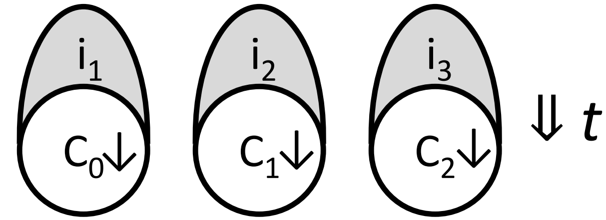
robust trace property preservation
(robust = in adversarial context)

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

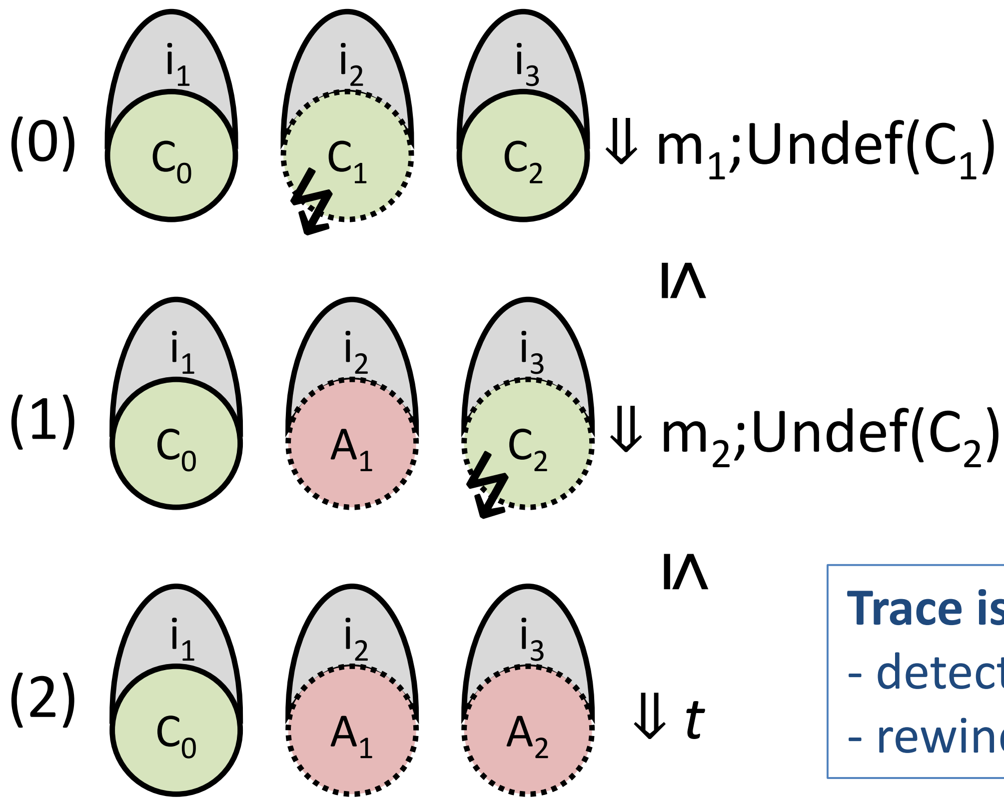
less extensional than full abstraction

Advantages: **easier to realistically achieve and prove at scale**
useful: preservation of **invariants** and other **integrity properties**
more intuitive to security people (generalizes to hyperproperties!)
extends to unsafe languages (supporting dynamic compromise)

