Formally Secure Compilation of Unsafe Low-level Components

Cătălin Hrițcu

Inria Paris, Prosecco team

https://secure-compilation.github.io

Collaborators

Cătălin Hrițcu

Yannis Juglaret

Guglielmo Fachini

Marco Stronati

Arthur Azevedo de Amorim

Ana Nora Evans

Rob Blanco

Carmine Abate

Boris Eng

Théo Laurent

Andrew Tolmach

Benjamin Pierce

Deepak Garg

Marco Patrignani

Inria Paris CMU U. Virginia U. Trento Paris 7 ENS Paris Portland State UPenn MPI-SWS

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

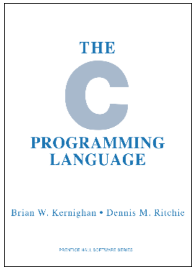
- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** running with **least privilege** & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component should not immediately 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

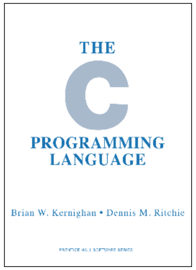
Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**



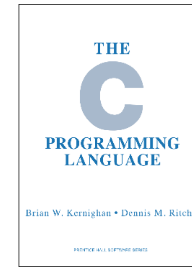
Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions:**
  - component separation, call-return discipline, ...

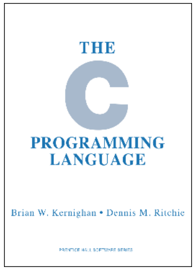


Goal 2: Build secure compilation chains

- **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



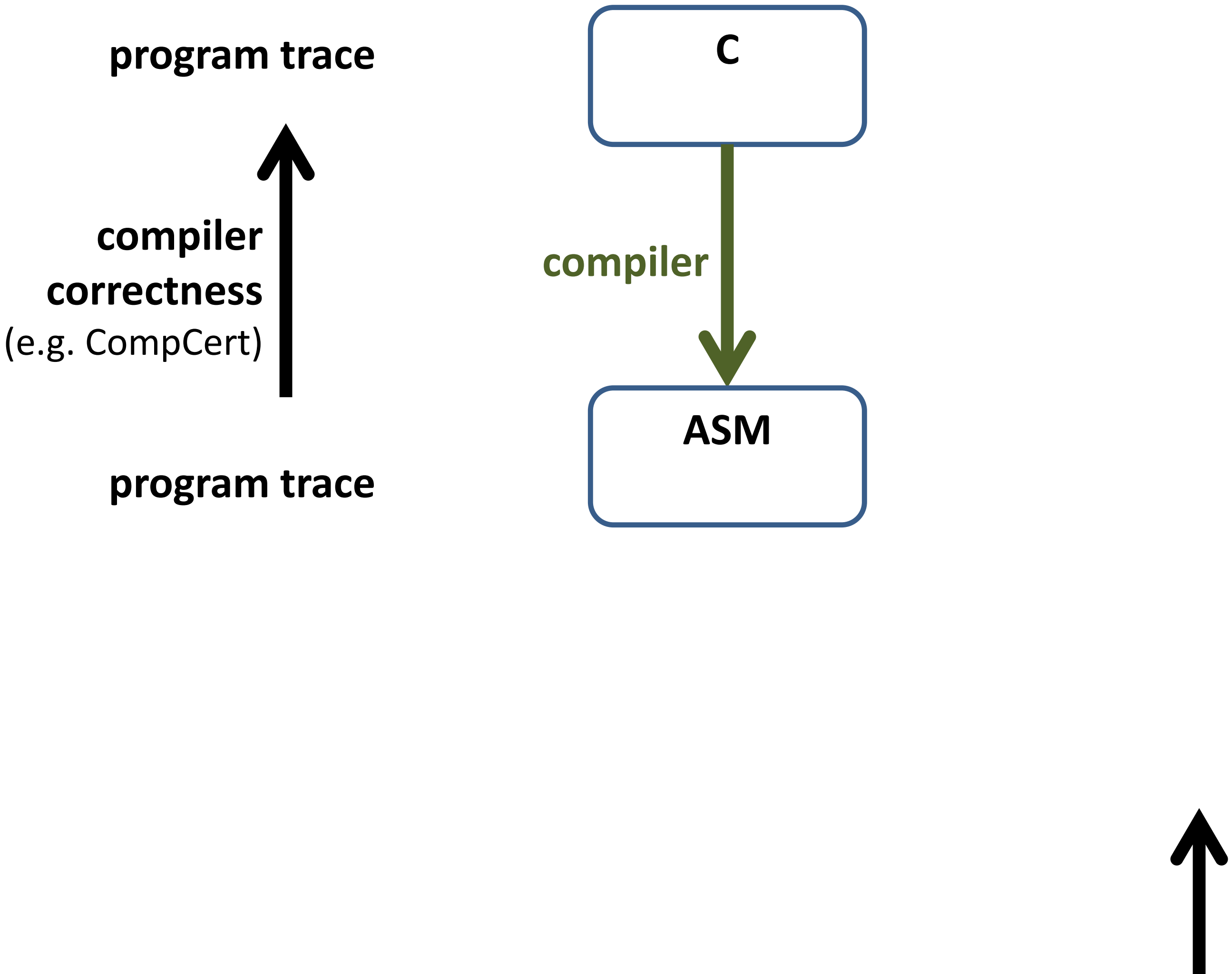
Goal 2: Build secure compilation chains



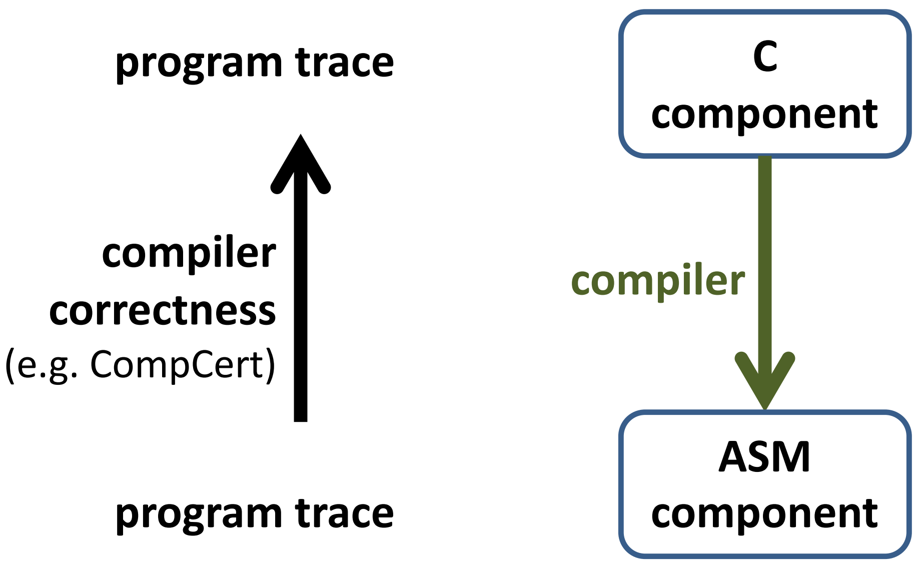
- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions:**
  - component separation, call-return discipline, ...
- **Secure compilation chain:**
  - compiler, linker, loader, runtime, system, hardware
- **Use efficient enforcement mechanisms:**
  - 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

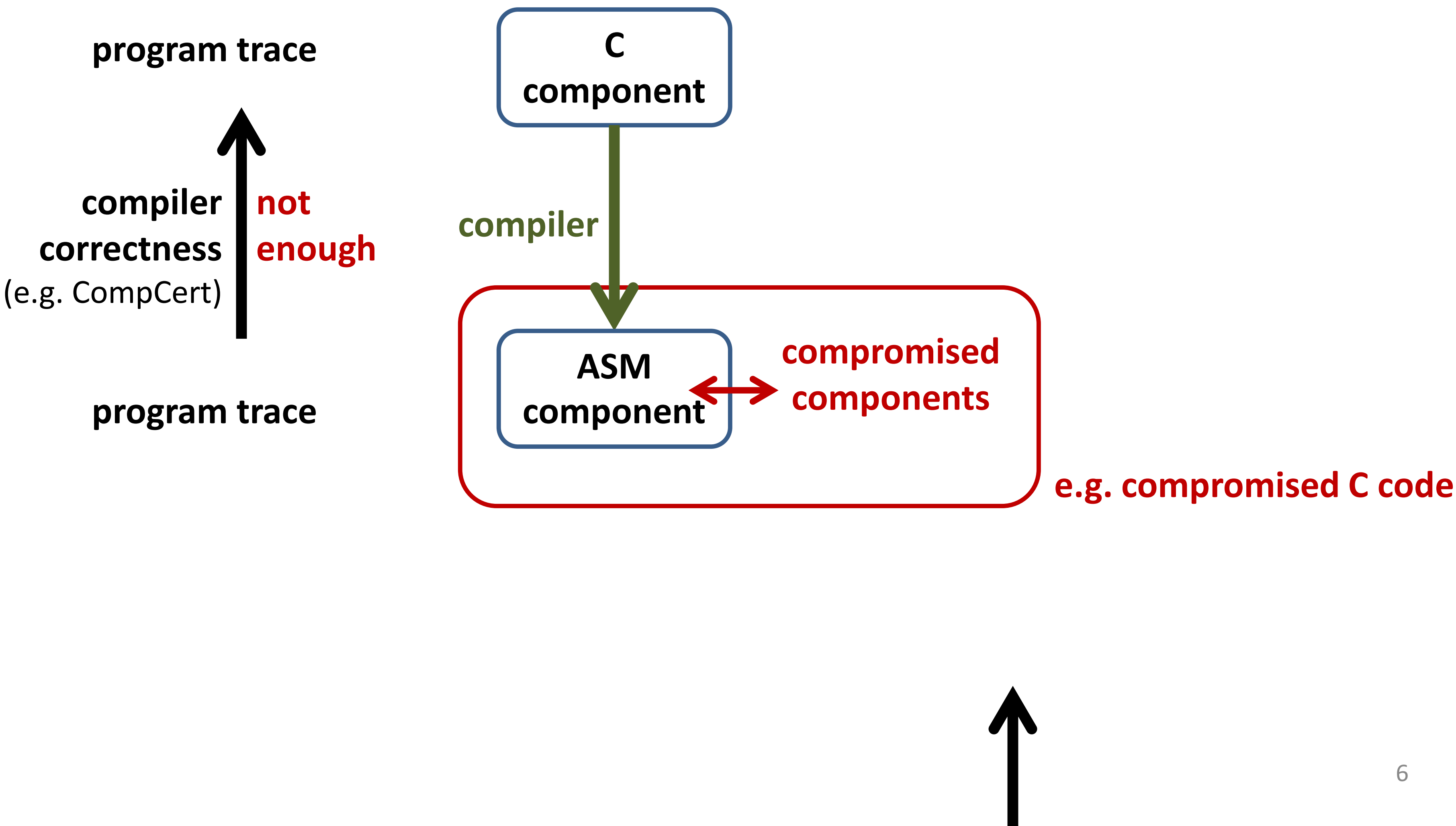
Goal 1: Formalizing security of compartmentalizing compilation



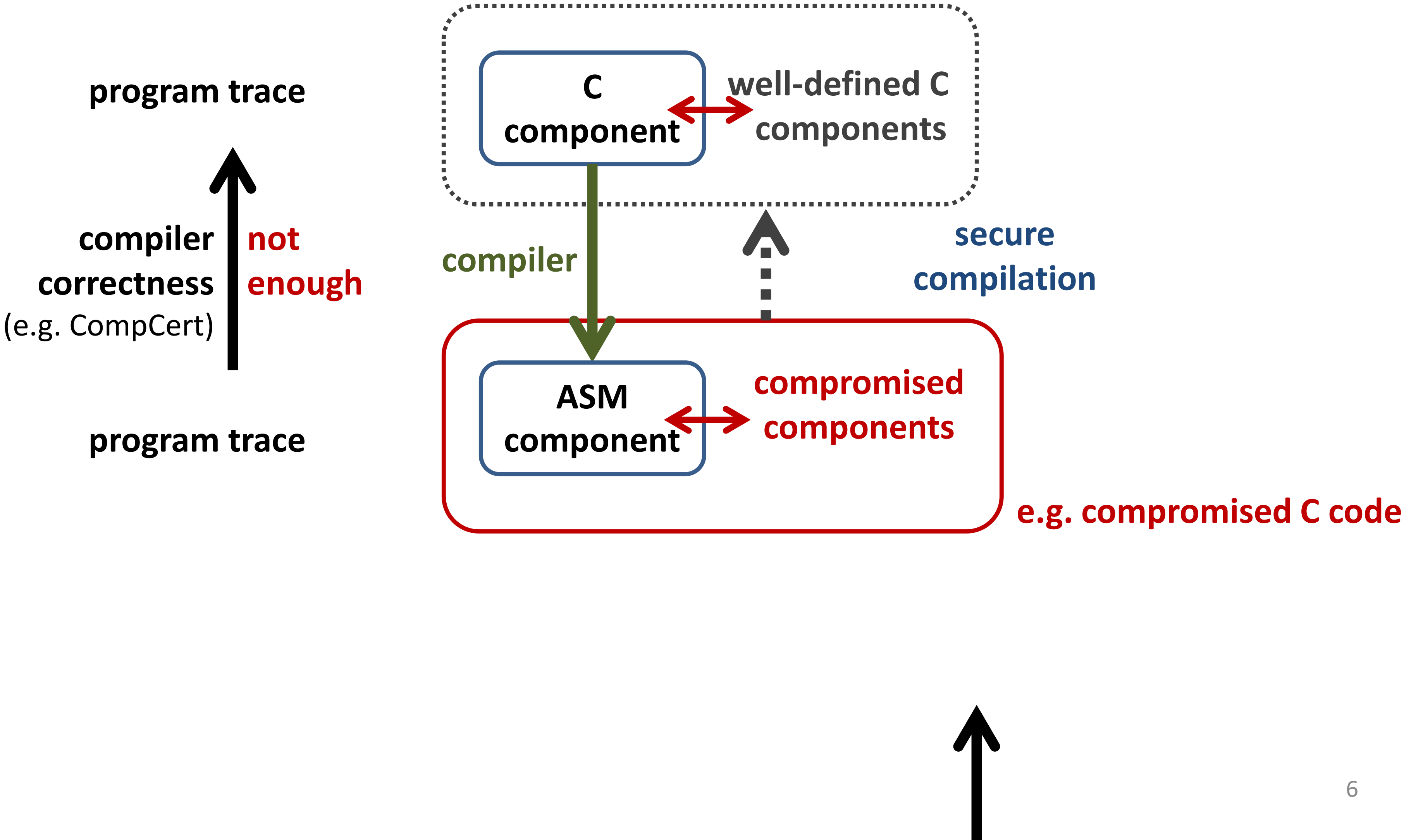
Goal 1: Formalizing security of compartmentalizing compilation



