Formally Secure Compilation of Unsafe C Compartments

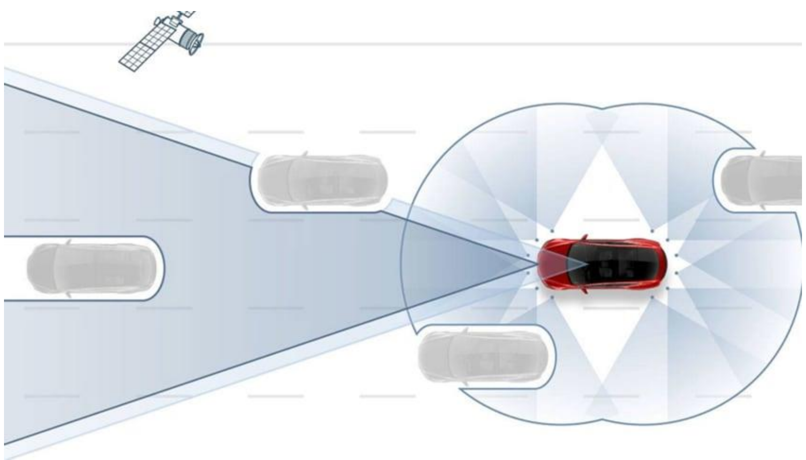
Cătălin Hrițcu, MPI-SP, Bochum

Joint work with
Carmine Abate, Cezar-Constantin Andrici, Arthur Azevedo de Amorim,
Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, **Akram El-Korashy**, Boris Eng,
Ana Nora Evans, Guglielmo Fachini, **Deepak Garg**, Aïna Linn Georges, Théo Laurent,
Guido Martínez, **Marco Patrignani**, Benjamin Pierce, Exequiel Rivas, Marco Stronati,
Éric Tanter, Jérémy Thibault, Andrew Tolmach, Théo Winterhalter, ...

In part supported by ERC Starting Grant SECOMP

We are increasingly reliant on computers

… trusting them with our ~~digital~~ lives



Computers vulnerable to hacking

Windows 10 zero-day exploit code released online

Security researcher 'SandboxEscaper' returns with new Windows LPE zero-day.

By Catalin Cimpanu for Zero Day | May 22,

Heartbleed vulnerability may have been exploited months before patch [Updated]

Fewer servers now vulnerable, but the potential damage rises.

GOOGLE \ TECH \ ANDROID

Google finds Android zero day that can take control of Pixel and Galaxy devices

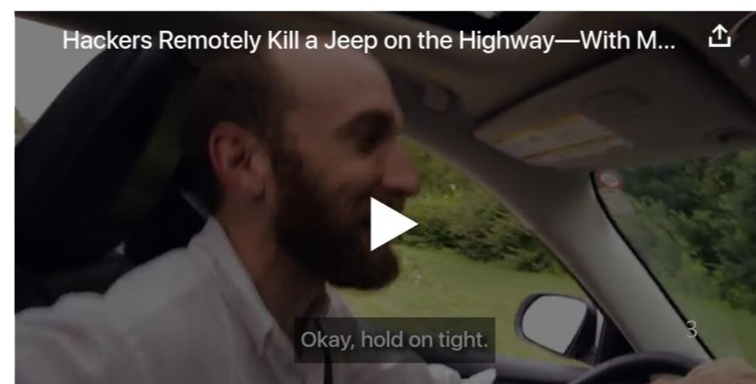
21

Affecting devices from Samsung, Huawei, and Google itself

By Jon Porter | @JonPorty | Oct 4, 2019, 8:42am EDT

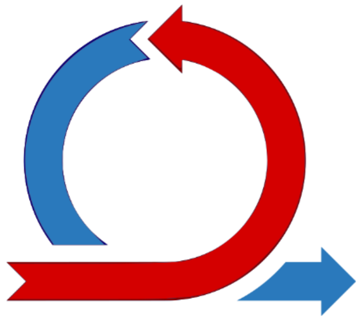
SHARE

Hackers Remotely Kill a Jeep on the Highway—With Me in It



Need to break the exploitation cycle

- Once the stakes are high enough, **attackers will find a way to exploit *any* vulnerability**

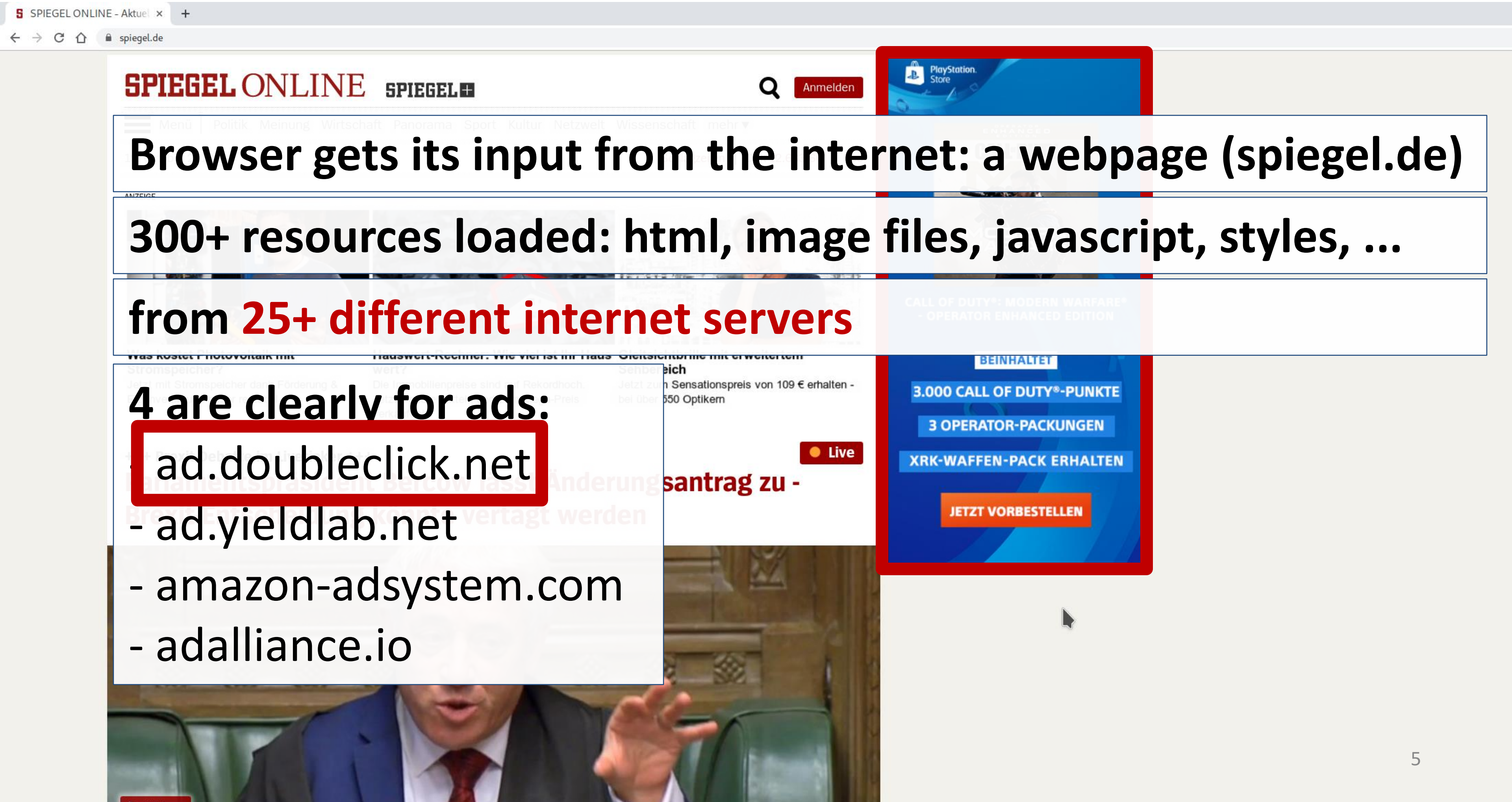


- **Weak security defenses** get deployed,

We need a deeper understanding that we can use to build provably secure defenses

-
  - defenders find clever ways to "increase attacker effort"
  - **attackers find clever ways around them**

Web browsers are frequently hacked



Browser gets its input from the internet: a webpage (spiegel.de)

300+ resources loaded: html, image files, javascript, styles, ...

