Formal Approaches to Secure Compilation

Amal Ahmed

Northeastern University & Inria Paris

Secure Compilation

building compilers that ensure
security properties of source programs
are **preserved** in target programs

Java source code

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

Java source code

```java
package Bank;

public class Account{
 private int balance = 0;

 public void deposit( int amount ) {
  this.balance += amount;
 }
}
```

compiled to C target code

```c
typedef struct account_t {
 int balance = 0;
 void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
 a→balance += amount;
 return;
}
```

Java source code

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

- gap between source and target abstractions

- need some mechanism to hide balance in target

compiled to C target code

```
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a->balance += amount;
  return;
}
```

context

- how do we prove that compiler preserves security properties

... and how are source properties expressed

Security Properties as Program Equivalences

Security Properties as Program Equivalences

Example: Integrity

```
public proxy( callback : Unit → Unit )
      : Int {
  var secret = 0;
  callback();

  return 0;


}
```

```
public proxy( callback : Unit → Unit )
      : Int {
  var secret = 0;
  callback();
  if ( secret == 0 ) {
    return 0;
  }
  return 1;
}
```

Security Properties as Program Equivalences

Example: Confidentiality

```
private secret : Int = 0;

public setSecret() : Int {
  secret = 1;
  return 0;
}
```

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

Security Properties as Program Equivalences

Example: Unbounded vs. finite memory

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  for (Int i = 0; i < n; i++){
   new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  callback();
  // security-relevant code
  return 0;
}
```

Security Properties as Program Equivalences

Example: Memory Allocation Order

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return y;
}
```

This Talk ...

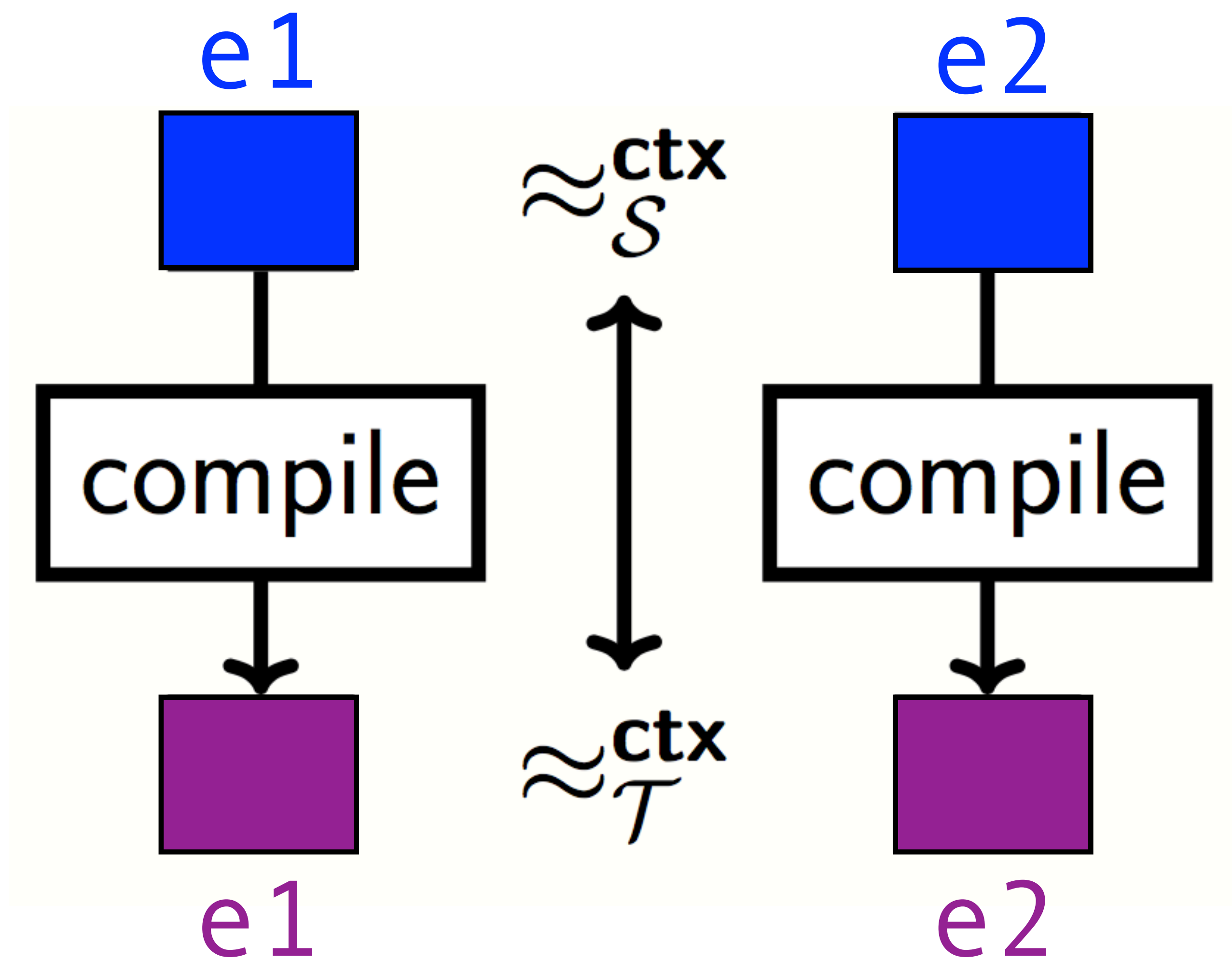
1. Preserving security properties expressed as some form of equivalence
   - contextual equivalence
     (different for C, ML, Gallina, DSLs)
   - observer-sensitive equivalence
     (e.g., noninterference in security-typed languages)
   - timing/resource-sensitive equivalence
     (e.g., security of constant-time code)