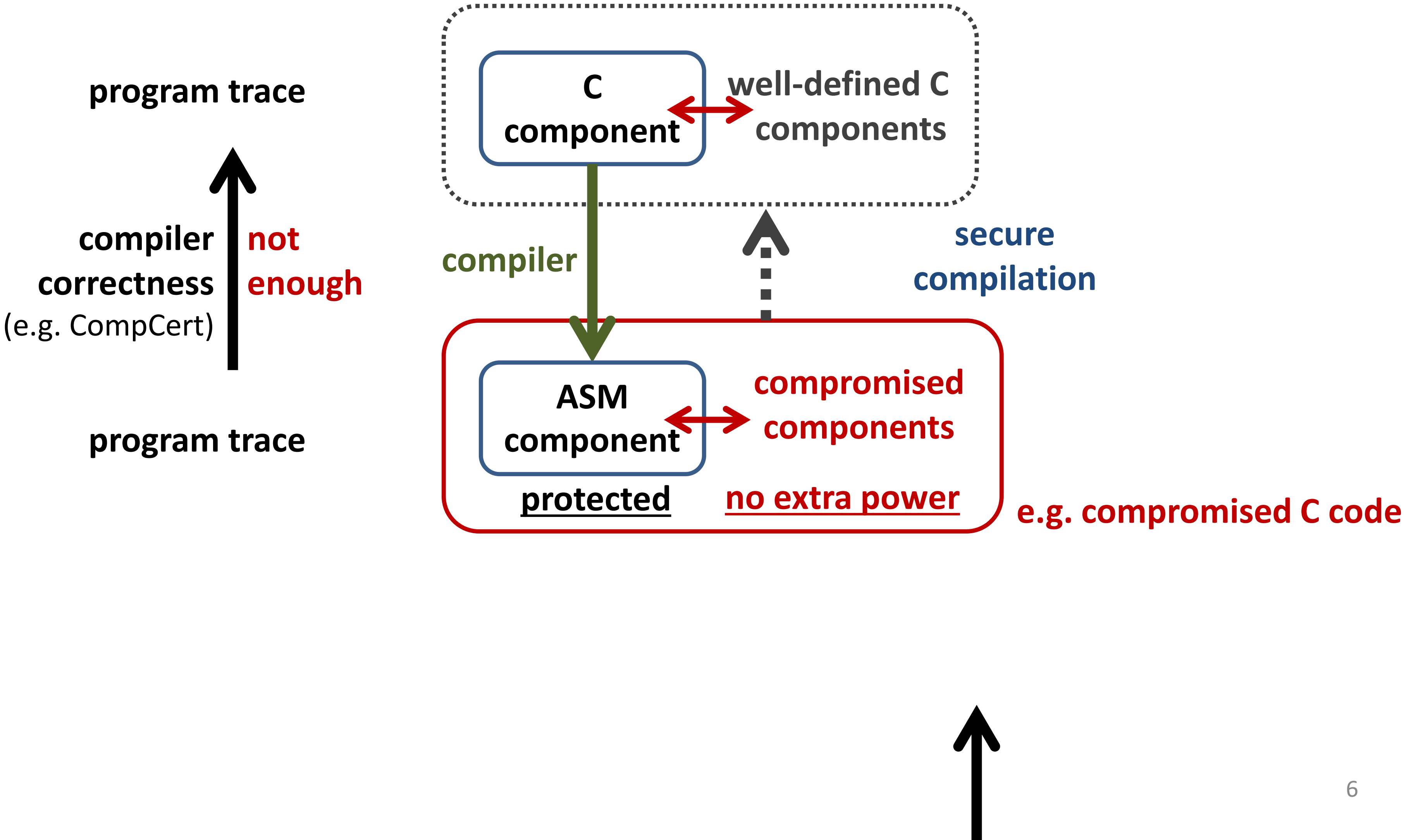
Goal 1: Formalizing security of compartmentalizing compilation



Goal 1: Formalizing security of compartmentalizing compilation



Goal 1: Formalizing security of compartmentalizing compilation



Goal 1: Formalizing security of compartmentalizing compilation

program trace

compiler correctness (e.g. CompCert)

not enough

program trace

C component

well-defined C components

secure

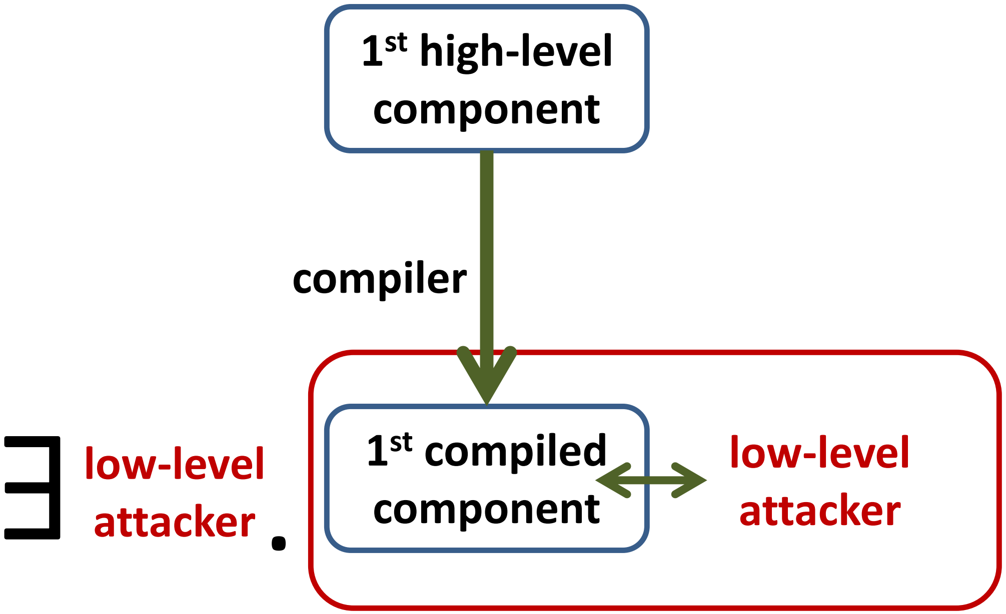
secure compilation

secure

Benefit: sound security reasoning in the source language

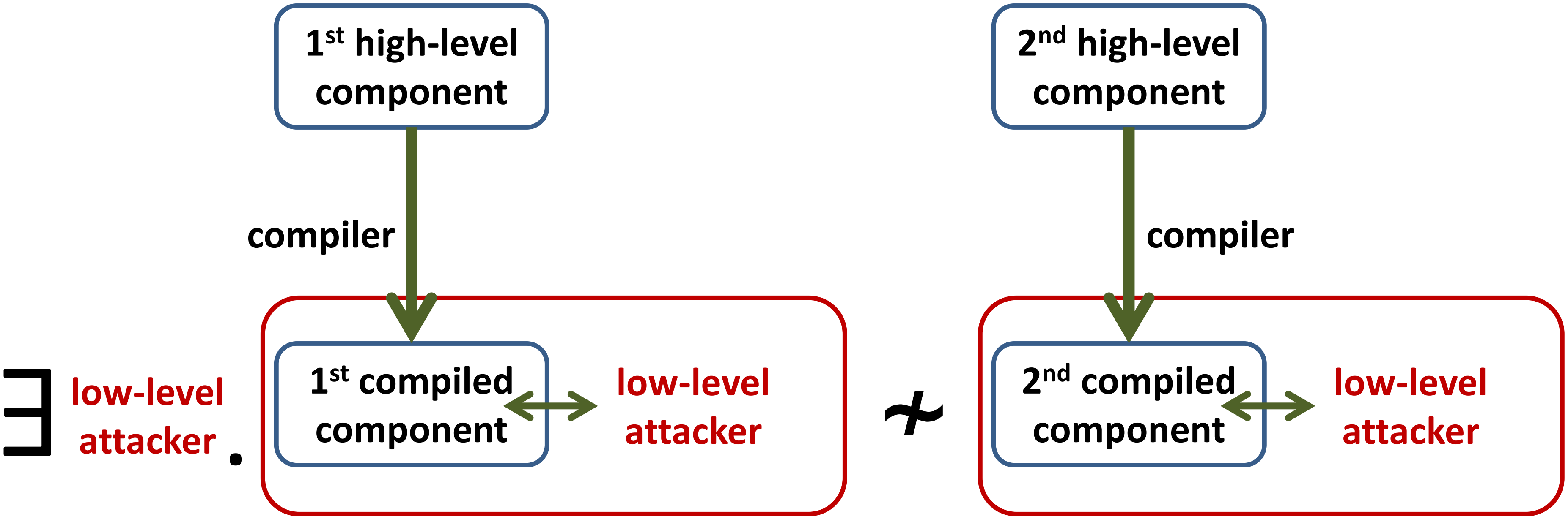
Fully abstract compilation

preservation of observational equivalence



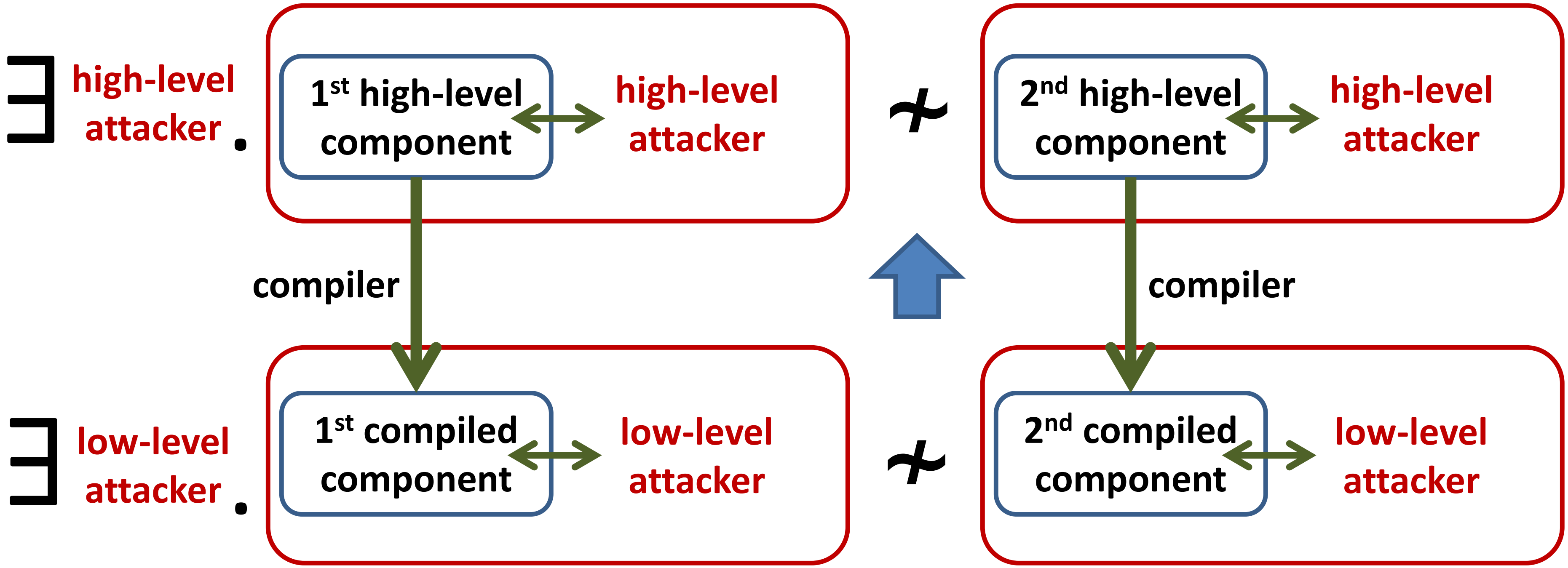
Fully abstract compilation

preservation of observational equivalence



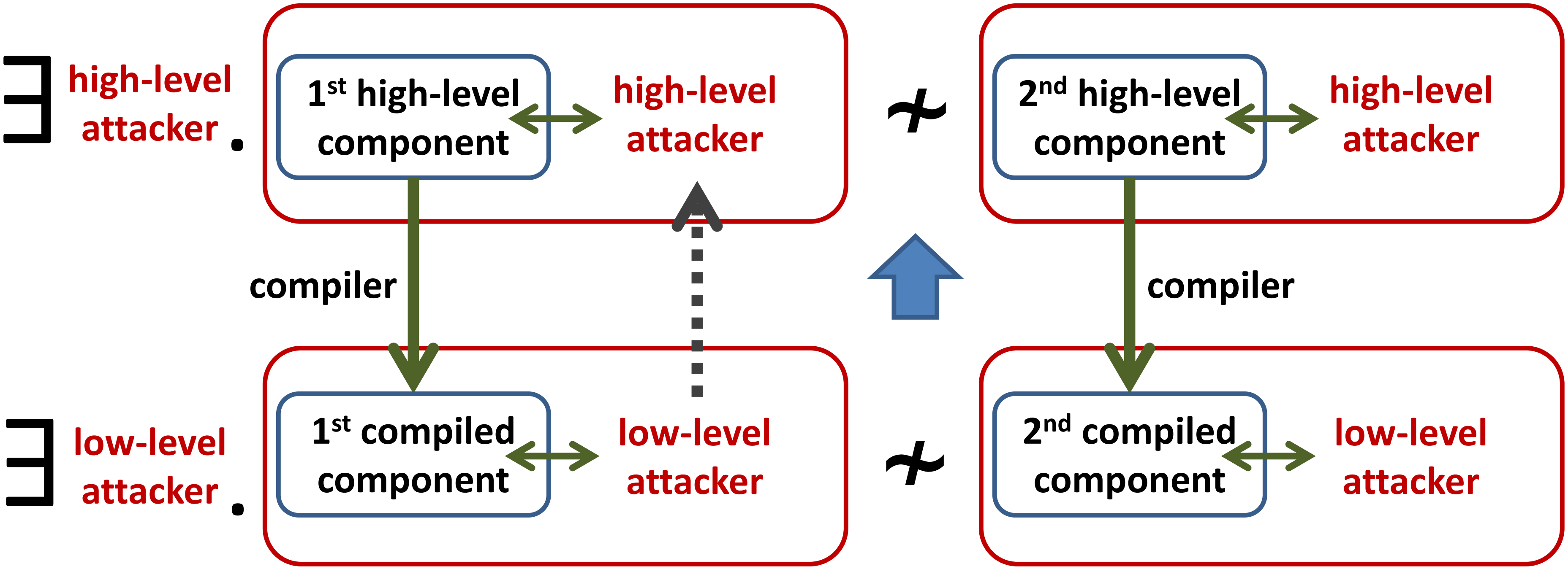
Fully abstract compilation

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

```
#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 hahahahahahahahahaha
zsh: segmentation fault (core dumped)
```

