# of Secure Compilation

Marco Patrignani[1,2]

Special thanks to:

Marco Guarnieri, Catalin Hritcu, Marco Vassena

for making their slides publicly available

07th March 2020

1 Stanford University   2 CISPA Helmholtz Center for Information Security

HACL* verified cryptographic library, in practice

~100.000 LOC in F*

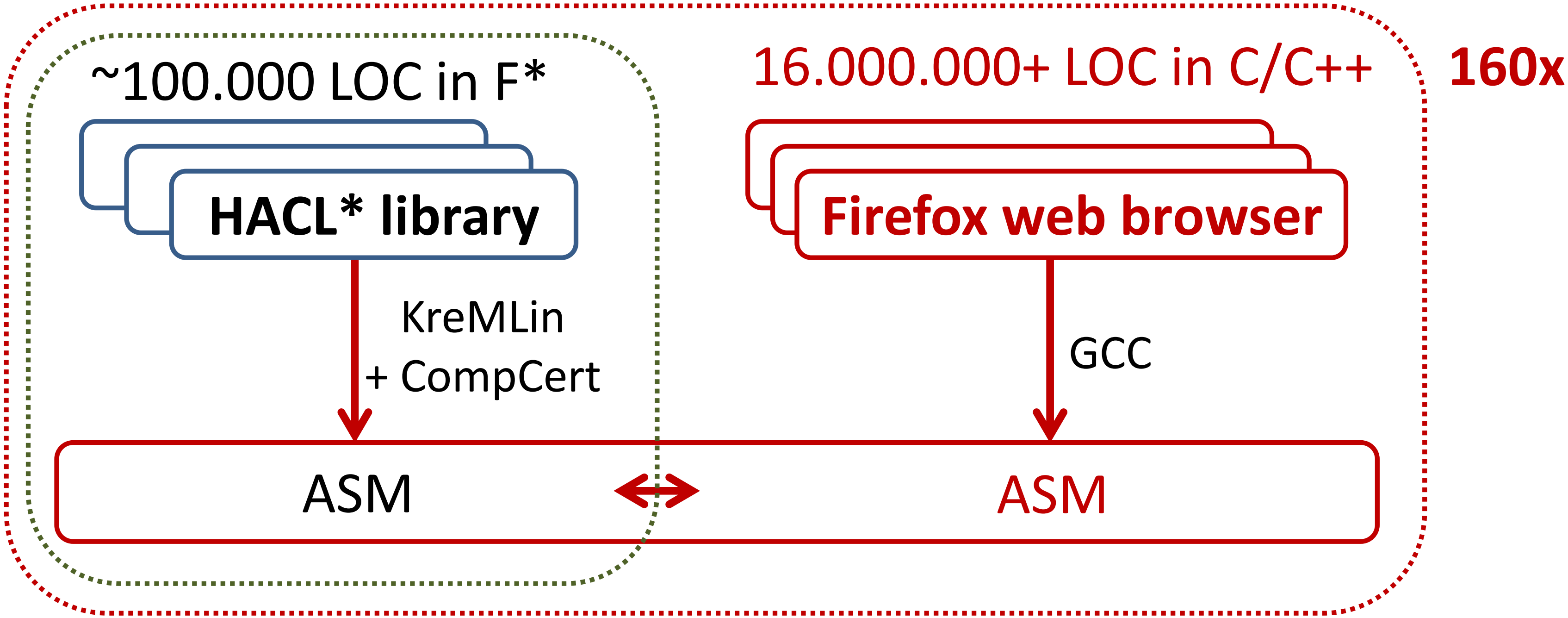
HACL* library

16.000.000+ LOC in C/C++

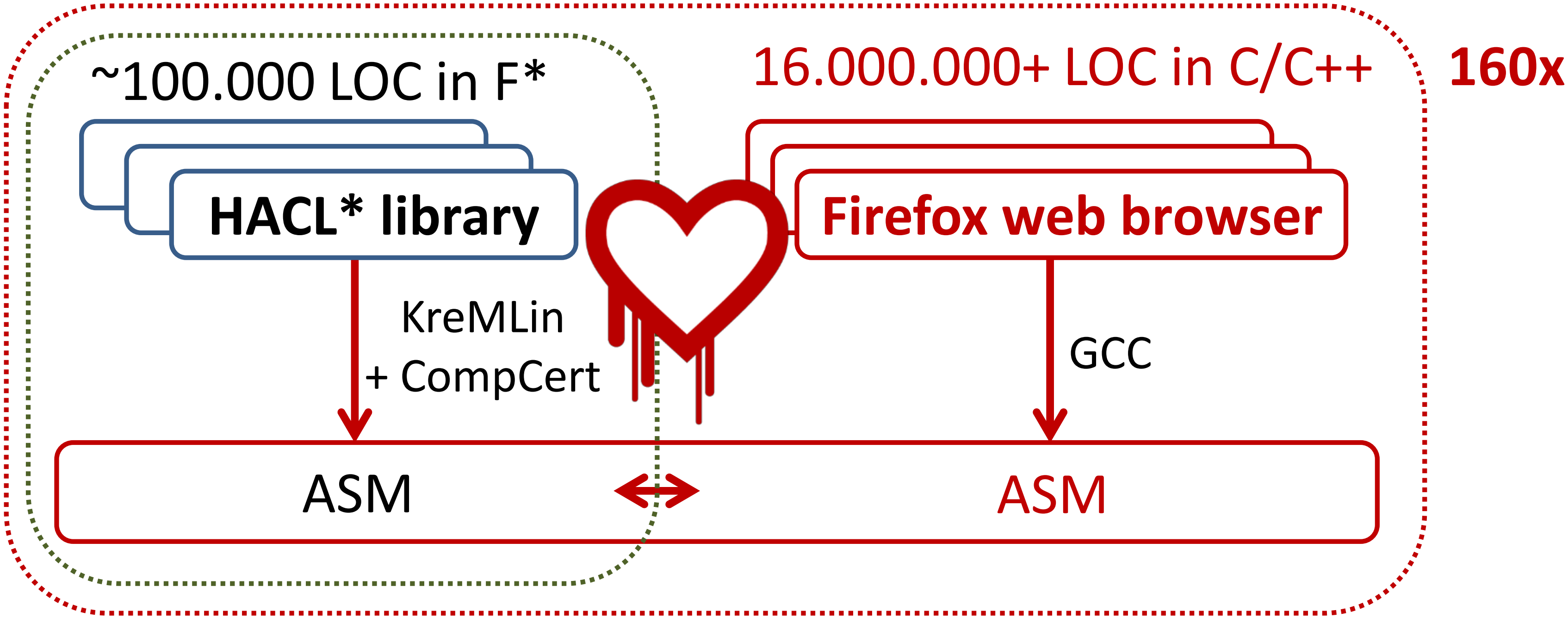
160x

Firefox web browser

HACL* verified cryptographic library, in practice



HACL* verified cryptographic library, in practice

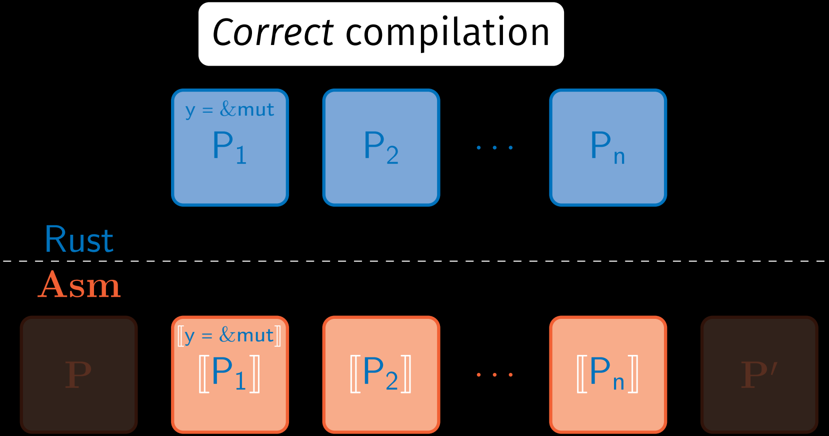


Insecure interoperability: **linked code can read and write data and code, jump to arbitrary instructions, smash the stack, ...**

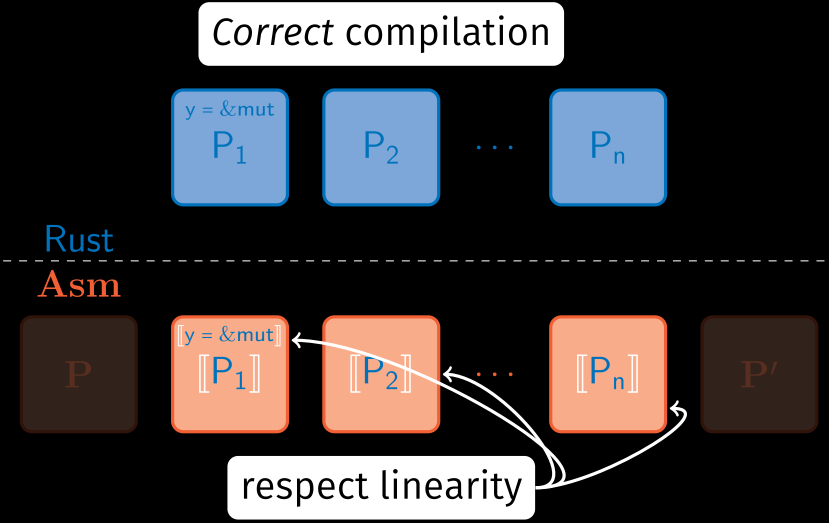
We need secure compilation chains

- **Protect source-level abstractions even against linked adversarial low-level code**
  - **various enforcement mechanisms**: processes, SFI, ...
  - shared responsibility: compiler, linker, loader, OS, HW
- **Goal: enable source-level security reasoning**
  - **linked adversarial target code cannot break the security of compiled program** any more than some linked source code
  - **no "low-level" attacks**

What is Secure Compilation?

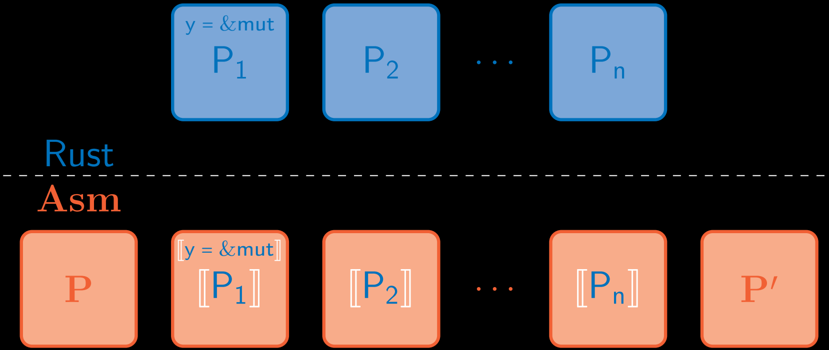


What is Secure Compilation?

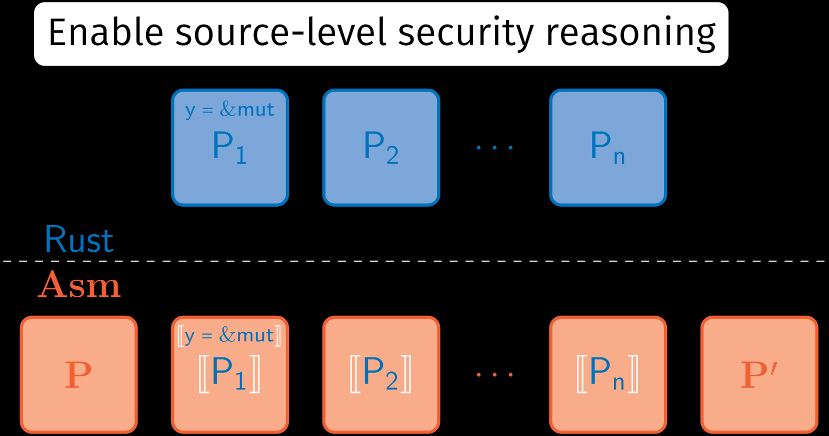


What is Secure Compilation?

Secure compilation



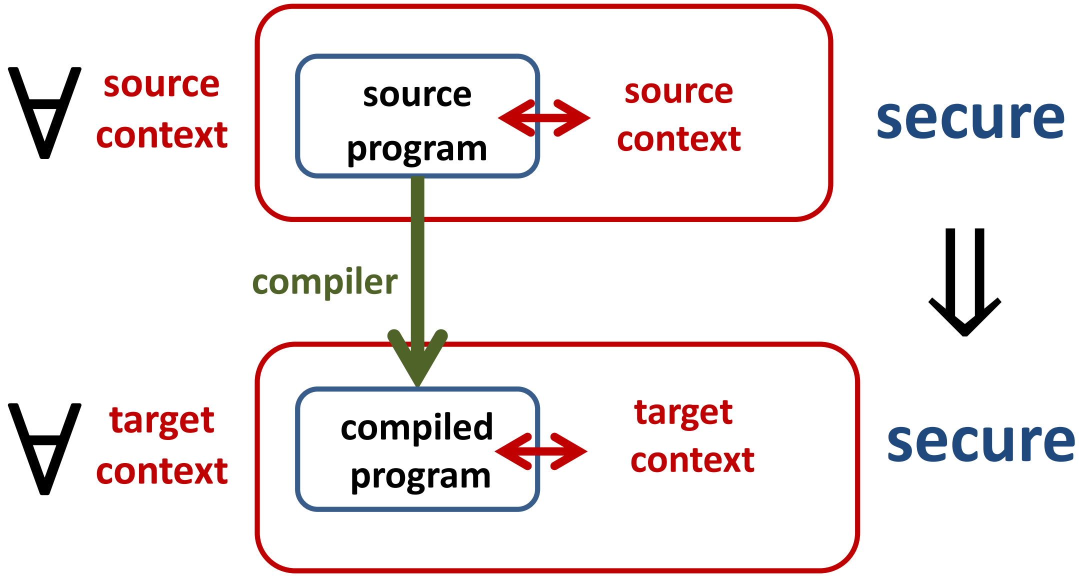
What is Secure Compilation?



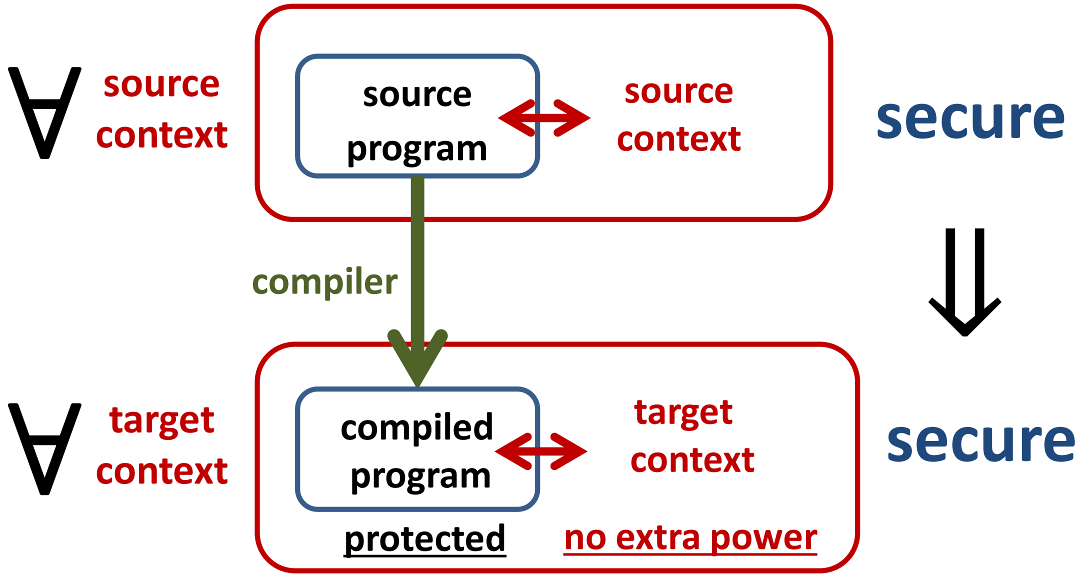
Robustly preserving security

∀ source context [source program ↔ source context] secure

Robustly preserving security



Robustly preserving security



But what should "secure" mean?

What properties should we robustly preserve?

The Origins of the Secure Compiler

Fully Abstract Compilation (FAC)

Theorem 1 *The compositional translation is fully-abstract, up to observational equivalence: for all join-calculus processes P and Q,*

$$P \approx Q \quad \textit{if and only if} \quad \mathcal{E}nv\,[\,[\![P]\!]\,] \approx \mathcal{E}nv\,[\,[\![Q]\!]\,]$$

Fully Abstract Compilation Influence

What properties should we robustly preserve?

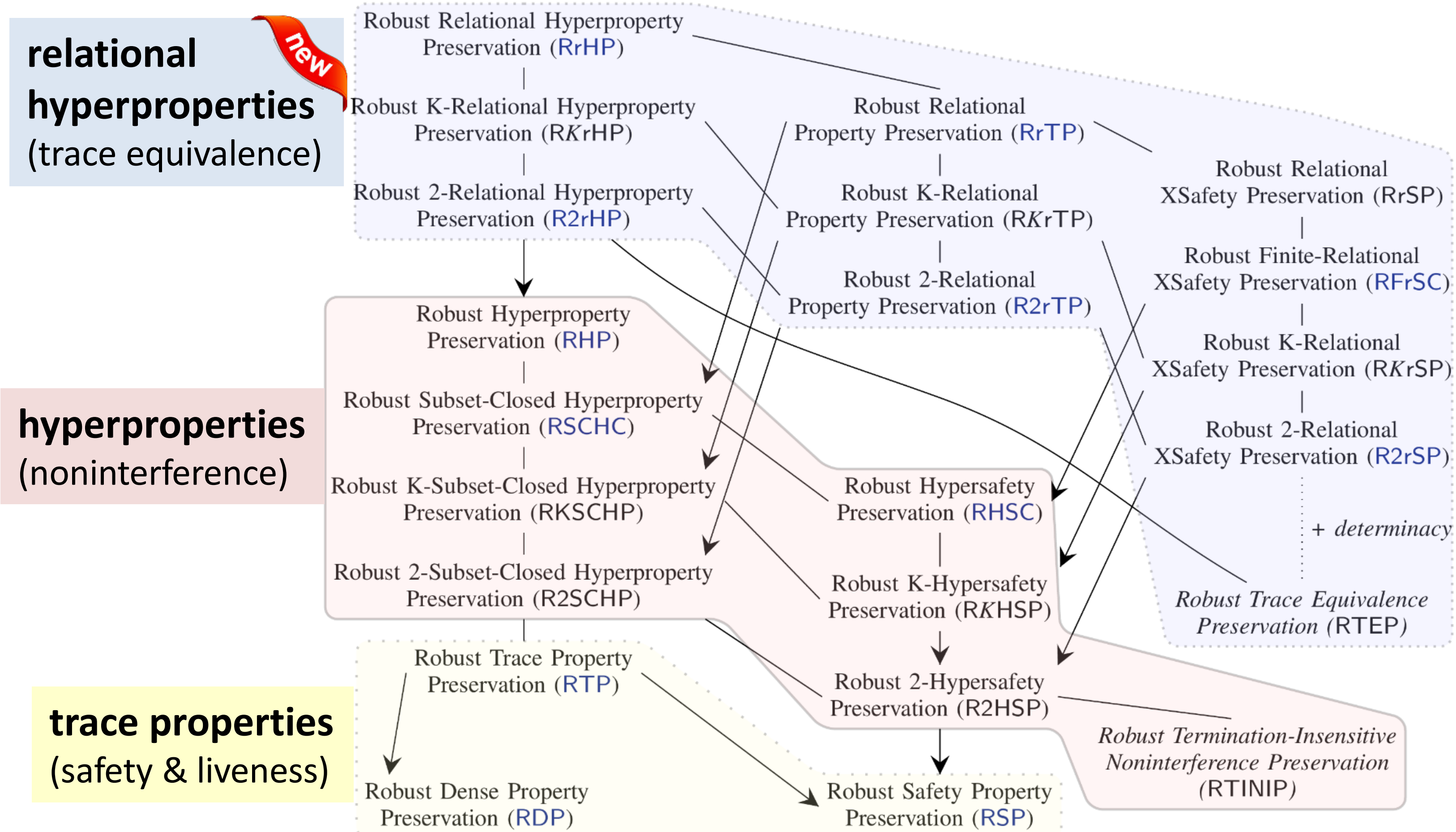
relational hyperproperties
(trace equivalence)