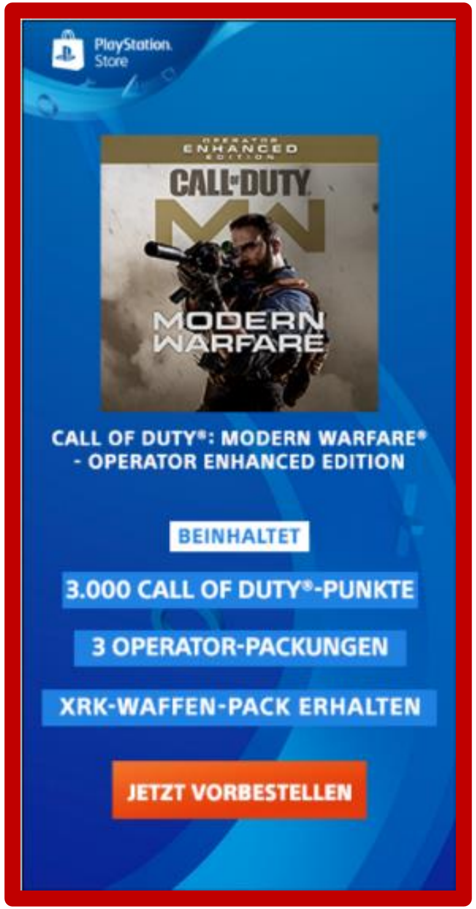
from **25+ different internet servers**

4 are clearly for ads:

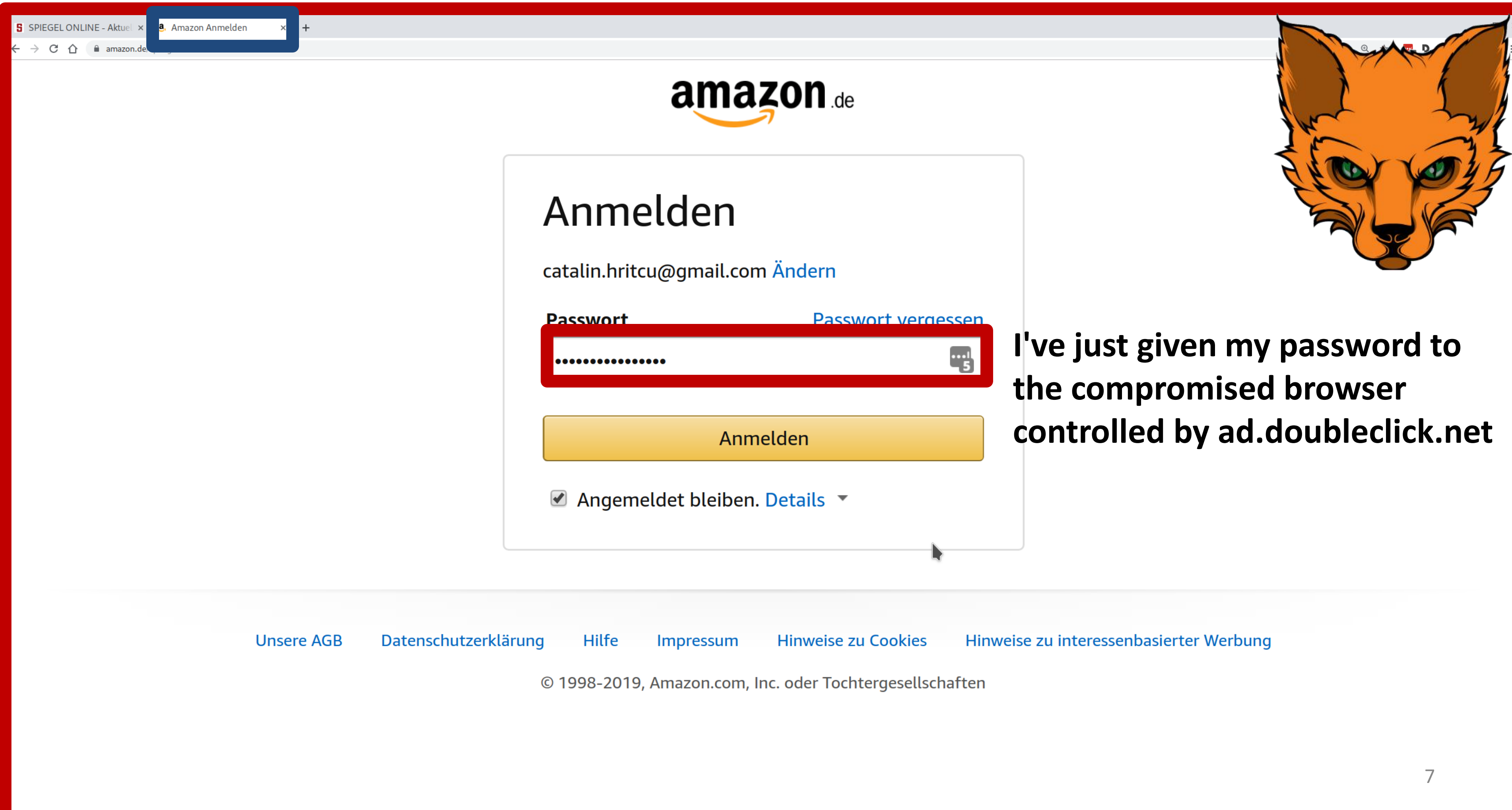
- ad.doubleclick.net
- ad.yieldlab.net
- amazon-adsystem.com
- adalliance.io

Malicious server can hack the browser

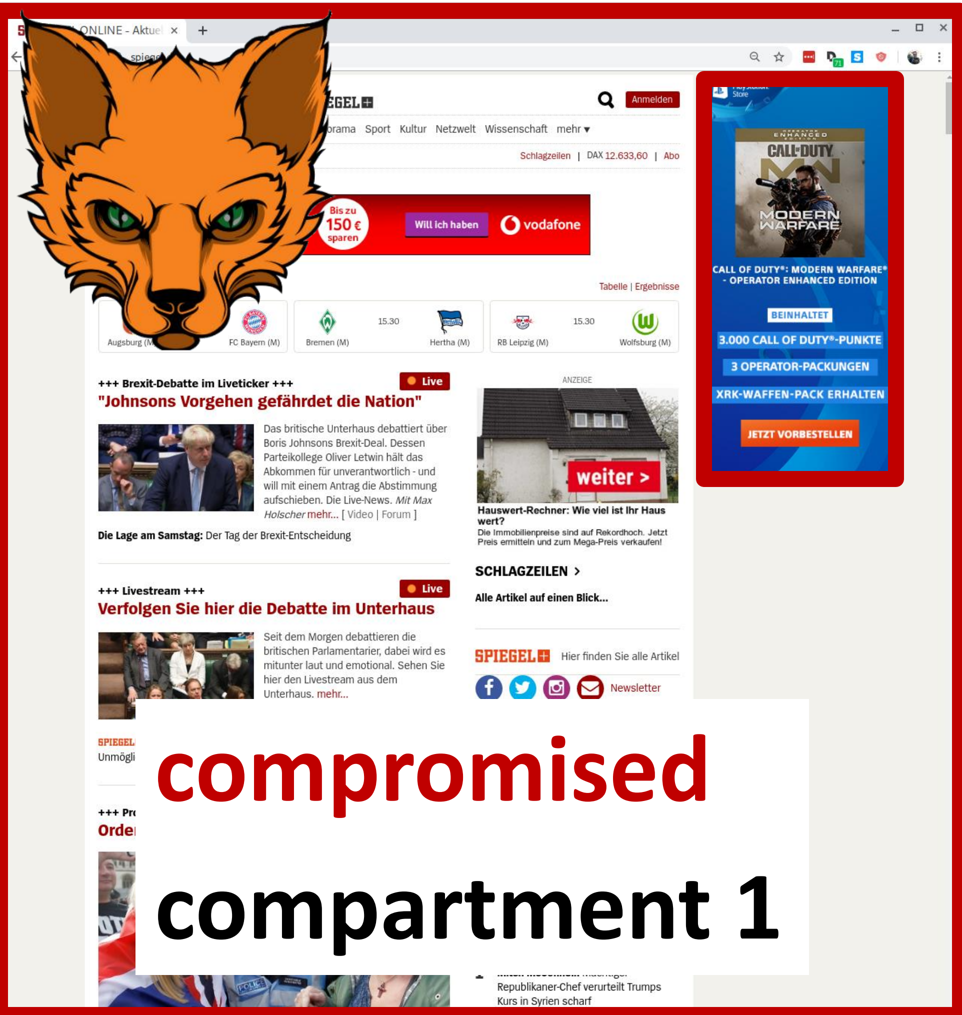
- send it an image that **looks like an** ad
- **specially crafted to exploit a vulnerability** in the browser's image drawing engine
- **this compromises the whole browser**
  - i.e. gives server **complete control** over it
- **malicious server can now:**
  - steal the user's data
  - take control of the victim's computer
  - encrypt victim's data and ask for ransom

