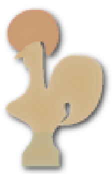
When Good Components Go Bad

Formally Secure Compilation Despite Dynamic Compromise



Cătălin Hrițcu

Inria Paris

https://secure-compilation.github.io

10 Co-authors ⇒ 100% acceptance rate

Carmine Abate

Arthur Azevedo de Amorim

Rob Blanco

Ana Nora Evans

Guglielmo Fachini

Cătălin Hrițcu

Théo Laurent

Benjamin Pierce

Marco Stronati

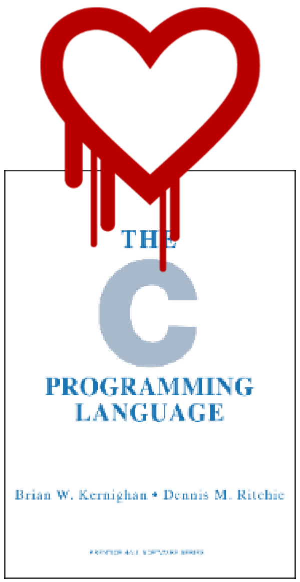
Andrew Tolmach

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

Compartmentalization can defend against devastating low-level attacks

Inherently insecure languages like C

- any **buffer overflow** can be catastrophic
- ~100 different **undefined behaviors** in the usual C compiler:
  - **use after frees and double frees, invalid casts, signed integer overflows, ...**
- **root cause**, but very challenging to fix:
  - **efficiency**, precision, scalability, backwards compatibility, deployment



Compartmentalization mitigation

- **Break up security-critical applications** into **mutually distrustful components** with **clearly specified privileges**
- **Protect component abstraction all the way down**
  - separation, static privileges, call-return discipline, types, ...
- **Compartmentalizing compilation chain:**
  - compiler, linker, loader, runtime, system, hardware
- **Base this on efficient enforcement mechanisms:**
  - OS processes (all web browsers)
  - software fault isolation (SFI)
  - hardware enclaves (SGX)
  - WebAssembly (web browsers)
  - capability machines
  - tagged architectures

Strong security!?

- **Security guarantees one can make fully water-tight**
  - beyond just "increasing attacker effort"
- **Intuitively, ...**

... **a vulnerability in one component does not immediately destroy the security of the whole application**

... **since each component is protected from all the others**

... and each component receives protection as long as it has not been compromised (e.g. by a buffer overflow)

Can we formalize this intuition?

What is a compartmentalizing compilation chain supposed to enforce precisely?

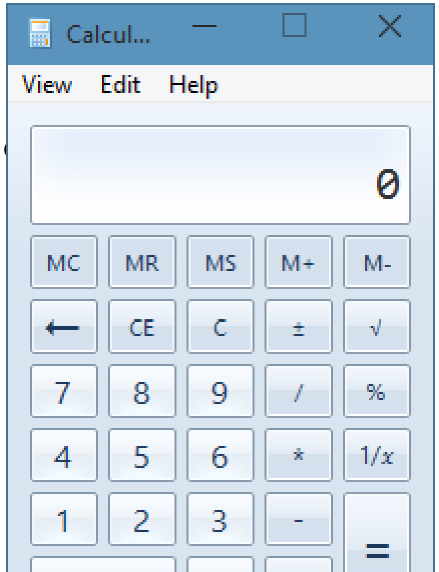
This paper answers this question:

Formal definition expressing the end-to-end security guarantees of compartmentalization

Challenge formalizing security of mitigations

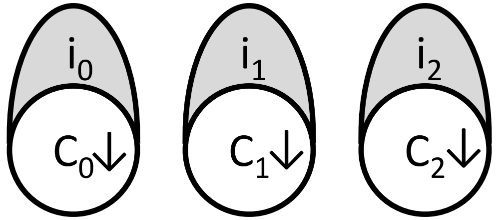
- **We want source-level security reasoning principles**
  - easier to **reason about security in the source language** if and application is compartmentalized
- **... even in the presence of undefined behavior**
  - can't be expressed at all by source language semantics!
  - **what does the following program do?**

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



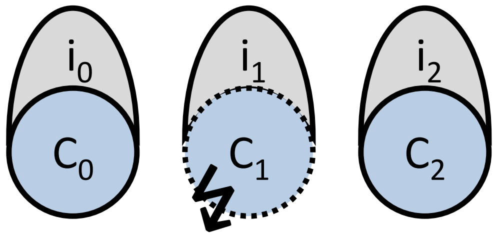
Compartmentalizing compilation should ...