new

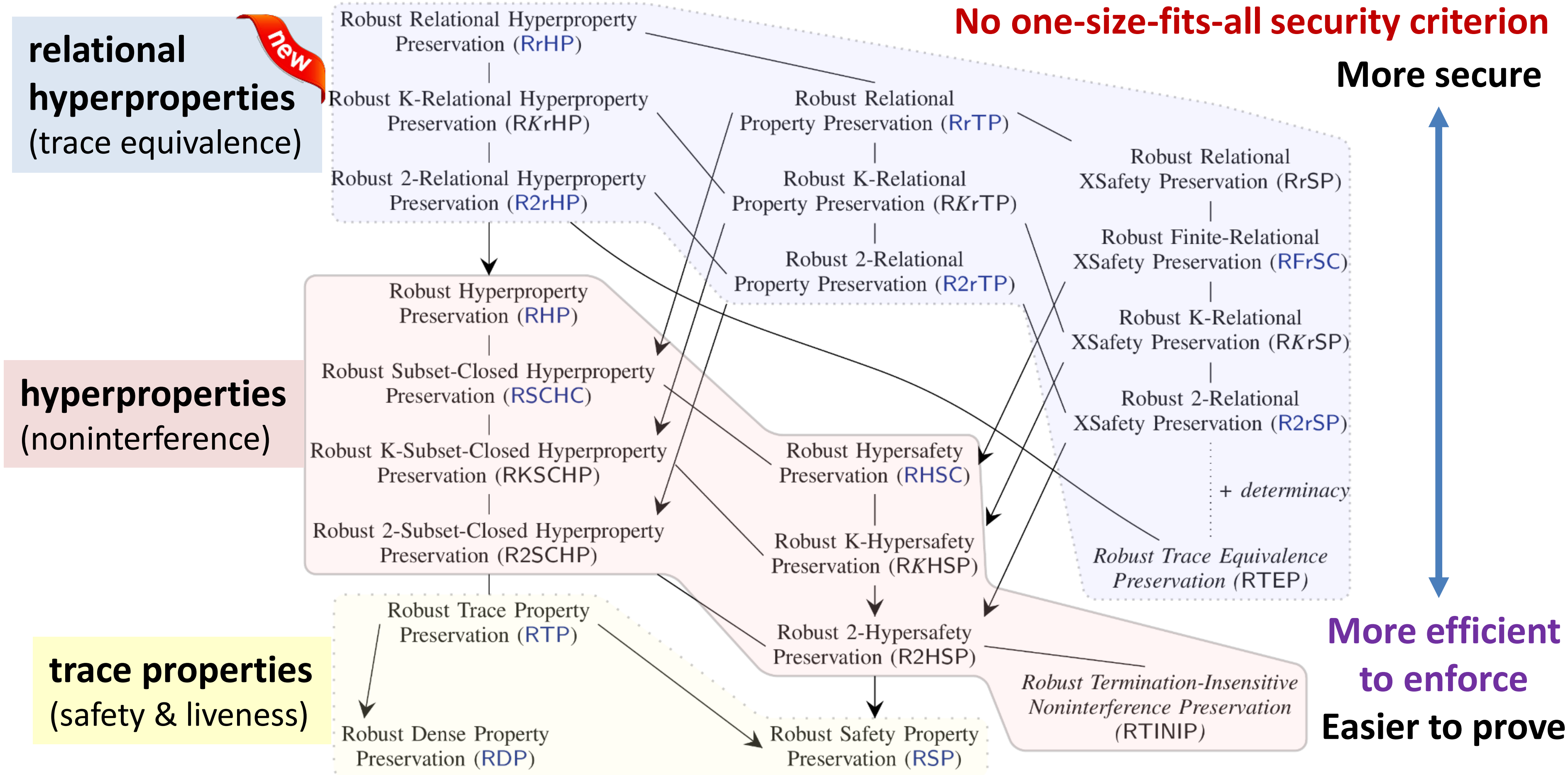
hyperproperties
(noninterference)

trace properties
(safety & liveness)

What properties should we robustly preserve?



What properties should we robustly preserve?



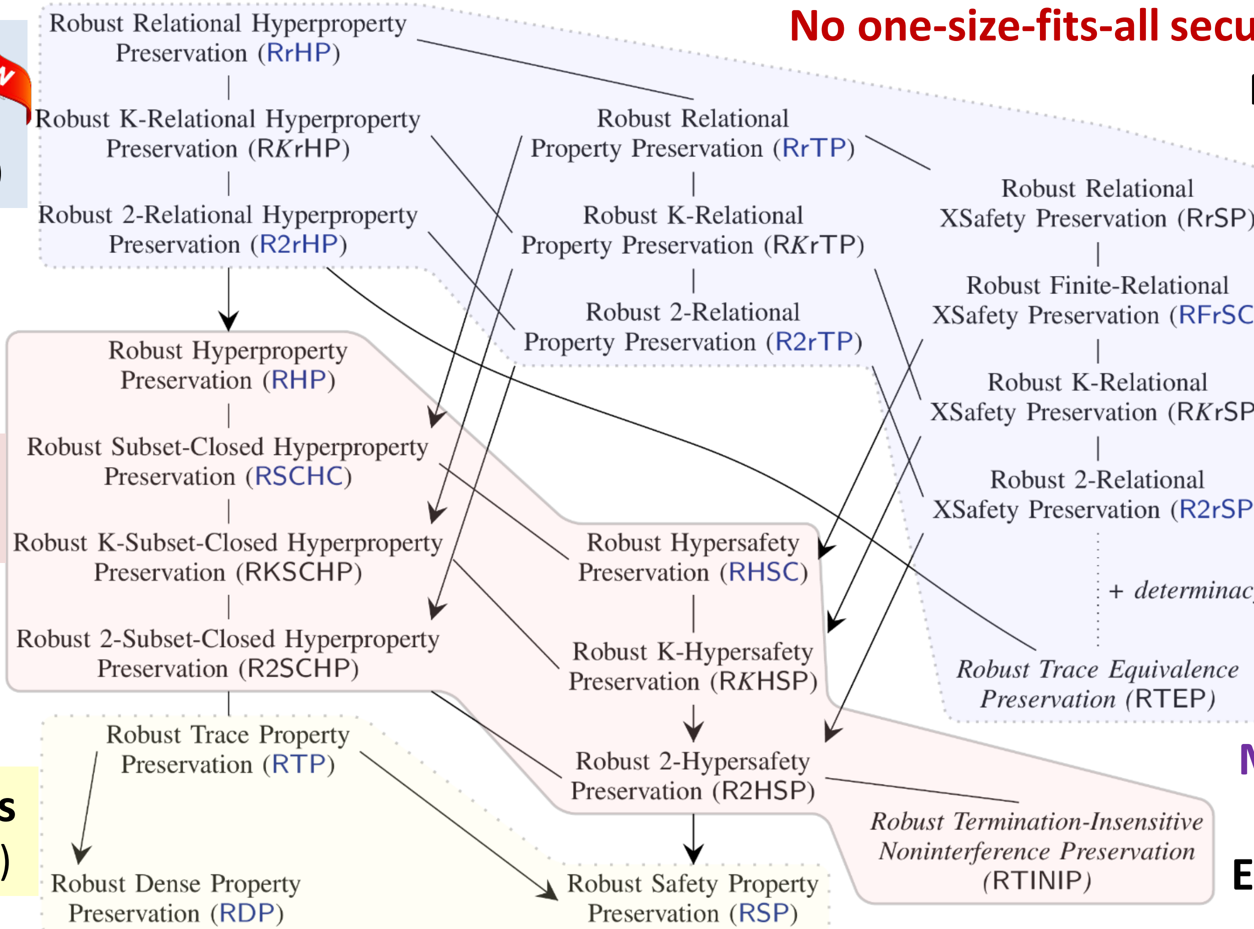
What properties should we robustly preserve?

relational hyperproperties new
(trace equivalence)

hyperproperties
(noninterference)

trace properties
(safety & liveness)

only integrity



No one-size-fits-all security criterion

More secure

More efficient to enforce

Easier to prove

What properties should we robustly preserve?

No one-size-fits-all security criterion

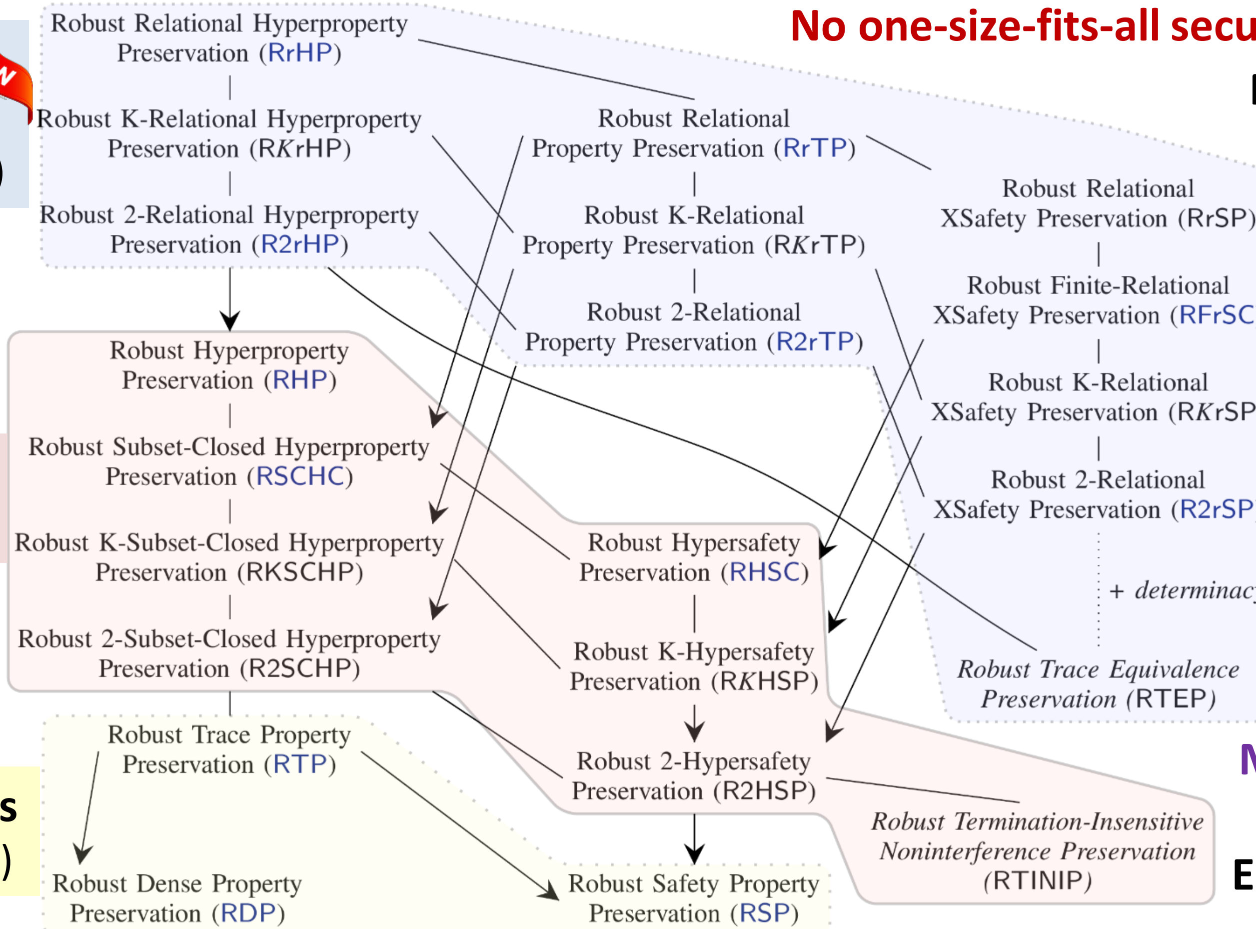
relational hyperproperties new
(trace equivalence)

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



More secure

More efficient to enforce
Easier to prove

What properties should we robustly preserve?

relational hyperproperties
(trace equivalence)

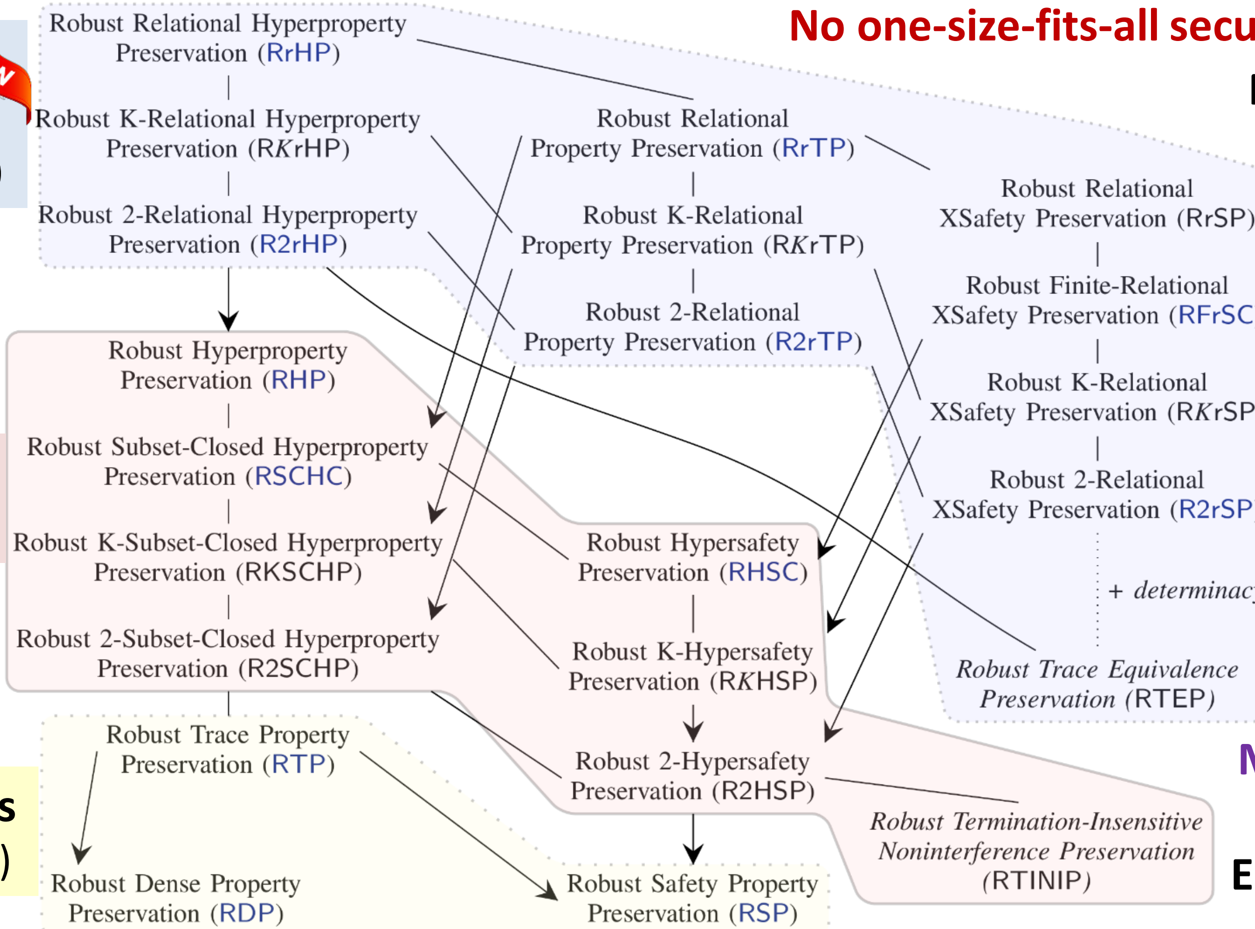
+ code confidentiality

hyperproperties
(noninterference)

+ data confidentiality

trace properties
(safety & liveness)

only integrity



No one-size-fits-all security criterion

More secure

More efficient to enforce

Easier to prove

Robust Safety

- for a safety property (M)
- no matter what we link against ($\forall A, \overline{\alpha}$)
- our program behaves in a way (if $A\,[P] \xrightarrow{\overline{\alpha}}$)
- that respects that safety property (then $M \vdash \overline{\alpha}$)

robust safety formally

$M \vdash P$

RSC and PF-RSC

RSC: given $\mathsf{M} \approx \mathbf{M}$ if $\mathsf{M} \vdash \mathsf{P}$ then $\mathbf{M} \vdash [\![\mathsf{P}]\!]$ $\iff$ PF-RSC: if $\forall \mathbf{A}.\mathbf{A}\left[[\![\mathsf{P}]\!]\right] \xrightarrow{\overline{\alpha}}$ then $\exists \mathsf{A}.\mathsf{A}\left[\mathsf{P}\right] \xrightarrow{\overline{\alpha}}$ and $\overline{\alpha} \approx \overline{\alpha}$

- $\iff$ must be proven (when needed)
- proof is (generally) trivial
- sanity-check for cross-language safety encoding ($\mathsf{M} \approx \mathbf{M}$)

Backtranslation: Build A From $\mathbf{A}$ or $\overline{\alpha}$

HP: P is RS

newBlk(c)
addBlk(b)
verifyCh()

+

lib_1 lib_2 lib_3

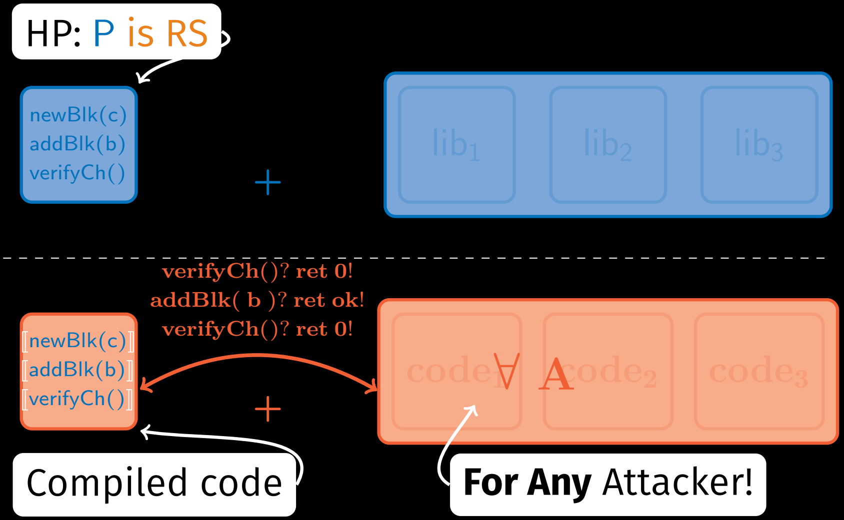
$[\![\mathrm{newBlk(c)}]\!]$
$[\![\mathrm{addBlk(b)}]\!]$
$[\![\mathrm{verifyCh()}]\!]$

+

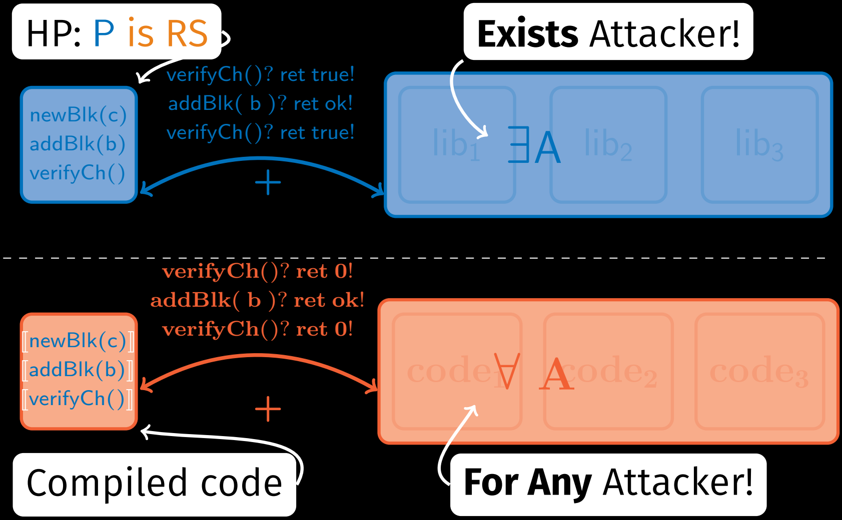
$\mathbf{code}_1$ $\mathbf{code}_2$ $\mathbf{code}_3$