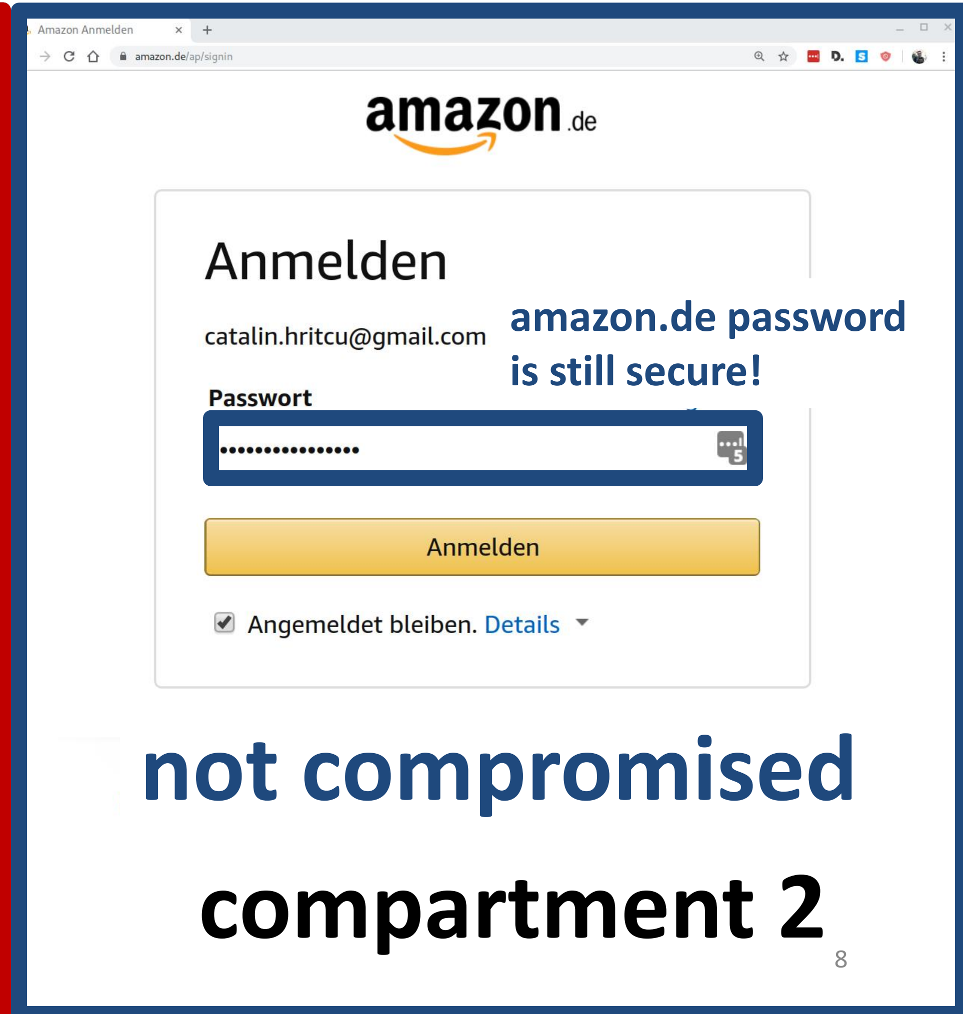


Compromised browser can steal user's data



Compartmentalization can help





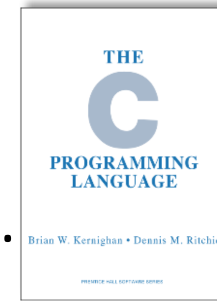
Good news: browsers now compartmentalized!

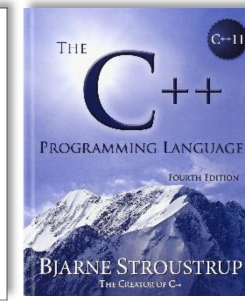
- **each tab indeed started in separate compartment**

Bad news, so far:

- **limited compartmentalization mechanisms**
  - **compartments coarse-grained, most often OS processes**
    - can compartmentalize tabs, but not origins or resources within a tab
  - **compartments can't naturally interact**
    - even for tabs this required big restructuring of web browsers

Source language compartments





- Mozilla Firefox mostly implemented in C/C++
- Programming languages like C/C++, Rust, Java, ... already provide **natural abstractions** for **fine-grained compartmentalization**:
  - procedures, interfaces, classes, objects, modules, libraries, ...
  - a **compartment** can be a library/module/class or even an object (e.g., an image or an origin)
- **In the source language fine-grained compartments are easy to define** and **can naturally interact**

Source language compartments
(simple example in simplified source language)

```
compartment C1 {
  private var x;

  private procedure p() {
    x := get_counter();
    x := password; ←not allowed
  }
}
```

```
compartment C2 {
  private var counter;
  private var password;

  public procedure get_counter() {
    counter := counter + 1;
    return counter;
  }
}
```

Abstractions lost during compilation

- **Computers don't directly run C/C++, Java, Rust, or F***
  – **Compiler** translates Firefox **from C/C++/Rust to machine code instructions**
- **All high-level abstractions lost during compilation**
  – no procedures, no interfaces, no classes, no objects, no modules, ...
- **Secure compilation**
  – **preserve compartment abstractions through compilation, enforce them all the way down**
- **Shared responsibility of the whole compilation chain:**
  – source language, compiler, operating system, and hardware
- **Goal: secure compilation chain for compartmentalized code**

Machine-code level

Compartment C_1

<<check rx∈C_1>>

load r ← [rx] **←not allowed**

put rc ← $a_{password}$

<<check rx∈C_1 **←not enough**

or rx∈C_2's interface>>

jump-and-link rx **←not allowed**

sub r ← r-1

Compartment C_2

put rc ← $a_{counter}$
load r ← [rc]
add r ← r+1
store r → [rc]
jump ra

compiled get_counter (public procedure)

