Formally Secure Compilation of Unsafe Low-level Components

Cătălin Hriţcu

Inria Paris, Prosecco team

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

• Inherently insecure C/C++-like languages
  – memory (and type) unsafe:
    any buffer overflow is catastrophic
Devastating low-level vulnerabilities

- Inherently insecure C/C++-like languages
  - memory (and type) unsafe: any buffer overflow is catastrophic
  - 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** running with **least privilege** & interacting via strictly enforced interfaces
Practical mitigation: compartmentalization

• Main idea:
  – break up security-critical C applications into **mutually distrustful components** running with **least privilege** & interacting via strictly enforced interfaces

• **Strong security guarantees & interesting attacker model**
  – "a vulnerability in one component should not immediately destroy the security of the whole application"
Practical mitigation: compartmentalization

• Main idea:
  – break up security-critical C applications into **mutually distrustful components** running with **least privilege** & interacting via strictly enforced interfaces

• **Strong security guarantees & interesting attacker model**
  – "a vulnerability in one component should not immediately destroy the security of the whole application"
  – "components can be compromised by buffer overflows"
Practical mitigation: compartmentalization

• **Main idea:**
  – break up security-critical C applications into **mutually distrustful components** running with **least privilege** & interacting via strictly enforced interfaces

• **Strong security guarantees & interesting attacker model**
  – "a vulnerability in one component should not immediately destroy the security of the whole application"
  – "components can be compromised by buffer overflows"
Practical mitigation: compartmentalization

• Main idea:
  – break up security-critical C applications into *mutually distrustful components* running with *least privilege* & interacting via strictly enforced interfaces

• **Strong security guarantees & interesting attacker model**
  – "a vulnerability in one component should not immediately destroy the security of the whole application"
  – "components can be compromised by buffer overflows"
  – "each component should be protected from all the others"
Practical mitigation: compartmentalization

• Main idea:
  – break up security-critical C applications into **mutually distrustful components** running with **least privilege** & interacting via strictly enforced interfaces

• **Strong security guarantees & interesting attacker model**
  – "a vulnerability in one component should not immediately destroy the security of the whole application"
  – "components can be compromised by buffer overflows"
  – "each component should be protected from all the others"

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)
  – software fault isolation (SFI)
  – hardware enclaves (SGX)
  – WebAssembly (web browsers)
  – capability machines
  – tagged architectures

• **Practical need for this** (e.g. crypto library/protocol)
Goal 1: Formalizing security of compartmentalizing compilation
Goal 1: Formalizing security of compartmentalizing compilation

- Program trace
- Compiler correctness (e.g. CompCert)
- Program trace

Diagram:

- C
- ASM
- Compiler

6
Goal 1: Formalizing security of compartmentalizing compilation

program trace

compiler
correctness
(e.g. CompCert)

program trace

C component

compiler

ASM component
Goal 1: Formalizing security of compartmentalizing compilation

- C component
- ASM component
- Compromised components
- e.g. compromised C code

Program trace

Compiler correctness (e.g. CompCert) not enough
Goal 1: Formalizing security of compartmentalizing compilation

- Program trace
- Compiler correctness (e.g., CompCert)
- Secure compilation
- Well-defined C components
- Compromised components
- E.g., compromised C code

C component

ASM component

Not enough
Goal 1: Formalizing security of compartmentalizing compilation

- C component
  - well-defined C components
  - not enough
  - compiler correctness (e.g. CompCert)
  - program trace

- ASM component
  - compromised components
  - protected
  - no extra power
  - e.g. compromised C code

- secure compilation
  - compiler
  - program trace

- secure compilation
  - not enough
  - compiler correctness (e.g. CompCert)
  - program trace
Goal 1: Formalizing security of compartmentalizing compilation

**Benefit:** sound security reasoning in the source language
Fully abstract compilation
preservation of observational equivalence
Fully abstract compilation
preservation of observational equivalence