Compiled code

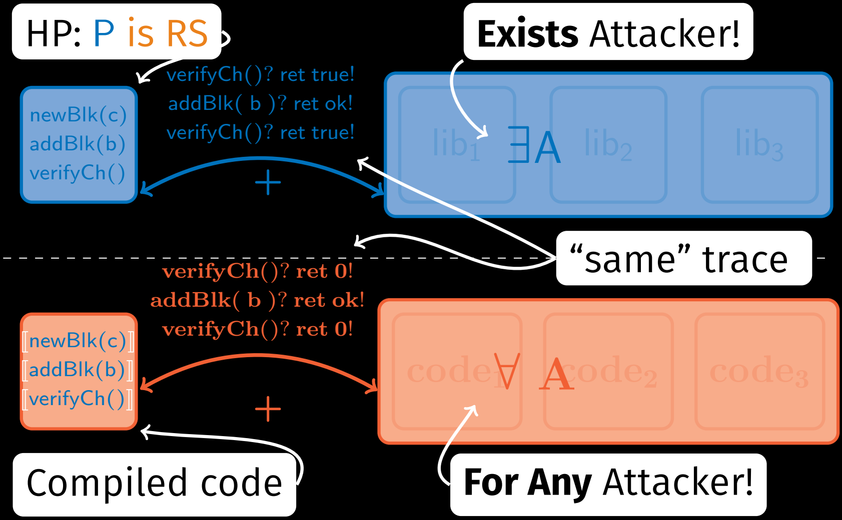
Backtranslation: Build A From A or $\overline{\alpha}$



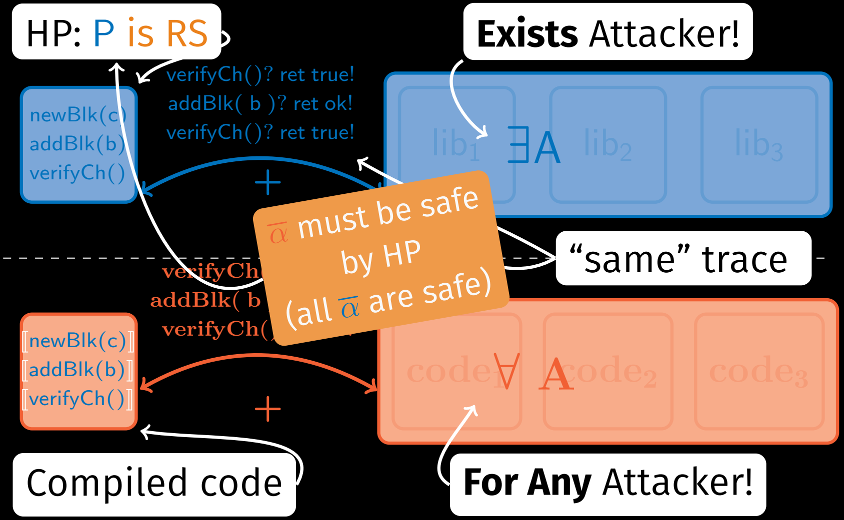
Backtranslation: Build A From A or $\overline{\alpha}$



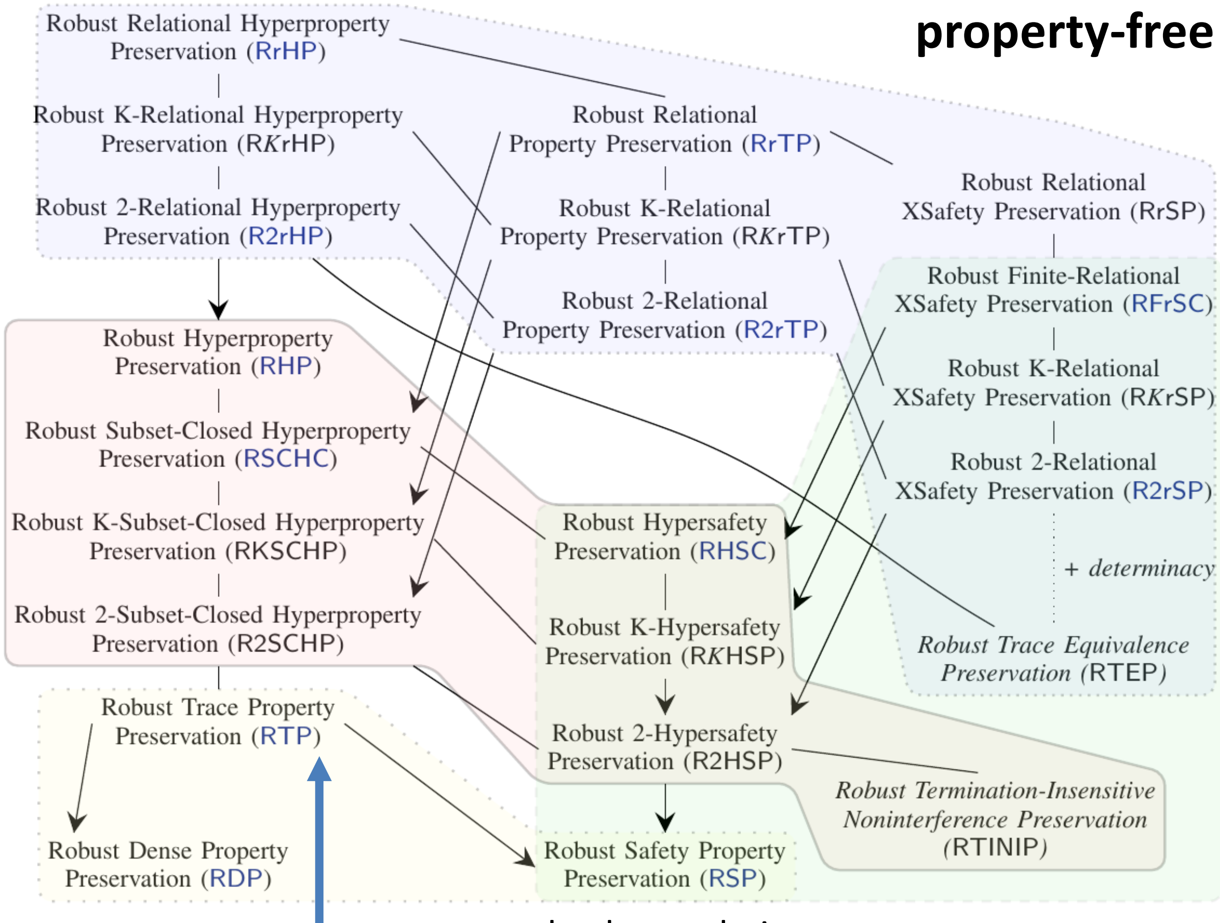
Backtranslation: Build A From A or $\overline{\alpha}$



Backtranslation: Build A From A or $\overline{\alpha}$



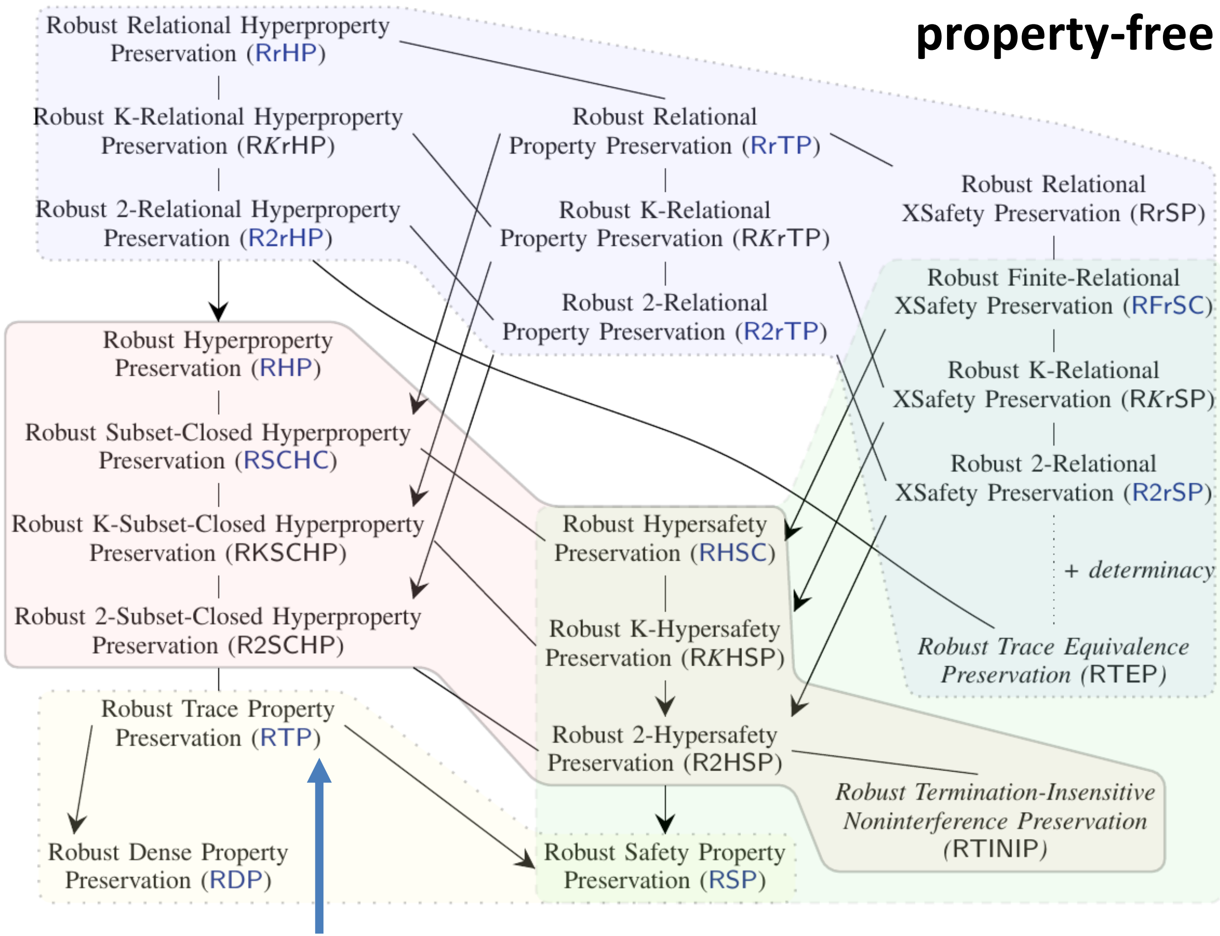
Some of the proof difficulty is manifest in property-free characterization



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

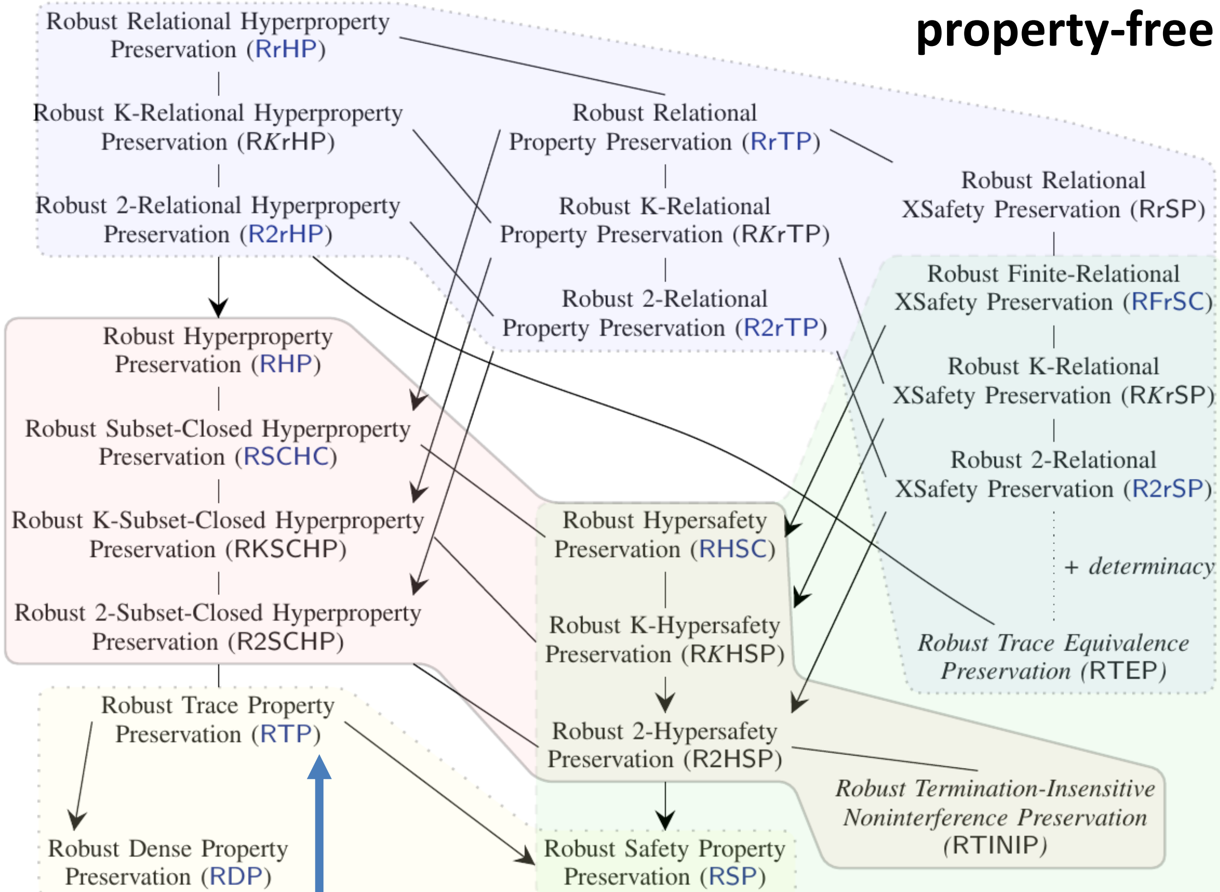
back-translating
finite trace prefix
$\forall P \forall C_T \forall m \leq t \exists C_S \ldots$

Some of the proof difficulty is manifest in property-free characterization



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

Some of the proof difficulty is manifest in property-free characterization



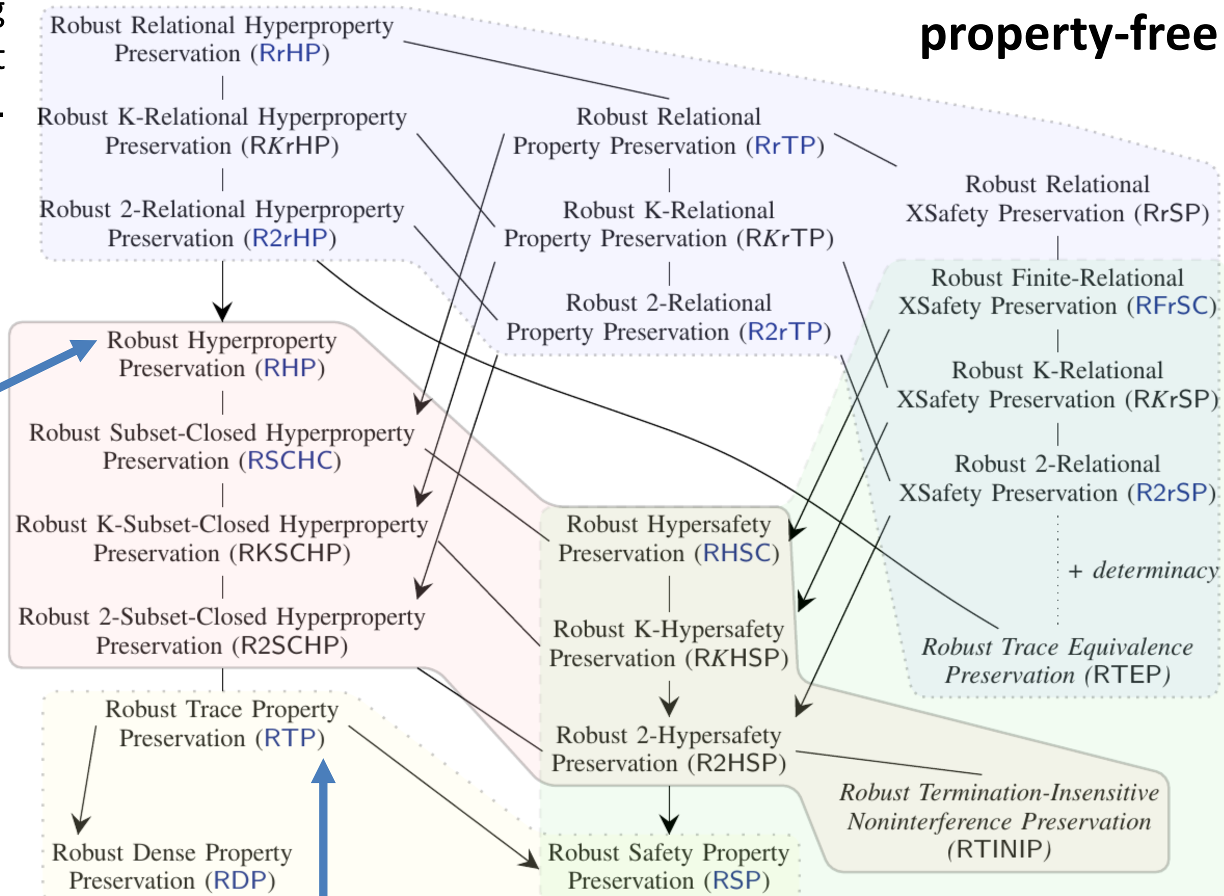
back-translating finite set of finite trace prefixes $\forall k \forall P_1..P_k \forall C_T$ $\forall \mathbf{m_1..m_k} \exists \mathbf{C_S}$...