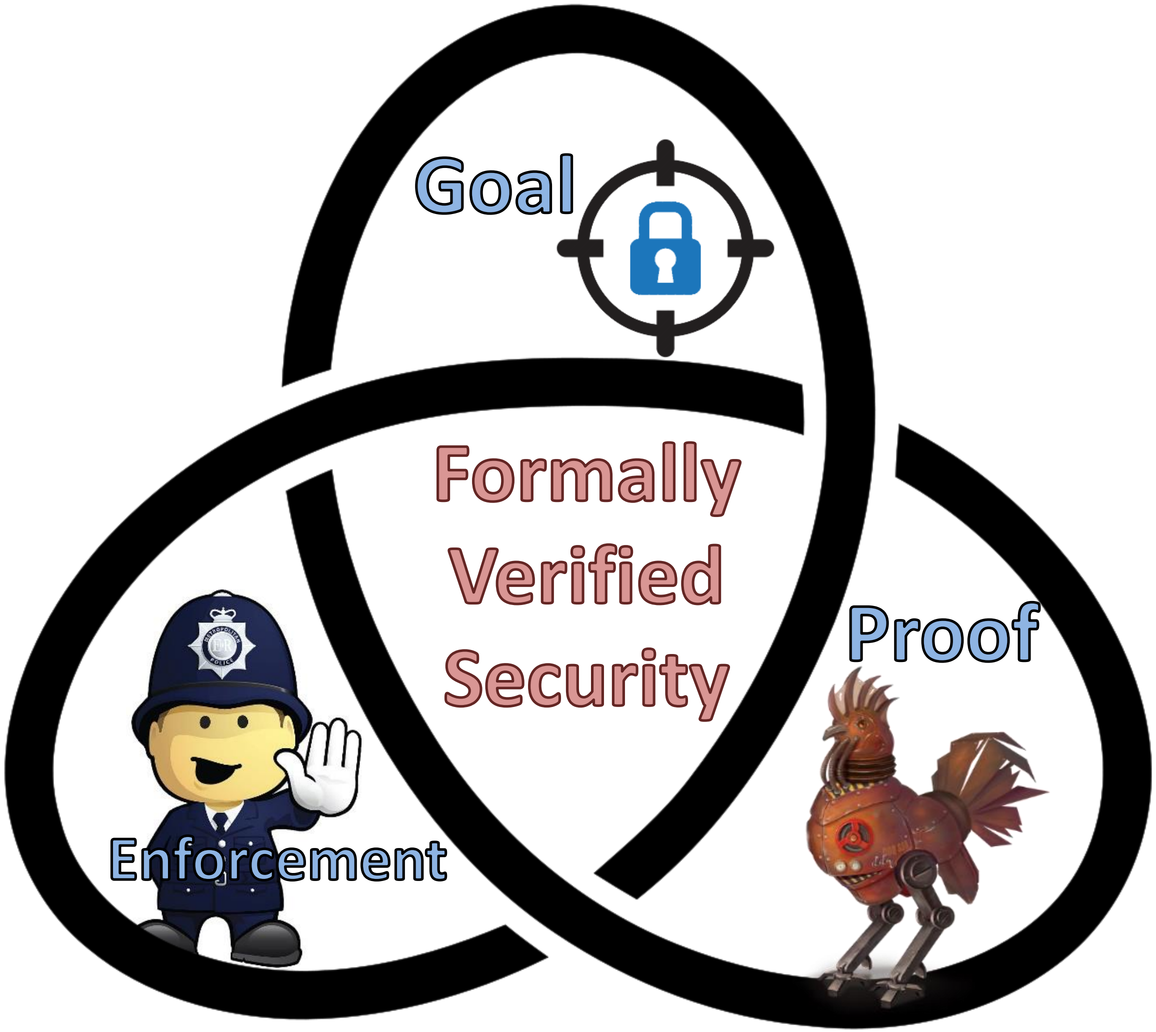
$a_{counter}$: 42

$a_{password}$: …

Securely enforcing source abstractions is challenging!

e.g. software checks complicated (uncircumventable, efficient)

Formally Secure Compilation of C Compartments



1. Security Goal

- **What does it mean for a compilation chain for unsafe C compartments to be secure?**
  - **formal definition expressing end-to-end security guarantees**
  - **these guarantees were not understood before**
- **Will only show an easier definition**
  - **protecting 1 trusted compartment from 1 untrusted one**
  - **untrusted compartment arbitrary** (e.g. compromised Firefox)
  - **trusted compartment has no vulnerabilities**

This is not just hypothetical!

Firefox

Mozilla shipping EverCrypt verified crypto library

(also used by Microsoft, Linux, ...)

[POPL'16,'17,'18,'20, ICFP'17,'19, ESOP'19, CPP'18, SNAPL'17]

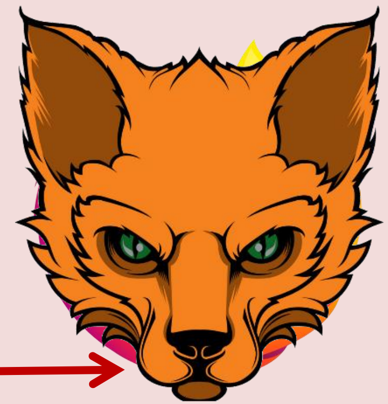
Formal verification milestone:

40.000+ lines of highly-efficient code, mathematically proved to be free of vulnerabilities

(and functionally correct and side-channel resistant)

Putting things into perspective

EverCrypt
(verified in F*)



Firefox

40.000 lines

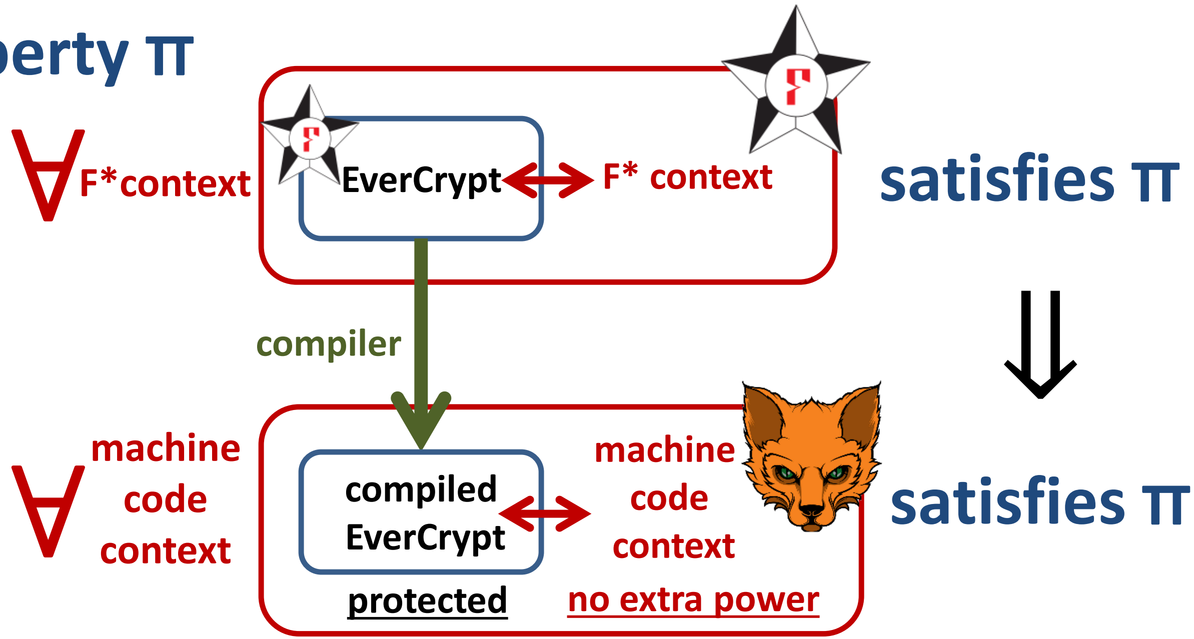
20.000.000 lines
+ external libraries
all unverified

Without compartmentalization interoperability is insecure:
if Firefox is compromised it can break security of verified code

What does secure compartmentalization mean in this setting?

Preserving security against adversarial contexts

$\forall$ security property π



Where "security property" can e.g., be $\boxed{\text{data confidentiality}}$

π = "private key is not leaked"

Journey Beyond Full Abstraction [CSF'19, ESOP'20, TOPLAS'21]

No one-size-fits-all security criterion

More secure

relational hyperproperties
(trace equivalence)

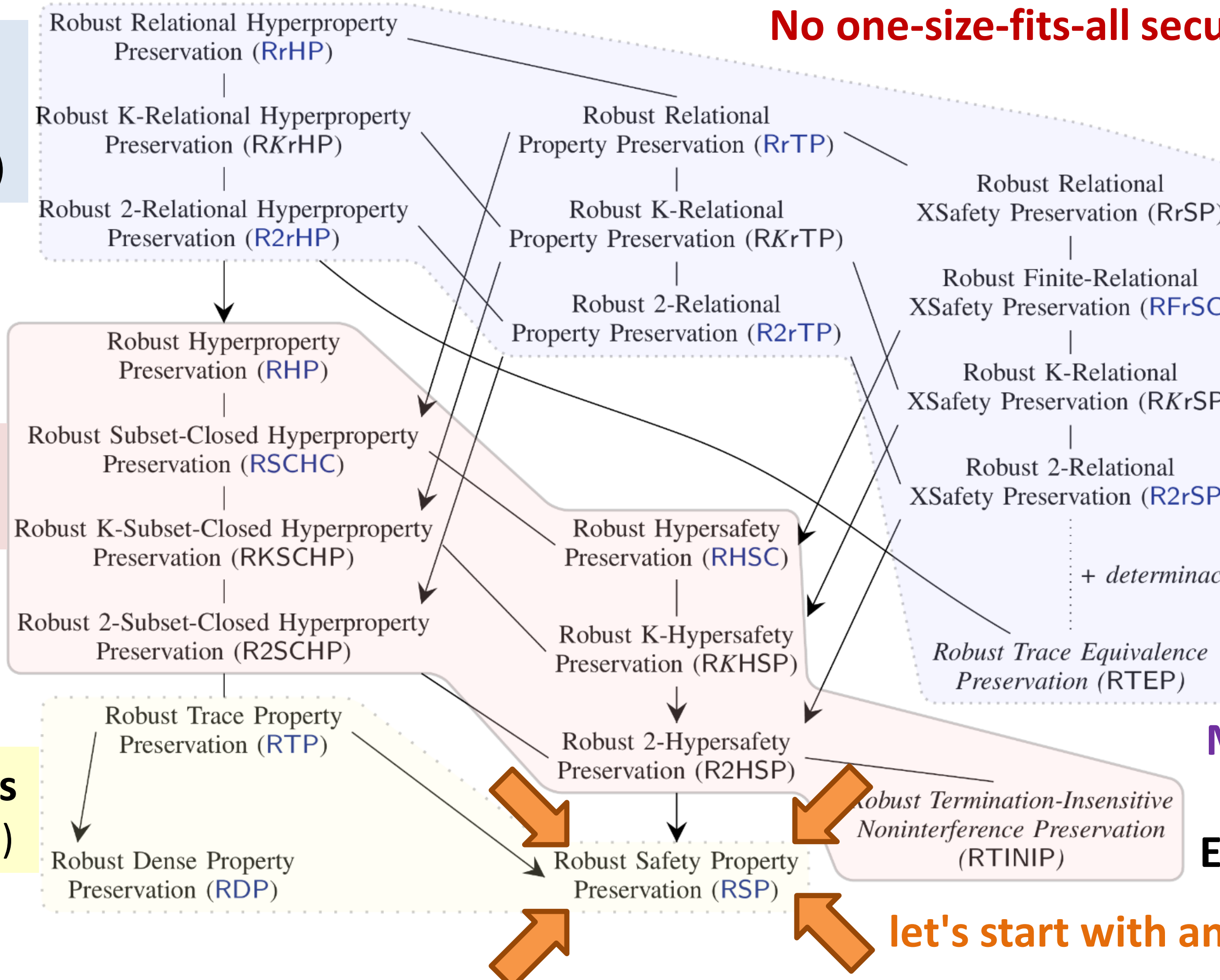
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