∃ low-level attacker

1\text{st} compiled component \iff \text{low-level attacker}

compiler

1\text{st high-level component}

\sim

2\text{nd compiled component} \iff \text{low-level attacker}

compiler

2\text{nd high-level component}
Fully abstract compilation
preservation of observational equivalence
Fully abstract compilation
preservation of observational equivalence
Undefined behavior

#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 -fno-stack-protector
$ ./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 -fno-stack-protector
$ ./a.out haha
$ ./a.out hahahahahahahahahahahahahaha
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 -fno-stack-protector
$ ./a.out haha
$ ./a.out hahahahahahahahahahahahahaha
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 -fno-stack-protector
$ ./a.out haha
$ ./a.out hahahahahahahahaha
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Source reasoning vs undefined behavior

• Source reasoning
  = We want to reason formally about security with respect to source language semantics
Source reasoning vs undefined behavior

• **Source reasoning**
  
  = We want to reason formally about security with respect to source language semantics

• **Undefined behavior**
  
  = can't be expressed at all by source language semantics!
Source reasoning vs undefined behavior

• Source reasoning
  = We want to reason formally about security with respect to source language semantics

• Undefined behavior
  = can't be expressed at all by source language semantics!

• Problem: observational equivalence doesn't work with undefined behavior!?
  – int buf[5]; buf[42] ~? int buf[5]; buf[43]
Source reasoning vs undefined behavior

- **Source reasoning**
  - We want to reason formally about security with respect to source language semantics

- **Undefined behavior**
  - Can't be expressed at all by source language semantics!

- **Problem: observational equivalence doesn't work with undefined behavior!?**
  - int buf[5]; buf[42] ~? int buf[5]; buf[43]

- **Can we somehow avoid undefined behavior?**
Full abstraction for mutually distrustful components

∀ compromise scenarios.

if $C_1$, $C_3$, $D_1$, $D_3$ **fully defined** and

$\exists$ low-level attack from compromised $C_2\downarrow$, $C_4\downarrow$, $C_5\downarrow$

[Beyond Good and Evil - Juglaret, Hriţcu, Azevedo de Amorim, Eng, Pierce, CSF’16]
Full abstraction for mutually distrustful components

∀ compromise scenarios.

∃ high-level attack from some fully defined $A_2, A_4, A_5$

if $C_1, C_3, D_1, D_3$ fully defined and

∃ low-level attack from compromised $C_2\downarrow, C_4\downarrow, C_5\downarrow$

[Beyond Good and Evil - Juglaret, Hrițcu, Azevedo de Amorim, Eng, Pierce, CSF’16]
Full abstraction for mutually distrustful components

∀ compromise scenarios.

∃ high-level attack from some **fully defined** $A_2$, $A_4$, $A_5$

if $C_1$, $C_3$, $D_1$, $D_3$ **fully defined** and

∃ low-level attack from compromised $C_2\downarrow$, $C_4\downarrow$, $C_5\downarrow$

Limitation: static compromise model: $C_1$, $C_3$, $D_1$, $D_3$ get guarantees only if perfectly safe
(i.e. fully defined = do not exhibit undefined behavior in any context)

[Beyond Good and Evil - Juglaret, Hrițcu, Azevedo de Amorim, Eng, Pierce, CSF’16]
Full abstraction for mutually distrustful components

∀ compromise scenarios.

∃ high-level attack from some **fully defined** A₂, A₄, A₅

if C₁, C₃, D₁, D₃ **fully defined** and

∃ low-level attack from compromised C₂↓, C₄↓, C₅↓

Limitation: static compromise model: C₁, C₃, D₁, D₃ get guarantees only if perfectly safe (i.e. fully defined = do not exhibit undefined behavior in any context)