- **Restrict spatial scope** of undefined behavior
  - **mutually-distrustful components**
    - **each component protected from all the others**
- **Restrict temporal scope** of undefined behavior
  - **dynamic compromise**
    - **each component gets guarantees as long as it has not encountered undefined behavior**
    - i.e. the mere existence of vulnerabilities doesn't necessarily make a component compromised

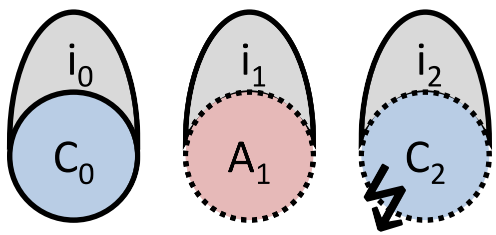
Security definition: If 

$\rightsquigarrow t$ then

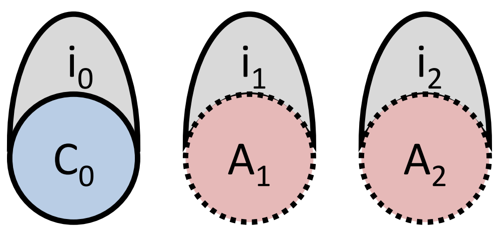
$\exists$ a sequence of component compromises explaining the finite trace t in the source language, for instance $t = m_1 \cdot m_2 \cdot m_3$ and

(1) 