Undefined behavior

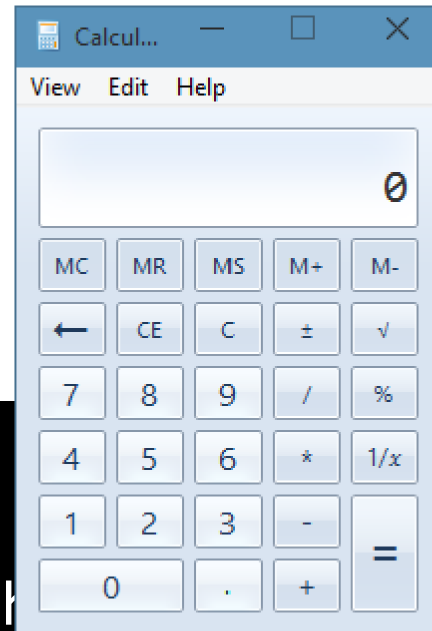
```
#include <string.h>
int main (int argc, char **argv) {
   char c[12];
   strcpy(c, argv[1]);
   return 0;
}
```

Buffer overflow

```
$ gcc target.c -fno-stack-protector
$ ./a.out haha
$ ./a.out hahahahahahahahahaha
zsh: segmentation fault (core dumped)
$ ./exploit.sh | a.out
```

Undefined behavior

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



erflow

```
$ gcc target.c                    protector
$ ./a.out haha
$ ./a.out hahah                   haha
zsh: segmentation fault (core dumped)
$ ./exploit.sh | a.out
```

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

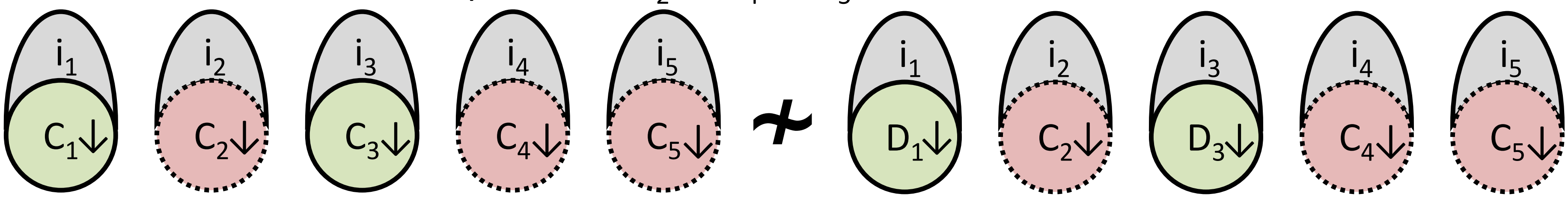
- **Source reasoning**
  = We want to reason formally about security
  with respect to source language semantics
- **Undefined behavior**
  = can't be expressed at all by source language semantics!
- **Problem: observational equivalence**
  doesn't work with undefined behavior!?
  – `int buf[5]; buf[42] ~? 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

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

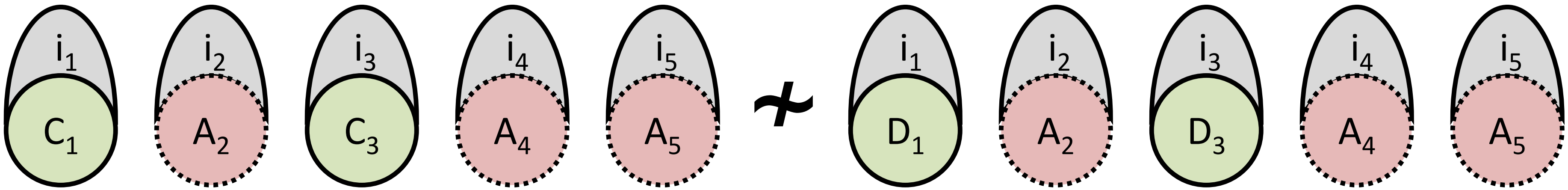


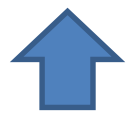
[Beyond Good and Evil - Juglaret, Hrițcu, Azevedo de Amorim, Eng, Pierce, CSF'16]

Full abstraction for mutually distrustful components

∀compromise scenarios.

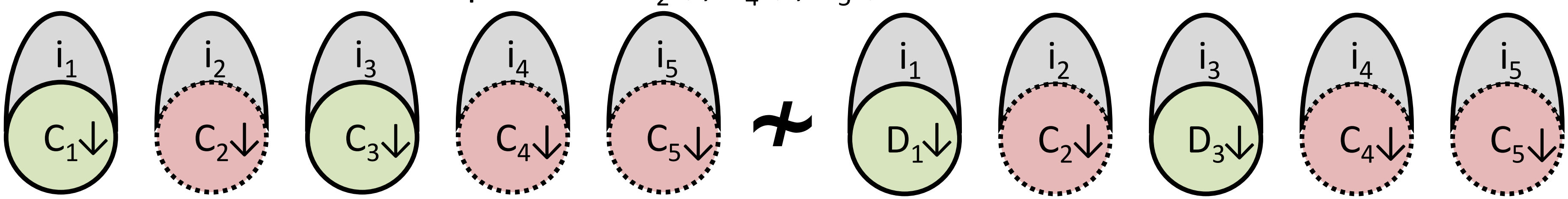
∃ high-level attack from some **fully defined** A_2, A_4, A_5





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

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

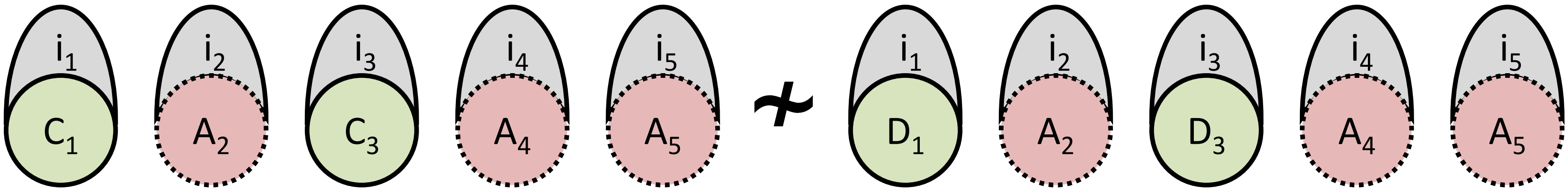


[Beyond Good and Evil - Juglaret, Hrițcu, Azevedo de Amorim, Eng, Pierce, CSF'16]

Full abstraction for mutually distrustful components

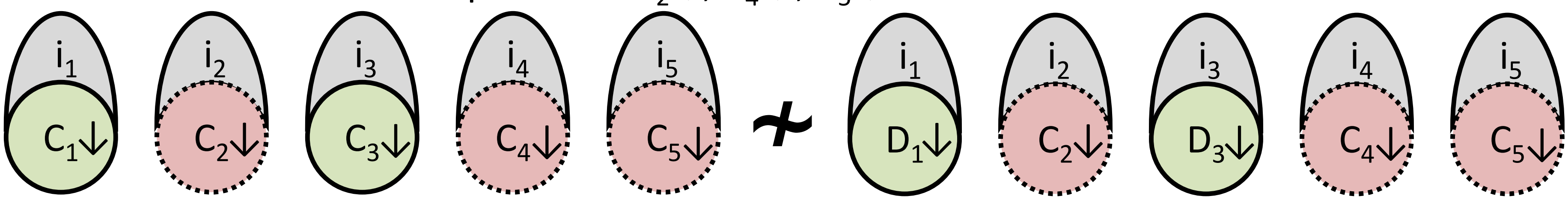
$\forall$compromise scenarios.

$\exists$ 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

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



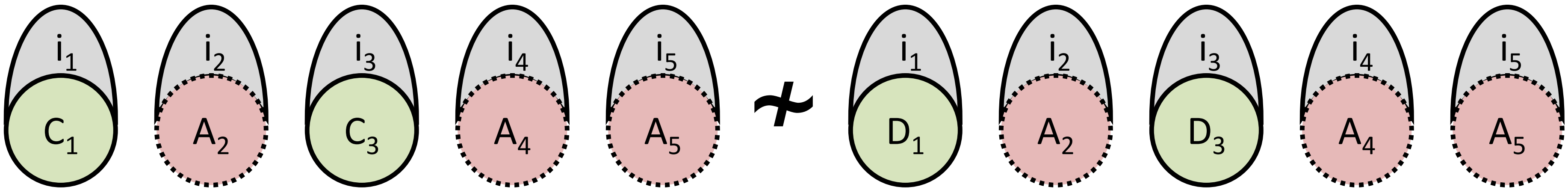
Limitation: static compromise model: C_1, C_3, D_1, D_3 get guarantees only if perfectly safe
(i.e. fully defined = do not exhibit undefined behavior in **any** context)

[Beyond Good and Evil - Juglaret, Hrițcu, Azevedo de Amorim, Eng, Pierce, CSF'16]

Full abstraction for mutually distrustful components

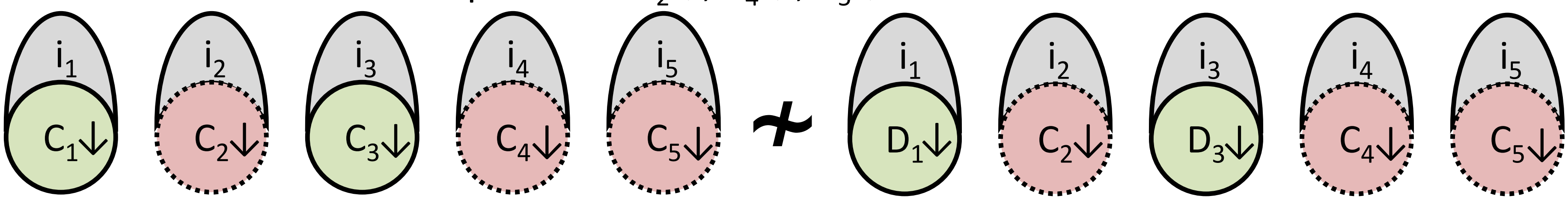
∀compromise scenarios.

∃ high-level attack from some **fully defined** 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)

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 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

yet C_1 is protected until calling C_1.parse

```
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

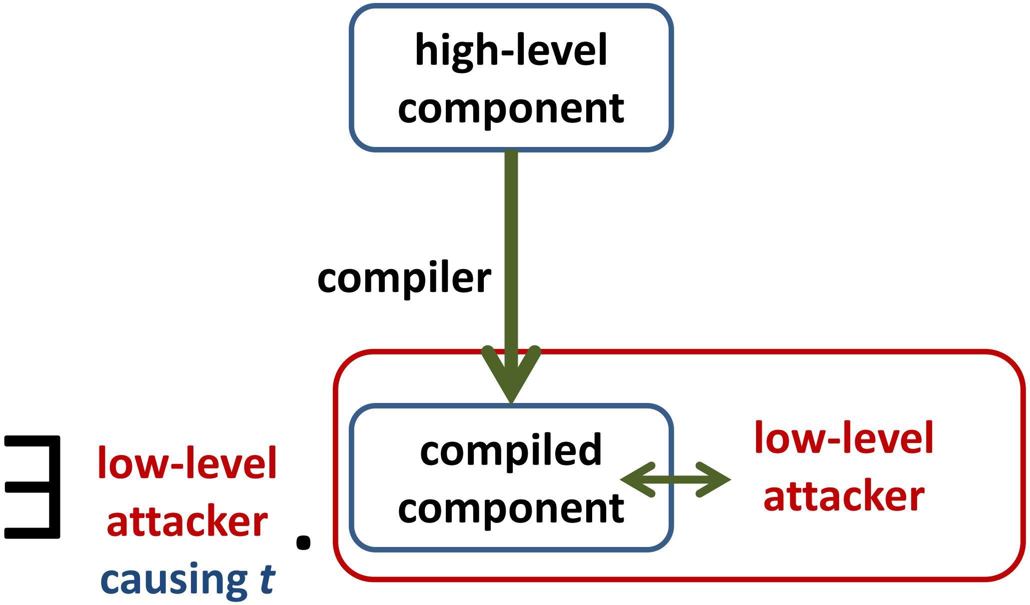
yet C_1 is protected until calling C_1.parse

and C_2 can't actually be compromised