**This is the most we were able to achieve on top of full abstraction!**

[Beyond Good and Evil - Juglaret, Hrițcu, Azevedo de Amorim, Eng, Pierce, CSF’16]
Static compromise not good enough

component C₀ {
    export valid;
    valid(data) { ... }
}

component C₁ {
    import E.read, C₂.init, C₂.process;
    main() {
        C₂.init();
        x := E.read();
        y := C₁.parse(x);  // (V₁) can UNDEF if x is malformed
        C₂.process(x,y);
    }
    parse(x) { ... }
}

component C₂ {
    import E.write, C₀.valid;
    export init, process;
    init() { ... }
    process(x,y) { ... }  // (V₂) can UNDEF if not initialized
Static compromise not good enough

neither $C_1$ not $C_2$ are fully defined

cOMPONENT C_0 {
  export valid;
  valid(data) { ... }
}

COMPONENT C_1 {
  import E.read, C_2.init, C_2.process;
  main() {
    C_2.init();
    x := E.read();
    y := C_1.parse(x);  // (V_1) can UNDEF if x is malformed
    C_2.process(x,y);
  }
  parse(x) { ... }
}

COMPONENT C_2 {
  import E.write, C_0.valid;
  export init, process;
  init() { ... }
  process(x,y) { ... }  // (V_2) can UNDEF if not initialized
}
Static compromise not good enough

neither $C_1$ not $C_2$ are fully defined

yet $C_1$ is protected until calling $C_1$.parse

```javascript
component C0 {
    export valid;
    valid(data) { ... }
}

component C1 {
    import E.read, C2.init, C2.process;
    main() {
        C2.init();
        x := E.read();
        y := C1.parse(x);  // (V1) can UNDEF if x is malformed
        C2.process(x, y);
    }
    parse(x) { ... }
}

component C2 {
    import E.write, C0.valid;
    export init, process;
    init() { ... }
    process(x, y) { ... }  // (V2) can UNDEF if not initialized
}
```
Static compromise not good enough

neither C₁ not C₂ are fully defined
yet C₁ is protected until calling C₁.parse
and C₂ can't actually be compromised

```plaintext
component C₀ {
    export valid;
    valid(data) { ... }
}

component C₁ {
    import E.read, C₂.init, C₂.process;
    main() {
        C₂.init();
        x := E.read();
        y := C₁.parse(x);  // (V₁) can UNDEF if x is malformed
        C₂.process(x,y);
    }
    parse(x) { ... }
}

component C₂ {
    import E.write, C₀.valid;
    export init, process;
    init() { ... }
    process(x,y) { ... }  // (V₂) can UNDEF if not initialized
}
```
We build instead on Robust Compilation

∀(bad attack) trace \( t \)

∃ low-level attacker causing \( t \)
We build instead on Robust Compilation

∀(bad attack) trace \( t \)

∃ high-level attacker causing \( t \)

∃ low-level attacker causing \( t \)

\( \exists \) high-level attacker

\( \exists \) low-level attacker

\( \forall \) (bad attack) trace \( t \)

high-level component

high-level attacker

compiler

compiled component

low-level attacker
We build instead on Robust Compilation

∀(bad attack) trace \( t \)

∃ high-level attacker causing \( t \)

⇒ high-level component

⇒ high-level attacker

↑ compiler

∃ low-level attacker causing \( t \)

⇒ compiled component

⇒ low-level attacker
We build instead on Robust Compilation

∀(bad attack) trace \( t \)

∃ high-level attacker causing \( t \)

∃ low-level attacker causing \( t \)

high-level component

compiler

compiled component

high-level attacker

low-level attacker

robust trace property preservation
(robust = in adversarial context)
We build instead 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)
- (when restricted to safety) seems weaker than full abstraction + compiler correctness
We build instead 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)
- (when restricted to safety) seems weaker than full abstraction + compiler correctness

less extensional than full abstraction
We build instead on Robust Compilation

∀(bad attack) trace \( t \)

Advantages: easier to realistically achieve and prove at scale

useful: preservation of invariants and other integrity properties

more intuitive to security people (generalizes to hyperproperties!)
We build instead 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)
- (when restricted to safety) seems weaker than full abstraction + compiler correctness
- less extensional than full abstraction

Advantages: easier to realistically achieve and prove at scale
useful: preservation of invariants and other integrity properties
more intuitive to security people (generalizes to hyperproperties!)
extends to unsafe languages, supporting dynamic compromise
Dynamic compromise

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
Dynamic compromise

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
∃ a dynamic compromise scenario explaining $t$ in source language

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
∃ a dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:

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

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
∃ a dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:

(0) $C_0 \Downarrow$; $C_1 \Downarrow$; $C_2 \Downarrow$; $m_1$; Undef($C_1$)
∃ a dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:

(0) $\exists A_1.