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

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

Some of the proof difficulty is manifest in property-free characterization

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



back-translating finite set of finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T$
$\forall m_1..m_k \exists C_S \ldots$

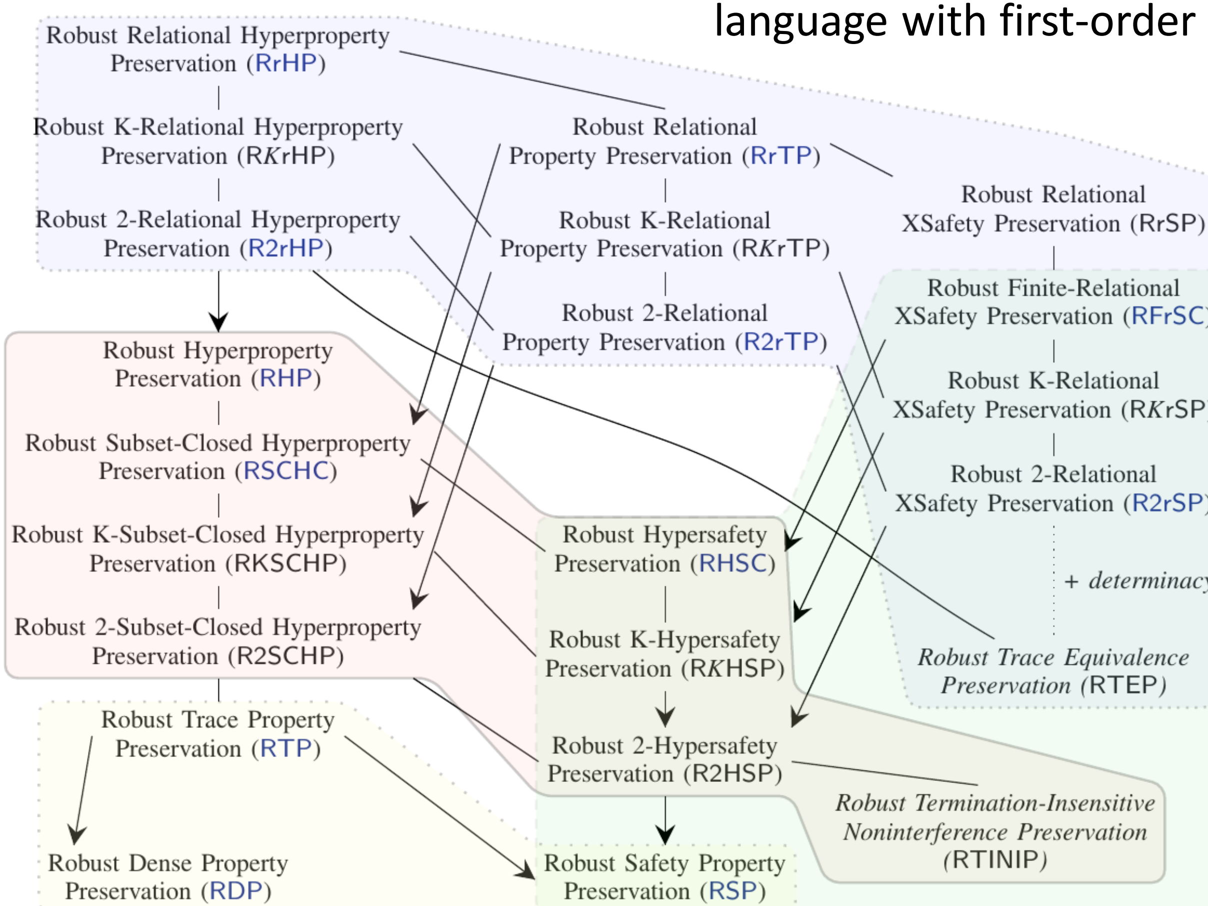
back-translating prog & context
$\forall P \forall C_T \exists C_S \forall t \ldots$

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

back-translating finite trace prefix
$\forall P \forall C_T \forall m \leq t \exists C_S \ldots$

Embraced and extended™ proof techniques

for simple translation from statically to dynamically typed language with first-order functions and I/O



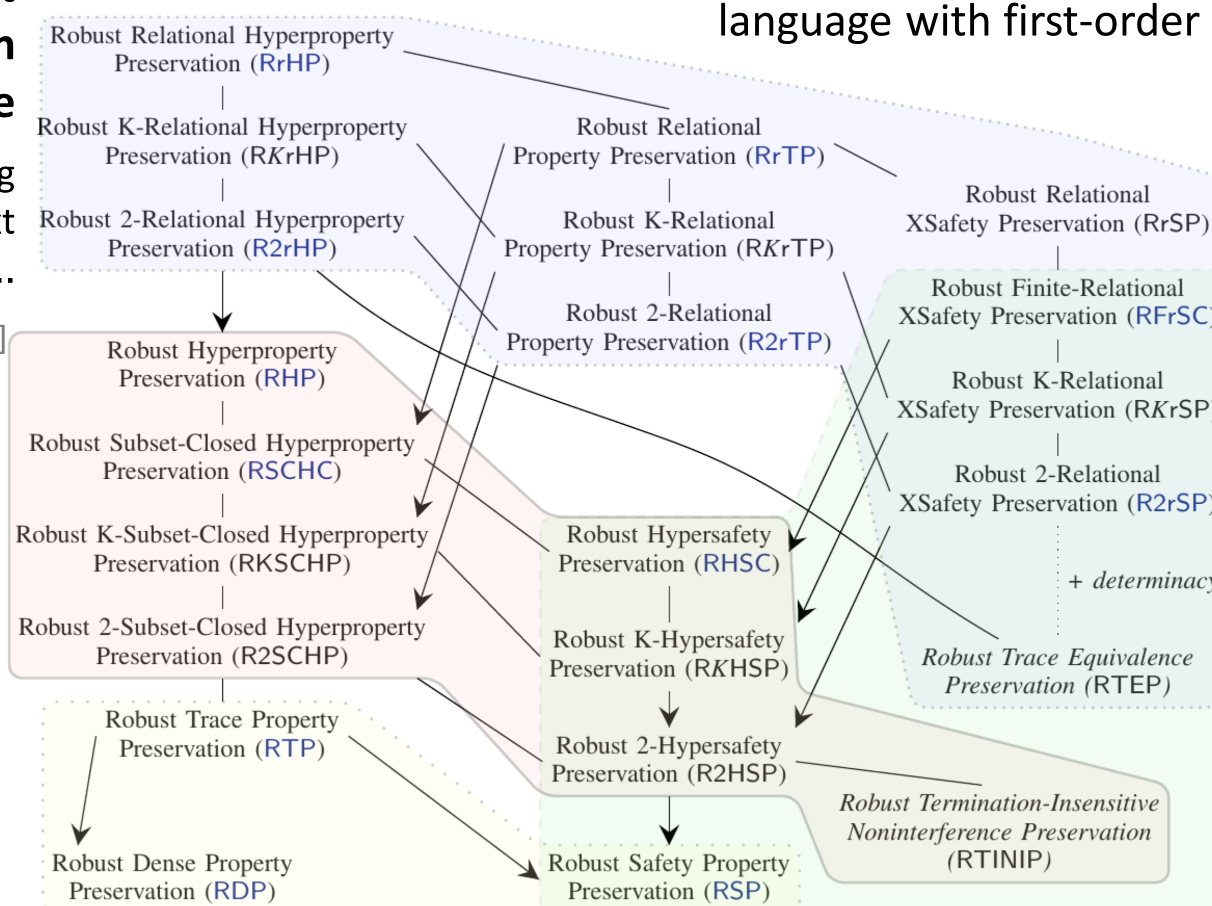
Embraced and extended™ proof techniques

strongest criterion achievable

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

[New et al,ICFP'16]

for simple translation from statically to dynamically typed language with first-order functions and I/O



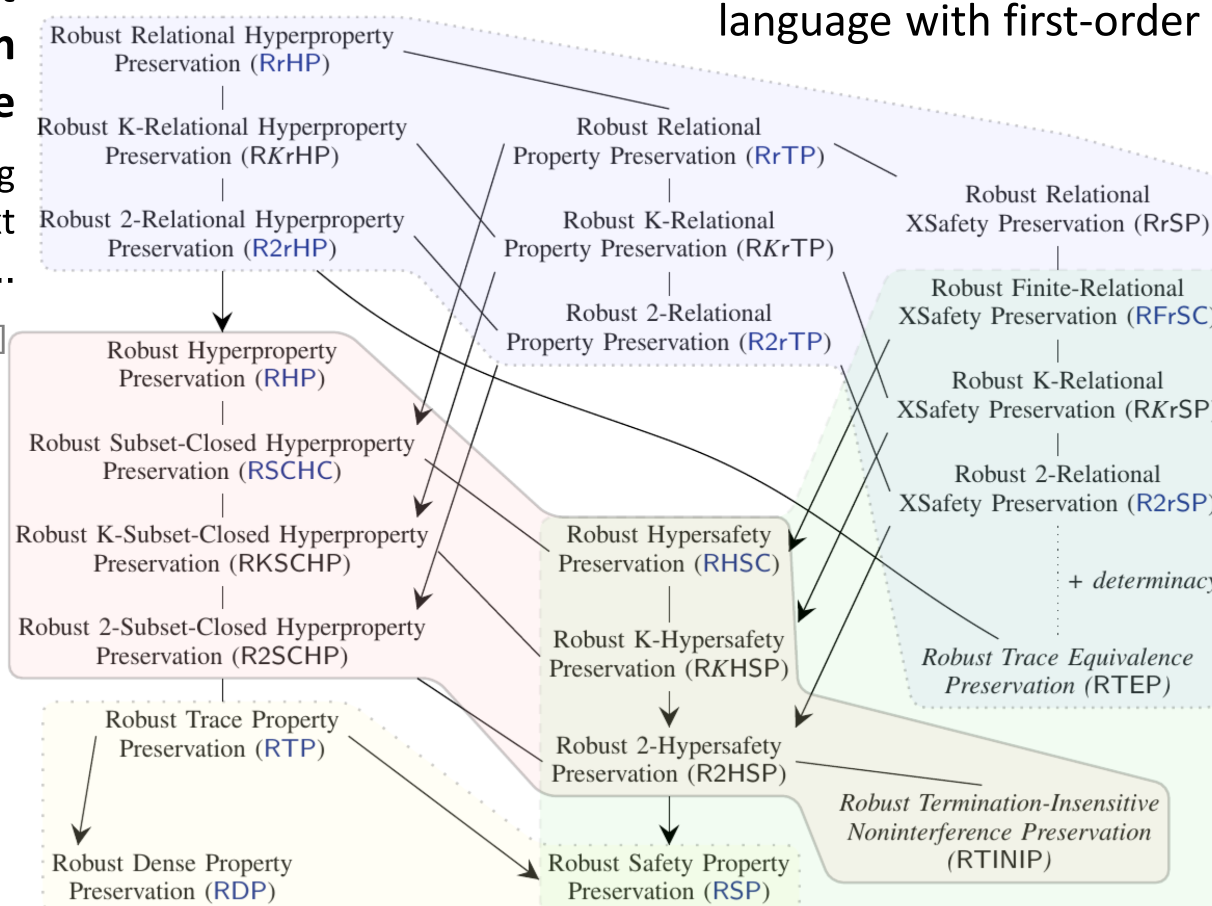
Embraced and extended™ proof techniques

for simple translation from statically to dynamically typed language with first-order functions and I/O

strongest criterion achievable

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

[New et al,ICFP'16]



generic technique applicable

back-translating finite set of finite trace prefixes
$\forall k \forall P_1..P_k \forall C_T$
$\forall m_1..m_k \exists C_S\ldots$

[Jeffrey & Rathke, ESOP'05]
[Patrignani et al,TOPLAS'15]

Proof Techniques for Secure Compilation

Proving FAC

$$\mathbf{P1} \;\; \simeq_{ctx} \;\; \mathbf{P2}$$

$$\Updownarrow$$

$$[\![\mathbf{P1}]\!] \simeq_{ctx} [\![\mathbf{P2}]\!]$$

Proving FAC (History)

$$\mathbf{P1} \quad \simeq_{ctx} \quad \mathbf{P2}$$

$$\Vert$$

Ahmed *et al.*'8'11'14'15'16'17,
Devriese *et al.*'16'17

$$[\![\mathbf{P1}]\!] \quad \sim_{\mathsf{n}} \quad [\![\mathbf{P2}]\!]$$

Proving FAC with Logical Relations

$$\mathbf{P1} \simeq_{ctx} \mathbf{P2}$$

approx. compiler security $\Downarrow$

$$[\![\mathbf{P1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{P2}]\!]$$

Proving FAC with Logical Relations

$$\mathbf{P1} \simeq_{ctx} \mathbf{P2}$$

approx. compiler security

$$\mathbb{C}\Big[\llbracket \mathbf{P1} \rrbracket\Big] \Downarrow_{\mathsf{n}} \quad \overset{?}{\Rightarrow} \quad \mathbb{C}\Big[\llbracket \mathbf{P2} \rrbracket\Big] \Downarrow_{\_}$$

$$\llbracket \mathbf{P1} \rrbracket \overset{?}{\simeq}_{ctx} \llbracket \mathbf{P2} \rrbracket$$

Proving FAC with Logical Relations

$$\mathbf{P1} \simeq_{ctx} \mathbf{P2}$$

$$\langle\!\langle \mathbb{C} \rangle\!\rangle_{\mathbf{n}} \sim_n \mathbb{C}$$

$$\mathbf{P1} \sim_{\_} [\![\mathbf{P1}]\!]$$

(1)

$$\mathbb{C}\Big[[\![\mathbf{P1}]\!]\Big]\Downarrow_{\mathsf{n}} \stackrel{?}{\Rightarrow} \mathbb{C}\Big[[\![\mathbf{P2}]\!]\Big]\Downarrow_{\_}$$

$$[\![\mathbf{P1}]\!] \stackrel{?}{\simeq}_{ctx} [\![\mathbf{P2}]\!]$$

approx. compiler security

Proving FAC with Logical Relations

$$\mathbf{P1} \simeq_{ctx} \mathbf{P2}$$

$$\langle\!\langle \mathbb{C} \rangle\!\rangle_{\mathbf{n}}\left[\mathbf{P1}\right] \Downarrow_{\_}$$

$$\langle\!\langle \mathbb{C} \rangle\!\rangle_{\mathbf{n}} \sim_n \mathbb{C}$$

$$\mathbf{P1} \sim_{\_} [\![\mathbf{P1}]\!] \qquad \Uparrow (1)$$

$$\mathbb{C}\left[[\![\mathbf{P1}]\!]\right] \Downarrow_{\mathsf{n}} \quad \overset{?}{\Rightarrow} \quad \mathbb{C}\left[[\![\mathbf{P2}]\!]\right] \Downarrow_{\_}$$

$$[\![\mathbf{P1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{P2}]\!]$$

$\Downarrow$ approx. compiler security

Proving FAC with Logical Relations

$$\mathbf{P1} \simeq_{ctx} \mathbf{P2}$$

$$\langle\!\langle \mathbb{C} \rangle\!\rangle_{\mathbf{n}}[\mathbf{P1}] \Downarrow_{\_} \overset{\Rightarrow}{(2)} \langle\!\langle \mathbb{C} \rangle\!\rangle_{\mathbf{n}}[\mathbf{P2}] \Downarrow_{\_}$$

$$\langle\!\langle \mathbb{C} \rangle\!\rangle_{\mathbf{n}} \sim_n \mathbb{C}$$

$$\mathbf{P1} \sim_{\_} [\![\mathbf{P1}]\!]$$

$$\Uparrow (1)$$

$$\mathbb{C}\Big[[\![\mathbf{P1}]\!]\Big] \Downarrow_{\mathsf{n}} \overset{?}{\Rightarrow} \mathbb{C}\Big[[\![\mathbf{P2}]\!]\Big] \Downarrow_{\_}$$

$$[\![\mathbf{P1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{P2}]\!]$$

$\Downarrow$ approx. compiler security

Proving FAC with Logical Relations

$$\mathbf{P1} \simeq_{ctx} \mathbf{P2}$$

$$\begin{tikzcd}
\langle\!\langle\mathbb{C}\rangle\!\rangle_{\mathbf{n}}[\mathbf{P1}]\Downarrow_ \arrow[r, phantom, "\underset{(2)}{\Rightarrow}"] & \langle\!\langle\mathbb{C}\rangle\!\rangle_{\mathbf{n}}[\mathbf{P2}]\Downarrow_ \arrow[d, Rightarrow, "(3)"] \\
\mathbb{C}\left[\llbracket\mathbf{P1}\rrbracket\right]\Downarrow_{\mathsf{n}} \arrow[u, Rightarrow, "(1)"] \arrow[r, phantom, "\overset{?}{\Rightarrow}"] & \mathbb{C}\left[\llbracket\mathbf{P2}\rrbracket\right]\Downarrow_
\end{tikzcd}$$

$\langle\!\langle\mathbb{C}\rangle\!\rangle_{\mathbf{n}} \sim_n \mathbb{C}$

$\mathbf{P1} \sim\_ \llbracket\mathbf{P1}\rrbracket$

$\langle\!\langle\mathbb{C}\rangle\!\rangle_{\mathbf{n}} \sim\_ \mathbb{C}$

$\mathbf{P2} \sim\_ \llbracket\mathbf{P2}\rrbracket$

$$\llbracket\mathbf{P1}\rrbracket \overset{?}{\simeq}_{ctx} \llbracket\mathbf{P2}\rrbracket$$

approx. compiler security

Open Problems

- Compiling to WebAssembly
- Proving security against Spectre

Memory Safety Preservation for WebAssembly

Marco Vassena [1]

Marco Patrignani [1,2]

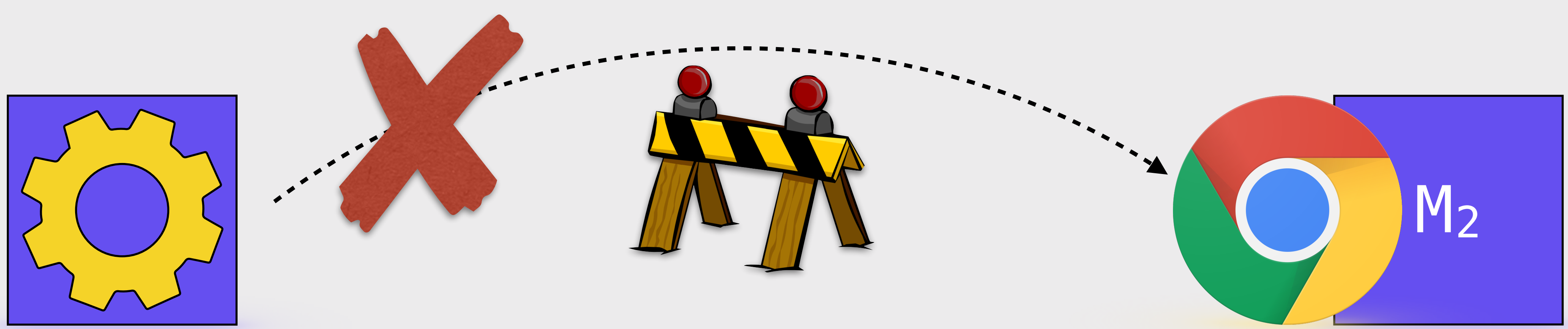
[1] CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