```
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
}
```

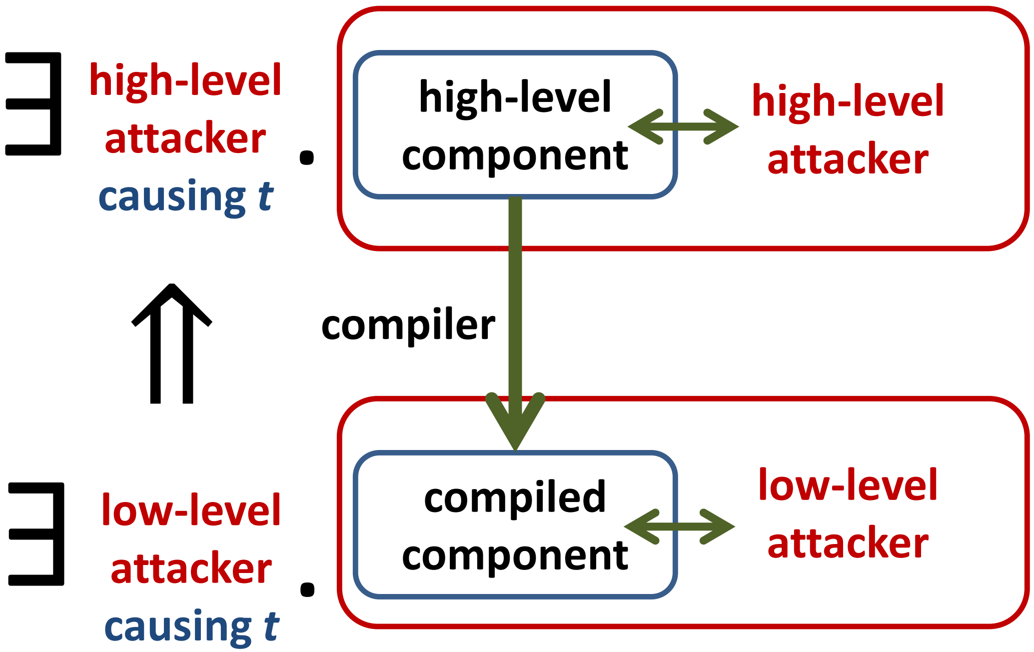
We build instead on Robust Compilation

∀(bad attack) trace *t*

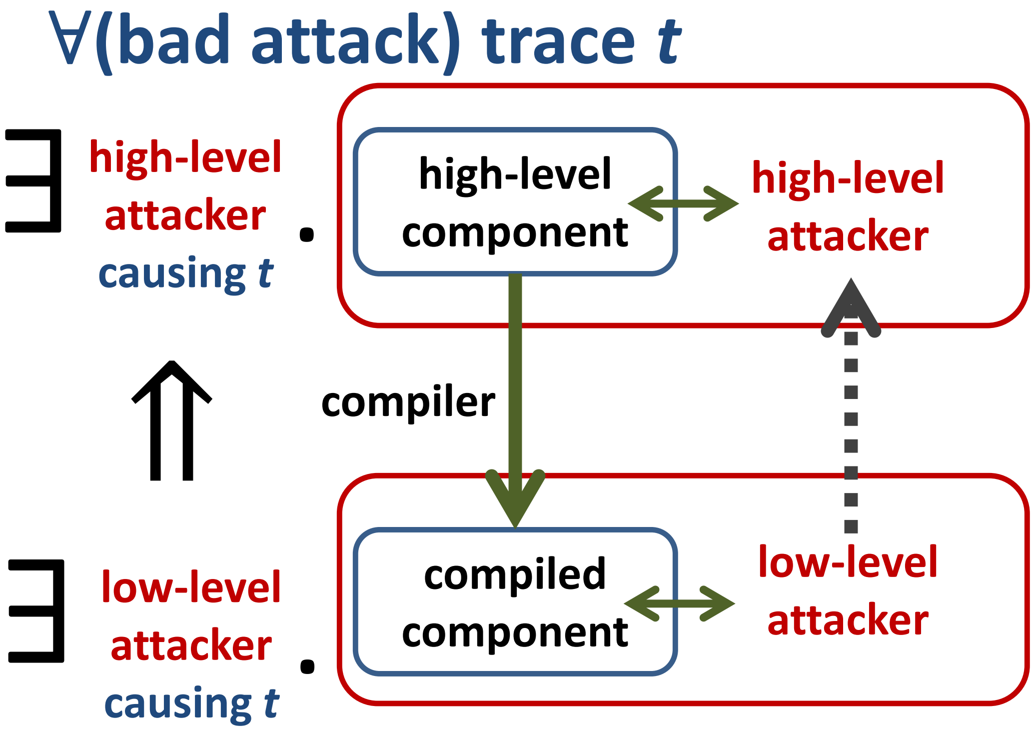


We build instead on Robust Compilation

∀(bad attack) trace t

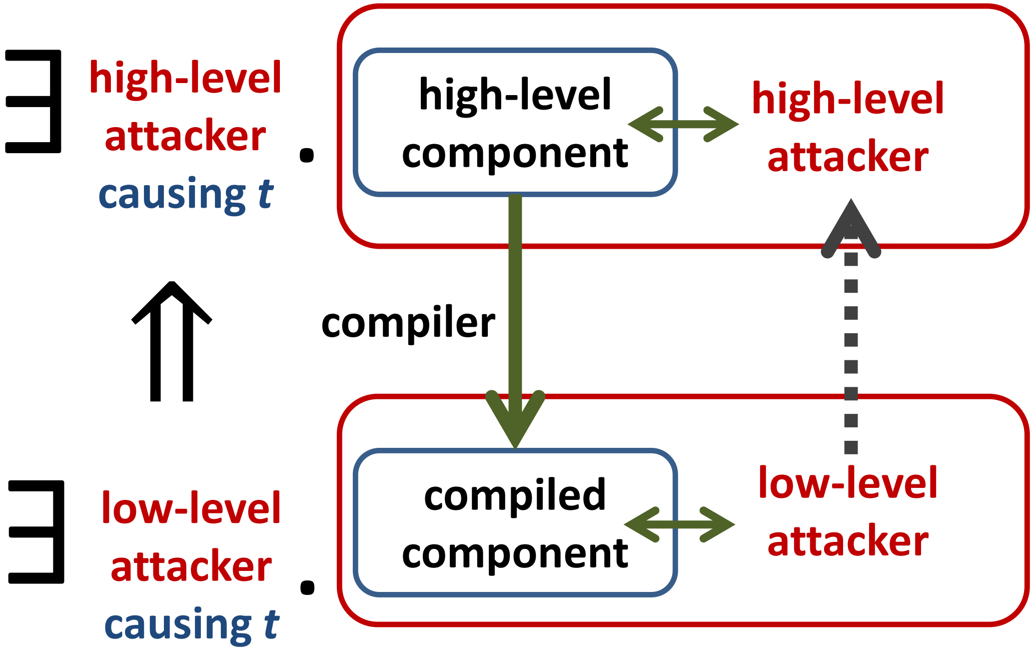


We build instead on Robust Compilation



We build instead on Robust Compilation

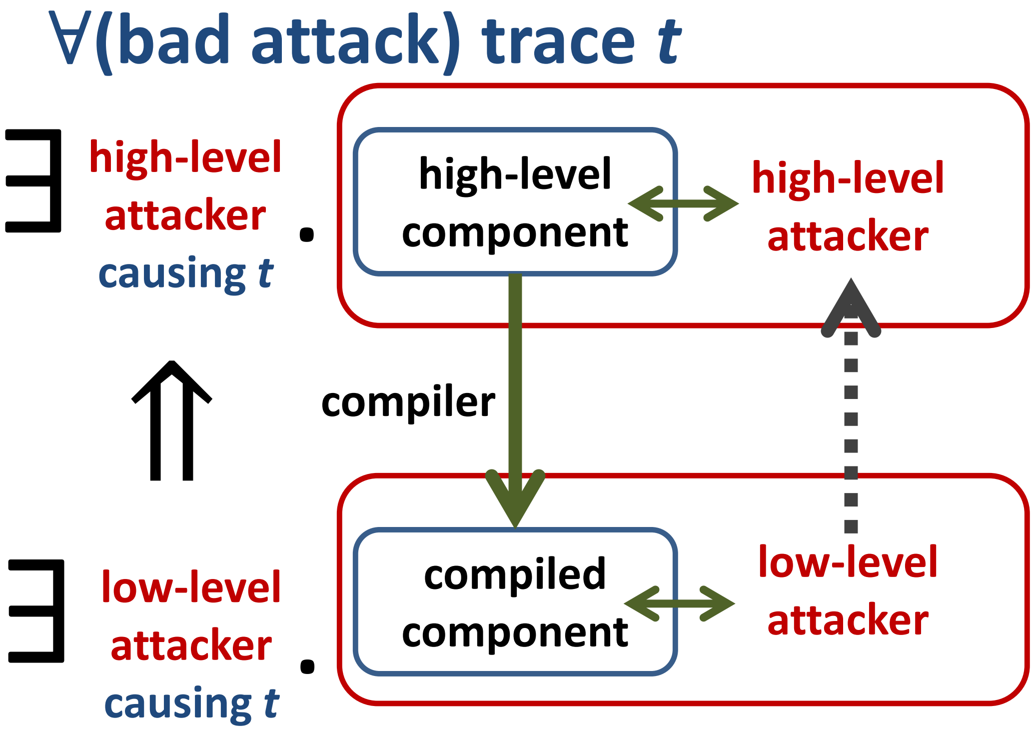
∀(bad attack) trace *t*



robust trace property preservation

(robust = in adversarial context)

We build instead on Robust Compilation



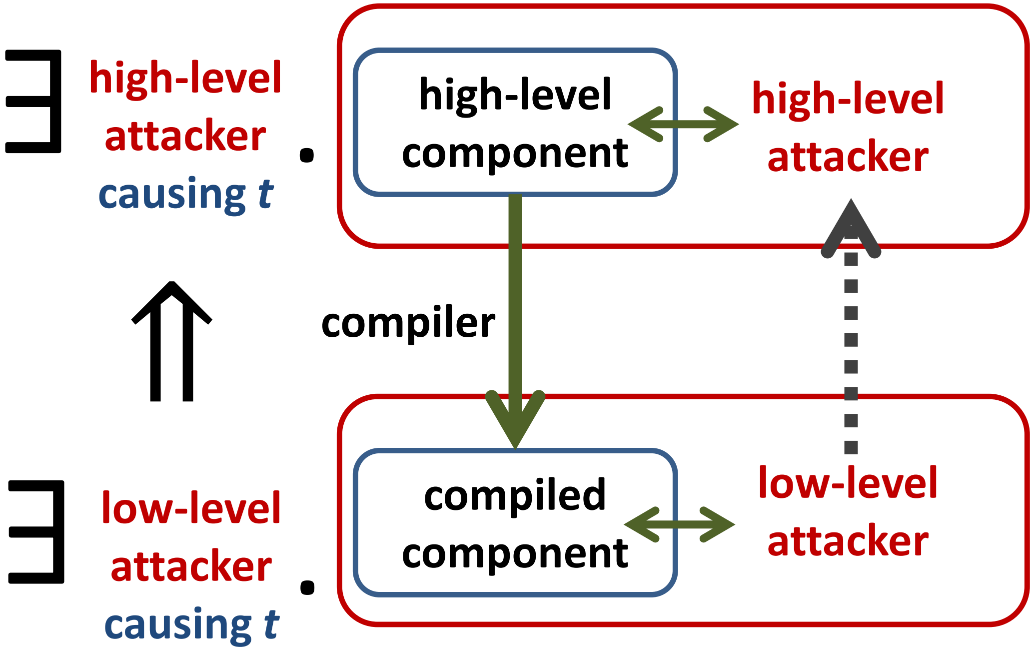
robust trace property preservation
(robust = in adversarial context)

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*



robust trace property preservation
(robust = in adversarial context)

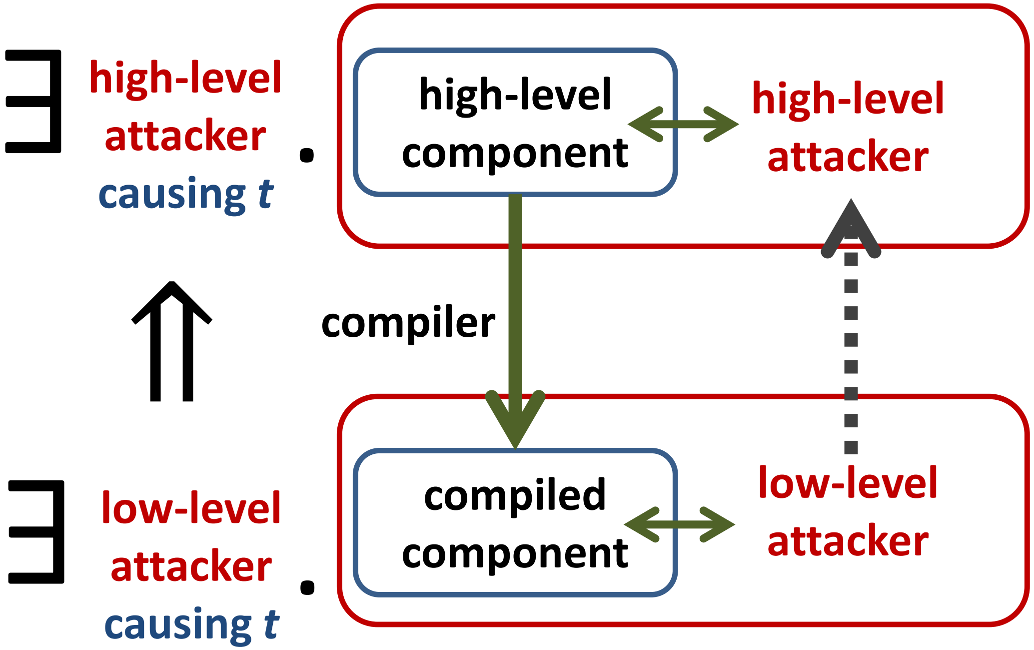
intuition:

- **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*



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

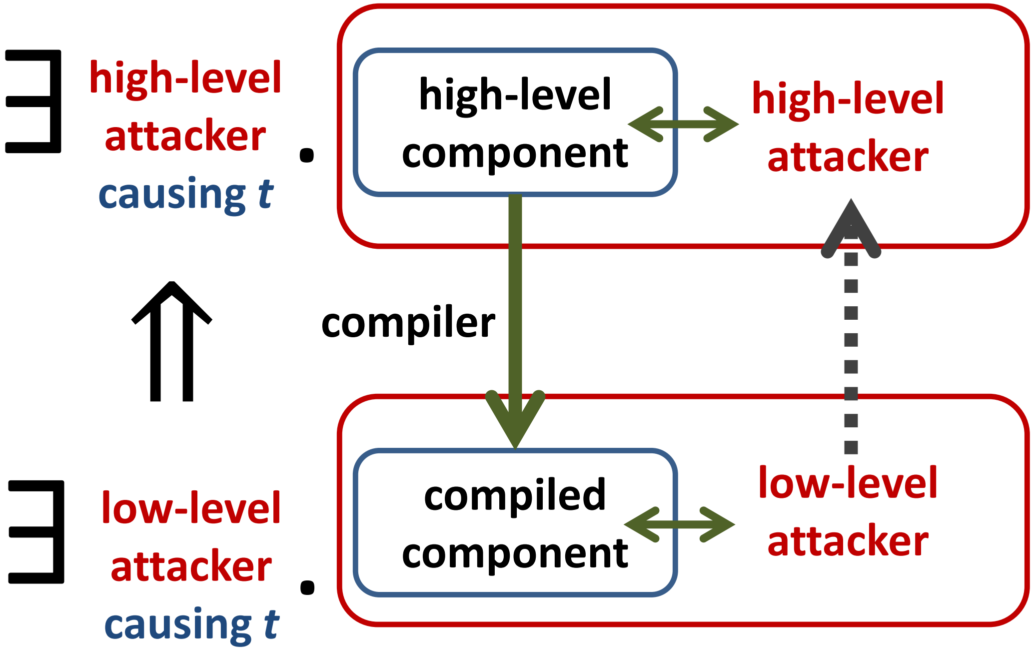
Advantages: **easier to realistically achieve and prove at scale**

useful: preservation of **invariants** and other **integrity properties**

more intuitive to security people (generalizes to hyperproperties!)

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**

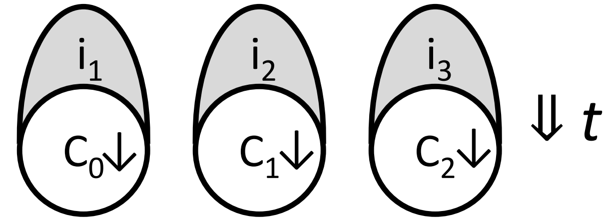
more intuitive to security people (generalizes to hyperproperties!)

extends to unsafe languages, supporting dynamic compromise

Dynamic compromise