Dynamic compromise



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

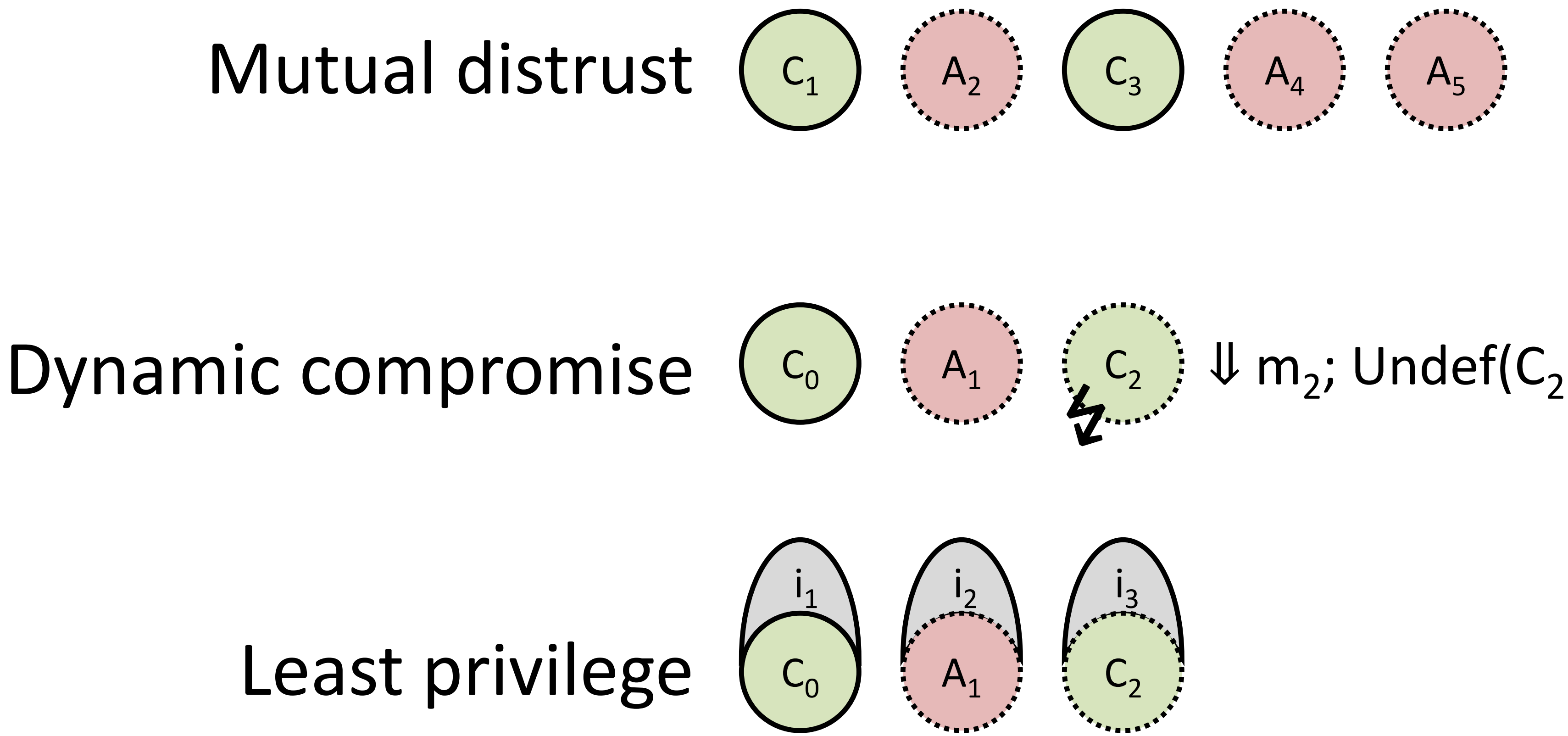


Trace is very helpful
- detect undefined behavior
- rewind execution

[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)



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

Simple Secure Compilation Chain

Verified (in Coq)