[2] Stanford University

WebAssembly

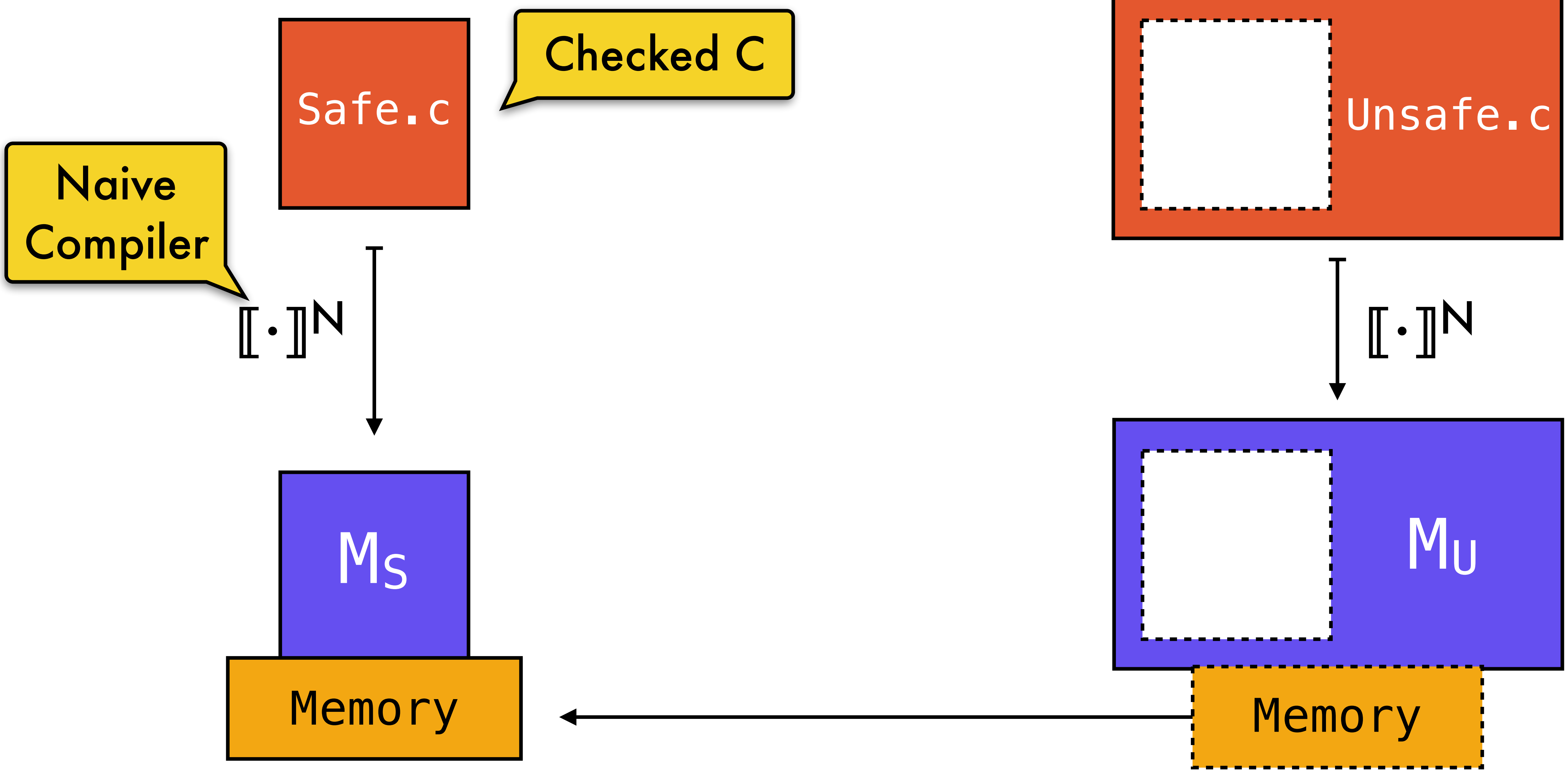
Validation

Sandboxed *Execution Environment*

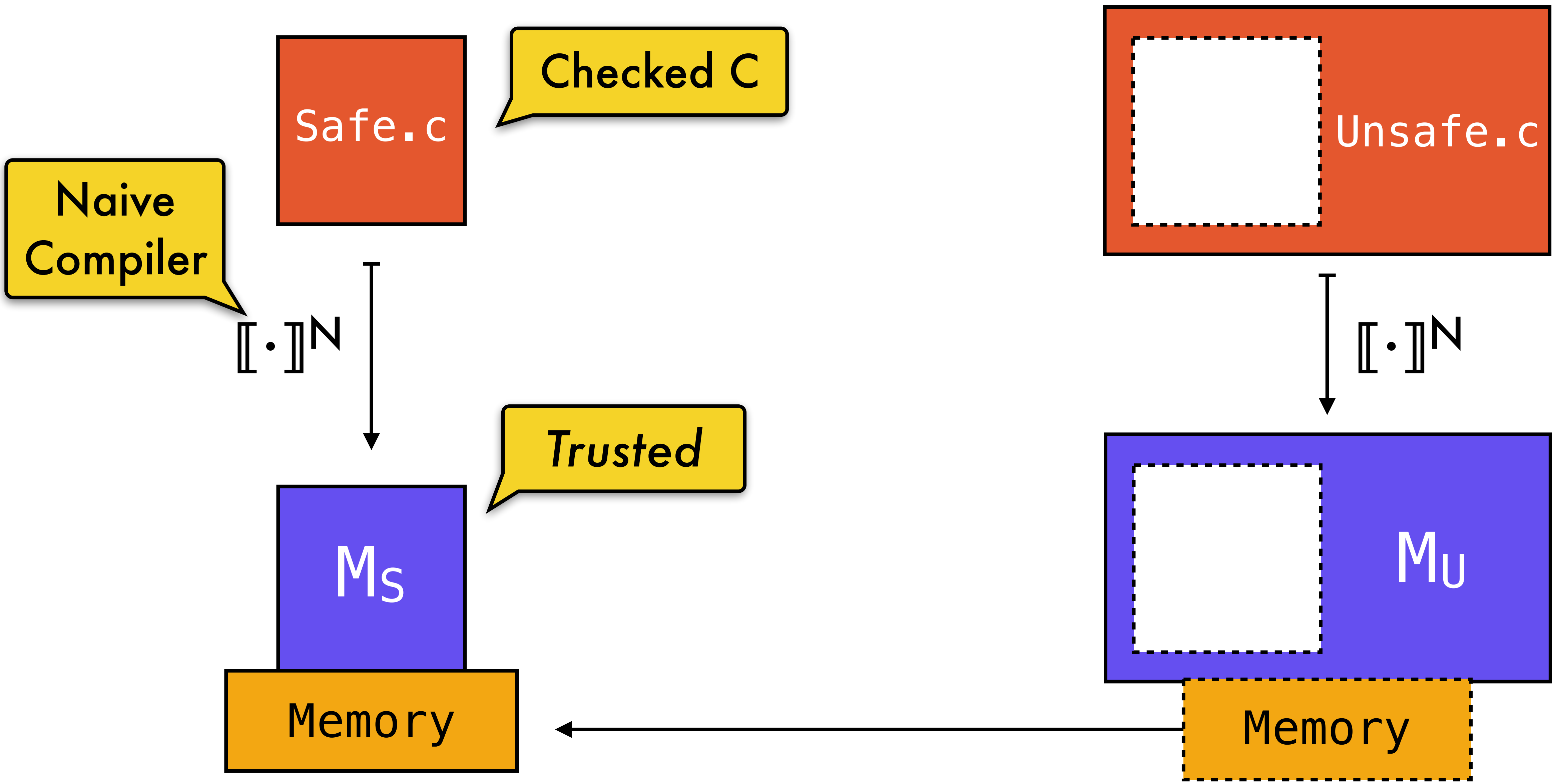


Problem

Problem

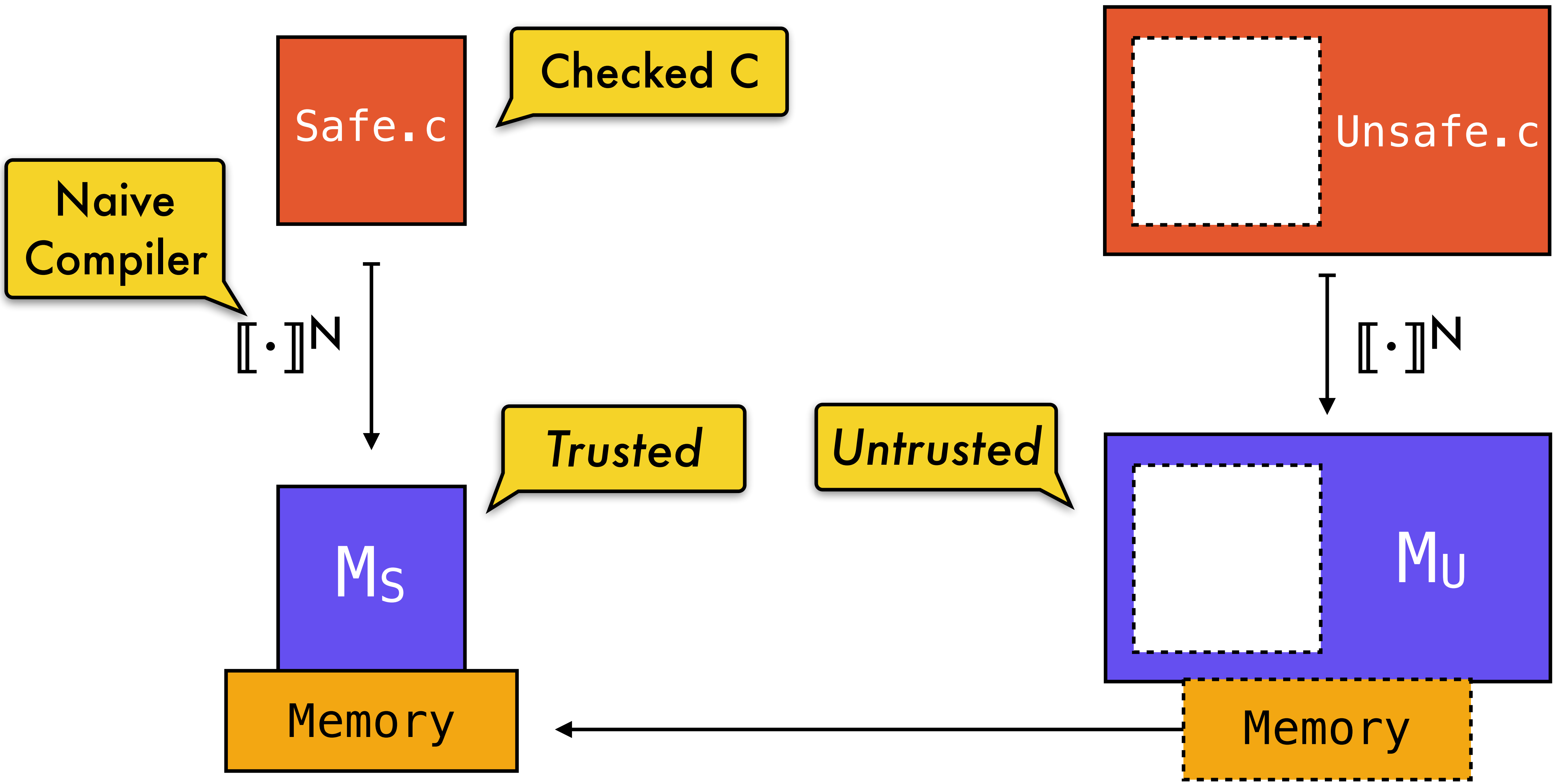

Checked C
Safe.c
Naive Compiler
$[\![\cdot]\!]^N$
M_S
Memory
Unsafe.c
$[\![\cdot]\!]^N$
M_U
Memory

Problem

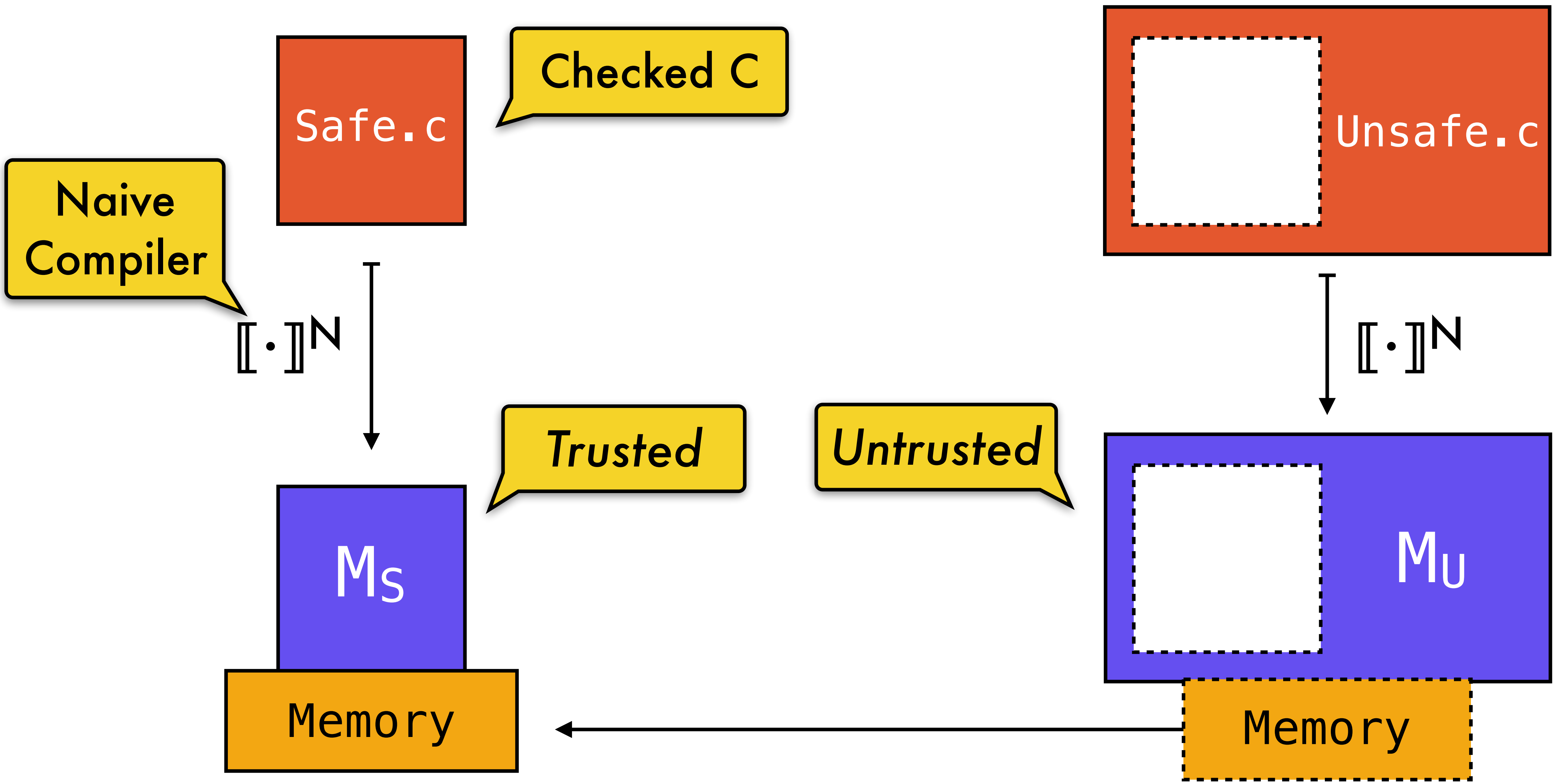


Safe.c
Checked C
Naive Compiler
⟦·⟧N
Trusted
MS
Memory
Unsafe.c
⟦·⟧N
MU
Memory

Problem


Checked C
Safe.c
Naive Compiler
⟦·⟧N
Trusted
Untrusted
Unsafe.c
⟦·⟧N
MS
MU
Memory
Memory

Problem



Module M_U *can corrupt the shared memory and break memory safety of* M_S

Memory Safety as a Trace Property

Pointers and memory locations are tagged with ***colors***

Trace

α = **read**(n^c) | **write**(n^c) | **alloc**(N,n^c) | **free**(n^c)

Memory Safety

$$MS(\overline{\alpha}, H) = \exists\, H' \cdot\ H \xrightarrow{\overline{\alpha}} H'$$