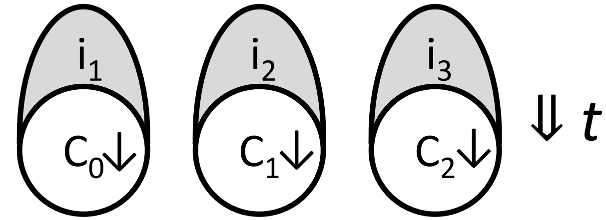
[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]

Dynamic compromise



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

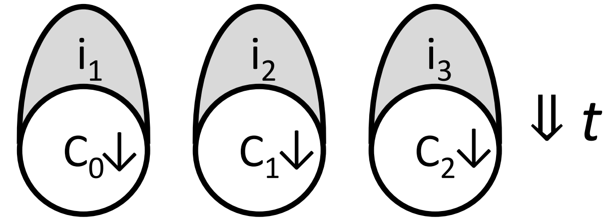
Dynamic compromise



∃ a **dynamic compromise scenario** explaining *t* in source language

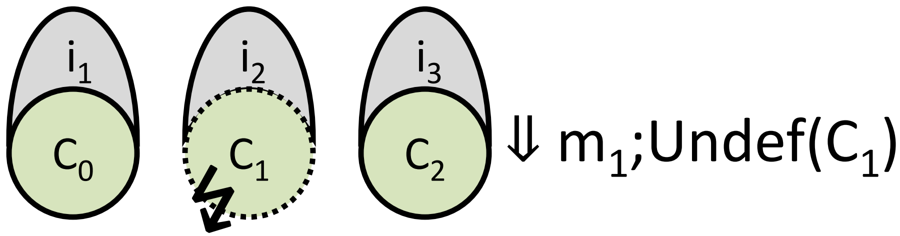
[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]

Dynamic compromise



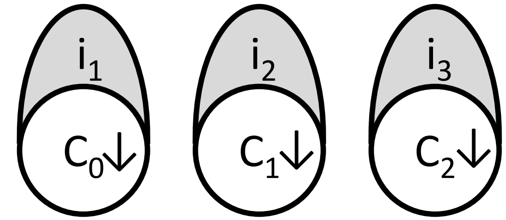
➡ ∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0)



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

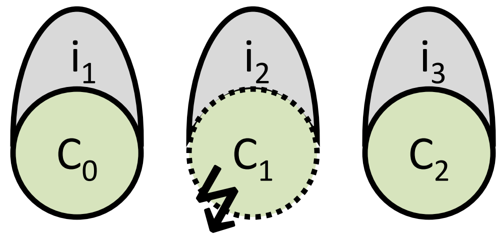
Dynamic compromise



$\Downarrow$ *t*

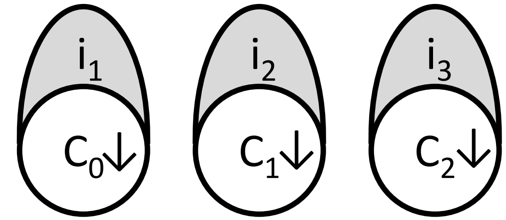
- ∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0)



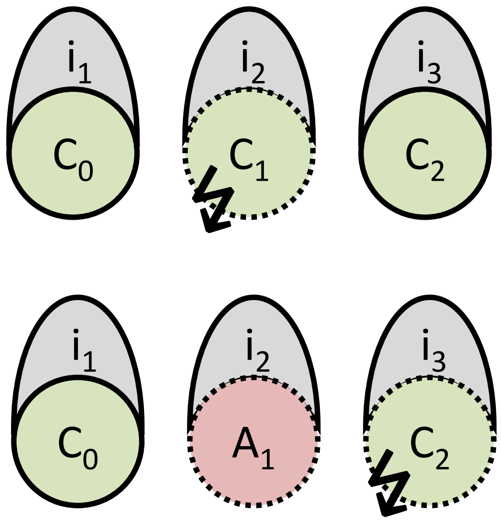
$\Downarrow$ m_1;Undef(C_1)

Dynamic compromise



$\Downarrow t$

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



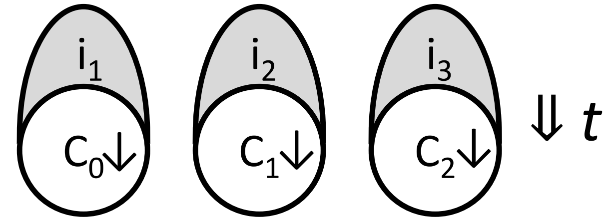
(0) $\Downarrow m_1;\mathrm{Undef}(C_1)$

$\leq$

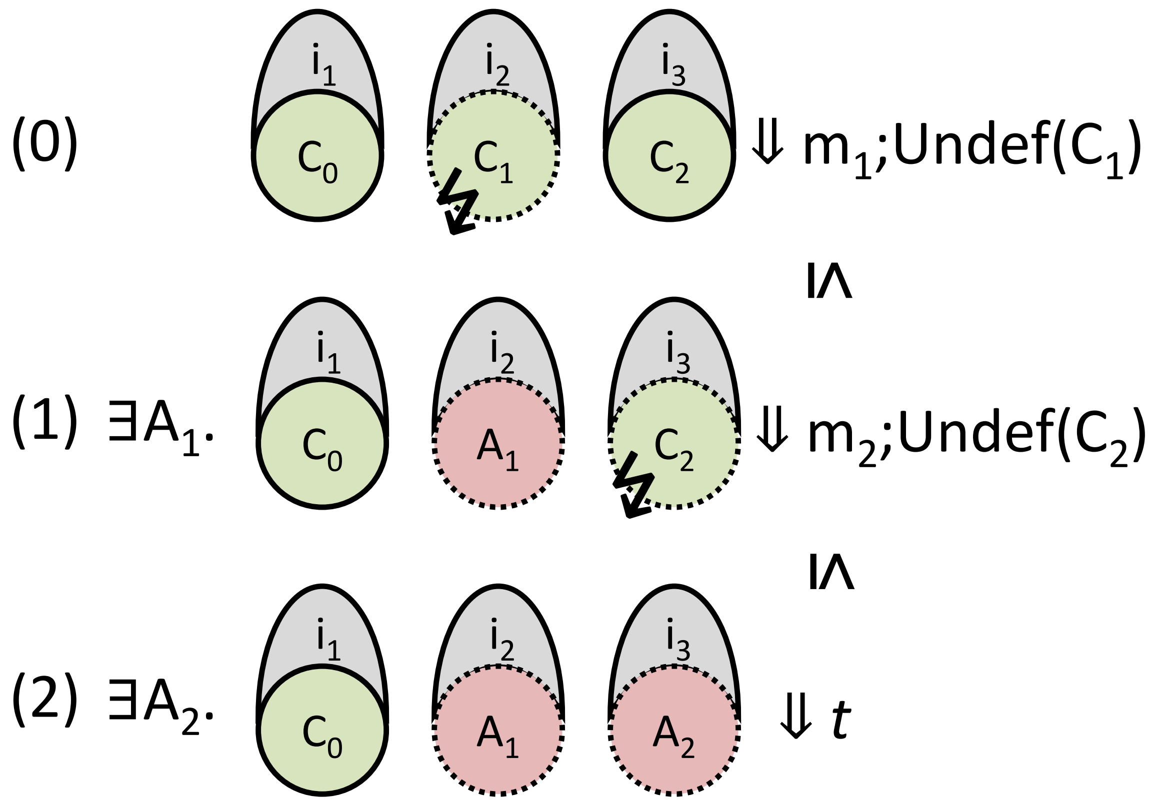
(1) $\exists A_1.$ $\Downarrow m_2;\mathrm{Undef}(C_2)$

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

Dynamic compromise

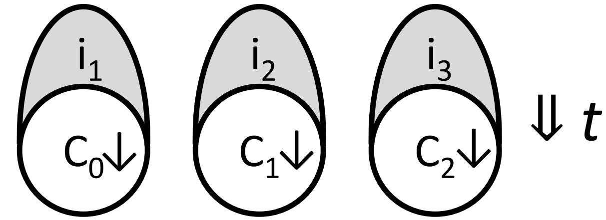


- ∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

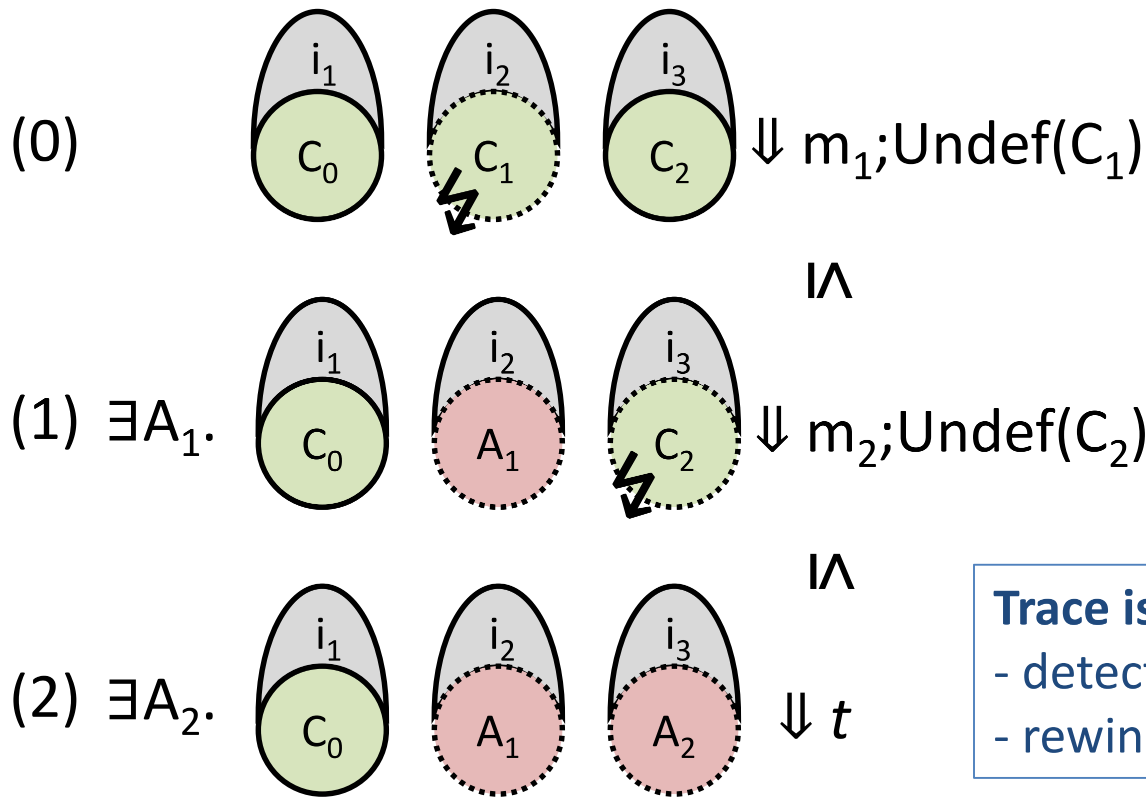


[When Good Components Go Bad - Fachini, Stronati, Hrițcu, et al]

Dynamic compromise



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