More efficient to enforce

Easier to prove

let's start with an "easier" one

fine for code without vulnerabilities (F*) , but ...

Extra challenges in defining secure compilation for unsafe C compartments [CSF'16, CCS'18]

- Program split into **many mutually distrustful compartments**
- **We don't know which compartments will be compromised**
  - **every compartment should be protected from all the others**
- **We don't know when a compartment will be compromised**
  - **every compartment should receive protection until compromised**

2. Security Enforcement

CompCert C with compartments

large subset of the C language (ISO C 2011)

CompCert verified C compiler extended with compartments

CompCert RISC-V ASM with compartments

magically secure semantics for RISC-V ASM

Software-Fault Isolation

vanilla ASM

Done for simplified languages, yet to be ported to RISC-V

Micro-Policies: ASM with programmable tags

[POPL'14, S&P'15, ASPLOS'15, POST'18, CCS'18, CSF'23 subm.]

CHERI RISC-V capability machine

(inspiration for ARM Morello)

Hardware-accelerated enforcement

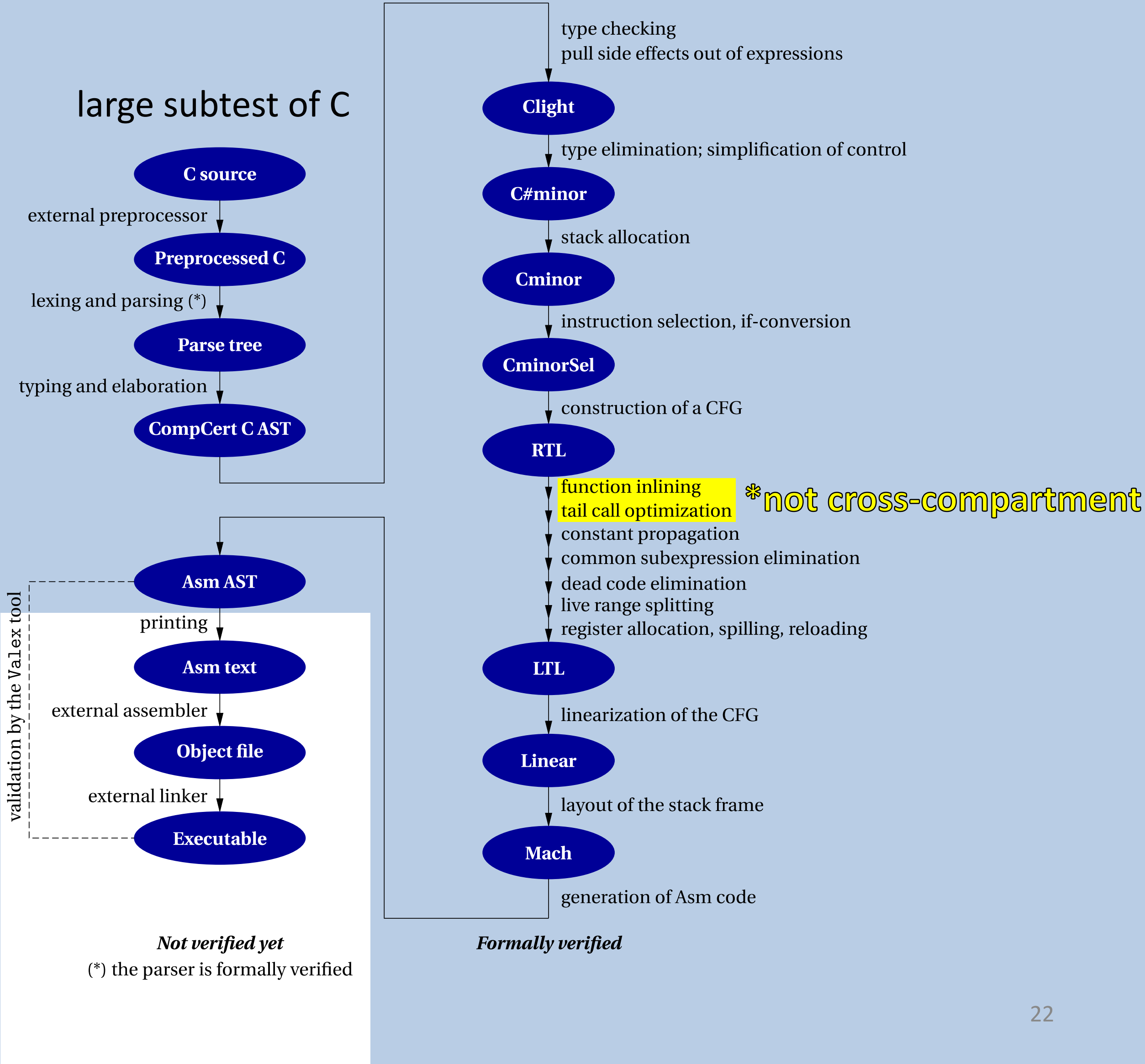
CompCert extended with compartments

mutually distrustful,
with clearly specified interfaces,
interacting via procedure calls

all 15 verified compilation passes*
from Clight to RISC-V ASM
(magically secure semantics)

compiler correctness proofs
a lot of work, reusing for security

large subtest of C



Capabilities Backend

- **Targeting the CHERI RISC-V capability machine**
  - capabilities = unforgeable pointers with base and bounds
- **Secure and efficient calling convention enforcing stack safety**
  [Aïna Linn Georges et al, Le temps de cerises, OOPSLA 2022]
  - **Uninitialized capabilities**: cannot read memory before initializing
  - **Directed capabilities**: cannot access old stack frames
- Mutual distrustful compartments: **capability-protected wrappers**
  - on calls and returns clear registers and
    prevent passing capabilities between compartments
- Also investigating **calling convention based solely on wrappers**
  - no new kind of capability over what CHERI already provides
  - but more interesting stack layout (not a single contiguous block)

3. Security Proof

- **Proving mathematically that our compilation chain for C compartments achieves secure compilation**
  - such proofs generally **very difficult and tedious**
    - wrong conjectures survived for decades
    - 250 pages of proof on paper for toy compiler
  - we propose a **more scalable proof technique**
  - focus on **machine-checked proofs** in the Coq proof assistant
    - with **property-based testing** stopgap to find bugs early

Testing and Proving secure compilation in Coq

Verification

CompCert C with compartments

CompCert with compartments

CompCert RISC-V ASM with compartments

Scalable proof technique for secure compilation
- applied to simpler languages [CCS'18, CSF'22]
- currently porting to CompCert with compartments
- reuses our extended compiler correctness proof
- aiming to finish this in the next couple of months

  –milestone for secure compilation in terms of scale/realism

Software-Fault Isolation

vanilla ASM

Micro-Policies: ASM with programmable tags

Done for simpler languages, yet to be ported to RISC-V

Next verification challenge

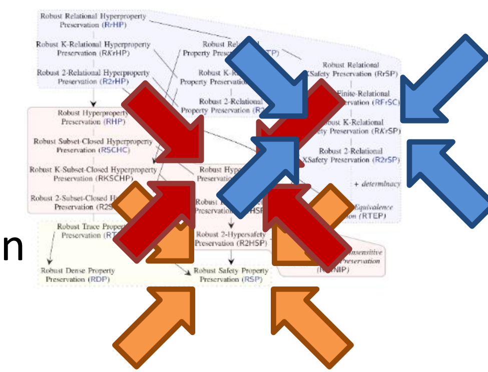
CHERI RISC-V capability machine

Systematic testing with QuickChick [POPL'17, ICFP'13, ITP'15, JFP'16]