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

(2) $\exists A_1.$ 

$\rightsquigarrow^* m_1 \cdot m_2 \cdot \mathrm{Undef}(C_2)$

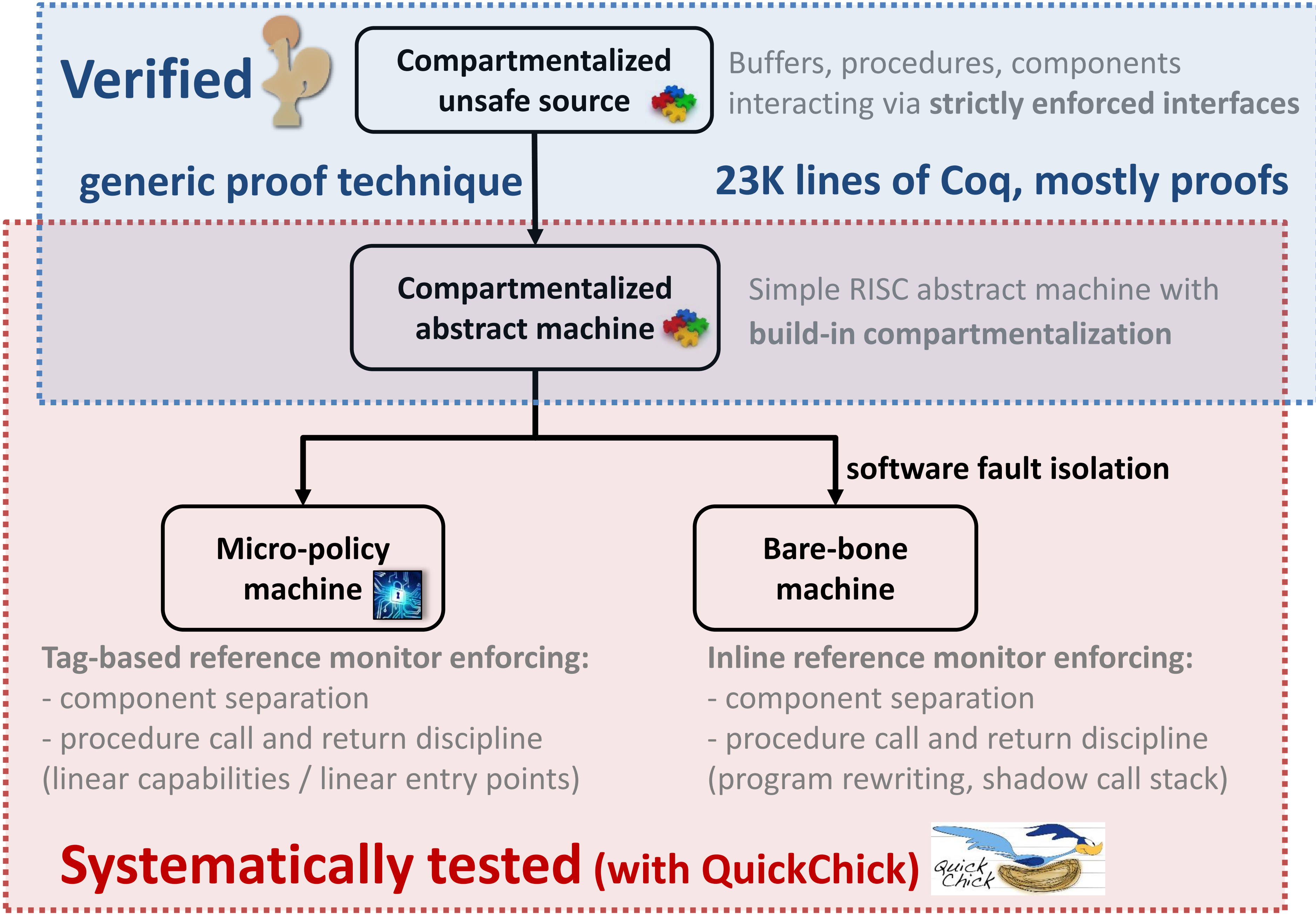
(3) $\exists A_2.$ 

$\rightsquigarrow m_1 \cdot m_2 \cdot m_3$

Finite trace records which component encountered undefined behavior and allows us to rewind execution

Proof-of-concept formally secure compilation chain in Coq

Illustrates our formal definition



https://secure-compilation.github.io

When Good Components Go Bad

- **Formalized security of compartmentalization**
  - **first definition** supporting **dynamic compromise**
  - **restricting undefined behavior spatially** and **temporally**

- **Proof-of-concept secure compilation chain in Coq**
  - **software fault isolation** or **tag-based reference monitor**
- **Generic definition and proof technique**
  - **we expect them to extend and scale well (ask me about it!)**

- **We're hiring!**
  - **PostDocs, Young Researchers**, Students

https://secure-compilation.github.io

Making this more practical ... next steps:

- **Scale formally secure compilation chain to C language**
  - ongoing: allow **shared memory** and **pointer passing** (capabilities)
  - eventually support enough of C to **measure and lower overhead**
  - eventually support **more enforcement mechanisms** (back ends)
- **Extend all this to dynamic component creation**
  - rewind to when compromised component was created
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, history-based access control, ...
- **From robust safety to hypersafety (eg confidentiality)**

  [Exploring robust property preservation for secure compilation, arXiv:1807.04603]

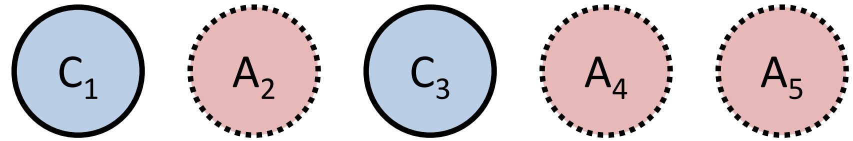
https://secure-compilation.github.io

BACKUP SLIDES

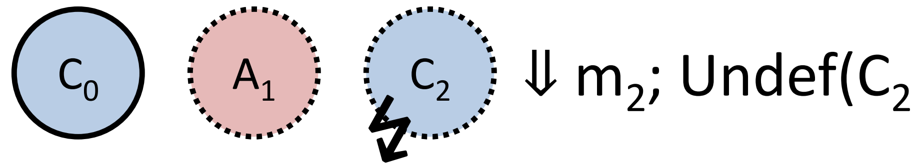
Now we know what these words mean!