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
Dynamic compromise

∃ a dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:

1. $\exists A_1$. $C_0 \Downarrow m_1; \text{Undef}(C_1)$

2. $\exists A_2$. $C_0 \Downarrow m_2; \text{Undef}(C_2)$

3. $\Downarrow t$
∃ a dynamic compromise scenario explaining $t$ in source language for instance leading to the following compromise sequence:

(0) $\exists A_1$. $\exists A_2$.

Trace is very helpful - detect undefined behavior - rewind execution
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 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 model
  - GCC and LLVM **currently violate** this model
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust

C₁  A₂  C₃  A₄  A₅
Now we know what these words mean!

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

Mutual distrust

Dynamic compromise

$C_0 \xrightarrow{\downarrow m_2} \text{Undef}(C_2)$
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust

Dynamic compromise  ↓ $m_2$; 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**
Towards Secure Compilation Chain

Compartmentalized unsafe source

Compartmentalized abstract machine

Micro-policy machine
Towards Secure Compilation Chain

Buffers, procedures, components interacting via strictly enforced interfaces
Towards Secure Compilation Chain

Compartmentalized unsafe source →

Compartmentalized abstract machine →

Micro-policy machine

Buffers, procedures, components interacting via strictly enforced interfaces

Simple RISC abstract machine with build-in compartmentalization
Towards Secure Compilation Chain

Compartamentalized unsafe source

Compartamentalized abstract machine

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)
Towards Secure Compilation Chain

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)

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
  (program rewriting, shadow call stack)
Towards Secure Compilation Chain

(mostly) Verified (in Coq)

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)

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)
Towards Secure Compilation Chain

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)

Inline reference monitor enforcing:
- component separation
- procedure call and return discipline (program rewriting, shadow call stack)

Systematically tested (with QuickChick)
Next steps towards making this practical
Next steps towards making this practical

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

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

• **Achieve confidentiality (hypersafety) preservation**
  – in a realistic attacker model with side-channels
Next steps towards making this practical

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

• **Achieve confidentiality (hypersafety) preservation**
  – in a realistic attacker model *with side-channels*

• **Devise scalable proof methods for (hyper)liveness**
Next steps towards making this practical

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

• Achieve confidentiality (hypersafety) preservation
  – in a realistic attacker model with side-channels

• Devise scalable proof methods for (hyper)liveness

• Support dynamic component creation
Grand Challenge

Build the first efficient formally secure compilers for realistic programming languages
Grand Challenge

Build the first efficient formally secure compilers for realistic programming languages

1. Provide secure semantics for low-level languages
   – C with protected components and memory safety
Grand Challenge

Build the first efficient formally secure compilers for realistic programming languages

1. Provide secure semantics for low-level languages
   – C with protected components and memory safety

2. Enforce secure interoperability with unsafe code
   – ASM, C, and Low*
     [= safe C subset embedded in F* for verification]
Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

miTLS*
Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

miTLS*

KremSec
Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

miTLS*
KremSec
memory safe C component
Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

miTLS*
KremSec
memory safe C component
Goal: achieving secure compilation at scale

**Low* language**
(safe C subset in F*)

**C language**
+ components
+ memory safety

**ASM language**
(RISC-V + micro-policies)
Goal: achieving secure compilation at scale

**Low* language**
(safe C subset in F*)

**C language**
+ components
+ memory safety

**ASM language**
(RISC-V + micro-policies)
Goal: achieving secure compilation at scale

- **Low* language**
  - (safe C subset in F*)

- **C language**
  - + components
  - + memory safety

- **ASM language**
  - (RISC-V + micro-policies)

Diagram: 
- miTLS* connects to KremSec
- KremSec connects to memory safe C component
- CompSec connects to legacy C component
- ASM component connects to protecting component boundaries
Goal: achieving secure compilation at scale

- **Low* language**
  - (safe C subset in F*)