Future work: extending proof technique

- **Verifying backends more challenging**
  - can't hide all information about compartment's code (memory layout)
  - proof step inspired by full abstraction no longer works (recomposition)
- **Fine-grained dynamic memory sharing** by **capability passing**
  - already proved in Coq in simpler setting [Akram El-Korashy et al, CSF'22]
- **Beyond preserving safety against adversarial contexts**
  - towards preserving **hyperproperties** (data confidentiality)
  - even **relational hyperproperties** (observational equivalence)
    - secure compilation criteria strictly stronger than full abstraction
    - can do this for CompCert, but won't hold for backends

[Jérémy Thibault et al, CSF'19, ICFP'21 submission]



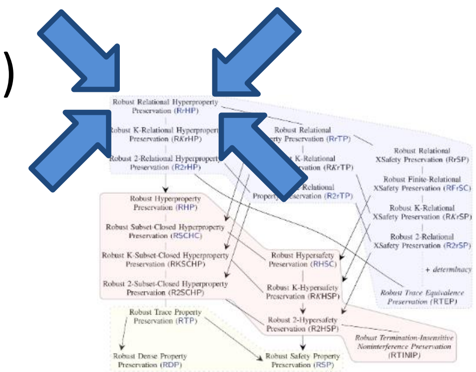
[CSF'19, ESOP'20, TOPLAS'21]

Future work (continued)

- **Enforcement beyond preserving safety against adversarial contexts**
  - towards preserving **hyperproperties** (data confidentiality)
  - **challenging at the lowest level: [micro-architectural] side-channels attacks**

SPECTRE

- **Dynamic component creation**
  - from code-based to data-based compartmentalization (e.g. browser tabs)
- **Dynamic privileges**
  - passing capabilities, dynamic interfaces, history-based access control, ...
- **Protecting higher-level abstractions** (than those of the C lang.)
  - **Securely Compiling Verified F* Programs With IO**
    [Cezar-Constantin Andrici et al, ICFP'23 submission]
    - using reference monitoring and higher-order contracts
    - preserving **all relational hyperproperties** against adversarial contexts
    - first step towards formally secure F*-OCaml interoperability



Formally Secure Compilation of Unsafe C Compartments

1. Goal: formalize end-to-end security guarantees

- preserve properties **against adversarial contexts**
- we overcame additional challenges to support **mutually distrustful components** and **dynamic compromise**

2. Enforcement: protect abstractions all the way down

- **SFI** or **tagged architecture** or **capability machine**

3. Proof: verify security of our compilation chain

- **scalable proof technique machine-checked in Coq**
- applying it to **CompCert extended with compartments**

Fine-grained compartmentalization

Fine-grained compartmentalization

My dream: secure compilation at scale

language

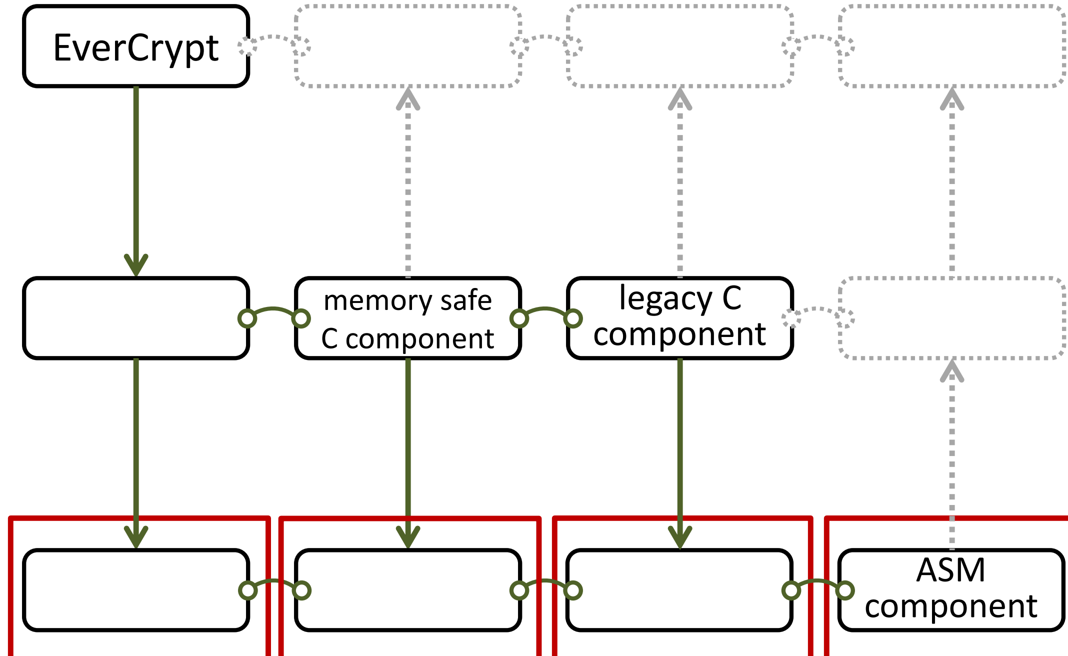
C language

+ components

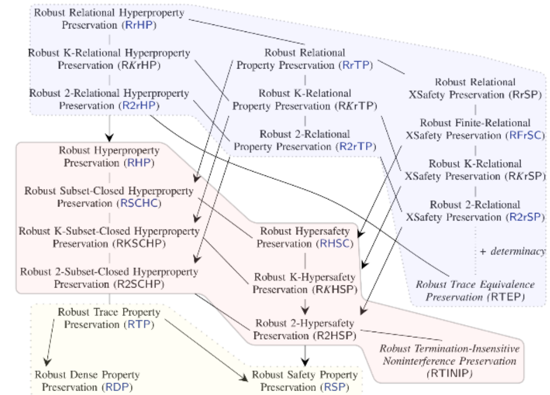
+ memory safety

ASM language

(RISC-V + micro-policies)



Going beyond Robust Preservation of Safety



Journey Beyond Full Abstraction (CSF 2019)

Carmine Abate
Inria Paris

Rob Blanco
Inria Paris

Deepak Garg
MPI-SWS

Cătălin Hrițcu
Inria Paris

Jérémy Thibault
Inria Paris

Marco Patrignani
Stanford & CISPA

Going beyond Robust Preservation of Safety [CSF'19]

No one-size-fits-all security criterion

relational hyperproperties
(trace equivalence)

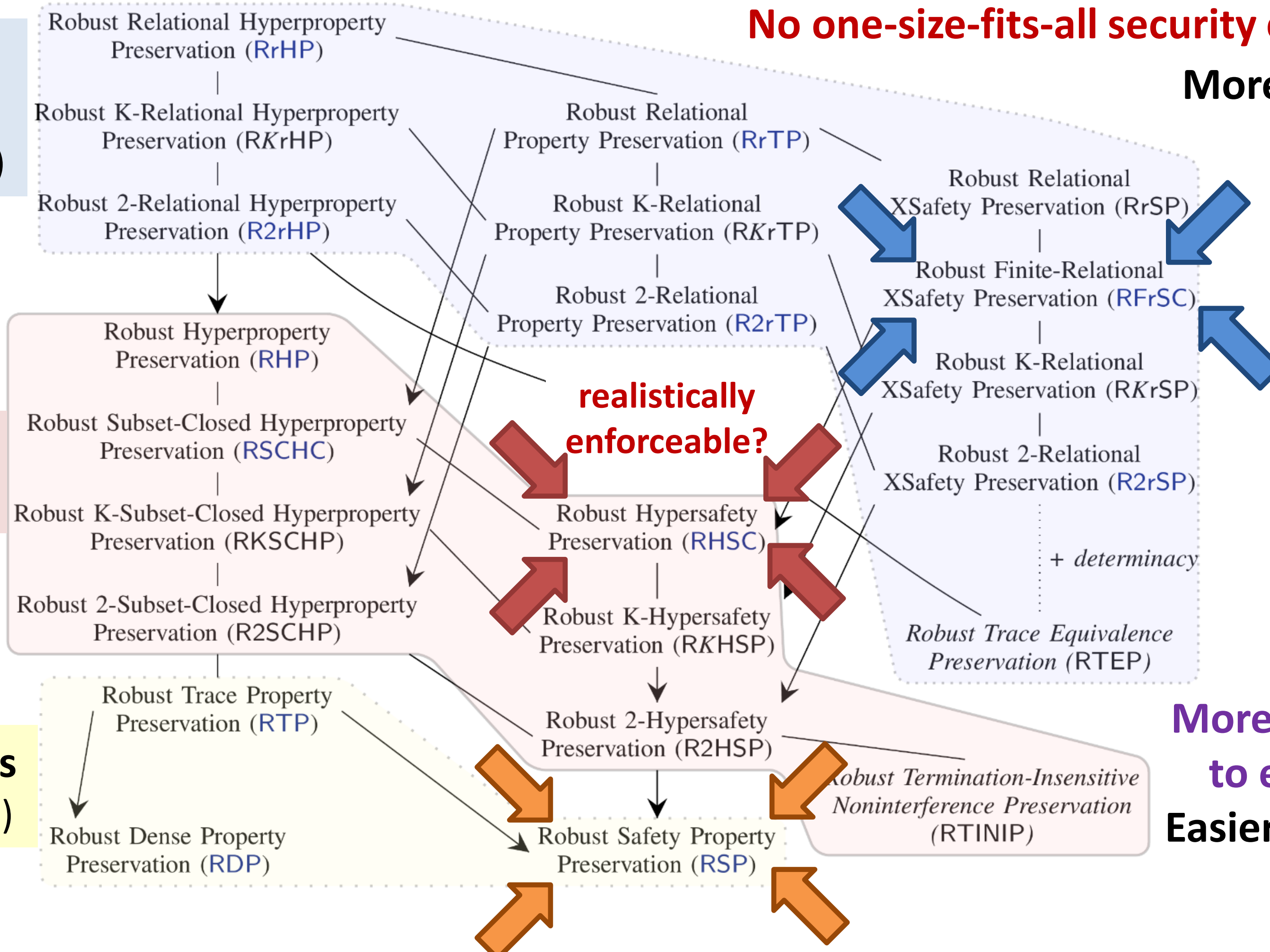
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