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 model

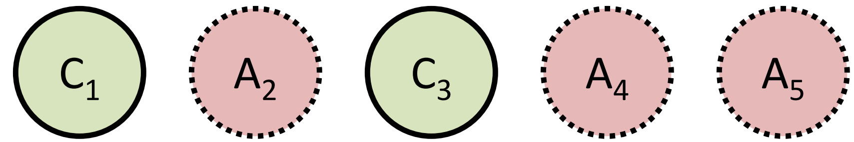
Restricting undefined behavior

- **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)

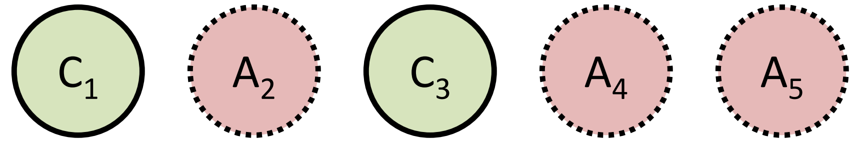
Mutual distrust



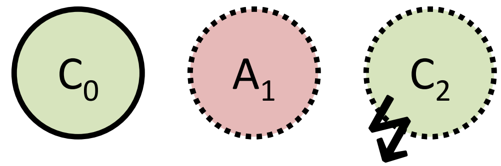
Now we know what these words mean!

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

Mutual distrust



Dynamic compromise

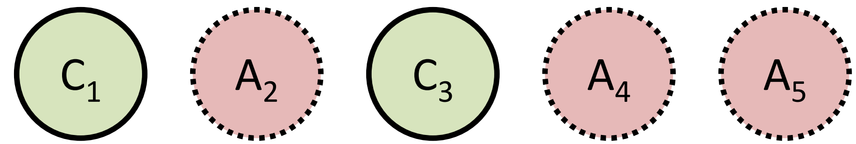


$\Downarrow m_2$; Undef(C_2)

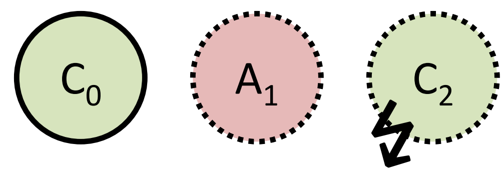
Now we know what these words mean!

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

Mutual distrust



Dynamic compromise

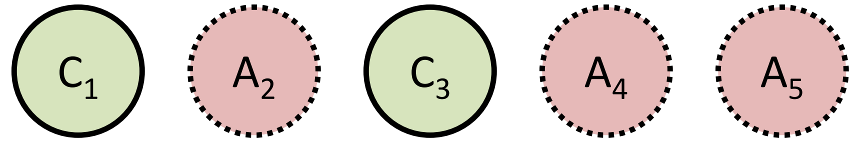


$\Downarrow m_2$; Undef(C_2)

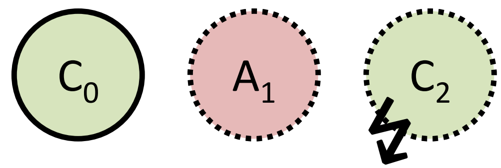
Now we know what these words mean!

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

Mutual distrust

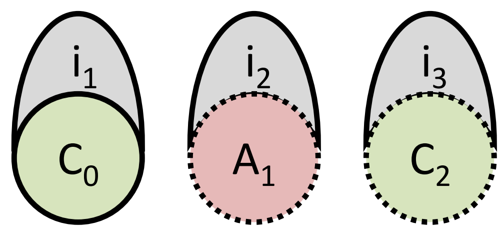


Dynamic compromise

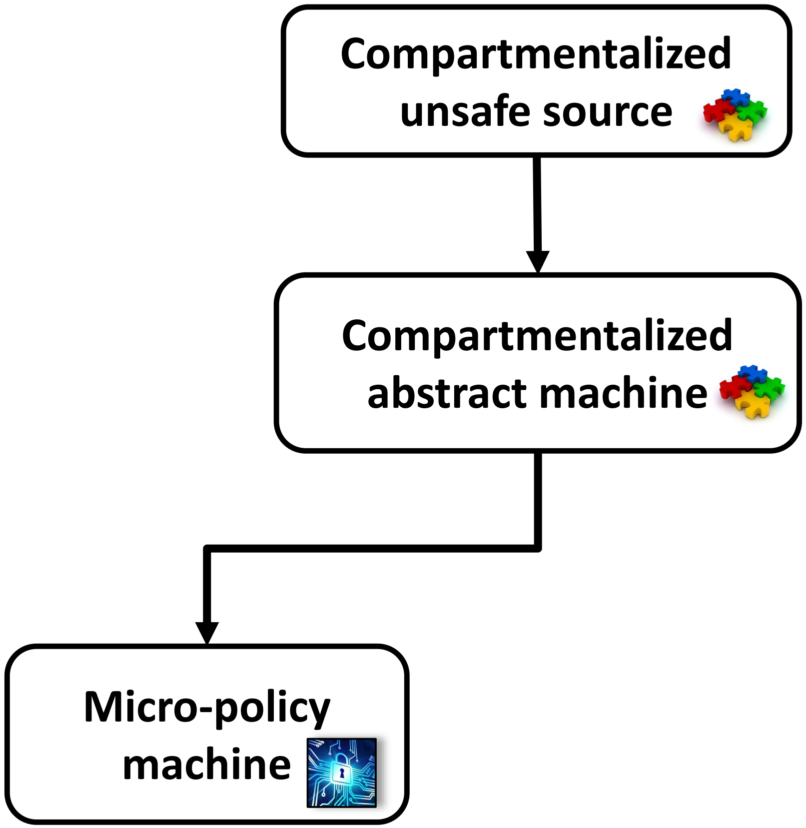


$\Downarrow m_2; \text{Undef}(C_2)$

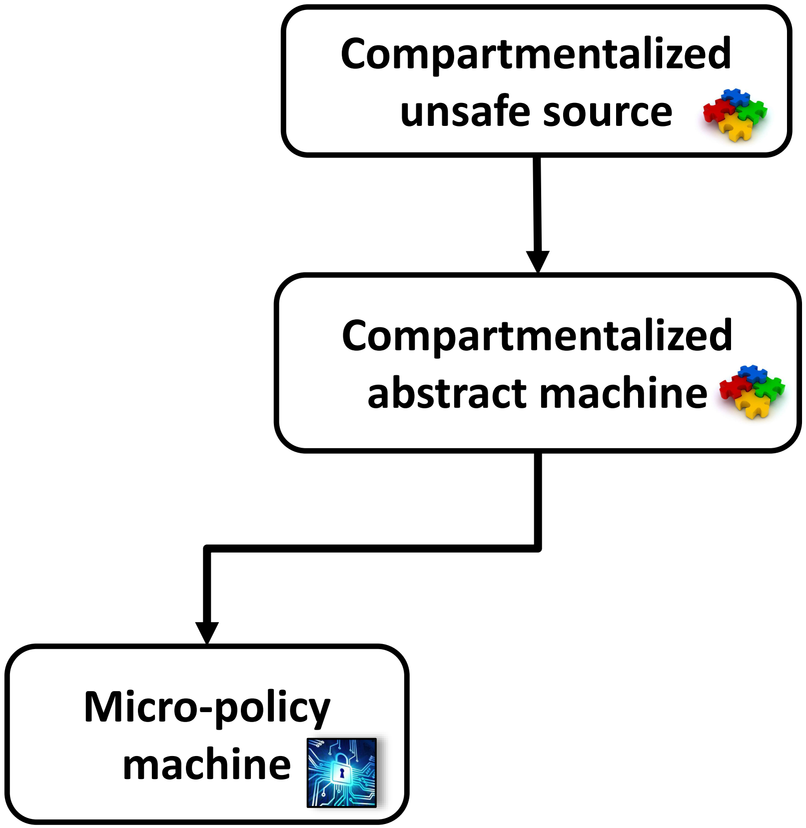
Static privilege



Towards Secure Compilation Chain

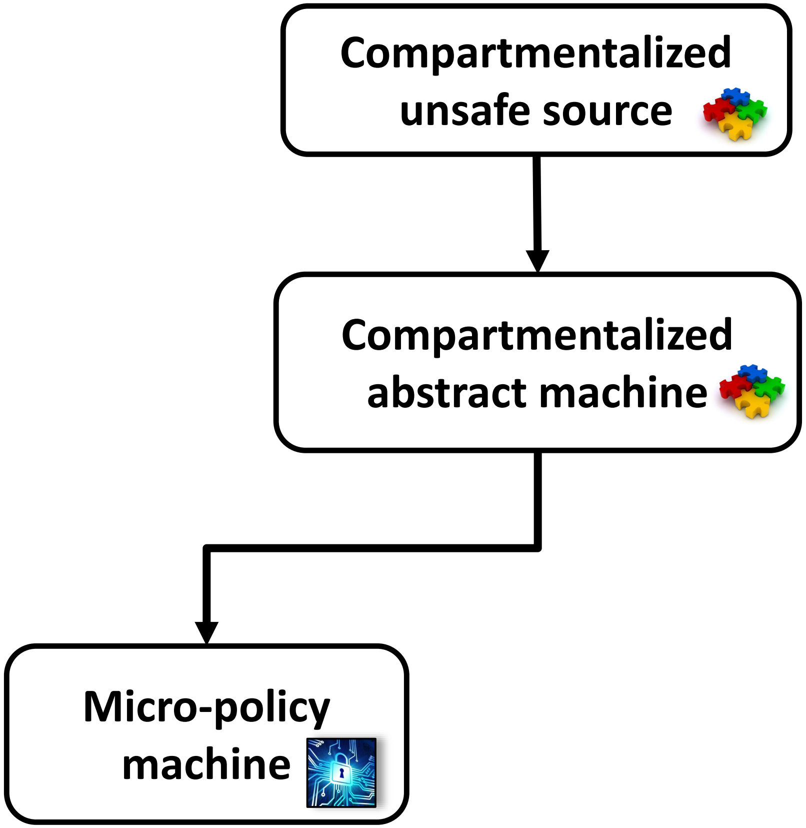


Towards Secure Compilation Chain



Buffers, procedures, components interacting via **strictly enforced interfaces**

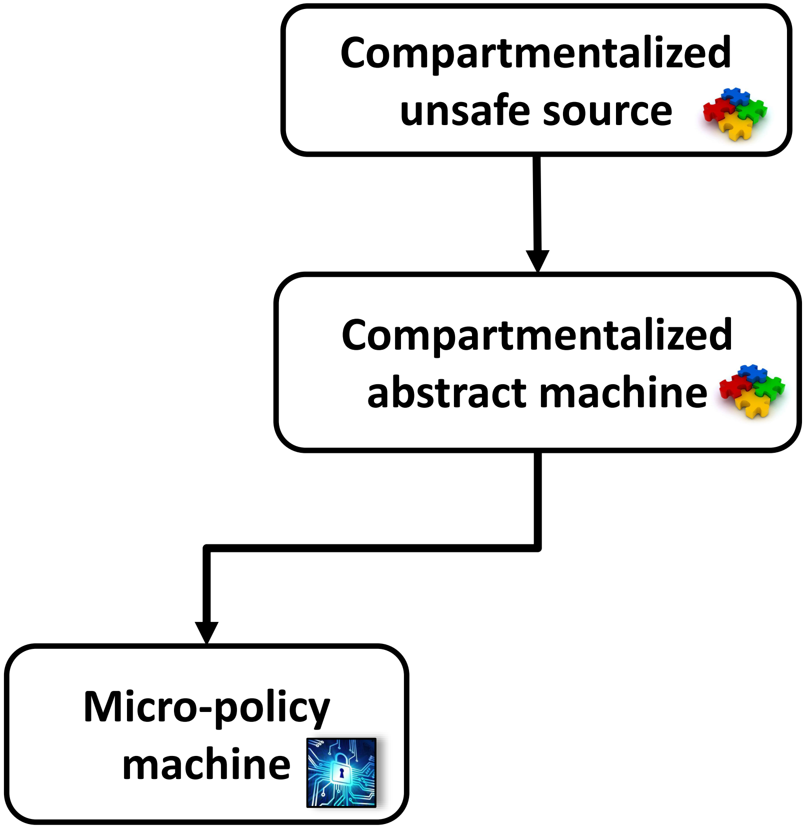
Towards Secure Compilation Chain



Buffers, procedures, components interacting via **strictly enforced interfaces**