Relaxing Memory Safety

Our definition can capture interesting variations

Array bounds & null pointer

Spatial Memory Safety

Including Relaxed Temporal safety

Temporal Memory Safety

Intra-object, e.g., struct fields

Fine-grained Memory Safety

Using colors & valid bit

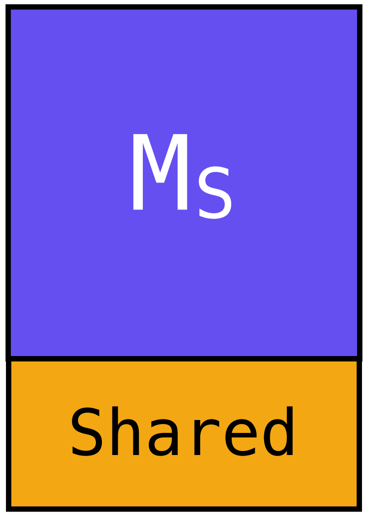
Pointer Integrity

Short Term Solution

Provide memory isolation through ***module encapsulation***

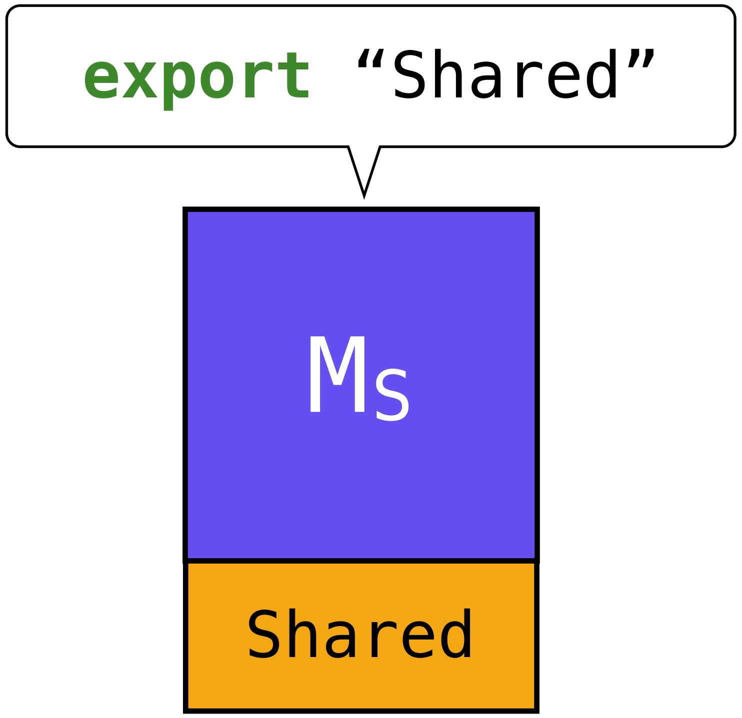
Short Term Solution

Provide memory isolation through ***module encapsulation***



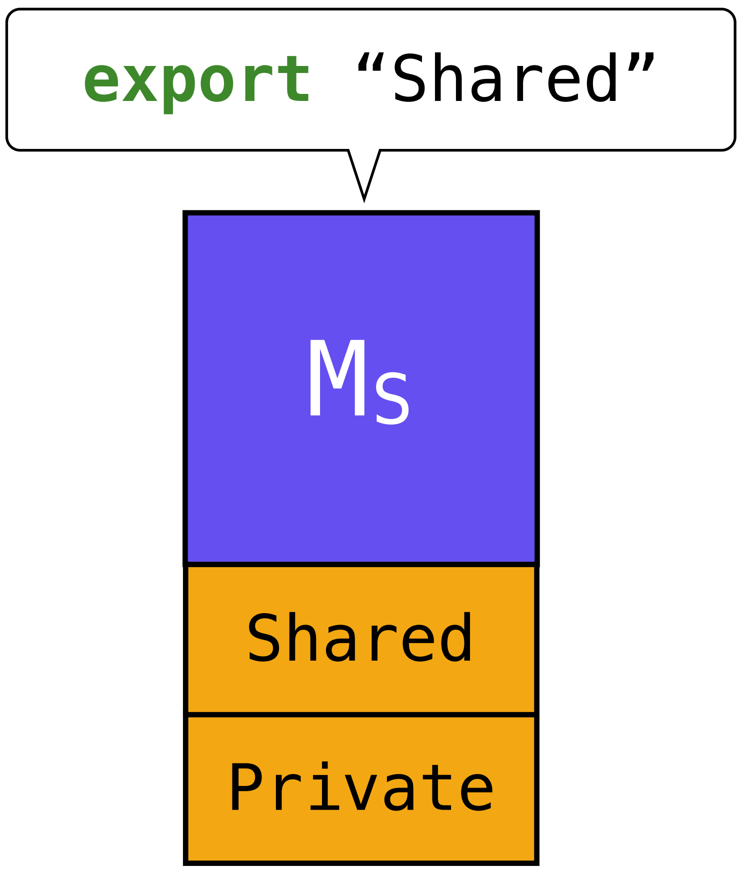
Short Term Solution

*Provide memory isolation through **module encapsulation***



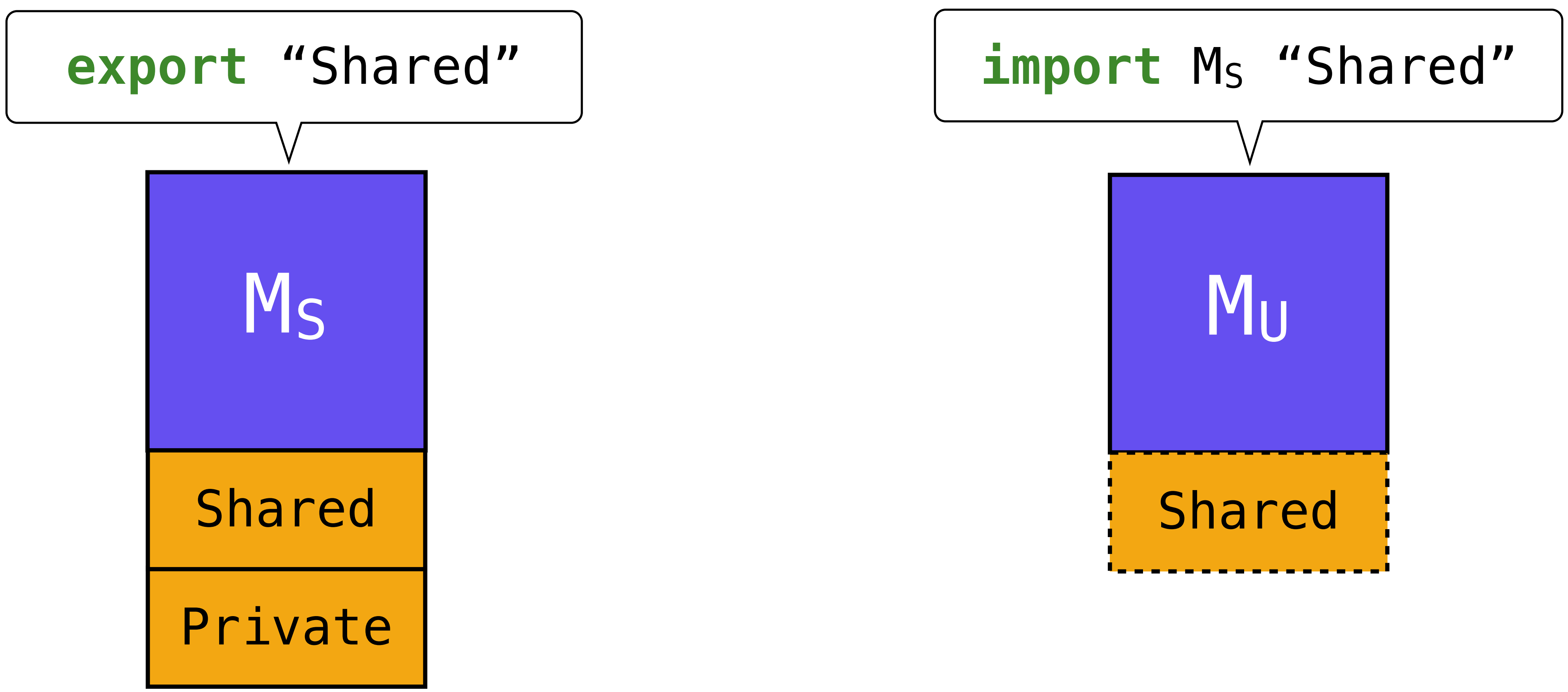
Short Term Solution

Provide memory isolation through ***module encapsulation***



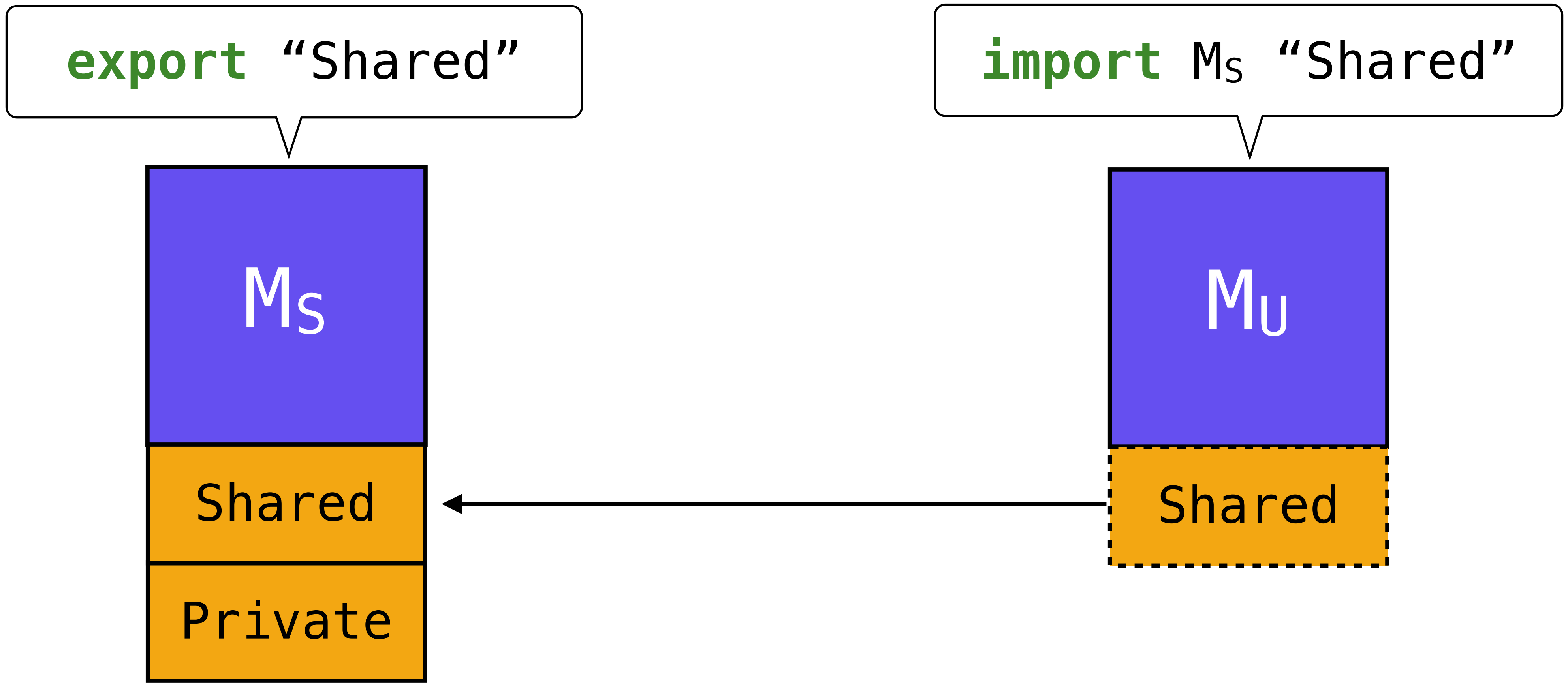
Short Term Solution

Provide memory isolation through ***module encapsulation***



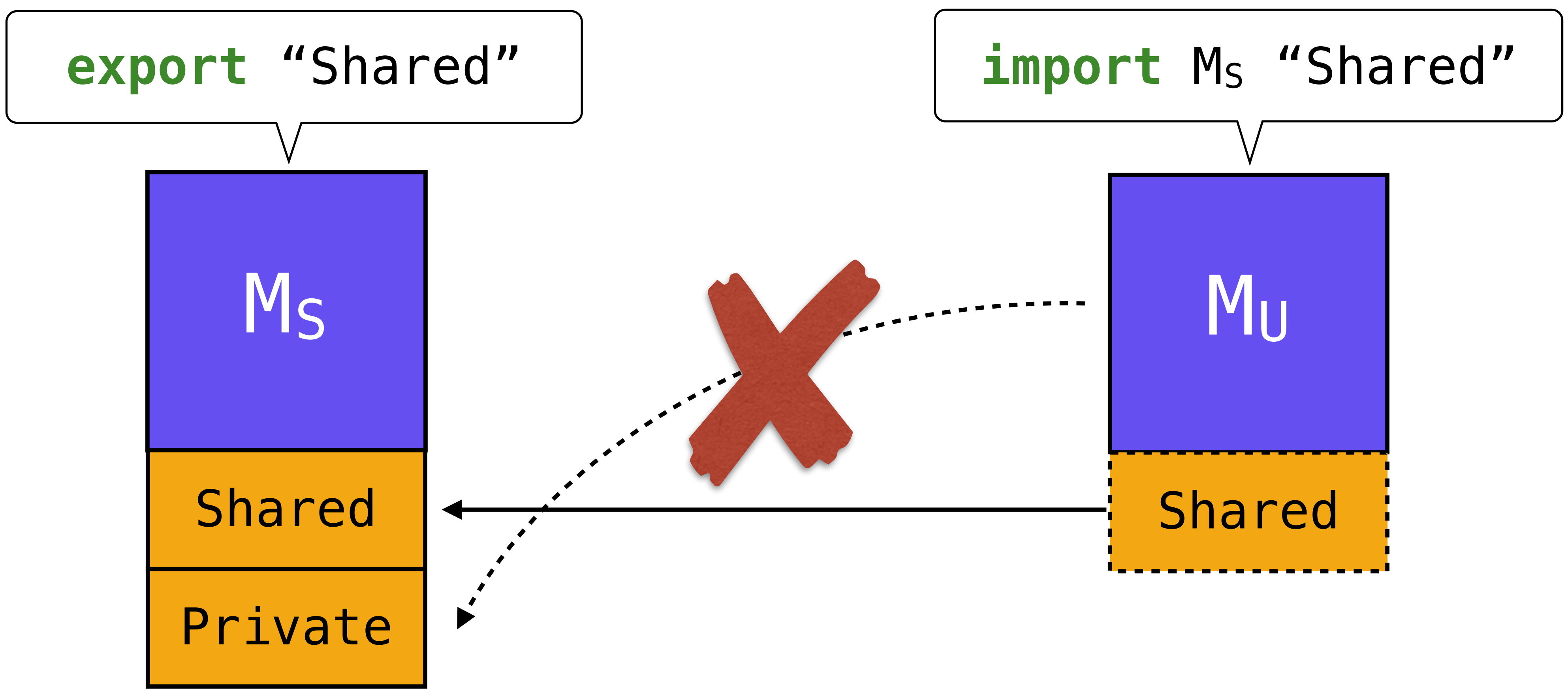
Short Term Solution

Provide memory isolation through ***module encapsulation***



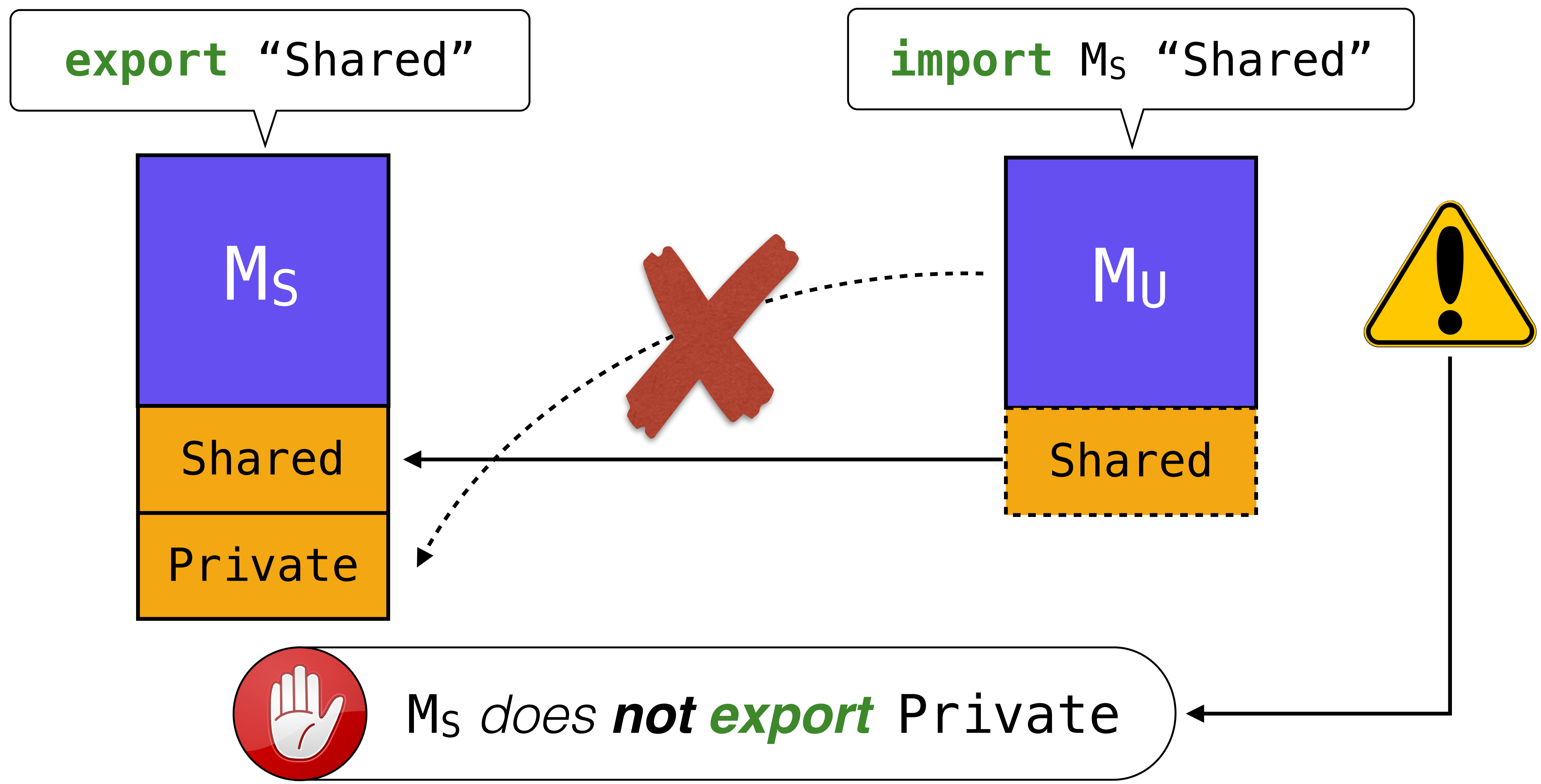
Short Term Solution

Provide memory isolation through ***module encapsulation***



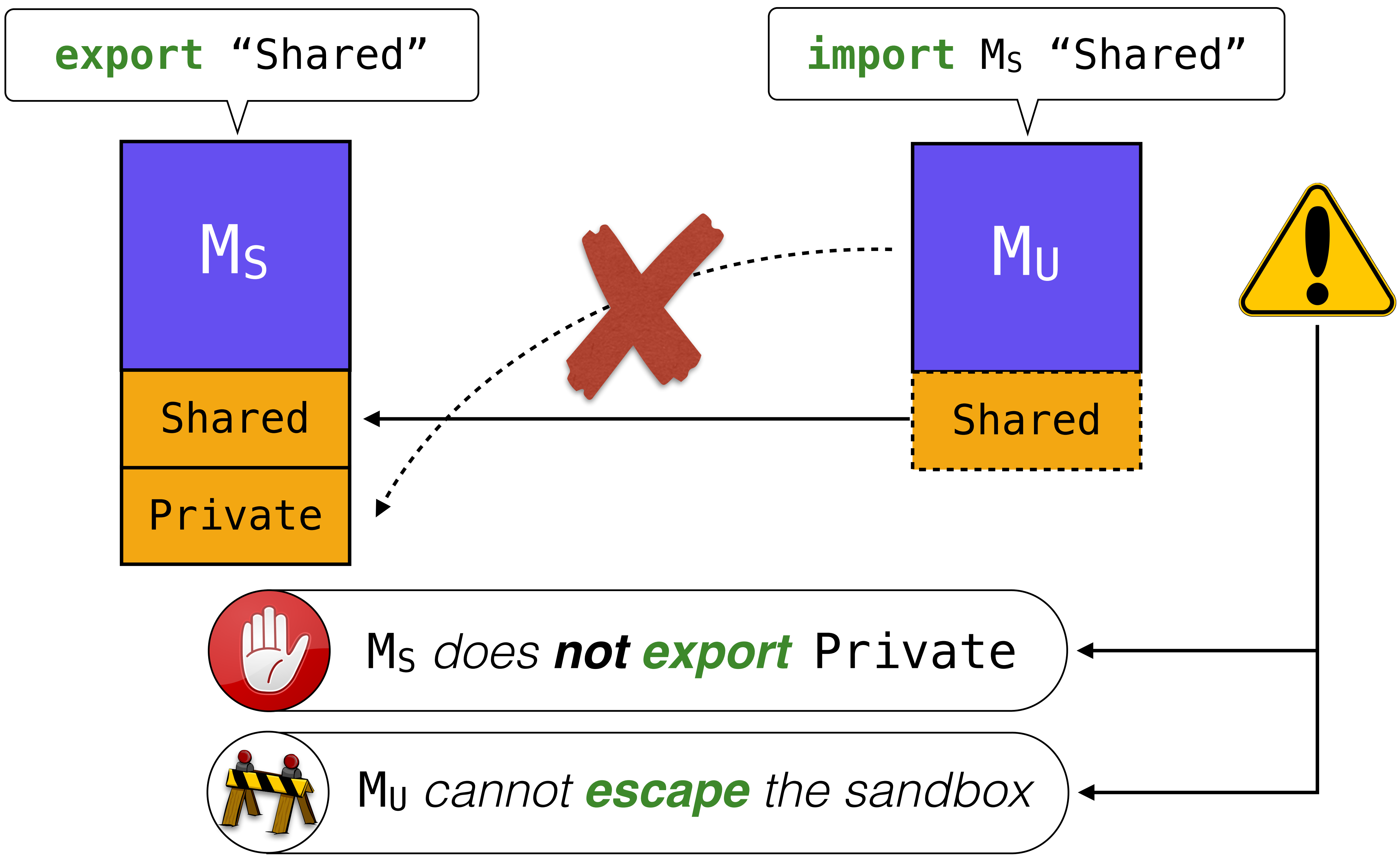
Short Term Solution

Provide memory isolation through ***module encapsulation***

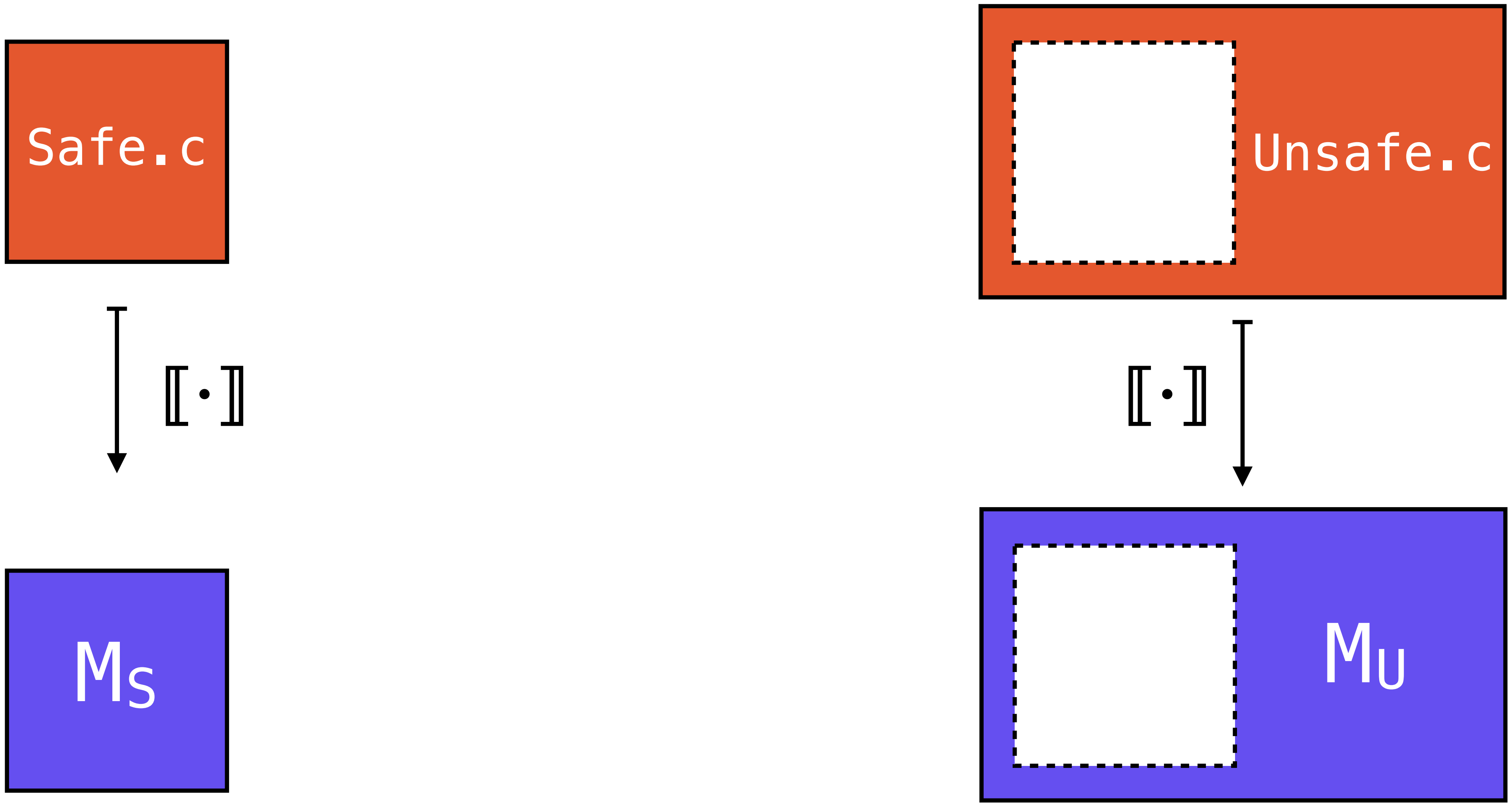


Short Term Solution

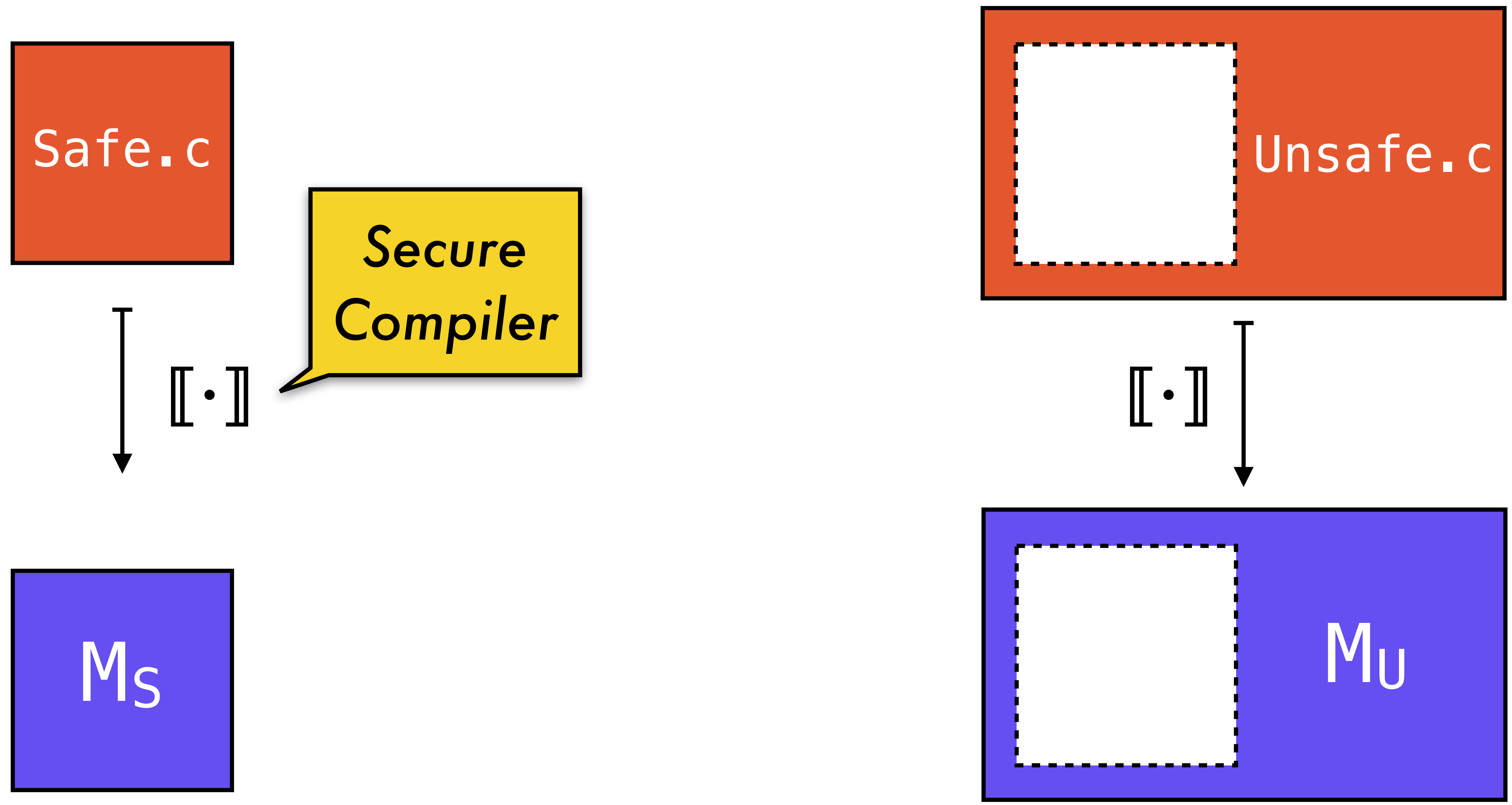
Provide memory isolation through ***module encapsulation***



