Formally Secure
Compartmentalizing Compilation

Cătălin Hrițcu
Inria Paris
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.

Heartbleed vulnerability may have been exploited months before patch [Updated]
Fewer servers now vulnerable, but the potential damage rises.

Google finds Android zero day that can take control of Pixel and Galaxy devices
Affecting devices from Samsung, Huawei, and Google itself

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

I've just given my password to the compromised browser controlled by ad.doubleclick.net
Compartmentalization can help

compartment 1

compartment 2

not compromised

compromised
Good news: browsers now compartmentalized!
• each tab indeed started in separate compartment

Bad news, so far:
• limited compartmentalization mechanism
  – compartments coarse-grained
    • can compartmentalize tabs, but not secrets within a tab
  – compartments can't naturally interact
    • even for tabs this required big restructuring of web browsers
Fine-grained compartmentalization
Fine-grained compartmentalization
Source language compartments

• Mozilla Firefox mostly implemented in C/C++
• Programming languages like C/C++, Java, F*, ... 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)
• In the source language fine-grained compartments are easy to define and can naturally interact
Source language compartments

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 run C/C++, Java, or F*
  – Compiler translates Firefox from C/C++ to machine code instructions

• All compartmentalization abstractions lost during compilation
  – no procedures, no interfaces, no classes, no objects, no modules, ...

• Secure compilation
  – preserve 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 compartmentalizing compilation chain
Securely enforcing source abstractions is challenging!
Formally Secure Compartmentalizing Compilation

Goal

Formal Security

Proof

Enforcement
1. Security Goal

• What does it mean for a compartmentalizing compilation chain to be secure?
  – formal definition expressing end-to-end security guarantees
  – these guarantees were not understood before

• Will start with an easier definition
  – protecting a 1 trusted compartment from 1 untrusted one
  – untrusted compartment arbitrary (e.g. compromised Firefox)
  – trusted compartment has no vulnerabilities
This is not just hypothetical!

Mozilla shipping EverCrypt verified crypto library
(also used by Microsoft, Linux, ...)

Firefox

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

∀ security property Π

∀ F*context EverCrypt F* context satisfies Π

∀ machine code context compiled EverCrypt protected satisfies Π

∀ machine code context

Where "security property" can e.g., be safety or integrity or confidentiality

Π = "EverCrypt's private key is not leaked"

