Devastating low-level vulnerabilities

- **Inherently insecure C/C++-like languages**
  - **memory (and type) unsafe**:
    any buffer overflow is catastrophic
  - **root cause**, but challenging to fix:
    - efficiency
    - precision
    - scalability
    - backwards compatibility
    - deployment

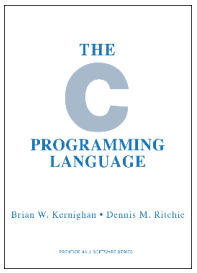
Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** running with **least privilege** & interacting via strictly enforced interfaces

- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component should not immediately destroy the security of the whole application"
  - "components can be compromised by buffer overflows"
  - "each component should be protected from all the others"

Goal 1: Formalize this

Goal 2: Build secure compilation chains



- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions**:
  - component separation, call-return discipline, ...
- **Secure compilation chain:**
  - compiler, linker, loader, runtime, system, hardware
- **Use efficient enforcement mechanisms:**
  - OS processes (all web browsers)
  - software fault isolation (SFI)
  - hardware enclaves (SGX)
  - — WebAssembly (web browsers)
  - — capability machines
  - — tagged architectures
- **Practical need for this** (e.g. crypto library/protocol)

Source reasoning vs undefined behavior

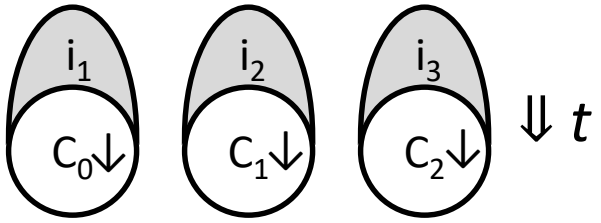
- **Source reasoning**

  = We want to reason formally about security with respect to source language semantics

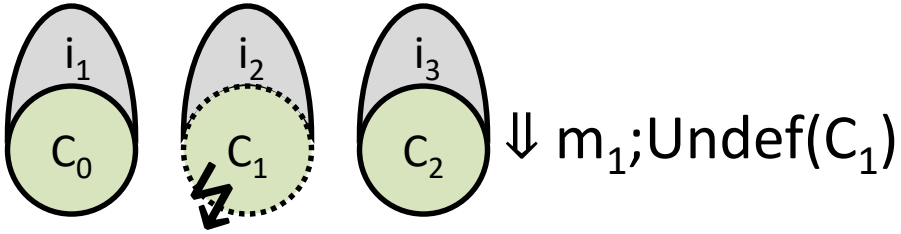
- **Undefined behavior**

  = can't be expressed at all by source language semantics!

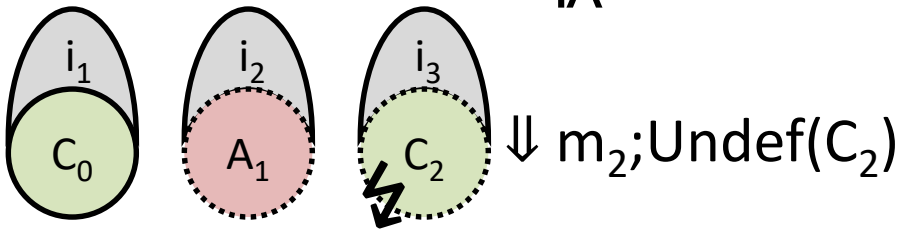
Dynamic compromise



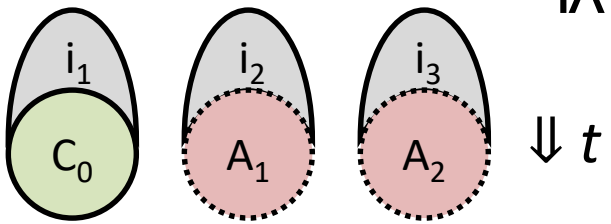
∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0) 

I∧

(1) $\exists A_1.$ 

I∧

(2) $\exists A_2.$ 

Trace is very helpful
- detect undefined behavior
- rewind execution

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

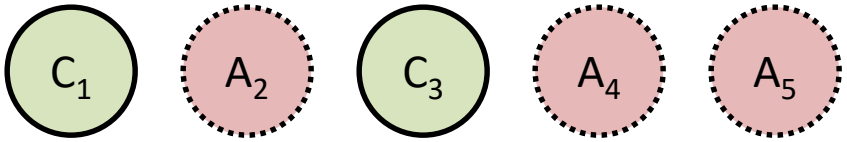
Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior** shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...
  - CompCert **already offers** this saner model
  - GCC and LLVM **currently violate** this model

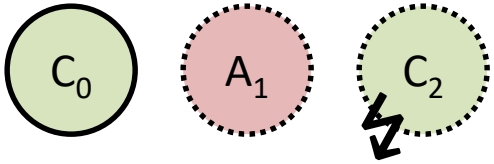
Now we know what these words mean!

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

Mutual distrust

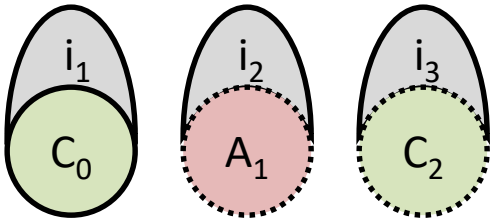


Dynamic compromise



$\Downarrow m_2$; Undef(C_2)

Static privilege



Towards Secure Compilation Chain

(mostly) 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

Bare-bone machine

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)

Systematically tested (with QuickChick)