More secure

current proof technique

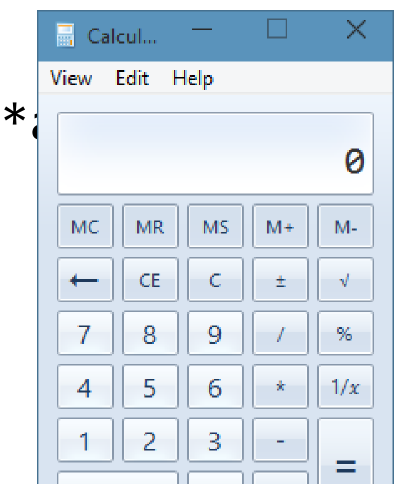
More efficient to enforce

Easier to prove

Formalizing security of mitigations is hard

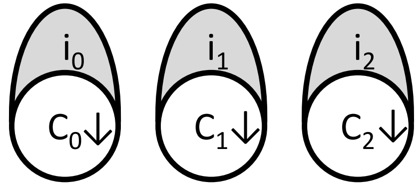
- **We want source-level security reasoning principles**
  - easier to **reason about security in the source language** if and application is compartmentalized
- **... even in the presence of undefined behavior**
  - can't be expressed at all by source language semantics!
  - **what does the following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

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



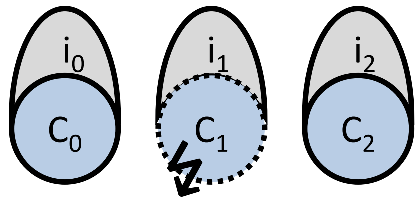
Compartmentalizing compilation should ...

- **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

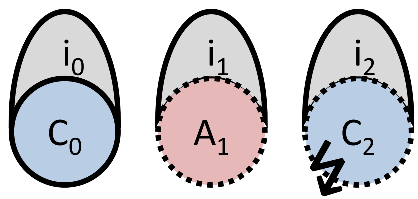
Security definition: If 

$\rightsquigarrow_{\text{machine}}$ m then

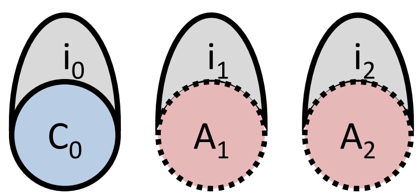
$\exists$ a sequence of component compromises explaining the finite trace m in the source language, for instance $m = m_1 \cdot m_2 \cdot m_3$ and

(1) 

$\rightsquigarrow_{\text{source}}$ $m_1 \cdot \text{Undef}(C_1)$

(2) $\exists A_1.$ 

$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot \text{Undef}(C_2)$

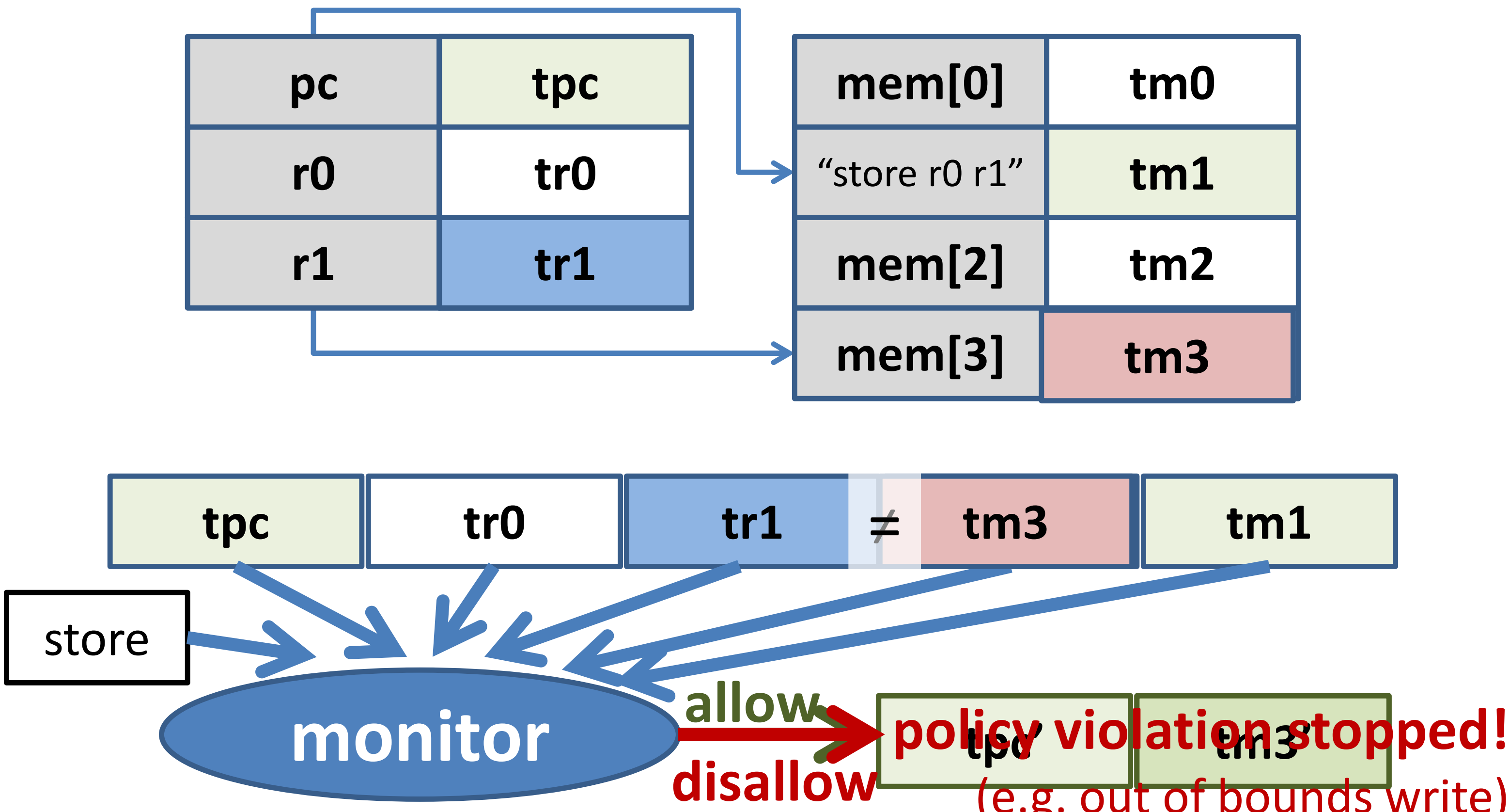
(3) $\exists A_2.$ 

$\rightsquigarrow_{\text{source}}$ $m_1 \cdot m_2 \cdot m_3$

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

Micro-Policies [POPL'14, Oakland'15, ASPLOS'15, POST'18, CCS'18]

software-defined, hardware-accelerated, tag-based monitoring



software monitor’s decision is hardware cached

new

Compartmentalization micro-policy

Compartment C_1

load r ← [rx]

put rc ← $a_{password}$

not allowed

pc@C_1

jump-and-link rx

sub r ← r-1 @NoEntry

Compartment C_2

a_1: put rc ← $a_{counter}$ @EntryPoint

a_2: load r ← [rc] @NoEntry

a_3: add r ← r+1 @NoEntry

a_4: store r → [rc] @ ...