- **C language**
  - + components
  - + memory safety

- **ASM language**
  - (RISC-V + micro-policies)

Protecting component boundaries
Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)
Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

protecting higher-level abstractions

miTLS*

KremSec

CompSec

memory safe C component

legacy C component

protecting component boundaries

CompSec+

ASM component
Beyond robust safety preservation

Legend
(Trace) property = set of traces
Hyperproperty = set of sets of traces
Beyond robust safety preservation

Legend
(Trace) property = set of traces
Hyperproperty = set of sets of traces
Beyond robust safety preservation

Legend
(Trace) property = set of traces
Hyperproperty = set of sets of traces

Robust 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 Property Preservation

Robust Safety Preservation

Robust Hypersafety Preservation

Robust K-Hypersafety Preservation

Robust 2-Hypersafety Preservation

Fully Abstract Compilation

back-translating contexts
∀P∀C_T∈C_S∀t...

back-translating finite trace prefixes
∀P∀C_T∀t∈C_S...
Beyond robust safety preservation

Legend
(Trace) property = set of traces
Hyperproperty = set of sets of traces

[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hriţcu, et al]
All of this is either work in progress ... or wild speculation ... but ...
All of this is either work in progress ... or wild speculation ... but ...

• **Working on it!**
  – *growing team* at Inria Paris: Rob Blanco (PostDoc), Carmine Abate (Intern), Jérémy Thibault (Intern)
  – *collaborators* at UPenn, MPI-SWS, Draper, Portland, ...
All of this is either work in progress ... or wild speculation ... but ...

• **Working on it!**
  – **growing team** at Inria Paris: Rob Blanco (PostDoc), Carmine Abate (Intern), Jérémy Thibault (Intern)
  – **collaborators** at UPenn, MPI-SWS, Draper, Portland, ...

• **We're hiring!**
  – Interns, PhD students, PostDocs, Researchers
All of this is either work in progress ... or wild speculation ... but ...

• **Working on it!**
  – **growing team** at Inria Paris: Rob Blanco (PostDoc), Carmine Abate (Intern), Jérémy Thibault (Intern)
  – **collaborators** at UPenn, MPI-SWS, Draper, Portland, ...

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

• **Open to new collaborations**
All of this is either work in progress ... or wild speculation ... but ...

• **Working on it!**
  – **growing team** at Inria Paris: Rob Blanco (PostDoc), Carmine Abate (Intern), Jérémy Thibault (Intern)
  – **collaborators** at UPenn, MPI-SWS, Draper, Portland, ...

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

• **Open to new collaborations**

• **Building a community**
  – Principles of Secure Compilation (PriSC) @ POPL
  – Dagstuhl seminar in May
BACKUP SLIDES
Scalable proof technique
for our extension of robustly safe compilation
Scalable proof technique
for our extension of robustly safe compilation

1. back-translating finite trace prefixes to whole source programs
Scalable proof technique
for our extension of robustly safe compilation

1. back-translating finite trace prefixes to whole source programs
   – limitation: only works for preserving (hyper)safety
Scalable proof technique
for our extension of robustly safe compilation

1. back-translating finite trace prefixes to whole source programs
   - limitation: only works for preserving (hyper)safety

2. generically defined semantics for partial programs
   - related to whole-program semantics via trace composition and decomposition lemmas
Scalable proof technique for our extension of robustly safe compilation

1. back-translating finite trace prefixes to whole source programs
   – limitation: only works for preserving (hyper)safety

2. generically defined semantics for partial programs
   – related to whole-program semantics via trace composition and decomposition lemmas

3. using whole-program compiler correctness proof (à la CompCert) as a black-box
   – for moving back and forth between source and target
Scalable proof technique
for our extension of robustly safe compilation

1. back-translating finite trace prefixes to whole source programs
   – limitation: only works for preserving (hyper)safety

2. generically defined semantics for partial programs
   – related to whole-program semantics via trace composition and decomposition lemmas

3. using whole-program compiler correctness proof (à la CompCert) as a black-box
   – for moving back and forth between source and target

all this yields much simpler and more scalable proofs