Formally Secure Compilation

Cătălin Hrițcu
Inria Paris

https://secure-compilation.github.io
Devastating low-level attacks
Devastating low-level attacks

inherently insecure languages like C/C++

- **e.g. memory unsafe**: any buffer overflow is catastrophic allowing remote attackers to gain complete control
Devastating low-level attacks

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
Devastating low-level attacks

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
Devastating low-level attacks

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
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 1 of 2

Secure Interoperability with Lower-Level Code

Carmine Abate  
Inria Paris  
U. Trento

Deepak Garg  
MPI-SWS

Cătălin Hrițcu  
Inria Paris

Jérémy Thibault  
Inria Paris  
ENS Rennes

Marco Patrignani  
CISPA  
Stanford
Good programming languages provide helpful abstractions for writing more secure code
Good programming languages provide helpful abstractions for writing more secure code

- e.g. **HACL*** and **miTLS** written in **Low*** which provides:
Good programming languages provide helpful abstractions for writing more secure code

- e.g. HACL* and miTLS written in Low* which provides:
  - low-level abstractions associated with safe C programs
Good programming languages provide helpful abstractions for writing more secure code

- e.g. HACL* and miTLS written in Low* which provides:
  - low-level abstractions associated with safe C programs
  - higher-level abstractions associated with ML-like languages
Good programming languages provide helpful abstractions for writing more secure code

- e.g. HACL* and miTLS written in Low* which provides:
  - low-level abstractions associated with safe C programs
  - higher-level abstractions associated with ML-like languages
  - most features of verification systems like Coq and Dafny
Good programming languages provide helpful abstractions for writing more secure code

- e.g. HACL* and miTLS written in Low* which provides:
  - low-level abstractions associated with safe C programs
  - higher-level abstractions associated with ML-like languages
  - most features of verification systems like Coq and Dafny
  - patterns specific to cryptographic code
Abstractions not enforced when linking with adversarial low-level code

~10,000 LOC in Low*

HACL* library
Abstractions not enforced when linking with adversarial low-level code

- HACL* library
  - ~10,000 LOC in Low*
- Firefox web browser
  - 16,000,000+ LOC in C/C++
Abstractions not enforced when linking with adversarial low-level code

Insecure interoperability: compromised (or malicious) application linking in miTLS can easily read and write miTLS’s data and code, jump to arbitrary instructions, smash the stack, …
Secure compilation

• Protect source-level abstractions even against linked adversarial low-level code
Secure compilation

• Protect source-level abstractions even against linked adversarial low-level code
• Enable source-level security reasoning
Secure compilation

• Protect source-level abstractions even against linked adversarial low-level code

• Enable source-level security reasoning
  – even an adversarial target-level context cannot break the security properties of the compiled program any more than some source-level context could
Secure compilation

• **Protect source-level abstractions**
  even against linked adversarial low-level code

• **Enable source-level security reasoning**
  – even **an adversarial target-level context** cannot break the security properties of the compiled program any more than some source-level context could
  – no "low-level" attacks
Secure compilation

• Protect source-level abstractions 
even against linked adversarial low-level code

• Enable source-level security reasoning
  – even an adversarial target-level context cannot 
    break the security properties of the compiled program 
    any more than some source-level context could
  – no "low-level" attacks
  – no need to worry about the compilation chain 
    (compiler, linker, loader, runtime, system, hardware)
Source-level security reasoning
Source-level security reasoning

∀ source context → secure

component

∀ source context → secure

component
Source-level security reasoning

∀ source context

source component
	source context

∀ target context

compiled component
	target context

compiler

secure

secure
Source-level security reasoning

∀ source context

source component

∀ target context

compiled component

protected

no extra power

∀ source context

source context

∀ target context

target context

secure

secure
Source-level security reasoning

∀ source context
source component
secure

∀ target context
compiled component
protected
no extra power

But what does "secure" mean?
What security properties should we preserve?
What security properties should we preserve?

• We explore a large space of security properties
What security properties should we preserve?

• We explore a large space of security properties
• Study preserving various classes of ...
  – trace properties (safety, liveness)
  – hyperproperties (e.g. noninterference)
  – relational hyperproperties (e.g. trace equivalence)

... against adversarial target-level contexts
What security properties should we preserve?

• We explore a large space of security properties

• Study preserving various classes of ...
  – trace properties (safety, liveness)
  – hyperproperties (e.g. noninterference)
  – relational hyperproperties (e.g. trace equivalence)

... against adversarial target-level contexts

• No “one-size-fits-all solution”
  – e.g. full abstraction does not imply the other criteria we study
  – stronger criteria are harder to achieve and prove, both challenging
More secure

More efficient

Easier to prove
Robust Trace Property Preservation
∀ source component.
∀ \pi trace property.
∀ source component. source context trace t \Rightarrow t \in \pi
∀ source component.
∀ \pi trace property.