a_5: jump ra ← pc@C_2

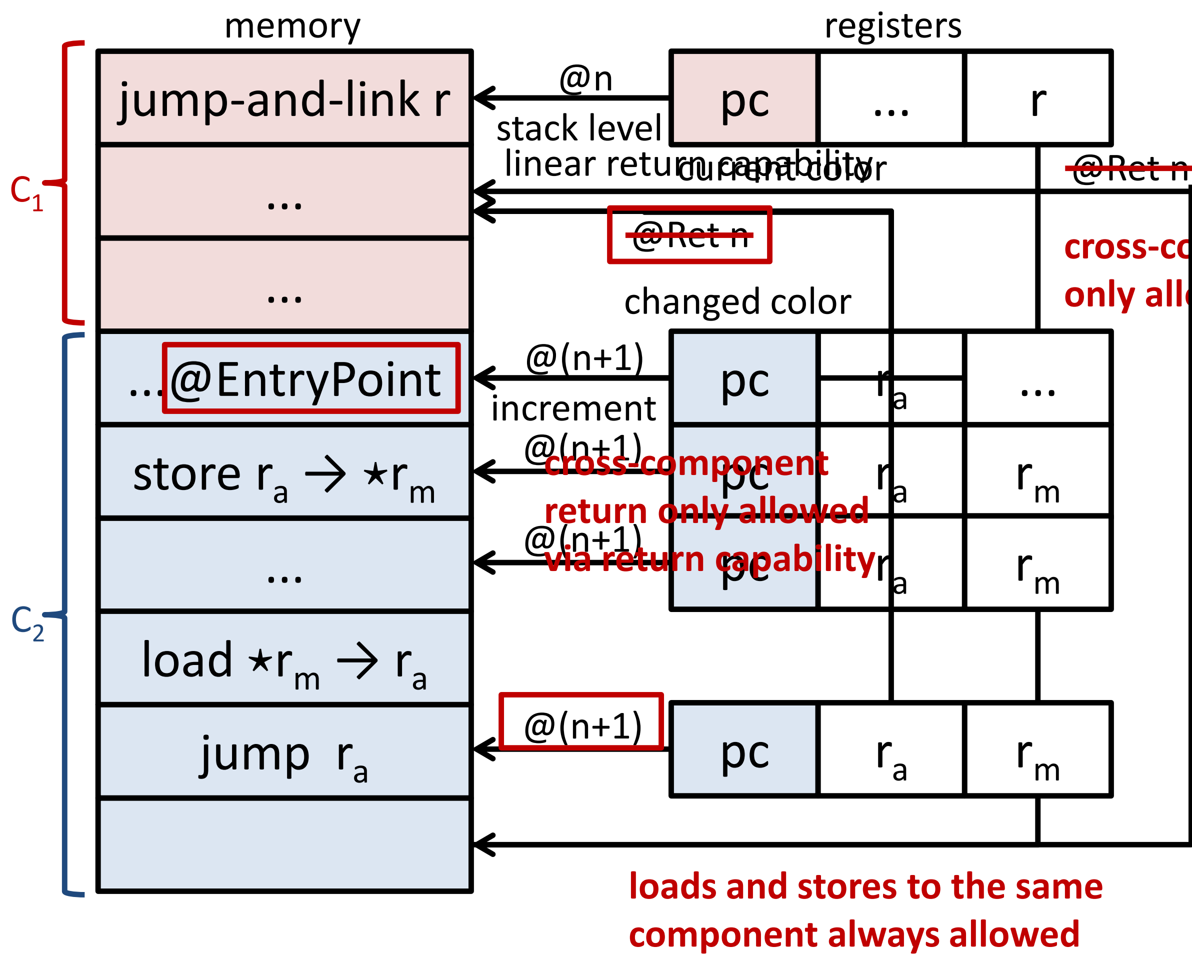
$a_{counter}$: 42

$a_{password}$: ...

Challenge: making sure returns go to the right place

Compartmentalization micro-policy

(calls and returns)

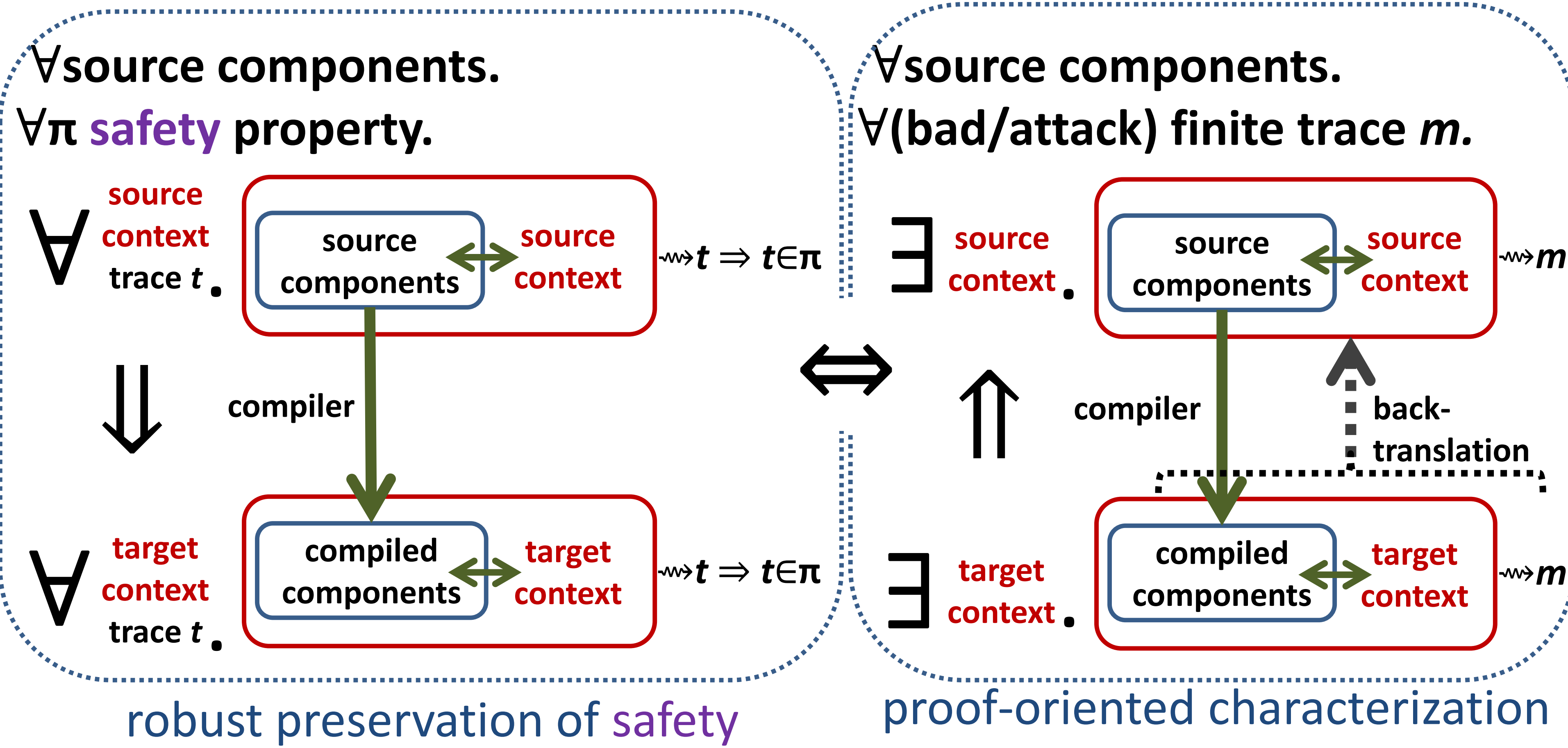


invariant:
at most one
return capability
per call stack level

Enforcement quickly gets complicated

We reduce our proof goal to a variant of:

Robust Safety Preservation



Scalable proof technique

(for our variant of Robust Safety Preservation)

1. back-translating **finite trace prefix** to **whole source program**

2+4. compiler correctness proof (à la CompCert) **used as a black-box**

3+5. also simulation proofs, but at a single level