This Talk ...

1. Preserving security by preserving equivalence
2. Different compilation targets and threat models
   - is the target language typed or untyped?
   - what observations can the attacker make?
3. Different ways of enforcing secure compilation
   - static checking
   - dynamic checking (e.g., runtime monitoring, cryptographic & hardware enforcement)
4. Proof techniques
   - "back-translating" target attackers to source

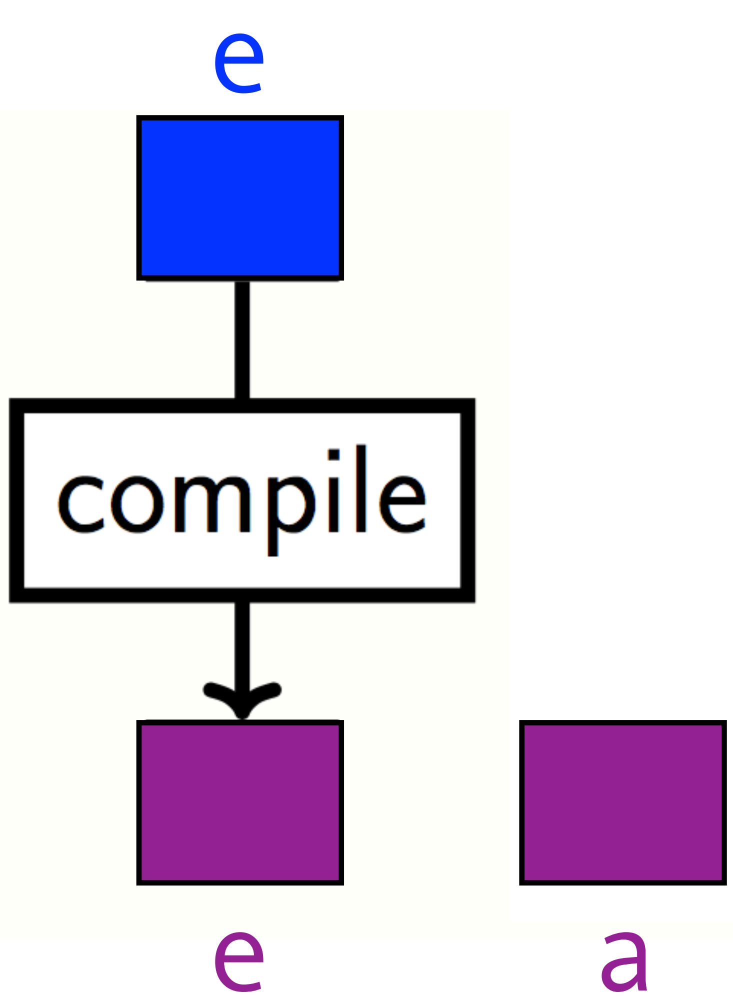
Fully Abstract Compilation

Preserve and reflect contextual equivalence



Fully Abstract Compilation

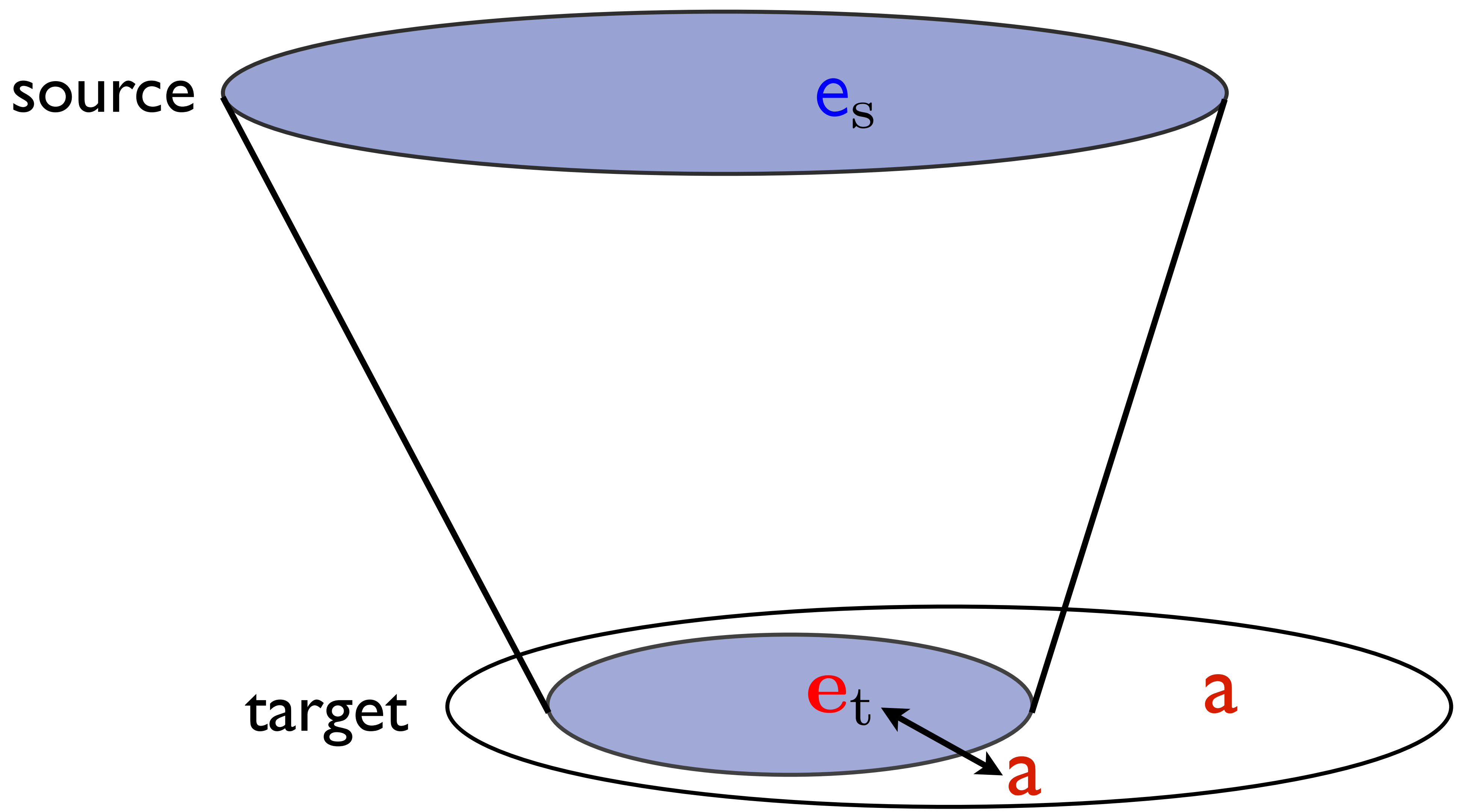
Preserve contextual equivalence



Guarantees that e will remain as secure as e when executed in arbitrary target-level contexts