∀ source context trace t.
∀ target context trace t.
∀ source component \Rightarrow t \Rightarrow t \in \pi
∀ target component \Rightarrow t \Rightarrow t \in \pi
Robust Trace Property Preservation

property-based characterization

∀source component.

∀π trace property.

∀

source context 

trace t

source component

source context

⇒

⇝

t ⇝ t ∈ π

compiler

∀

target context 

trace t

compiled component

target context

⇒

⇝

t ⇝ t ∈ π

preservation of robust satisfaction
Robust Trace Property Preservation

property-based characterization

∀source component.
∀π trace property.

∀ source context trace t.

source component
source context

compiler

⇒
⇝
t ⇒ t ∈ π

∀ target context trace t.

compiled component
target context

property-free characterization

⇔

preservation of robust satisfaction

how one can prove it
Robust Trace Property Preservation

**property-based characterization**

\[ \forall \text{source component}. \]
\[ \forall \pi \text{ trace property}. \]

\[ \forall \text{source context} \text{ trace } t. \]

\[ \text{compiler} \]

\[ \forall \text{target context} \text{ trace } t. \]

preservation of **robust** satisfaction

**property-free characterization**

\[ \forall \text{source component}. \]

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

\[ \text{how one can prove it} \]

\[ \Rightarrow \Rightarrow t \implies t \in \pi \]

\[ \text{compiler} \]

\[ \Rightarrow \Rightarrow t \implies t \in \pi \]

\[ \text{target context} \text{ compiled component} \text{ target context} \]
Robust Trace Property Preservation

**property-based characterization**

\[ \forall \text{source component.} \]
\[ \forall \pi \text{ trace property.} \]
\[ \forall \text{ source context trace } t. \]
\[ \text{source component} \rightarrow \text{source context} \]
\[ \downarrow \]
\[ \text{compiler} \]
\[ \forall \text{ target context trace } t. \]
\[ \text{target component} \rightarrow \text{target context} \]

**property-free characterization**

\[ \forall \text{source component.} \]
\[ \forall (\text{bad attack}) \text{ trace } t. \]
\[ \forall \text{ target context.} \]
\[ \text{target component} \rightarrow \text{target context} \]

preservation of robust satisfaction

how one can prove it
Robust Trace Property Preservation

property-based characterization

\[ \forall \text{source component.} \]
\[ \forall \pi \text{ trace property.} \]

property-free characterization

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

preservation of robust satisfaction

how one can prove it
back-translating
prog & context & trace
∀P∀C_T ∀t∈C_S...
∀P∀C \text{T} \forall m \leq t \exists C S...
back-translating finite sets of finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T \forall m_1..m_k \exists C_S...$

back-translating prog & context & trace
$\forall P \forall C_T \forall t \exists C_S...$

back-translating finite trace prefixes
$\forall P \forall C_T \forall m \exists t \exists C_S...$
back-translating contexts
\( \forall C_T \exists C_S \forall \forall t \ldots \)

back-translating prog & context
\( \forall \forall \forall C_T \exists C_S \forall t \ldots \)

Trace Equivalence Preservation

Robust Relational Hyperproperty Preservation
Robust k-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation

Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust k-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Trace Property Preservation

Robust Hypersafety Preservation
Robust k-Hypersafety Preservation
Robust 2-Hypersafety Preservation

Robust Safety Preservation

Robust Relational Hypersafety Preservation
Robust Relational Safety Preservation

Robust Finite-Relational Safety Preservation
Robust k-Relational Safety Preservation
Robust 2-Relational Safety Preservation

Observational Equivalence Preservation

back-translating finite sets of finite trace prefixes
\( \forall k \forall P_1..P_k \forall C_T \forall m_1..m_k \exists C_S \ldots \)

back-translating finite trace prefixes
\( \forall P \forall C_T \forall t \exists C_S \ldots \)

back-translating prog & context & trace
\( \forall P \forall C_T \forall t \exists C_S \ldots \)
Results

• Mapped the space of secure compilation criteria based on robust "property" preservation
Results

• Mapped the space of secure compilation criteria based on robust "property" preservation
  – Property-free characterizations and implications in Coq
Results

• Mapped the space of secure compilation criteria based on robust "property" preservation
  – Property-free characterizations and implications in Coq
  – Separation results (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)
Results

• Mapped the space of secure compilation criteria based on robust "property" preservation
  – Property-free characterizations and implications in Coq
  – Separation results (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)
  – **Surprising collapse** between preserving all hyperproperties and preserving just hyperliveness
Results

