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
Théo Laurent

Deepak Garg
Marco Patrignani
Carmine Abate
Andrew Tolmach
Benjamin Pierce
Yannis Juglare
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**
  - use compartmentalization to *efficiently enforce:* component separation, call and return discipline, ...

- **Interesting attacker model**
  - *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

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)

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

∀ (bad, attack) trace t

∃ high-level attacker causing t

∃ low-level attacker causing t

∀ (bad, attack) trace t

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]
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
}
∃ a dynamic compromise scenario explaining t in source language for instance ∃[A₁, A₂] leading to the following compromise sequence:

(0) C₀ ↓ C₁ ↓ C₂ ↓ m₁; Undef(C₁)

(1) C₀ ↓ A₁ ↓ C₂ ↓ m₂; Undef(C₂)

(2) C₀ ↓ A₁ ↓ A₂ ↓ t

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

Least privilege
Beyond trace properties

[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)

miTLS*

KremSec

memory safe C component

legacy C component

CompSec

CompSec

ASM component

protecting higher-level abstractions

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
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, ...