Formally Secure Compilation of Unsafe Low-level Components

Cătălin Hrițcu
Inria Paris, Prosecco team

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

- Inherently insecure C/C++-like languages
  - type and memory unsafe:
    e.g. any buffer overflow is catastrophic
Devastating low-level vulnerabilities

• Inherently insecure C/C++-like languages
  – type and memory unsafe:
    e.g. 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"
  – "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
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 all this
  – e.g. crypto libraries/protocols ... verified (HACL*/miTLS*) or not
Goal 1: Formalizing the security of compartmentalizing compilation
Goal 1: Formalizing the security of compartmentalizing compilation
Goal 1: Formalizing the security of compartmentalizing compilation

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

Diagram:
- C component
- ASM component
- Compiler
Goal 1: Formalizing the security of compartmentalizing compilation

Program trace

Compiler correctness (e.g. CompCert) not enough

Program trace

C component

ASM component

Malicious components

e.g. compromised ASM obtained from C
Goal 1: Formalizing the security of compartmentalizing compilation

*program trace*

**compiler correctness**
(e.g. CompCert)

**not enough**

**program trace**

C component

well-defined C components

**compiler**

secure compilation

**ASM component**

malicious components

e.g. compromised ASM obtained from C
Goal 1: Formalizing the security of compartmentalizing compilation

- C component
- well-defined C components
- secure compilation
- ASM component
- malicious components
- protected
- no extra power
- compiler correctness (e.g. CompCert)
- not enough
- program trace
- compiler
- e.g. compromised ASM obtained from C

- program trace
Goal 1: Formalizing the security of compartmentalizing compilation

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

\[ \exists \text{low-level attacker} \]
Fully abstract compilation
preservation of observational equivalence

∃ low-level attacker

1\textsuperscript{st} high-level component

compiler

1\textsuperscript{st} compiled component

≈

2\textsuperscript{nd} compiled component

compiler

2\textsuperscript{nd} high-level component

low-level attacker

low-level attacker
Fully abstract compilation
preservation of observational equivalence
Fully abstract compilation
preservation of observational equivalence
#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

```bash
$ 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 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
$ ./a.out haha
$ ./a.out hahahahahahahahahahahahahahahahaha
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 haha
$ ./.a.out hahahaha
zsh: segmentation fault (core dumped)
$ ./.exploit.sh | a.out
```

Buffer overflow
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, 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 $C_2\downarrow, C_4\downarrow, C_5\downarrow$

[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, Hriţcu, Azevedo de Amorim, Eng, Pierce, CSF’16]
Full abstraction for mutually distrustful components

∀ compromise scenarios.

∃ high-level attack from some fully defined \( A_2, A_4, A_5 \)

\[
\begin{array}{c}
\begin{array}{cccc}
C_1 & A_2 & C_3 & A_4 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
D_1 & A_2 & D_3 & A_4 \\
\end{array}
\end{array}
\]

if \( C_1, C_3, D_1, D_3 \) fully defined and

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

\[
\begin{array}{c}
\begin{array}{cccc}
C_1 \downarrow & C_2 \downarrow & C_3 \downarrow & C_4 \downarrow \\
\downarrow & \downarrow & \downarrow & \downarrow \\
D_1 \downarrow & C_2 \downarrow & D_3 \downarrow & C_4 \downarrow \\
\end{array}
\end{array}
\]

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)

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]


```
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 $C_1$ nor $C_2$ are fully defined