• Mapped the space of secure compilation criteria based on robust "property" preservation
  – Property-free characterizations and implications in Coq
  – Separation results (e.g. robust safety/liveness preservation strictly weaker than robust trace property preservation)
  – Surprising collapse between preserving all hyperproperties and preserving just hyperliveness

• Showed that even strongest criterion is achievable
  – for simple translation from a statically to a dynamically typed language with first-order functions and I/O
Some open problems

• Practically achieving secure interoperability with lower-level code
Some open problems

• Practically achieving secure interoperability with lower-level code
  – More realistic languages and secure compilation chains
Some open problems

• Practically achieving secure interoperability with lower-level code
  – More realistic languages and secure compilation chains
  – Achieve noninterference preservation
    in realistic attacker model with side-channels
Some open problems

• Practically achieving secure interoperability with lower-level code
  – More realistic languages and secure compilation chains
  – Achieve noninterference preservation in realistic attacker model with side-channels
  – Efficient enforcement mechanisms
Some open problems

• Practically achieving secure interoperability with lower-level code
  – More realistic languages and secure compilation chains
  – Achieve noninterference preservation in realistic attacker model with side-channels
  – Efficient enforcement mechanisms

• Scalable proof techniques for other criteria
  – (hyper)liveness preservation (possible?)
Some open problems

- Practically achieving secure interoperability with lower-level code
  - More realistic languages and secure compilation chains
  - Achieve noninterference preservation in realistic attacker model with side-channels
  - Efficient enforcement mechanisms

- Scalable proof techniques for other criteria
  - (hyper)liveness preservation (possible?)

- Nontrivial relation between source and target traces
Where is full abstraction?
Where is full abstraction?

with internal nondeterminism

Robust Relational Hyperproperty Preservation
  Robust k-Relational Hyperproperty Preservation
  Robust 2-Relational Hyperproperty Preservation

Robust Hyperproperty Preservation
  Robust Subset-Closed Hyperproperty Preservation
  Robust K-Subset-Closed Hyperproperty Preservation
  Robust 2-Subset-Closed Hyperproperty Preservation
  Robust Trace Property Preservation

Robust Relational Property Preservation
  Robust k-Relational Property Preservation
  Robust 2-Relational Property Preservation

Robust Relational Hypersafety Preservation
  Robust Finite-Relational Safety Preservation
  Robust K-Relational Safety Preservation
  Robust 2-Relational Safety Preservation

Robust Hypersafety Preservation
  Robust 2-Hypersafety Preservation

Robust Safety Preservation

Observational Equivalence Preservation

+ determinacy
Where is full abstraction?

with internal nondeterminism

without internal nondeterminism

Trace Equivalence Preservation

Robust Relational Hyperproperty Preservation
Robust k-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation

Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust K-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Trace Property Preservation

Robust Relational Property Preservation
Robust K-Relational Property Preservation
Robust 2-Relational Property Preservation

Robust Relational Hypersafety Preservation
Robust Finite-Relational Safety Preservation
Robust K-Relational Safety Preservation
Robust 2-Relational Safety Preservation

Observational Equivalence Preservation

+ determinacy
Where is full abstraction?

with internal nondeterminism

without internal nondeterminism

+what extra assumptions? compiler correctness enough??

???

Trace Equivalence Preservation

Robust Relational Hyperproperty Preservation

Robust k-Relational Hyperproperty Preservation

Robust 2-Relational Hyperproperty Preservation

Robust Hyperproperty Preservation

Robust Subset-Closed Hyperproperty Preservation

Robust K-Subset-Closed Hyperproperty Preservation

Robust 2-Subset-Closed Hyperproperty Preservation

Robust Trace Property Preservation

Robust Safety Preservation

Robust Hypersafety Preservation

Robust Finite-Relational Safety Preservation

Robust K-Relational Safety Preservation

Robust 2-Relational Safety Preservation

Observational Equivalence Preservation

+ determinacy
Collaborators for Part 2

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
Undefined behavior

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

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

Buffer overflow

```
$ gcc target.c
$ ./a.out haha
```
Undefined behavior

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

Buffer overflow

```
$ gcc target.c
$ ./a.out haha
$ ./a.out hahahahahahahahahahahahaha
zsh: segmentation fault (core dumped)
```
### Undefined behavior

```c
#include <string.h>

int main (int argc, char **argv) {
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```

**Buffer overflow**

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

```c
#include <string.h>

int main (int argc, char **argv) {
    char c[12];
    strcpy(c, argv[1]);
    return 0;
}
```

