$$\underbrace{\text{Formally Secure Compilation}}^{\text{Compartmentalizing}}$$

Compartmentalizing Formally Secure Compilation

Cătălin Hrițcu

Inria Paris

https://secure-compilation.github.io

Course outline

- **0. Compiler correctness**
  - as Trace Property Preservation
- **1. Secure interoperability with lower-level code**
  - Secure 2-Compartmentalizing Compilation as **Robust** Property Preservation
  - Property ∈ **Trace Properties**, Hyperproperties, Relational ...
- **2. Secure compilation despite dynamic compromise**
  - Compartmentalizing Compilation for **Unsafe Languages**
  - Restricting the temporal + spatial scope of **undefined behavior**

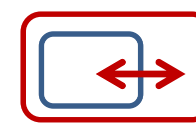
Devastating low-level attacks

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

Part 2 of 2

When Good Components Go Bad

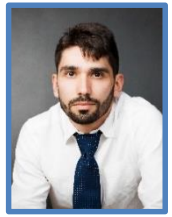
Secure Compilation Despite Dynamic Compromise

To appear @ Computer and Communications Security (CCS 2018)

https://arxiv.org/abs/1802.00588

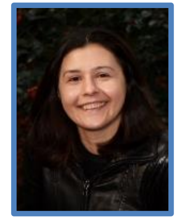
Collaborators for Part 2

Carmine Abate



Arthur Azevedo de Amorim

Rob Blanco

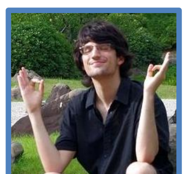


Ana Nora Evans

Guglielmo Fachini

Florian Groult

Cătălin Hrițcu



Yannis Juglaret

Théo Laurent

Benjamin Pierce

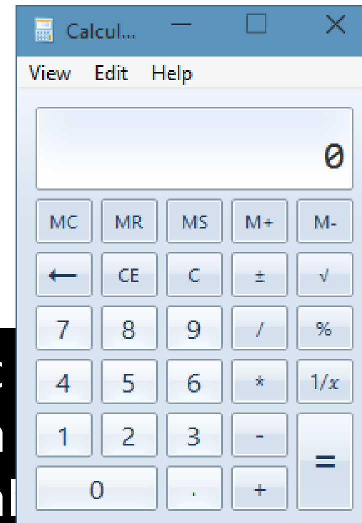
Marco Stronati

Andrew Tolmach

Inria Paris CMU U. Virginia U. Trento Paris 7 ENS Paris Portland State UPenn

Undefined behavior

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



erflow

```
$ gcc target.c
$ ./a.out haha
$ ./a.out hahal                haha
zsh: segmentation fault (core dumped)
$ ./exploit.sh | a.out
```

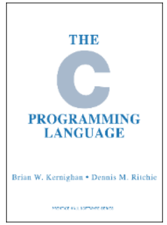
Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"
  - "each components receives guarantees as long as it has not encountered undefined behavior"

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



Goal 1: Formalizing the security of compartmentalizing compilation

Restricting undefined behavior

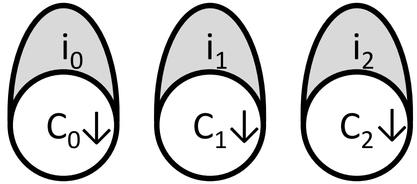
- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable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

Dynamic compromise

- each component gets guarantees as long as it has not encountered undefined behavior

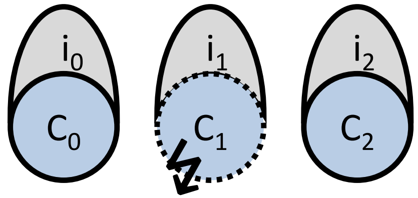
- a component only loses guarantees after an attacker discovers and exploits a vulnerability

- the mere existence of vulnerabilities doesn't immediately make a component compromised

If 

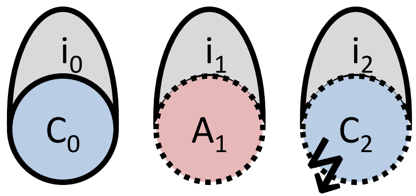
$\rightsquigarrow t$ then

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

(0) 

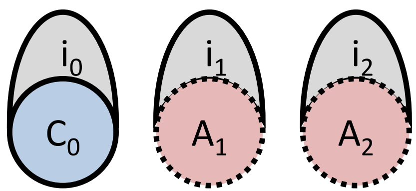
$\rightsquigarrow^* m_1 \cdot \mathrm{Undef}(C_1)$

$\leq$

(1) $\exists A_1.$ 

$\rightsquigarrow^* m_2 \cdot \mathrm{Undef}(C_2)$

$\leq$

(2) $\exists A_2.$ 

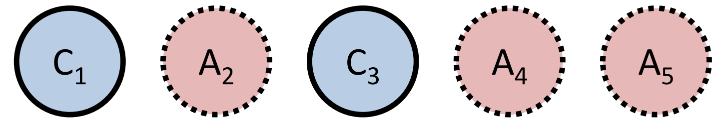
$\rightsquigarrow t$

Trace is very helpful
- detect undefined behavior
- rewind execution

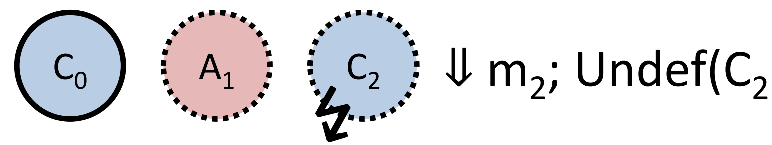
Now we know what these words mean!

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

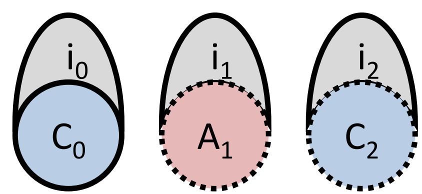
Mutual distrust



Dynamic compromise



Static privilege



Goal 2: Towards building secure compilation chains

(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

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

Bare-bone machine

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)

Systematically tested (with QuickChick)

Making this **more practical** ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...
- **Support other enforcement mechanisms (back ends)**
- **Measure & lower overhead**

Wrapping up

- **1. Secure interoperability with lower-level code**
  - **exploring a continuum, security vs efficiency tradeoff**
- **2. Secure compilation despite dynamic compromise**
  - **restricting the scope of undefined behavior**
    - **spatially** to the component that caused it
    - **temporally** by treating UB as an observable trace event
- **We're hiring! now!**
  - **PostDocs, Young Researchers,** Interns, PhD students