```java
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 \( C_1.\text{parse} \)

```scala
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

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
}
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 \( t \)

high-level attacker

compiler

high-level component

compiled component

∀ (bad attack) trace \( t \)
We build instead on Robust Compilation

∀(bad attack) trace t

∃ high-level attacker causing t

∃ low-level attacker causing t

high-level component

compiler

low-level component

∀ (bad attack) trace t

∃ high-level attacker

∃ 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

low-level component

compiler

∀ (bad attack) trace \( t \)
We build instead on Robust Compilation

∀(bad attack) trace $t$

∃ high-level attacker causing $t$

high-level component

⇒

∃ low-level attacker causing $t$

compiled component

robust trace property preservation
(robust = in adversarial context)
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

robust trace property preservation
(robust = in adversarial context)

intuition:
- stronger than compiler correctness
  (i.e. trace property preservation)
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)
- confidentiality not preserved (i.e. no hyperproperties)
We build instead on Robust Compilation

∀ (bad attack) trace \( t \)

- high-level attacker causing \( t \)
- compiler
- 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
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!)

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 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!)

extends to unsafe languages, supporting dynamic compromise

intuition:

– stronger than compiler correctness (i.e. trace property preservation)

– confidentiality not preserved (i.e. no hyperproperties)

– less extensional than fully abstract compilation
Dynamic compromise

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

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

∃ 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:

\[
\begin{align*}
(0) & \quad C_0 \quad C_1 \quad C_2 \\
\downarrow & \quad m_1; \text{Undef}(C_1)
\end{align*}
\]
∃ a dynamic compromise scenario explaining \( t \) in source language for instance leading to the following compromise sequence:

(0) \( \exists A \).

(1) \( \exists A_1 \).

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:

1. $\exists A_1$. $\exists A_2$. $\downarrow t$
∃ a dynamic compromise scenario explaining \( t \) in source language for instance leading to the following compromise sequence:

1. \( \exists A_1. \) 

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

1. $\exists A_1$. $\exists A_2$. Trace is very helpful
   - detect undefined behavior
   - rewind execution

[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]
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
Now we know what these words mean!
(at least in the setting of compartmentalization for unsafe low-level languages)

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

Mutual distrust \[ \text{Mutual distrust} \quad C_1 \quad A_2 \quad C_3 \quad A_4 \quad A_5 \]

Dynamic compromise \[ \text{Dynamic compromise} \quad C_0 \quad A_1 \quad C_2 \quad \downarrow m_2; \text{Undef}(C_2) \]
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
Towards Secure Compilation Chain

Compartmentalized unsafe source

Compartmentalized abstract machine

Micro-policy machine
Towards Secure Compilation Chain

Compartmentalized unsafe source

Compartmentalized abstract machine

Micro-policy machine

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

Buffers, procedures, components interacting via strictly enforced interfaces

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

Compartmentalized unsafe source

Compartmentalized abstract machine

Simple RISC abstract machine with build-in compartmentalization

Buffers, procedures, components interacting via strictly enforced interfaces

Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
  (linear capabilities / linear entry points)
Towards Secure Compilation Chain

Compartamentalized unsafe source

Buffers, procedures, components interacting via strictly enforced interfaces

Compartamentalized abstract machine

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)
Towards Secure Compilation Chain

(mostly) 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)
Towards Secure Compilation Chain

(mostly) 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

**Towards 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)

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)

• **Verify compartmentalized applications**
  – put the source-level reasoning principles to work
Next steps towards making this 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 this 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, HBAC, ...
Next steps towards making this 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, HBAC, ...**

• **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)

• Verify compartmentalized applications
  – put the source-level reasoning principles to work

• Extend all this to dynamic component creation

• ... and dynamic privileges: capabilities, HBAC, ...

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

• Devise scalable proof techniques for (hyper)liveness preservation (possible?)
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: achieve secure compilation at scale

Low* language
(safe C subset in F*)

miTLS*

C language
+ components
+ memory safety
Goal: achieve secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

miTLS*

KremSec
Goal: achieve secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

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

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety
Goal: achieve secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

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

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

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

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

**C language**
+ components
+ memory safety

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

Diagram:
- miTLS*
- KremSec
- CompSec+
- memory safe C component
- legacy C component
- ASM component

Protecting component boundaries
Goal: achieve 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: achieve 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: achieve 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**

**Protecting higher-level abstractions**

- miTLS*
- KremSec
- CompSec*
- CompSec
- ASM component

- Memory safe C component
- Legacy C 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

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

Robust Relational Hypersafety Preservation
- Robust Relational Safety Preservation

[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, Patrignani, et al]
Beyond robust safety preservation

Legend
- Trace property = set of traces
- Hyperproperty = set of sets of traces

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 Property Preservation

Robust Relational Hypersafety Preservation
  | Robust Relational Safety Preservation

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

Robust Safety Preservation

back-translating finite trace prefixes
∀ P∀ C T ∀m ≤ t ∈ C S...

[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, Patrignani, et al]
Beyond robust safety preservation

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

Robust Hyperproperty 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 Property Preservation

Robust Relational Hypersafety Preservation
- Robust Relational Safety Preservation
  - Robust Finite-Relational Safety Preservation
    - Robust Hypersafety Preservation
      - Robust K-Hypersafety Preservation
        - Robust 2-Hypersafety Preservation
  - Fully Abstract Compilation

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

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

[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, Patrignani, et al]
Beyond robust safety preservation

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

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 Property Preservation

Robust Relational Hyperproperty Preservation
  Robust Relational Hypersafety Preservation

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

Robust Safety Preservation

back-translating contexts & progs
∀P∀C_T∃C_S∀t...

back-translating finite sets of finite trace prefixes
∀k∀P_1..P_k∀C_T
∀m_1..m_k∃C_S...

Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, Patrignani, et al
Beyond robust safety preservation

Legend
- Trace property = set of traces
- Hyperproperty = set of sets of traces

∀CT∃CS∀P∀t...

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 K-Relational Hyperproperty Preservation
| Robust K-Hyperproperty Preservation
| Robust 2-Hyperproperty Preservation

Robust Hypersafety Preservation
| Robust Finite-Relational Safety Preservation
| Robust Finite-Safety Preservation

Robust Relational Hypersafety Preservation
| Robust Relational Safety Preservation

∀k∀P1..Pk ∀CT
∀m1..mk ∈ CS...

∀P∀CT∃CS ∀t...

back-translating contexts & progs

back-translating contexts

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

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

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

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

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

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

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

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

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

- **Open to new collaborations**

- **Building a community**
  - Workshop on Principles of Secure Compilation (PriSC) @ POPL
  - Dagstuhl Seminar on Secure Compilation in May