Simple RISC abstract machine with **build-in compartmentalization**

Towards Secure Compilation Chain



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)

Towards Secure Compilation Chain

Compartmentalized unsafe source

Buffers, procedures, components interacting via **strictly enforced interfaces**

Compartmentalized 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**

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**

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)
- **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*)

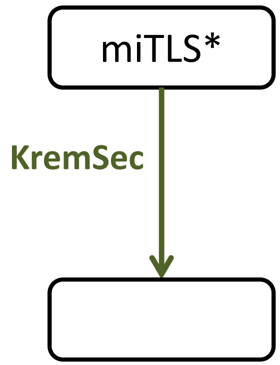
miTLS*

C language
+ components
+ memory safety

Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

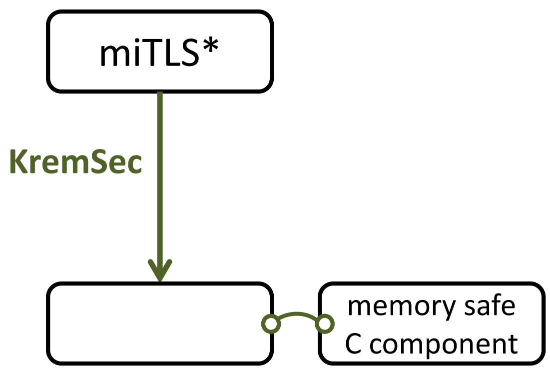
C language
+ components
+ memory safety



Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

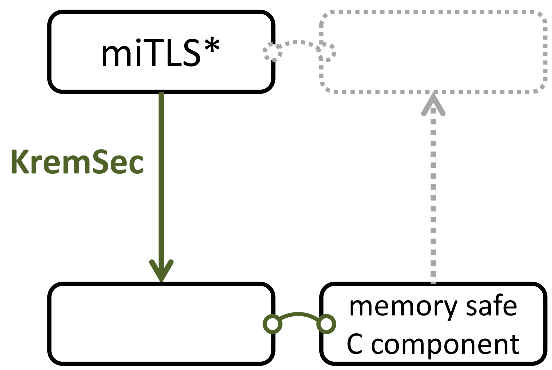
C language
+ components
+ memory safety



Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

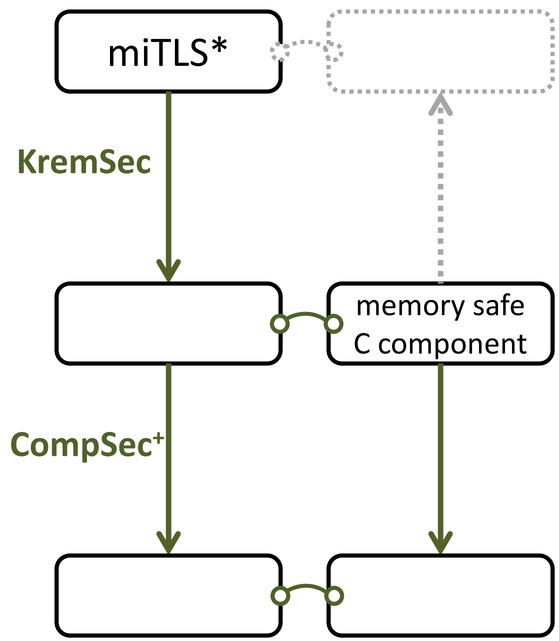


Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

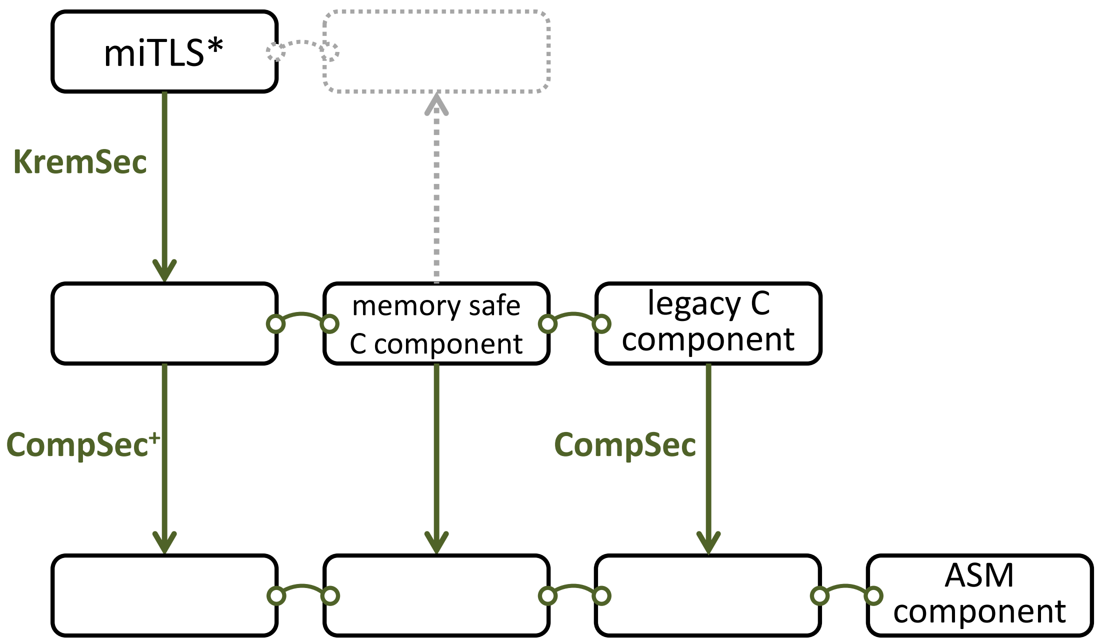


Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

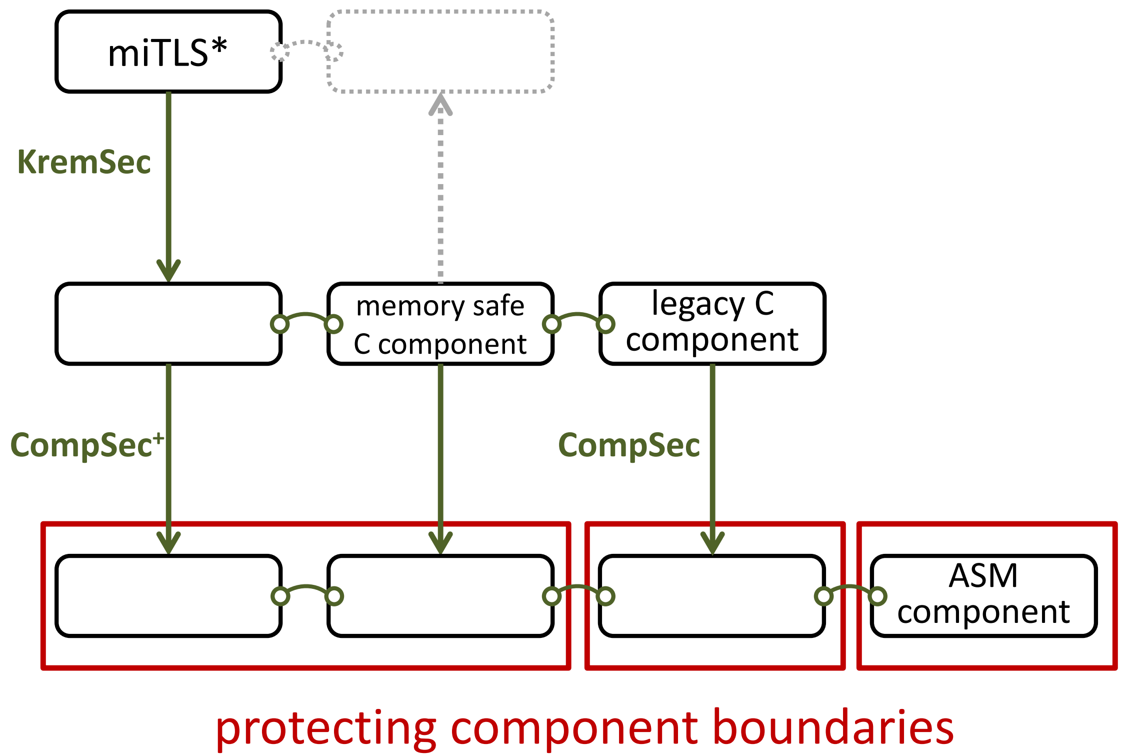


Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

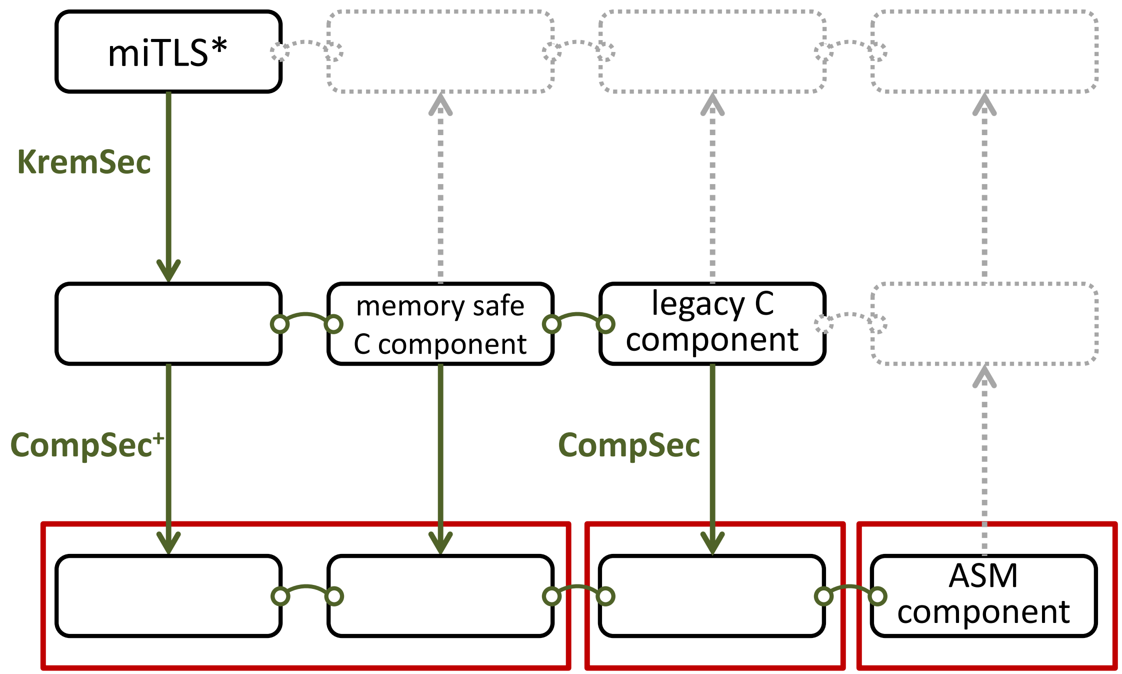


Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)



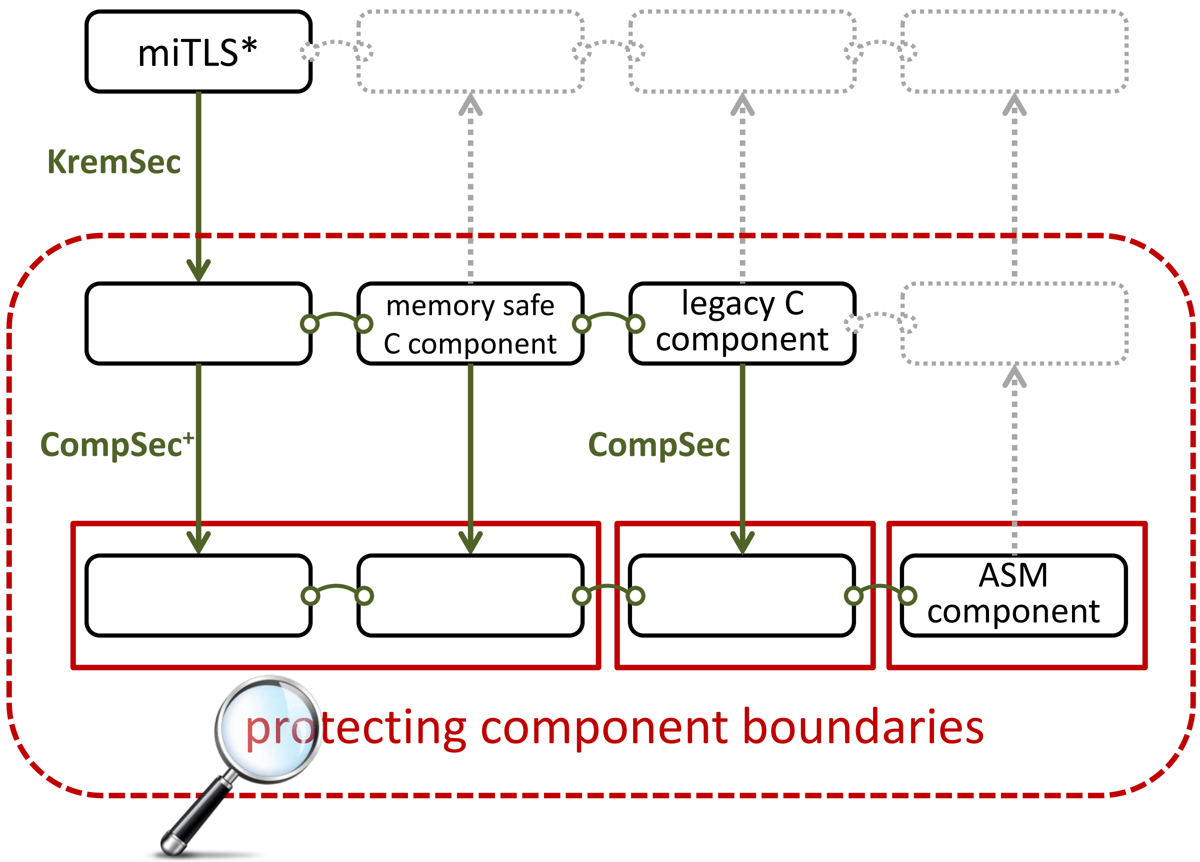
protecting component boundaries

Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