Compartmentalized unsafe source

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

Compartmentalized abstract machine

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

software fault isolation

Micro-policy machine

Standard machine

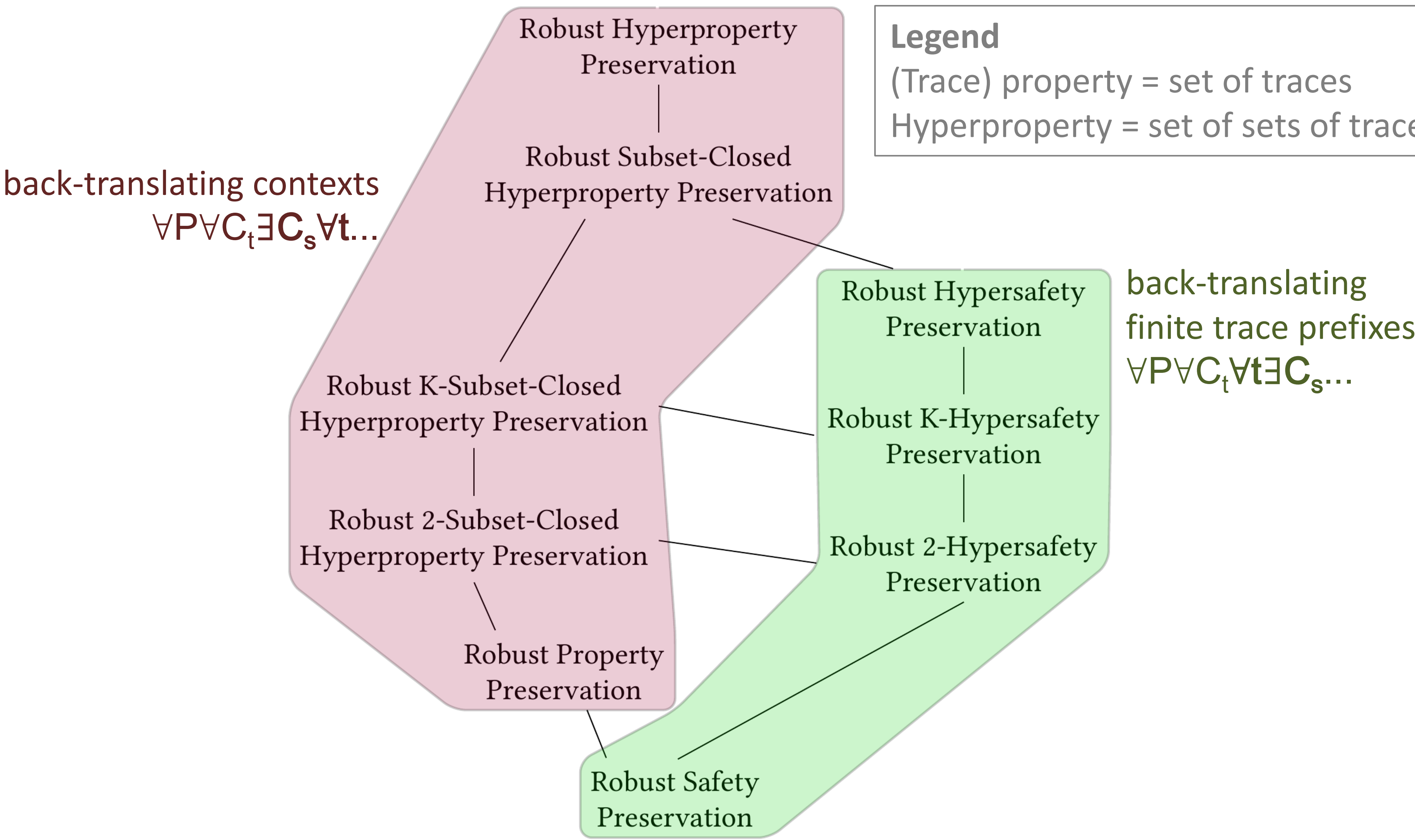
fallback

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline

Systematically tested (with QuickChick)

Beyond trace properties



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

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

Compartmentalization mechanisms

- **practically deployed ones**
  - process-level privilege separation (all web browsers)
  - software fault isolation (SFI, Google Native Client)
  - hardware enclaves (Intel SGX, ARM TrustZone)
- **and more on drawing boards**:
  - WebAssembly (WASM)
  - capability machines (CHERI)
  - tagged architectures (micro-policies)