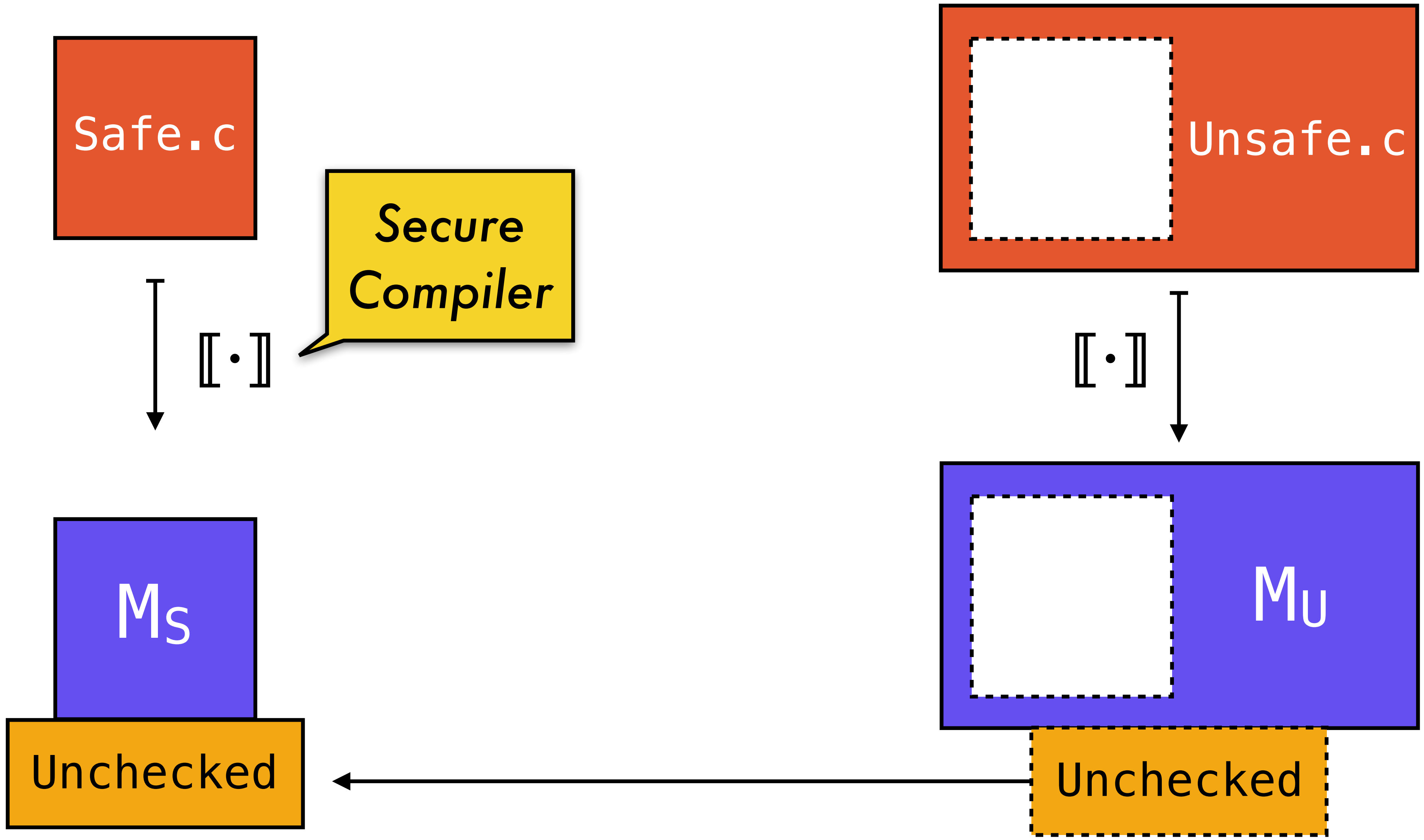
Memory Isolation in Action



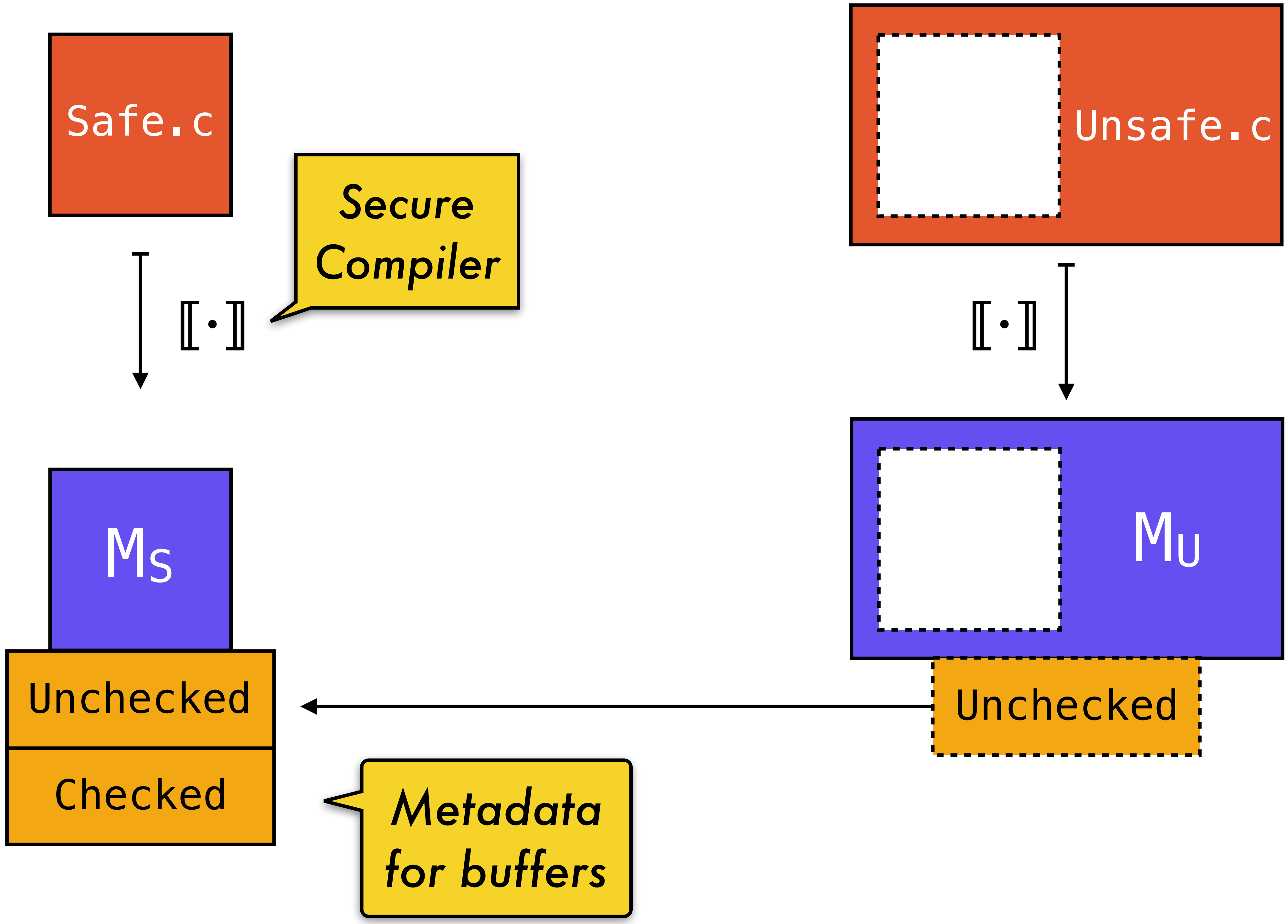
Memory Isolation in Action



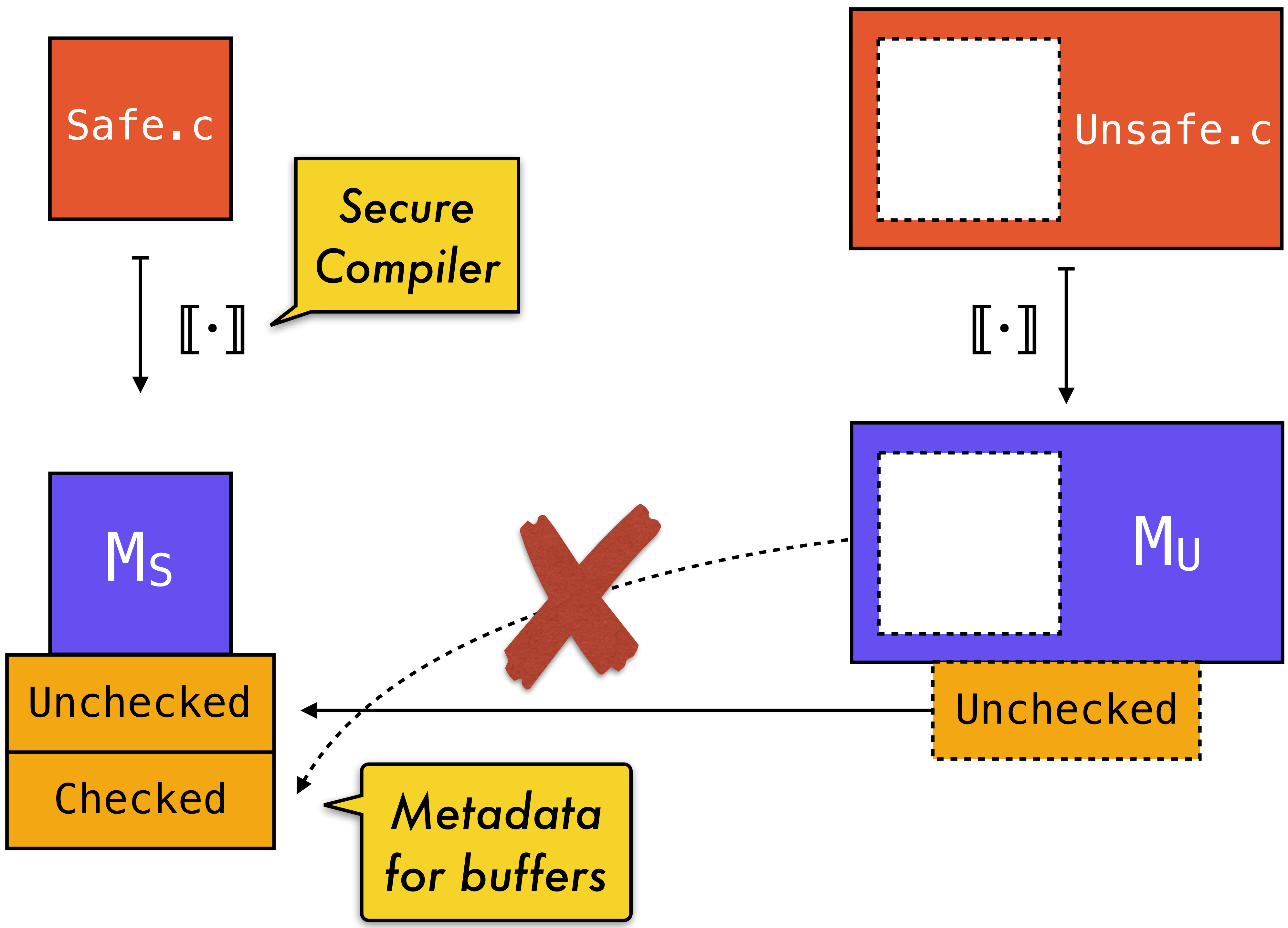
Memory Isolation in Action



Memory Isolation in Action



Memory Isolation in Action



Compiling Safe Modules

*Pointers as **capabilities** stored in private memory:*

Long Term Solution

*Target a language designed with **first-class support** for memory safety*

MS-WASM

Position Paper: Progressive Memory Safety for WebAssembly

Craig Disselkoen
UC San Diego
cdisselk@cs.ucsd.edu

John Renner
UC San Diego
jmrenner@eng.ucsd.edu

Conrad Watt
University of Cambridge
conrad.watt@cl.cam.ac.uk

Tal Garfinkel
Stanford University
talg@cs.stanford.edu

Amit Levy
Princeton University
aalevy@cs.princeton.edu

Deian Stefan
UC San Diego
deian@cs.ucsd.edu

ABSTRACT

WebAssembly (Wasm) is a low-level platform-independent bytecode language. Today, developers can compile C/C++ to Wasm and run it everywhere, at almost native speeds. Unfortunately, this compilation from C/C++ to Wasm also preserves classic memory safety vulnerabilities, such as buffer overflows and use-after-frees.

New processor features (e.g., tagged memory, pointer authentication, and fine grain capabilities) are making it increasingly possible to detect, mitigate, and prevent such vulnerabilities with low overhead. Unfortunately, Wasm JITs and compilers cannot exploit these features. Critical high-level information—e.g., the size of an array—is lost when lowering to Wasm.

1 INTRODUCTION

WebAssembly (Wasm) is a platform-independent bytecode designed to run C/C++ and similar languages at near native speed in the browser. Wasm's *linear memory model*—i.e., loads and stores to an untyped array of bytes, is the key feature that makes it possible for C/C++ compilers like Clang to easily and efficiently target Wasm. Unfortunately, this is also the reason memory safety vulnerabilities, like buffer overflows and use-after-frees (UAFs), remain a problem when C/C++ programs are compiled to Wasm [29, 32].

Wasm is designed to allow browsers to run code in a sandbox, isolating the impact of vulnerabilities in Wasm code from the rest of the browser.[1] But keeping the browser safe from Wasm code

Long Term

First instance of usable secure compiler

Exorcising Spectres with Secure Compilers

Marco Guarnieri
IMDEA Software Institute

Joint work with Marco Patrignani @ CISPA/Stanford University

SPECTRE

Exploits ***speculative execution***

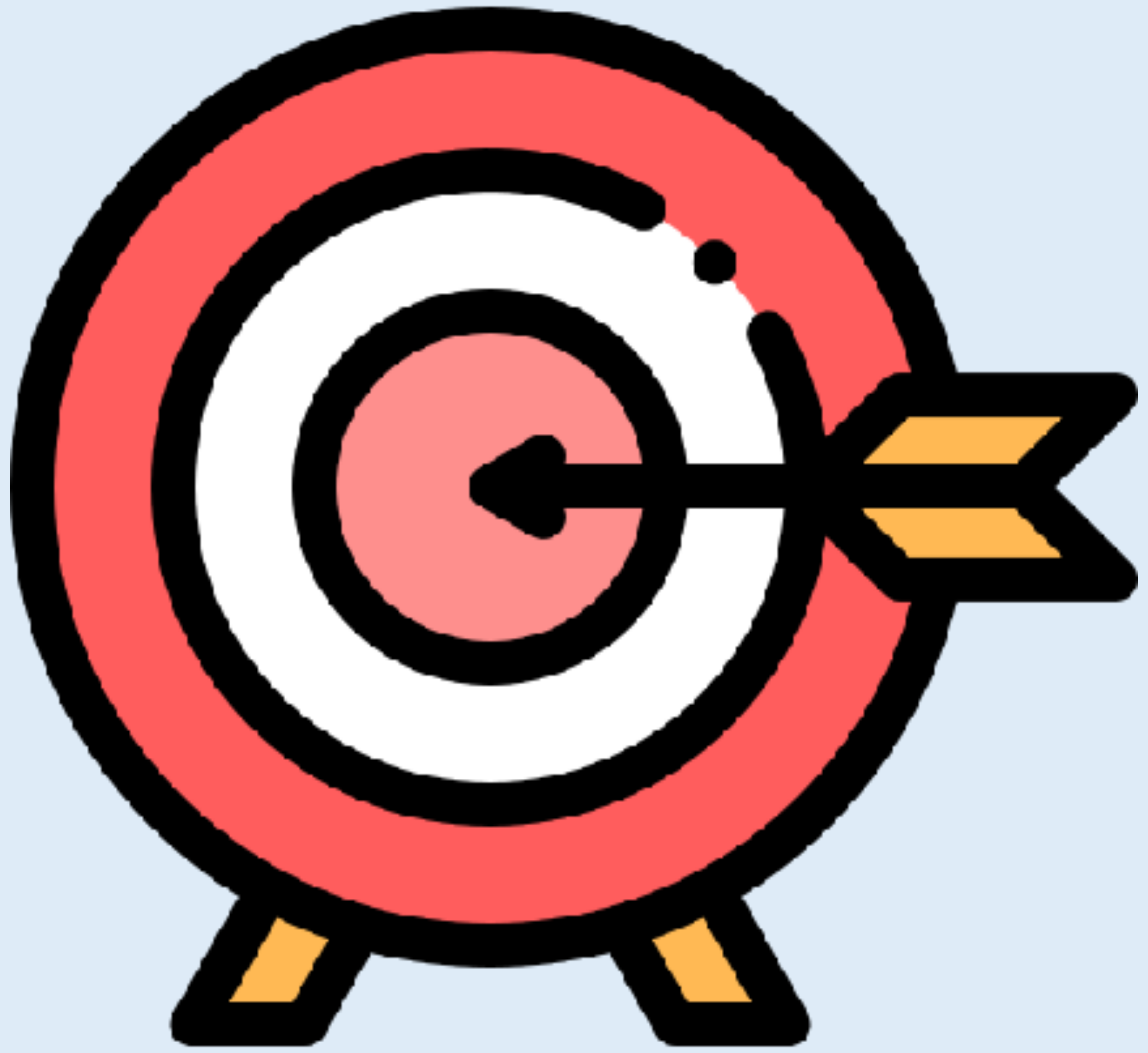
Almost ***all*** modern ***CPUs*** are ***affected***

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom — Spectre Attacks: Exploiting Speculative Execution — S&P 2019

Compiler-level countermeasures

- Example: insert LFENCE to selectively stop speculative execution
- Implemented in major compilers (Microsoft Visual C++, Intel ICC, Clang)

No security guarantees!