ASM language
(RISC-V + micro-policies)

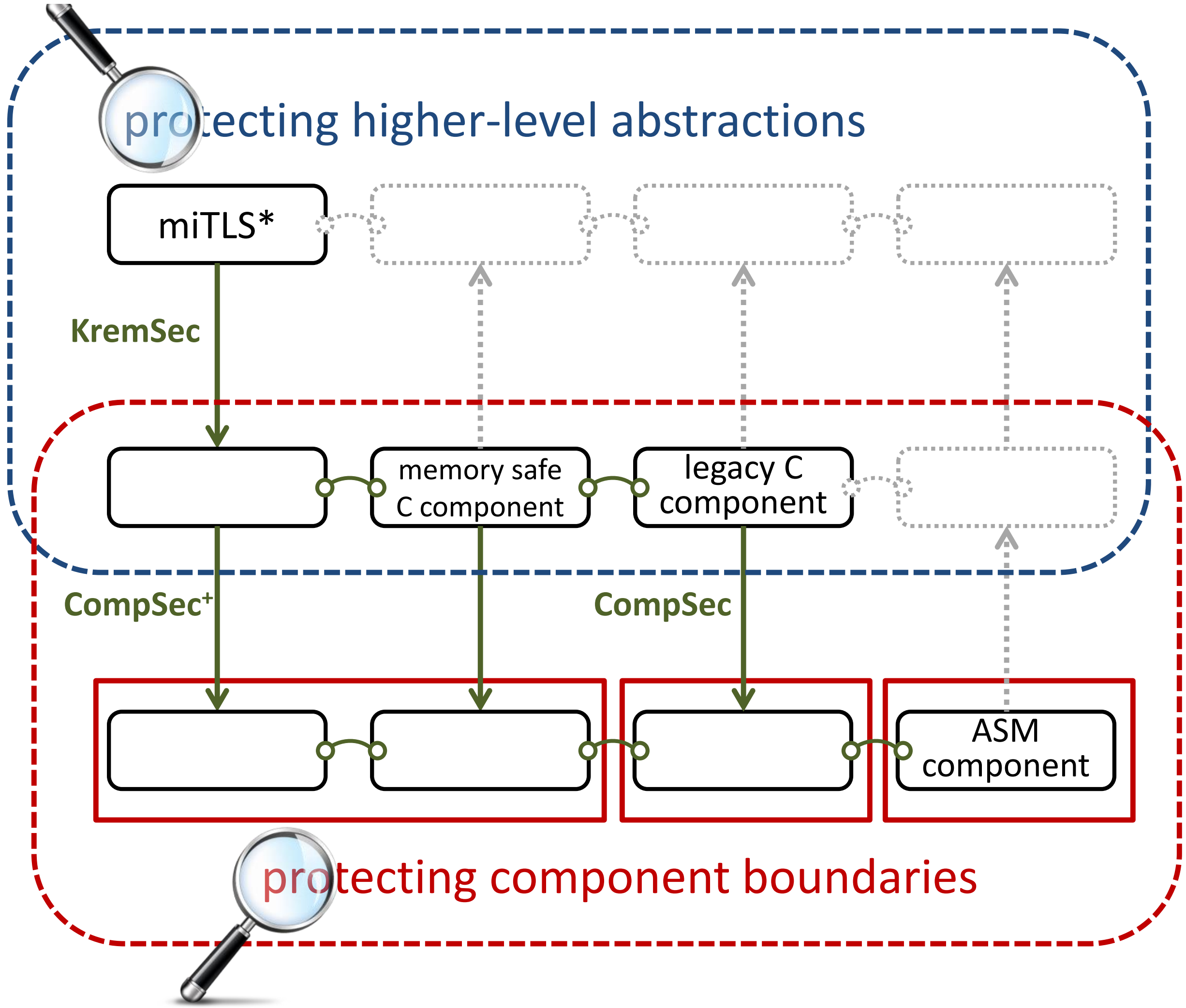


Goal: achieving secure compilation at scale

Low* language
(safe C subset in F*)

C language
+ components
+ memory safety

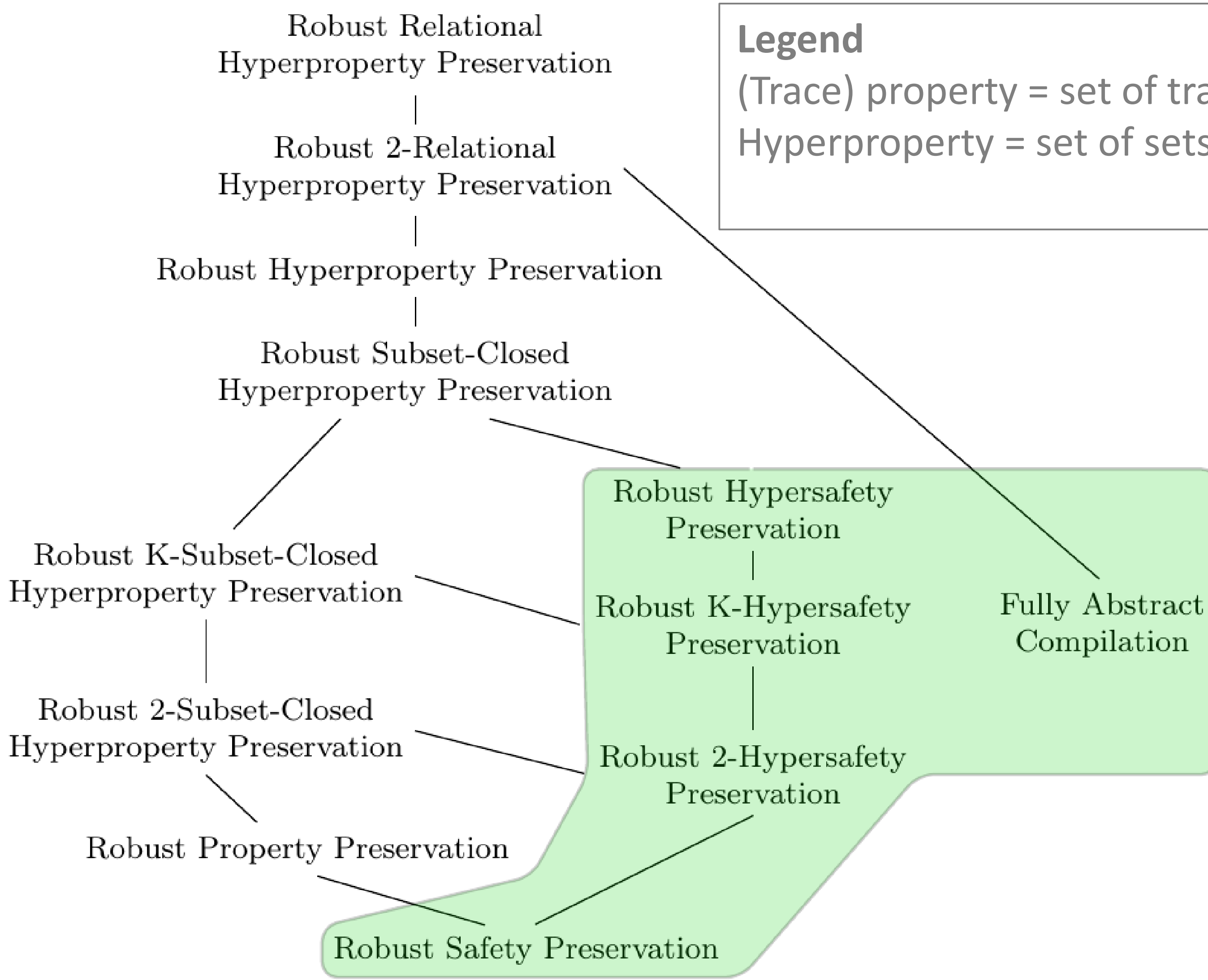
ASM language
(RISC-V + micro-policies)



Beyond robust safety preservation

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

Beyond robust safety preservation

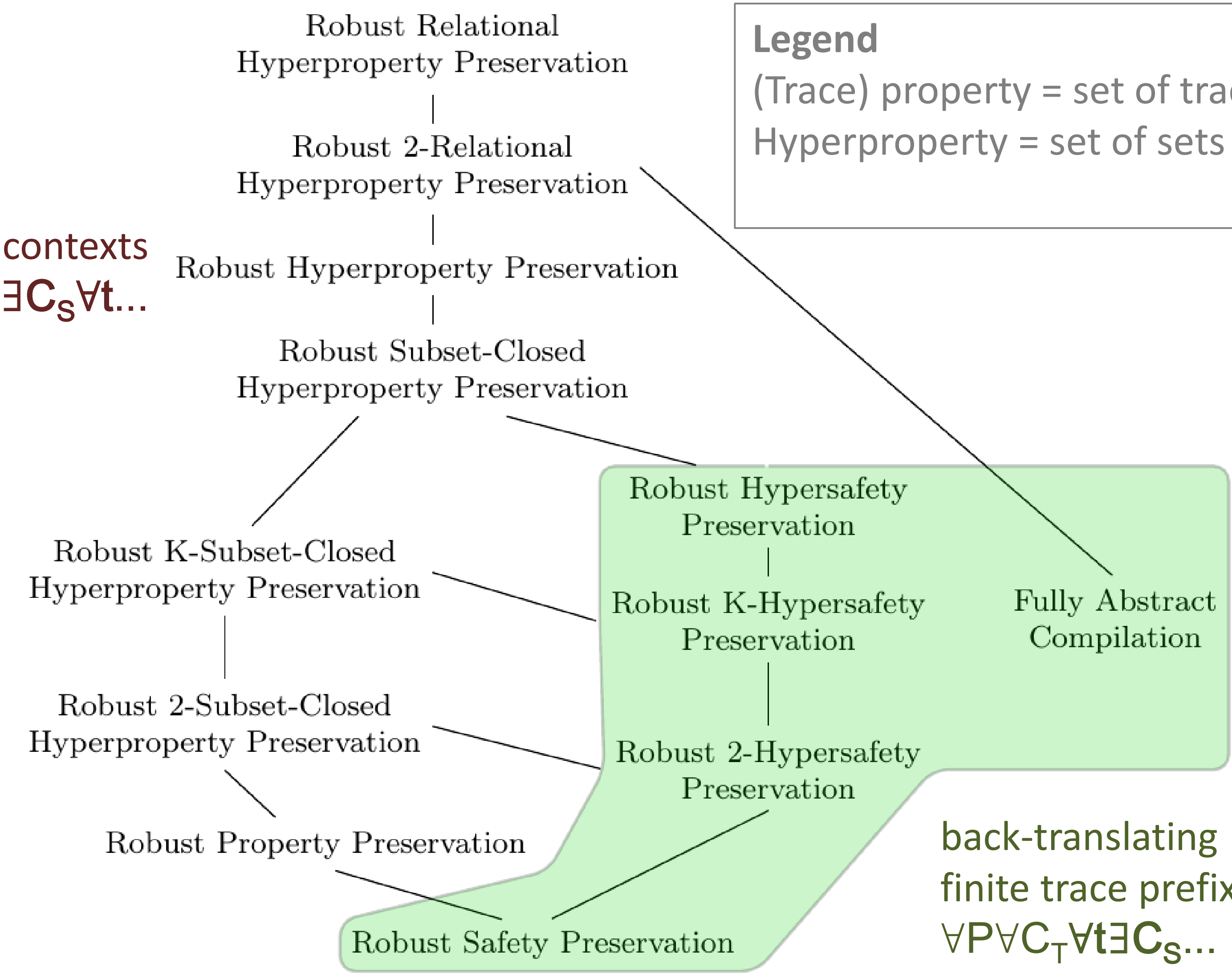


Legend

(Trace) property = set of traces

Hyperproperty = set of sets of traces

Beyond robust safety preservation



back-translating contexts
$\forall P \forall C_T \exists \mathbf{C_S} \forall \mathbf{t}$...

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

back-translating finite trace prefixes
$\forall P \forall C_T \forall \mathbf{t} \exists \mathbf{C_S}$...

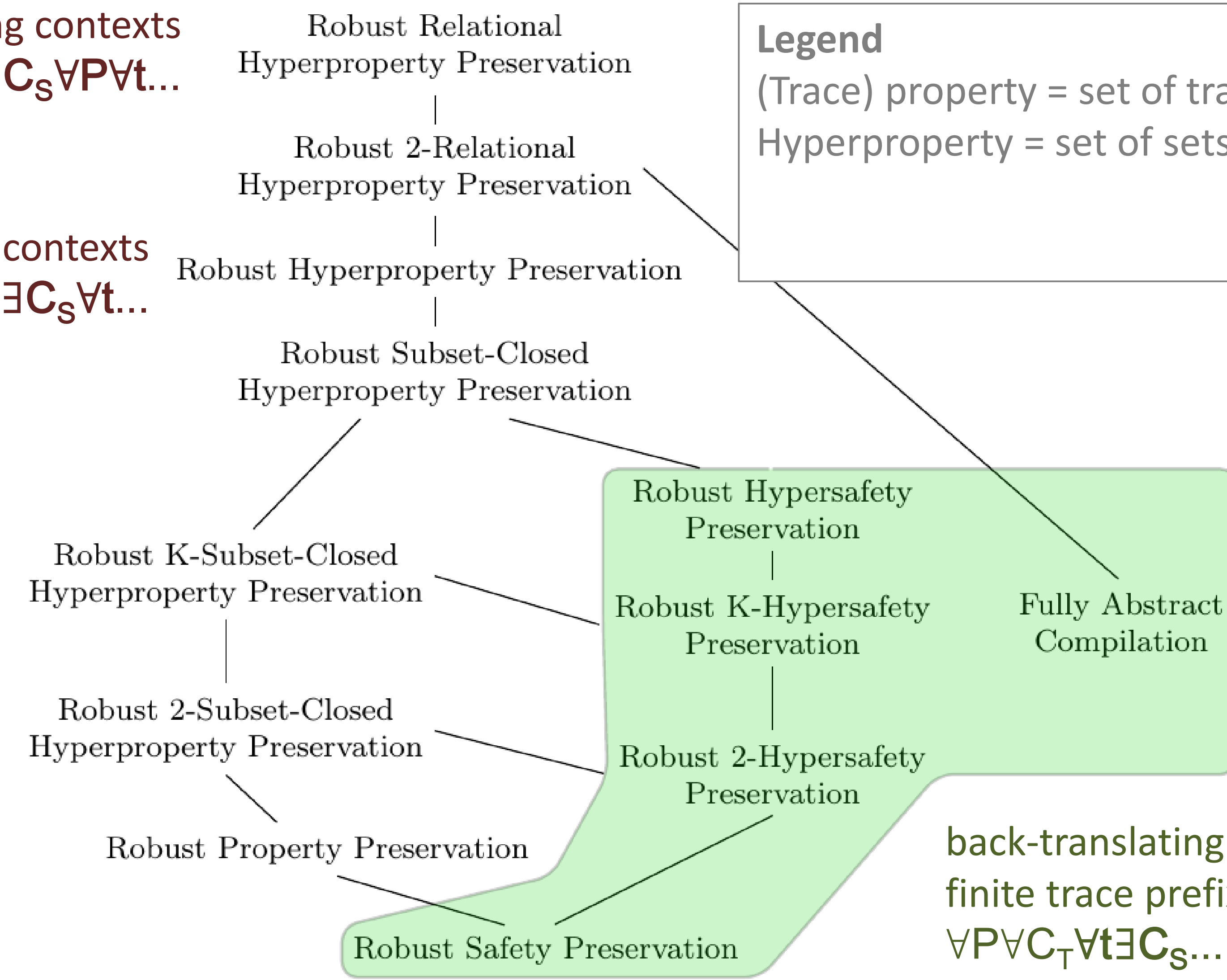
[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, et al]

Beyond robust safety preservation

back-translating contexts
$\forall C_T \exists C_S \forall P \forall t \ldots$

back-translating contexts
$\forall P \forall C_T \exists C_S \forall t \ldots$

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



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

[Robust Hyperproperty Preservation for Secure Compilation - Garg, Hrițcu, 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

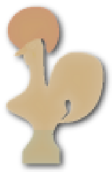
BACKUP SLIDES

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