When Good Components Go Bad
Formally Secure Compilation
Despite Dynamic Compromise

Cătălin Hrițcu
Inria Paris

https://secure-compilation.github.io
Collaborators

Carmine Abate
Arthur Azevedo de Amorim
Rob Blanco
Ana Nora Evans
Guglielmo Fachini

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
Devastating low-level vulnerabilities
Devastating low-level vulnerabilities

• Inherently insecure C-like languages
  – type and memory unsafe:
    e.g. any buffer overflow is catastrophic
  – ~100 different undefined behavior
    reasons in the usual C compiler
Devastating low-level vulnerabilities

• Inherently insecure C-like languages
  – type and memory unsafe:
    e.g. any buffer overflow is catastrophic
  – ~100 different undefined behavior reasons in the usual C compiler
  – root cause, but challenging to fix:
    • efficiency
    • precision
    • scalability
    • backwards compatibility
    • deployment
Practical mitigation: compartmentalization
Practical mitigation: compartmentalization

• Main idea:
  – break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
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"
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"
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"
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 component receives guarantees as long as it has not encountered undefined behavior"

Goal 1: Formalize this
Goal 2: Build secure compilation chains
Goal 2: Build secure compilation chains

• Add components to C
  – interacting only via strictly enforced interfaces
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, ...
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
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) — WebAssembly (web browsers)
  - software fault isolation (SFI) — capability machines
  - hardware enclaves (SGX) — tagged architectures
Goal 1: Formalizing the security of compartmentalizing compilation
Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict *spatial* scope of undefined behavior
Restricting undefined behavior

• **Mutually-distrustful components**
  – restrict *spatial* scope of undefined behavior

• **Dynamic compromise**
  – restrict *temporal* scope of undefined behavior
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, ...
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
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
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
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 \( C_0 \downarrow \) \( i_0 \) \( C_1 \downarrow \) \( i_1 \) \( C_2 \downarrow \) \( i_2 \) \( \leadsto t \) then
∃ a **dynamic compromise scenario** explaining $t$ in source language
∃ a **dynamic compromise scenario** explaining \( t \) in source language for instance leading to the following compromise sequence:

\[
\begin{align*}
&\exists \quad \text{a dynamic compromise scenario} \quad \text{explaining} \quad t \quad \text{in source language} \\
&\text{for instance leading to the following compromise sequence:}
\end{align*}
\]
∃ a **dynamic compromise scenario** explaining \( t \) in source language for instance leading to the following compromise sequence:

\[
\begin{align*}
(0) & \quad \exists A_1. \quad C_0 \Rightarrow C_1 \Rightarrow C_2 \not\Rightarrow^* m_1; \text{Undef}(C_1) \\
(1) & \quad C_0 \Rightarrow C_1 \Rightarrow A_1 \Rightarrow C_2 \not\Rightarrow^* m_2; \text{Undef}(C_2)
\end{align*}
\]
∃ a **dynamic compromise scenario** explaining $t$ in source language for instance leading to the following compromise sequence:

1. $\exists A_1. C_0 \xrightarrow{\exists A_1} C_1 \xrightarrow{*} m_1; \text{Undef}(C_1)$
2. $\exists A_2. C_0 \xrightarrow{\exists A_1} C_2 \xrightarrow{*} m_2; \text{Undef}(C_2)$
3. $\exists A_2. C_0 \xrightarrow{\exists A_1} A_1 \xrightarrow{\exists A_2} A_2 \xrightarrow{t}$

If $i_0$ then $i_1$ then $i_2$ then $t$. 

---

$\exists \alpha$ dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:
∃ a **dynamic compromise scenario** explaining \( t \) in source language for instance leading to the following compromise sequence:

\[
\begin{array}{c}
(0) \\
\downarrow \\
C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ast m_1; \text{Undef}(C_1) \\
\end{array}
\]

\[
\begin{array}{c}
(1) \exists A_1. \\
\downarrow \\
C_0 \rightarrow A_1 \rightarrow C_2 \rightarrow \ast m_2; \text{Undef}(C_2) \\
\end{array}
\]

\[
\begin{array}{c}
(2) \exists A_2. \\
\downarrow \\
C_0 \rightarrow A_1 \rightarrow A_2 \rightarrow \ast t \\
\end{array}
\]

**Trace is very helpful**
- detect undefined behavior
- rewind execution
We build this on Robust Compilation

∀(bad attack) trace $t$

∃ low-level attacker causing $t$
We build this on Robust Compilation

∀(bad attack) trace $t$

∃ high-level attacker causing $t$

∃ low-level attacker causing $t$

high-level component

low-level component

compiler
We build this on Robust Compilation

∀(bad attack) trace $t$

∃ high-level attacker causing $t$

∃ low-level attacker causing $t$

compiler

high-level component

compiled component

∀ (bad attack) trace $t$
We build this on Robust Compilation

∀(bad attack) trace $t$

∃ high-level attacker causing $t$

∃ low-level attacker causing $t$

high-level component \[\xrightarrow{\text{compiler}}\] compiled component

∀ (bad attack) trace $t$ ⇒ robust trace property preservation (robust = in adversarial context)
We build this on Robust Compilation

