Compartimentalizing

Formally Secure Compilation

Cătălin Hrițcu

Inria Paris

https://secure-compilation.github.io
Secure compilation has various goals

• Preventing low-level attacks and Enabling source-level security reasoning
  — by compartmentalizing compilation

• Making the source language safer
  — memory and type safety, less/no undefined behavior

• Making it easier to express security intent
  — marking secrets, specifying security properties

• Making exploits more difficult
  — CFI, CPI, stack protection, randomization, diversity

This course is only about
Devastating low-level attacks

Part 2: give meaning to mitigation (protected components)

inherently insecure languages like C/C++
- e.g. memory unsafe: any buffer overflow is catastrophic allowing remote attackers to gain complete control
- ~100 different undefined behaviors in usual C compiler

insecure interoperability with lower-level code
- even code in more secure languages (Java, OCaml, Rust) has to interoperate with low-level code (C, C++, ASM)
- insecure interoperability: all source-level guarantees lost

Part 1: formalize what it means to solve this problem
Secure Interoperability with Lower-Level Code

Journey Beyond Full Abstraction:
Exploring Robust Property Preservation for Secure Compilation

https://arxiv.org/abs/1807.04603
Good programming languages provide helpful abstractions for writing more secure code

- e.g. HACL* and miTLS written in Low* which provides:
  - low-level abstractions associated with safe C programs
    - structured control flow, procedures, abstract memory model
  - higher-level abstractions associated with ML-like languages
    - modules, interfaces, and parametric polymorphism
  - most features of verification systems like Coq and Dafny
    - effects, dependent types, logical pre- and post-conditions
  - patterns specific to cryptographic code
    - abstract types and interfaces for mitigating side-channel attacks
Abstractions not enforced when linking with adversarial low-level code

Insecure interoperability: compromised (or malicious) application linking in miTLS can easily read and write miTLS’s data and code, jump to arbitrary instructions, smash the stack, ...
Secure compilation

• Protect source-level abstractions even against linked adversarial low-level code
• Enable source-level security reasoning
  – even adversarial target-level context cannot break security properties of compiled program any more than some source-level context could
  – no "low-level" attacks
Three important concerns for secure compilation

1. **What are we trying to achieve?**
   - Identifying and formalizing secure compilation criteria and attacker models

2. **How can we achieve it efficiently?**
   - Compartmentalization can be achieved using: OS processes, software-fault isolation, hardware enclaves, tagged architectures, capability machines

3. **How can we prove it effectively?**
   - E.g. (bi)simulations, logical relations, game semantics, ...
Source-level security reasoning

But what does "secure" mean?
What security properties should we preserve?

• We explored a large space of security properties

• Studied preserving various classes of ...
  – trace properties (safety, liveness)
  – hyperproperties (e.g. noninterference)
  – relational hyperproperties (e.g. trace equivalence)

... against adversarial target-level contexts

• No “one-size-fits-all solution”
  – e.g. full abstraction does not imply the other criteria we study
  – stronger criteria are harder to achieve and prove, both challenging
More secure

More efficient

Easier to prove
Robust Trace Property Preservation

**property-based characterization**

\[ \forall \text{source component.} \]

\[ \forall \pi \text{ trace property.} \]

\[ \forall \text{source component.} \]

\[ \forall t \Rightarrow t \in \pi \]

**preservation of robust satisfaction**

**property-free characterization**

\[ \forall \text{source component.} \]

\[ \forall (\text{bad attack}) \text{ trace } t. \]

\[ \forall t \Rightarrow t \Rightarrow t \in \pi \]

**how one can prove it**
Some of proof difficulty is manifest in property-free characterization

back-translating context
∀C_T∃C_S∀P∀t...

back-translating prog & context
∀P∀C_T∃C_S∀t...

back-translating
finite set of
finite trace prefixes
∀k∀P_1..P_k∀C_T
∀m_1..m_k∃C_S...

back-translating
finite trace prefix
∀P∀C_T∀m≤t∃C_S...

Some of proof difficulty is manifest in property-free characterization
Summarizing recent results [arXiv:1807.04603]

• Mapped the space of secure compilation criteria based on robust "property" preservation
  – **Property-free characterizations** and **implications** in Coq
  – **Separation results** (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)
  – **Collapse** between preserving all hyperproperties and preserving just hyperliveness

• Showed that **even strongest criterion is achievable**
  – for simple translation from a statically to a dynamically typed language with first-order functions and I/O
Some open problems

• Practically achieving secure interoperability with lower-level code
  – More realistic languages and secure compilation chains
  – Achieve robust noninterference preservation in realistic attacker model with side-channels
  – Efficient enforcement mechanisms

• Scalable proof techniques for other criteria
  – robust (hyper)liveness preservation (possible?)

• Proving robust satisfaction for source programs
  – partial semantics, program logics, logical relations, ...
Where is full abstraction?

Q: Under what extra assumptions does full abstraction imply anything?

compiler correctness not enough!

divergence finitely observable

with internal nondeterminism

without internal nondeterminism
Wrapping up the intro

• 1. Secure interoperability with lower-level code
   – exploring a continuum, security vs efficiency tradeoff

• We're hiring! now!
  – PostDocs, Young Researchers, Interns, PhD students
Plan for the rest of this course

• **0. Compiler correctness**
  – as Trace Property Preservation

• **1. Secure interoperability with lower-level code**
  – Secure 2-Compartmentalizing Compilation as **Robust** Property Preservation
  – Property $\in$ Trace Properties, Hyperproperties, Relational ...

• **2. Secure compilation despite dynamic compromise**
  – Compartmentalizing Compilation for **Unsafe Languages**
  – Restricting the temporal + spatial scope of **undefined behavior**
And pay no attention to the little man behind the curtain...