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

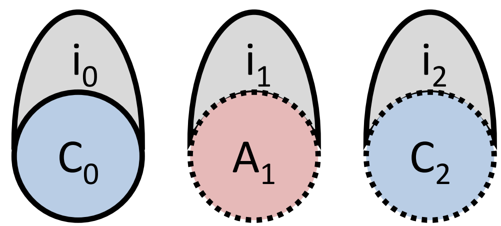
Mutual distrust

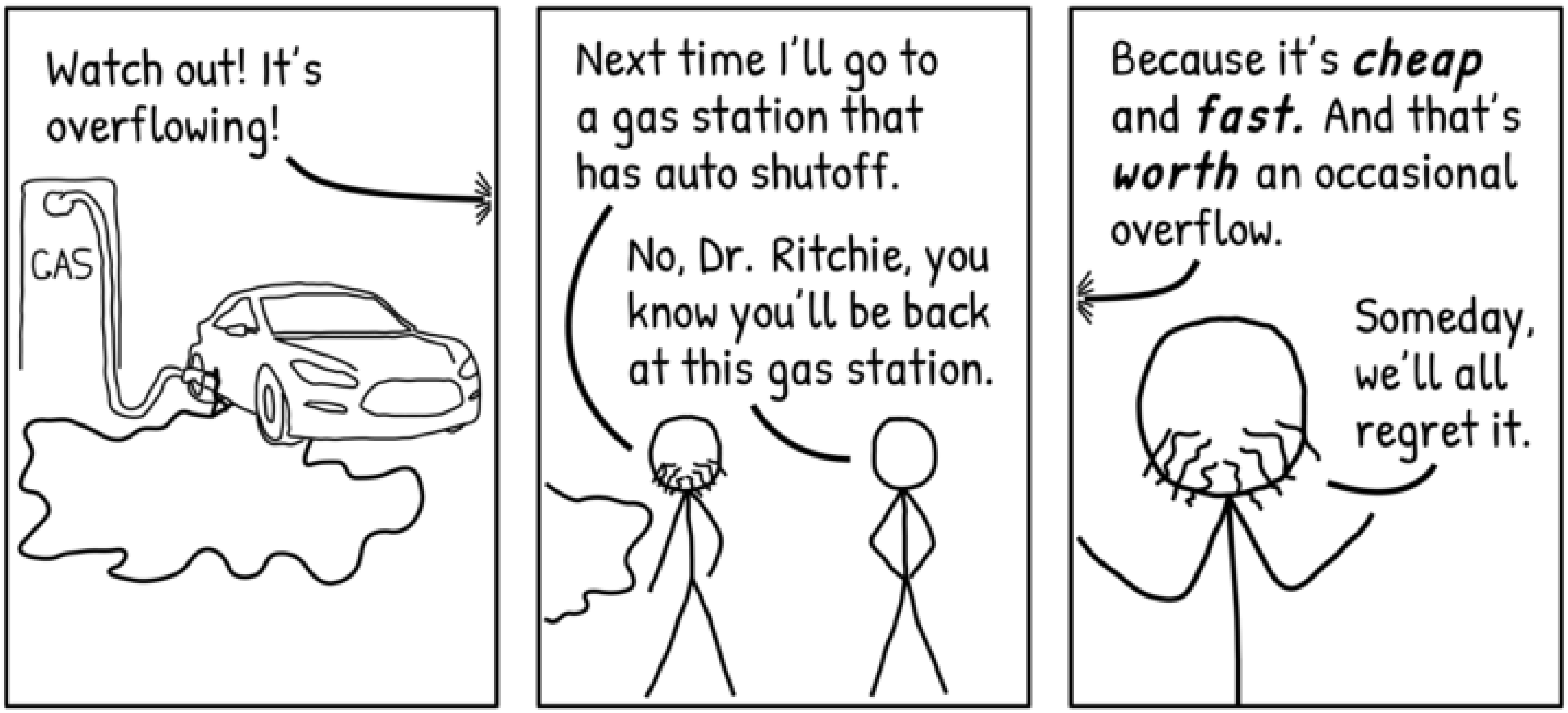


Dynamic compromise



Static privilege





Buffer Overflow.

icanbarelydraw.com

Restricting undefined behavior

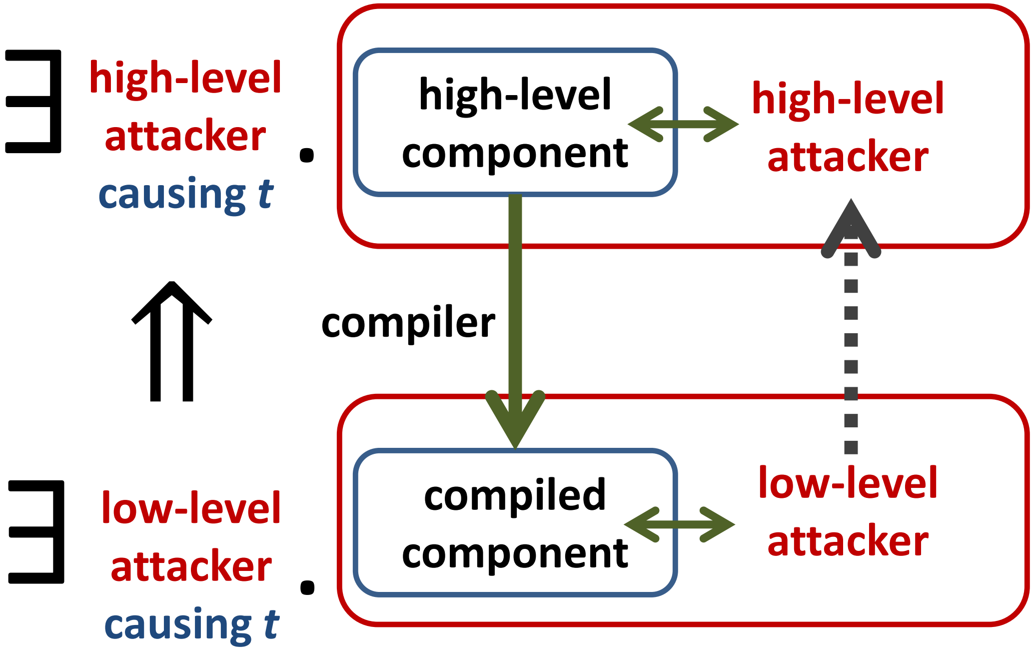
- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable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

We build this on Robust Compilation

∀(bad attack) trace *t*



robust trace property preservation
(robust = in adversarial context)

intuition:

- **stronger** than compiler correctness (i.e. trace property preservation)
- **confidentiality not preserved** (i.e. no hyperproperties)
- **less extensional** than fully abstract compilation

Advantages: **easier to realistically achieve and prove at scale**

useful: preservation of **invariants** and other **integrity properties**

generalizes to preserving [relational] hyperproperties!

extends to unsafe languages, supporting dynamic compromise

Scalable proof technique

for our extension of robustly safe compilation

1. **back-translating finite trace prefixes to whole source programs**
   – limitation: only works for preserving (hyper)safety
2. **generically defined semantics for partial programs**
   – related to whole-program semantics via **trace composition** and **decomposition lemmas**
3. **using whole-program compiler correctness proof (à la CompCert) as a black-box**
   – for moving back and forth between source and target

all this yields much simpler and more scalable proofs

Making this stronger ... beyond safety

[Exploring Robust Property Preservation For Secure Compilation, arXiv...]