[CSF'19]
Extra challenges for our real security definition
[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
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?**

```c
#include <string.h>
int main (int argc, char **argv) {
    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
∃ 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

Finite trace $m$ records which component encountered undefined behavior and allows us to rewind execution.
Prototype compartmentalizing compilation chain

2. Security Enforcement

Buffers, procedures, compartments

Intermediate language with built-in compartmentalization

Programmable tagged architecture

Machine code

Hardware-accelerated enforcement

[POPL'14, Oakland'15, ASPLOS'15, POST'18, CCS'18]
Software-fault isolation

Compartment $C_1$

$\text{check } rx \in C_1\rangle$

load $r \leftarrow [rx]$

put $rc \leftarrow a_{password}$

$\text{check } rx \in C_1\leftarrow \text{not enough}$

or $rx \in C_2$'s interface

jump-and-link $rx$

sub $r \leftarrow r-1$

Compartment $C_2$

$a_1$: put $rc \leftarrow a_{counter}$

$a_2$: load $r \leftarrow [rc]$

$a_3$: add $r \leftarrow r+1$

$a_4$: store $r \rightarrow [rc]$

$a_5$: jump $ra$

$a_{counter} : 42$

$a_{password} : \ldots$

Idea: rewrite $C_1$'s (& $C_2$'s) code to insert all the required checks

Challenges: checks complicated (uncircumventable, efficient)
Micro-Policies  

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

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

<table>
<thead>
<tr>
<th>pc</th>
<th>tpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>tr0</td>
</tr>
<tr>
<td>r1</td>
<td>tr1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mem[0]</th>
<th>tm0</th>
</tr>
</thead>
<tbody>
<tr>
<td>“store r0 r1”</td>
<td>tm1</td>
</tr>
<tr>
<td>mem[2]</td>
<td>tm2</td>
</tr>
<tr>
<td>mem[3]</td>
<td>tm3</td>
</tr>
</tbody>
</table>

store

| tpc | tr0 | tr1 | tm3 | tm1 |

| monitor |

| allow | disallow |

policy violation stopped! (e.g. out of bounds write)

software monitor’s decision is hardware cached
Compartmentalization micro-policy

**Compartment C₁**

- load r ← [rx]
- put rc ← a_{password}
- jump-and-link rx
- sub r ← r-1

**Compartment C₂**

- a₁: put rc ← a_{counter} @EntryPoint
- a₂: load r ← [rc] @NoEntry
- a₃: add r ← r+1 @NoEntry
- a₄: store r → [rc] @...
- a₅: jump ra pc@C₂

**Challenge:** making sure returns go to the right place
3. Security Proof

• Proving mathematically that a compartmentalizing compilation chain achieves the security goal
  – formally verifying the security of the whole compilation chain
  – such proofs very difficult and tedious
    • wrong conjectures survived for decades; 250pg for toy compiler
  – we propose a more scalable proof technique
  – focus on machine-checked proofs in the Coq proof assistant
  – Proof-of-concept formally secure compilation chain in Coq
Verified

Compartmentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

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)

Software fault isolation

Systematically tested (with QuickChick)

Verified

Buffers, procedures, components interacting via strictly enforced interfaces

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

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)

verified

Buffers, procedures, components interacting via strictly enforced interfaces

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

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)

https://secure-compilation.github.io
Compartmentalizing compilation is an important security defense in practice

1. Goal: formalize end-to-end security guarantees
   - first definition supporting mutually distrustful components and dynamic compromise

2. Enforcement: protect abstractions all the way down
   - software fault isolation or tag-based architecture

3. Proof: verify security of entire compilation chain
   - scalable proof technique machine-checked in Coq
Making this **more practical** ... next steps:

- **Scale formally secure compilation chain to C language**
  - allow **pointer passing** (capabilities for fine-grained memory sharing)
  - eventually support enough of C to **measure and lower overhead**
  - check whether hardware support (tagged architecture) is faster

- **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 (confidentiality)** [CSF'19]

- **Secure compilation of EverCrypt, miTLS, ...**
My dream: secure compilation at scale

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]

relational hyperproperties (trace equivalence)

+ code confidentiality

hyperproperties (noninterference)

+ data confidentiality

trace properties (safety & liveness)

only integrity

No one-size-fits-all security criterion

More secure

current proof technique

realistically enforceable?

More efficient to enforce

Easier to prove

Several robust preservation properties are shown, including:

- Robust Relational Hyperproperty Preservation (RrHP)
- Robust K-Relational Hyperproperty Preservation (RKrHP)
- Robust 2-Relational Hyperproperty Preservation (R2rHP)
- Robust Hyperproperty Preservation (RHP)
- Robust Subset-Closed Hyperproperty Preservation (RSCHC)
- Robust K-Subset-Closed Hyperproperty Preservation (RKSCHP)
- Robust 2-Subset-Closed Hyperproperty Preservation (R2SCHP)
- Robust Trace Property Preservation (RTP)
- Robust Dense Property Preservation (RDP)
- Robust Safety Property Preservation (RSP)
- Robust Termination-Insensitive Noninterference Preservation (RTINIP)

These properties are compared in terms of security, efficiency, and enforceability.
Summary

Compartmentalizing compilation is an important security defense in practice

1. Goal: formalize end-to-end security guarantees
   – first definition supporting mutually distrustful components and dynamic compromise

2. Enforcement: protect abstractions all the way down
   – software fault isolation or tag-based architecture

3. Proof: verify security of entire compilation chain
   – scalable proof technique machine-checked in Coq