Our Goal

Formally reasoning about the security of compiler-level countermeasures

Compiler-level countermeasures

For ***Spectre V1***

Injecting speculation barriers

```
if (x < A_size)
    y = B[A[x]]
```

```
if (x < A_size)
    lfence
    y = B[A[x]]
```

- In x86, ***LFENCE*** act as ***speculation barrier***
- Compiler injects LFENCE after each branch instruction
  - Microsoft Visual C++
  - Intel ICC
- Effectively ***stop speculative execution***!

Speculative load-hardening (SLH)

```
if (x < A_size)
    y = B[A[x]]
```

```
if (x < A_size)
    y = B[mask(A[x])]
```

- Injects ***data dependencies*** and ***masking operations***
    - Combines ***conditional moves*** and ***binary operations***
- Stops ***speculative leaks***
- Does not block speculative execution!
- Implemented in Clang

Speculative load-hardening (SLH)

```
if (x < A_size)
    y = B[A[x]]
```

```
mov     rax, A_size
mov     rcx, x
mov     rdx, 0
cmp     rcx, rax
jae     END
cmovae  -1, rdx
mov     rax, A[rcx]
shl     rax, 9
or      rax, rdx
mov     rax, B[rax]
```

- Injects ***data dependencies*** and ***masking operations***
  - Combines ***conditional moves*** and ***binary operations***
- Stops ***speculative leaks***
- Does not block speculative execution!
- Implemented in Clang

Modeling speculative execution

See M. Guarnieri, B. Köpf, J. Morales, J. Reineke, A. Sánchez
Spectector: Principled Detection of Speculative Information Flows
@ IEEE S&P 2020

https://spectector.github.io

How to capture leakage?

Capture attacker's observational power

Non-speculative semantics

Speculative semantics

+

Attacker model

Model program's behavior

How to capture leakage?

Starts ***speculative transactions*** upon branch instructions

- ***Committed*** upon correct speculation
- ***Rolled-back*** upon misprediction

Speculative semantics

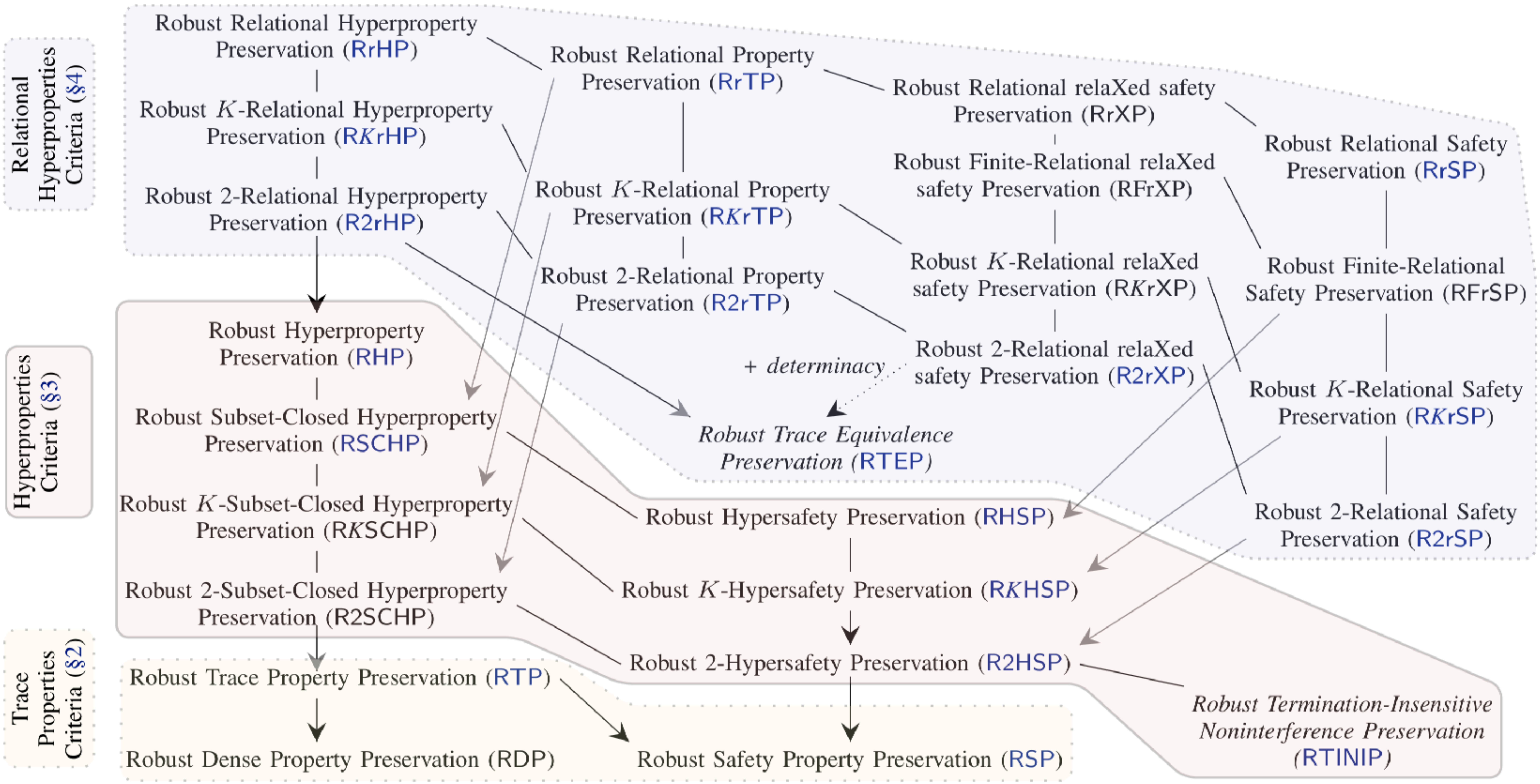
+

Capture attacker's observational power

Attacker model

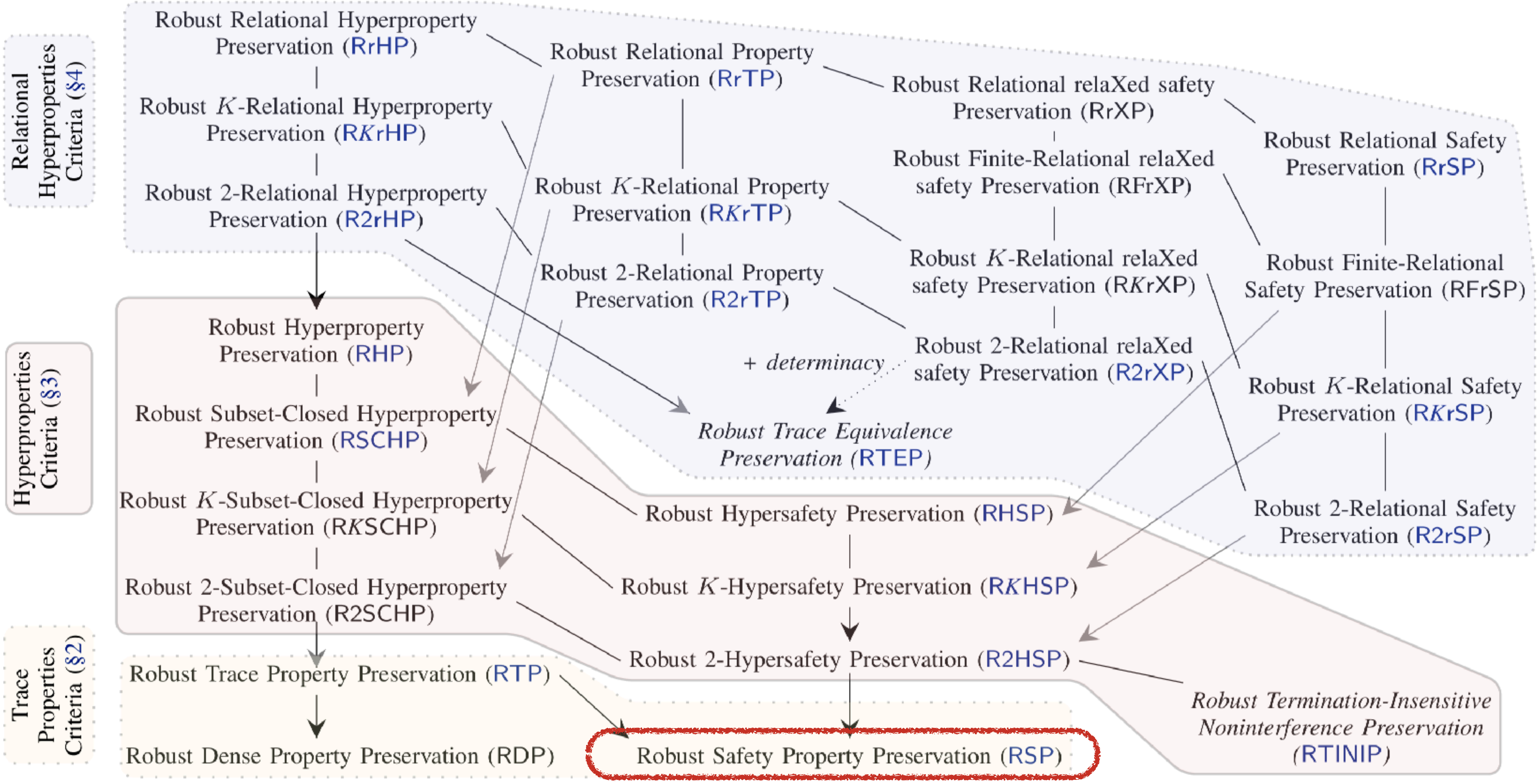
Model program's behavior

Secure compilation for Spectre



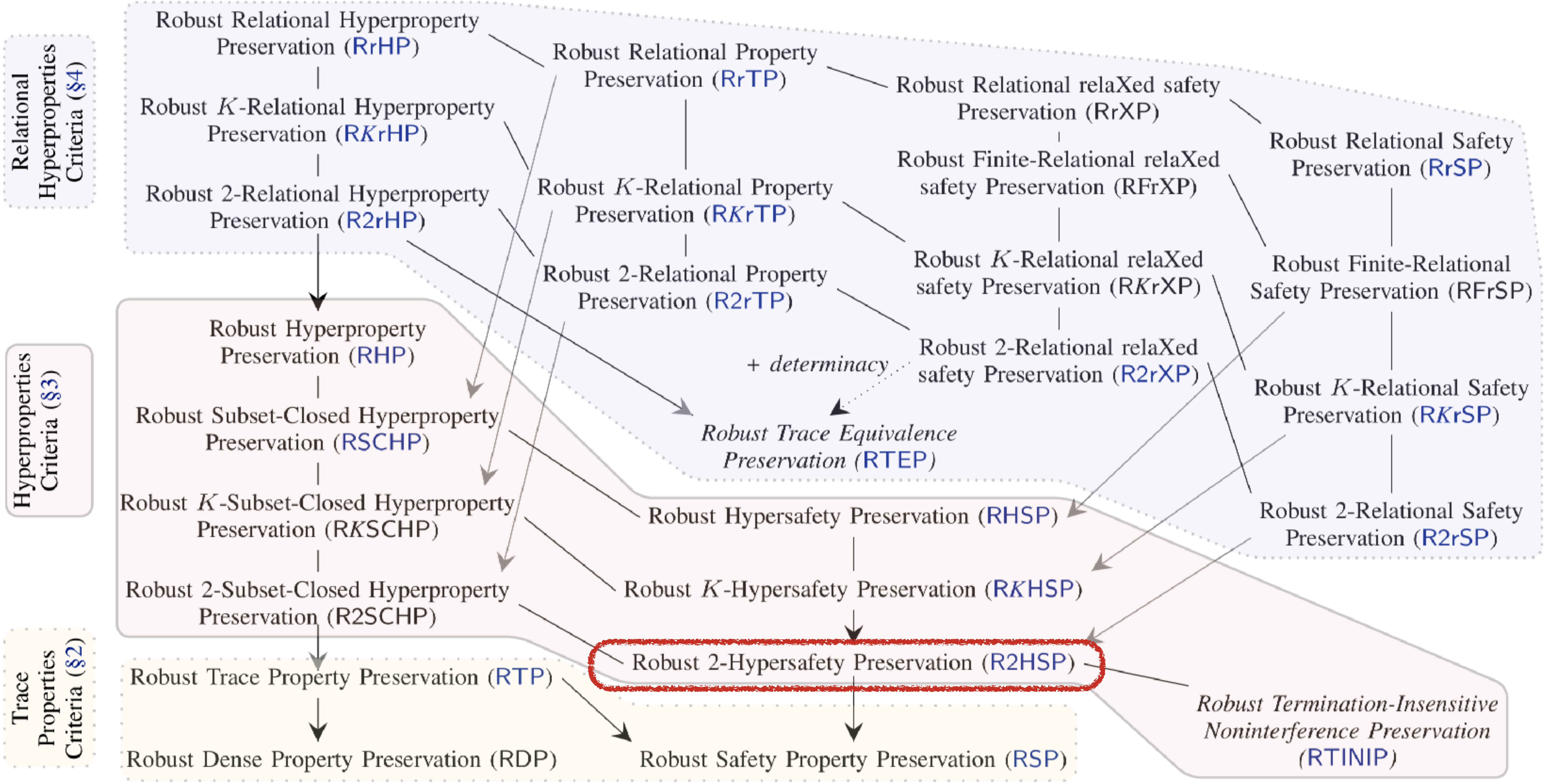
[Abate et al. — Journey Beyond Full Abstraction, CSF19]

Secure compilation for Spectre



[Abate et al. — Journey Beyond Full Abstraction, CSF19]

Secure compilation for Spectre



[Abate et al. — *Journey Beyond Full Abstraction*, CSF19]

A hierarchy of security conditions

Strict speculative safety

Speculative safety

Speculative non-interference

[Guarnieri et al., S&P 2020]

Strict speculative safety

Strict speculative safety

Program **P** is **strictly speculative safe** if

Strict speculative safety

Program **P** is **strictly speculative safe** if

Informally:

No speculative observations (i.e., no speculative execution ***at all***)

Strict speculative safety

Program $\mathbf{P}$ is **strictly speculative safe** if

Informally:

No speculative observations (i.e., no speculative execution ***at all***)

Formally:

For all program states s:

$$\mathbf{P}_{\texttt{non-spec}}(s) = \mathbf{P}_{\texttt{spec}}(s)$$

Strict speculative safety

Program $\mathbf{P}$ is **strictly speculative safe** if

Informally:

No speculative observations (i.e., no speculative execution ***at all***)

Formally:

For all program states s:

$$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s)$$

Speculative safety

Speculative safety

Program **P** is **speculatively safe** if

Speculative safety

Program **P** is **speculatively safe** if

Informally:

Speculative observations may refer only to non-speculatively accessed data!

Speculative safety

Program $\mathbf{P}$ is **speculatively safe** if

Informally:

Speculative observations may refer only to non-speculatively accessed data!

Formally:

For all program states s:

$$\mathbf{P}_{\mathbf{spec}}(s) = \mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{safe}}$$

Speculative safety

Program **P** is **speculatively safe** if

Informally:

Speculative observations may refer only to non-speculatively accessed data!

Formally:

For all program states s:

$$\mathbf{P}_{\mathbf{spec}}(s) = \mathbf{P}_{\mathbf{spec}}(s)|_{\mathbf{safe}}$$

Drops ***unsafe*** observations

Speculative non-interference

Speculative non-interference

Program **`P`** is **speculatively non-interferent** if

Speculative non-interference

Program **P** is **speculatively non-interferent** if

Informally:

Leakage of **P** in ***non-speculative*** execution	=	Leakage of **P** in ***speculative*** execution

Speculative non-interference

Program **P** is **speculatively non-interferent** if

Informally:

Leakage of **P** in ***non-speculative*** execution	=	Leakage of **P** in ***speculative*** execution

Formally:

Speculative non-interference

Program $\mathbf{P}$ is **speculatively non-interferent** if

Informally:

Leakage of $\mathbf{P}$ in ***non-speculative*** execution $=$ Leakage of $\mathbf{P}$ in ***speculative*** execution

Formally:

For all program states s and s':

$$\mathbf{P}_{\texttt{non-spec}}(s) = \mathbf{P}_{\texttt{non-spec}}(s') \implies \mathbf{P}_{\texttt{spec}}(s) = \mathbf{P}_{\texttt{spec}}(s')$$

Speculative non-interference

Program $\mathbf{P}$ is **speculatively non-interferent** if

Informally:

Leakage of $\mathbf{P}$ in ***non-speculative*** execution	$=$	Leakage of $\mathbf{P}$ in ***speculative*** execution

Formally:

For all program states s and s':

$$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathbf{ns}} = \mathbf{P}_{\mathbf{spec}}(s')|_{\mathbf{ns}}$$

$$\Rightarrow \mathbf{P}_{\mathbf{spec}}(s) = \mathbf{P}_{\mathbf{spec}}(s')$$

A hierarchy of security conditions

$\mathbf{P}_{\mathtt{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathtt{spec}}(s)$

Strict speculative safety

$\mathbf{P}_{\mathtt{spec}}(s)|_{\mathsf{safe}} = \mathbf{P}_{\mathtt{spec}}(s)$

Speculative safety

Speculative non-interference

[Guarnieri et al., S&P 2020]

$$\mathbf{P}_{\mathtt{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathtt{spec}}(s')|_{\mathsf{ns}}$$
$$\Rightarrow \mathbf{P}_{\mathtt{spec}}(s) = \mathbf{P}_{\mathtt{spec}}(s')$$

A hierarchy of security conditions

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s)$

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{safe}} = \mathbf{P}_{\mathbf{spec}}(s)$

Strict speculative safety

Speculative safety

Speculative non-interference

[Guarnieri et al., S&P 2020]

$$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s\,')|_{\mathsf{ns}}$$
$$\Rightarrow \mathbf{P}_{\mathbf{spec}}(s) = \mathbf{P}_{\mathbf{spec}}(s\,')$$

A hierarchy of security conditions

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s)$

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{safe}} = \mathbf{P}_{\mathbf{spec}}(s)$

Strict speculative safety

Speculative safety

Speculative non-interference

[Guarnieri et al., S&P 2020]

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s')|_{\mathsf{ns}}$

$\Rightarrow \mathbf{P}_{\mathbf{spec}}(s) = \mathbf{P}_{\mathbf{spec}}(s')$

A hierarchy of security conditions

$\mathbf{P}_{\mathsf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathsf{spec}}(s)$ $\mathbf{P}_{\mathsf{spec}}(s)|_{\mathsf{safe}} = \mathbf{P}_{\mathsf{spec}}(s)$

Strict speculative safety $\Rightarrow$ **Speculative safety** $\Rightarrow$ **Speculative non-interference**

[Guarnieri et al., S&P 2020]

$$\mathbf{P}_{\mathsf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathsf{spec}}(s')|_{\mathsf{ns}} \Rightarrow \mathbf{P}_{\mathsf{spec}}(s) = \mathbf{P}_{\mathsf{spec}}(s')$$

A hierarchy of security conditions

$\mathbf{P}_{\mathtt{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathtt{spec}}(s)$ $\quad$ $\mathbf{P}_{\mathtt{spec}}(s)|_{\mathsf{safe}} = \mathbf{P}_{\mathtt{spec}}(s)$

Strict speculative safety $\Rightarrow$ **Speculative safety** $\Rightarrow$ **Speculative non-interference**

[Guarnieri et al., S&P 2020]

$$\mathbf{P}_{\mathtt{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathtt{spec}}(s')|_{\mathsf{ns}}$$
$$\Rightarrow \mathbf{P}_{\mathtt{spec}}(s) = \mathbf{P}_{\mathtt{spec}}(s')$$

LFENCE Injection

A hierarchy of security conditions

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s)$ $\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{safe}} = \mathbf{P}_{\mathbf{spec}}(s)$

Strict speculative safety $\Rightarrow$ **Speculative safety** $\Rightarrow$ **Speculative non-interference**

[Guarnieri et al., S&P 2020]

$\mathbf{P}_{\mathbf{spec}}(s)|_{\mathsf{ns}} = \mathbf{P}_{\mathbf{spec}}(s')|_{\mathsf{ns}}$

$\Rightarrow \mathbf{P}_{\mathbf{spec}}(s) = \mathbf{P}_{\mathbf{spec}}(s')$

↑ *LFENCE Injection*

↑ *Speculative load hardening*