Formally Secure Compilation
of Unsafe Low-level Components

Cătălin Hrițcu
Inria Paris

https://secure-compilation.github.io
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**
  - **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

holly 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

 Advantage: easier to realistically achieve and prove

 useful: preservation of invariants and other integrity properties

 works for unsafe languages (supporting dynamic compromise)

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
Mutually distrustful components

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

∃ high-level attack from some fully defined A₂, A₄, A₅

↓ t

Limitation: static compromise
C₁ and C₃ fully defined

∃ low-level attack from compromised C₂↓, C₄↓, C₅↓

↓ t

C₁ and C₃ 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

```javascript
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
}
```

neither C1 nor C2 are fully defined
yet C1 is protected until calling C1.parse
and C2 can't actually be compromised
∃ a **dynamic compromise scenario** explaining t in source language for instance ∃[A₁,A₂] leading to the following compromise sequence:

1. (0) \[ C₀ \downarrow \xrightarrow{t} C₁ \downarrow \text{m}_{1};\text{Undef}(C₁) \]
2. (1) \[ C₀ \xleftarrow{t} A₁ \downarrow \text{m}_{2};\text{Undef}(C₂) \]
3. (2) \[ C₀ \xleftarrow{t} A₁ \xleftarrow{t} A₂ \downarrow \text{t} \]

**Trace is very helpful**
- detect undefined behavior
- rewind execution
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]

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

back- translating contexts
∀P∀C_t∃C_s∀t...

finite trace prefixes
∀P∀C_t∀t∃C_s...
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 component boundaries

protecting higher-level abstractions
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, ...