```
$ gcc target.c
$ ./a.out hahahahahahahahahaha
zsh: segmentation fault (core dumped)
$ ./exploit.sh | a.out
```
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 components 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 \textit{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 \( C_0 \downarrow, C_1 \downarrow, C_2 \downarrow \rightarrow t \) 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:

$$
\begin{align*}
\text{If } & C_0 \downarrow C_1 \downarrow C_2 \leadsto t \text{ then} \\
\exists \text{ a dynamic compromise scenario explaining } t \text{ in source language for instance leading to the following compromise sequence:}
\end{align*}
$$

(0) $C_0 \leadsto C_1 \leadsto * \, m_1; \text{Undef}(C_1)$
∃ a **dynamic compromise scenario** explaining $t$ in source language for instance leading to the following compromise sequence:

(0) \[ i_0 \downarrow C_0 \xrightarrow{} C_1 \xrightarrow{} C_2 \xrightarrow{} m_1;\text{Undef}(C_1) \]

(1) \[ i_0 \downarrow C_0 \xrightarrow{} A_1 \xrightarrow{} i_1 \xrightarrow{} C_2 \xrightarrow{} m_2;\text{Undef}(C_2) \]
∃ a **dynamic compromise scenario** explaining $t$ in source language for instance leading to the following compromise sequence:

(0) $C_0 \xrightarrow{m_1; \text{Undef}(C_1)} C_1$  

(1) $\exists A_1. C_0 \xrightarrow{m_2; \text{Undef}(C_2)} A_1$  

(2) $\exists A_2. C_0 \xrightarrow{} A_1 \xrightarrow{} t$
∃ a dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:

1. $\exists A_1$.
   - $C_0 \leadsto \cdots \leadsto m_1;\text{Undef}(C_1)$
   - Trace is very helpful
     - detect undefined behavior
     - rewind execution

2. $\exists A_2$.
   - $C_0 \leadsto \cdots \leadsto t$
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust

C\textsubscript{1} \hspace{1cm} A\textsubscript{2} \hspace{1cm} C\textsubscript{3} \hspace{1cm} A\textsubscript{4} \hspace{1cm} A\textsubscript{5}
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust

Dynamic compromise

\[ m_2; \text{Undef}(C_2) \]
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
Goal 2: Towards building secure compilation chains
Compartmentalized unsafe source

Compartmentalized abstract machine

Buffers, procedures, components interacting via strictly enforced interfaces

Micro-policy machine
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
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)
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
Compartimentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

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)

Verified in Coq

(mostly)
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)

_verified in Coq (mostly)

Systematically tested (with QuickChick)
Making this more practical ... next steps:
Making this more practical ... next steps:

- **Scale up to more of C**
  - first step: allow pointer passing (capabilities)
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
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
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, ...
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)**
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

• Secure interoperability with lower-level code
  – exploring a continuum, security vs efficiency tradeoff

• 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
Wrapping up

• Secure interoperability with lower-level code
  – exploring a continuum, security vs efficiency tradeoff

• 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!
  – PostDocs, Young Researchers, Interns, PhD students
More goals of secure compilation

• Enabling source-level security reasoning
• Making the source language safer
  – memory and type safety, less/no undefined behavior
• Making it easier to express security intent
  – marking secrets, specifying security properties
• Making exploits more difficult
  – CFI, CPI, stack protection, randomization, diversity
Micro-Policies

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

```
mem[0]  "store r0 r1"
mem[2]
mem[3]
```
### 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>“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>
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>“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

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>“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>

Monitor:
- Store: tpc
- Allow: tpc' tm3'

Diagram:
- tpc connects to tr0 and tr1
- tm0 connects to “store r0 r1” and tm1
- tm2 connects to mem[2]
- tm3 connects to mem[3]
Micro-Policies

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

\[
\begin{array}{|c|c|}
\hline
\text{pc} & \text{tpc'} \\
\hline
\text{r0} & \text{tr0} \\
\hline
\text{r1} & \text{tr1} \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
\text{mem}[0] & \text{tm0} \\
\text{“store r0 r1”} & \text{tm1} \\
\text{mem}[2] & \text{tm2} \\
\text{mem}[3] & \text{tm3'} \\
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{tpc} \\
\text{tr0} \\
\text{tr1} = \text{tm3} \\
\text{tm1}
\end{array}
\]

\[
\begin{array}{c}
\text{store} \\
\text{monitor} \\
\text{allow}
\end{array}
\begin{array}{c}
\text{tpc'} \\
\text{tm3'}
\end{array}
\]
Micro-Policies

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

store

monitor

allow

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>“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

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
- ...

Expressiveness
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 (<10% runtime overhead) [ASPLOS’15]