∀(bad attack) trace \( t \)

\[ \exists \text{high-level attacker causing } t \]

high-level component \[ \xrightarrow{} \] high-level attacker

\[ \xleftarrow{} \]

compiler

\[ \exists \text{low-level attacker causing } t \]

compiled component \[ \xrightarrow{} \] low-level attacker

robust trace property preservation (robust = in adversarial context)

intuition:

– stronger than compiler correctness (i.e. trace property preservation)
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)
We build this on Robust Compilation

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

- high-level attacker causing \( t \)
- compiled component
- low-level attacker causing \( t \)
- high-level component
- compiler

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
We build this on Robust Compilation

∀(bad attack) trace $t$

- high-level attacker causing $t$

- low-level attacker causing $t$

---

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!
We build this on Robust Compilation

∀(bad attack) trace \( t \)

- high-level attacker causing \( t \)
- low-level attacker causing \( t \)

- compiler

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
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust  \( C_1 \quad A_2 \quad C_3 \quad A_4 \quad A_5 \)
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust

Dynamic compromise

\( \downarrow m_2; \text{Undef}(C_2) \)
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

(at least in the setting of compartmentalization for unsafe low-level languages)
Goal 2: Towards building secure compilation chains
Compartmentalized unsafe source

Compartmentalized abstract machine

Micro-policy machine
Compartmentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

Compartmentalized abstract machine

Micro-policy machine
Buffers, procedures, components interacting via **strictly enforced interfaces**

Simple RISC abstract machine with **build-in compartmentalization**
Buffers, procedures, components interacting via *strictly enforced interfaces*

Simple RISC abstract machine with *build-in compartmentalization*

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

Compartmentalized abstract machine

Buffers, procedures, components interacting via strictly enforced interfaces

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)
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)

(mostly) Verified in Coq
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

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)
Micro-Policies

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

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

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>&quot;store r0 r1&quot;</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>
Micro-Policies

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>&quot;store r0 r1&quot;</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>
```

```
| tpc | tr0 | tr1 | tm3 | tm1 |
```

store

monitor
Micro-Policies

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>&quot;store r0 r1&quot;</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>
```

```
| tpc | tr0 | tr1 | tm3 | tm1 |
```

store

allow

monitor
Micro-Policies

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>

mem[0] | tm0
```
```
| mem[2] | tm2
```
```
| mem[3] | tm3'
```
```
| "store r0 r1" | tm1
```

```
| tpc | tr0 | tr1 | tm3 | tm1
```

store

monitor

allow

```
| tpc' | tm3'
```

Micro-Policies

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

software monitor’s decision is hardware cached
Micro-Policies

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>&quot;store r0 r1&quot;</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>
```

```
<table>
<thead>
<tr>
<th>tpc</th>
<th>tr0</th>
<th>tr1</th>
<th>tm3</th>
<th>tm1</th>
</tr>
</thead>
</table>

store

monitor

disallow

policy violation stopped!
(e.g. out of bounds write)
Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure and simple**: enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V
Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V
Expressiveness

- information flow control (IFC)  [POPL’14]
Expressiveness

- information flow control (IFC) [POPL’14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

16
Expressiveness

- information flow control (IFC)  [POPL’14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

Verified (in Coq)  [Oakland’15]
Expressiveness

- information flow control (IFC)  [POPL’14]
- monitor self-protection
- protected compartments
- dynamic sealing

- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking  [ASPLOS’15]

Verified
(in Coq)
[Oakland’15]

Evaluated
(<10% runtime overhead)
Next steps towards making our secure compilation chain more practical
Next steps towards making our secure compilation chain more practical

• Scale up secure compilation to more of C
  – first step: allow pointer passing (capabilities)
Next steps towards making our secure compilation chain more practical

• **Scale up secure compilation to more of C**
  – first step: allow pointer passing (capabilities)

• **Verify compartmentalized applications**
  – put the source-level reasoning principles to work
Next steps towards making our secure compilation chain more practical

• Scale up secure compilation 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
Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation 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, ...
Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation 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, ...
- **Achieve confidentiality (hypersafety) preservation**
  - in a realistic attacker model with side-channels,
    but for this we probably need to clearly identify secrets
Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation 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, ...
- **Achieve confidentiality (hypersafety) preservation**
  - in a realistic attacker model with side-channels,
    but for this we probably need to clearly identify secrets
- **Support other enforcement mechanisms (back ends)**
Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation 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, ...
- **Achieve confidentiality (hypersafety) preservation**
  - in a realistic attacker model with side-channels,
    but for this we probably need to clearly identify secrets
- **Support other enforcement mechanisms (back ends)**
- **Measure & lower overhead**
Formally Secure Compilation
Despite Dynamic Compromise

• restrict scope of undefined behavior
  – spatially to the component that caused it
  – temporally by treating UB as an observable trace event
Formally Secure Compilation
Despite Dynamic Compromise

• restrict scope of undefined behavior
  – spatially to the component that caused it
  – temporally by treating UB as an observable trace event

• We're hiring!
  – Interns, PhD students, PostDocs, Young Researchers
Formally Secure Compilation Despite Dynamic Compromise

• restrict scope of undefined behavior
  – spatially to the component that caused it
  – temporally by treating UB as an observable trace event

• We're hiring!
  – Interns, PhD students, PostDocs, Young Researchers

• Another interesting event
  – Workshop on Principles of Secure Compilation (PriSC) @ POPL