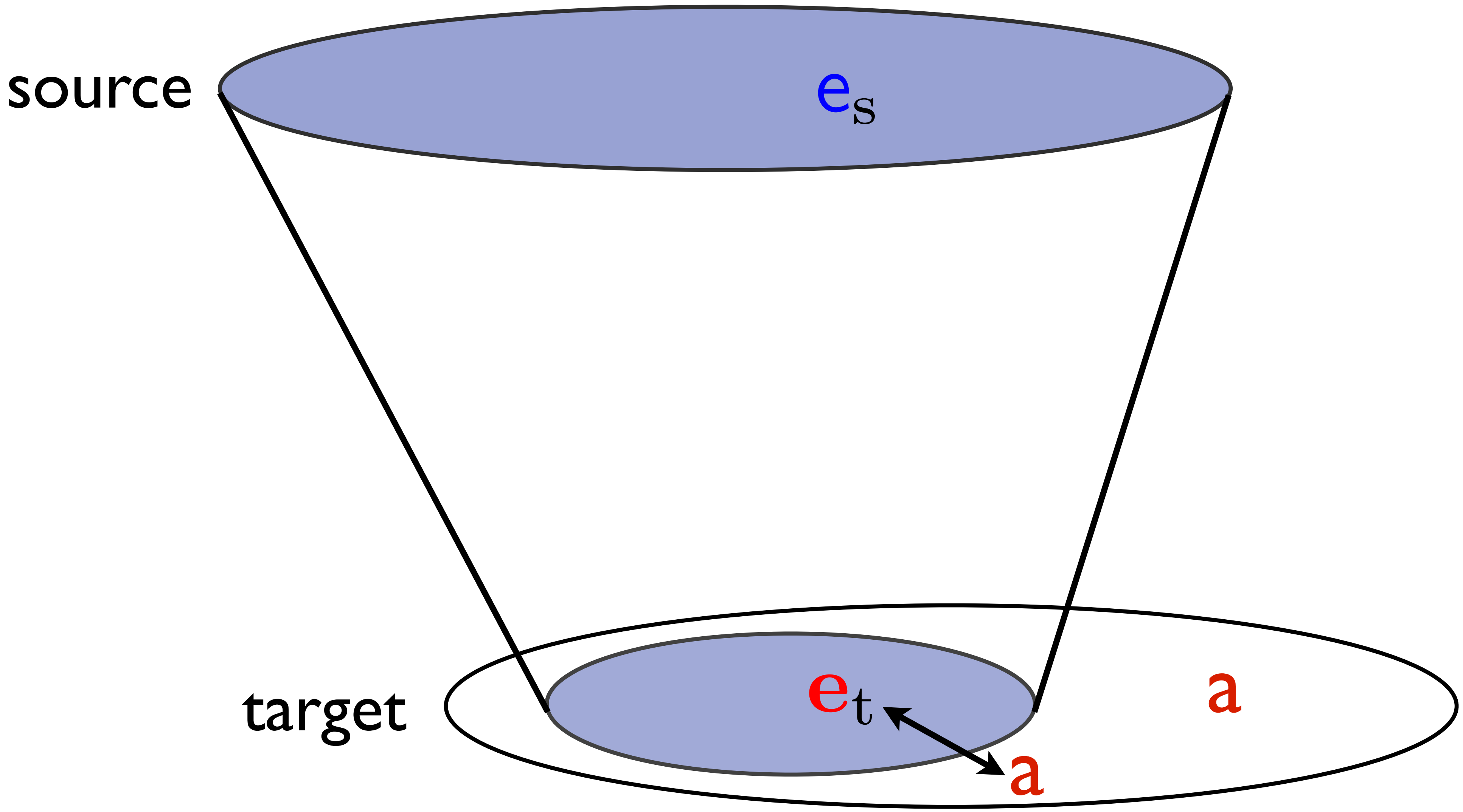
i.e. target contexts (attackers a) can make no more observations about e than a source context can make about e

Ensuring Full Abstr. / Secure Comp.



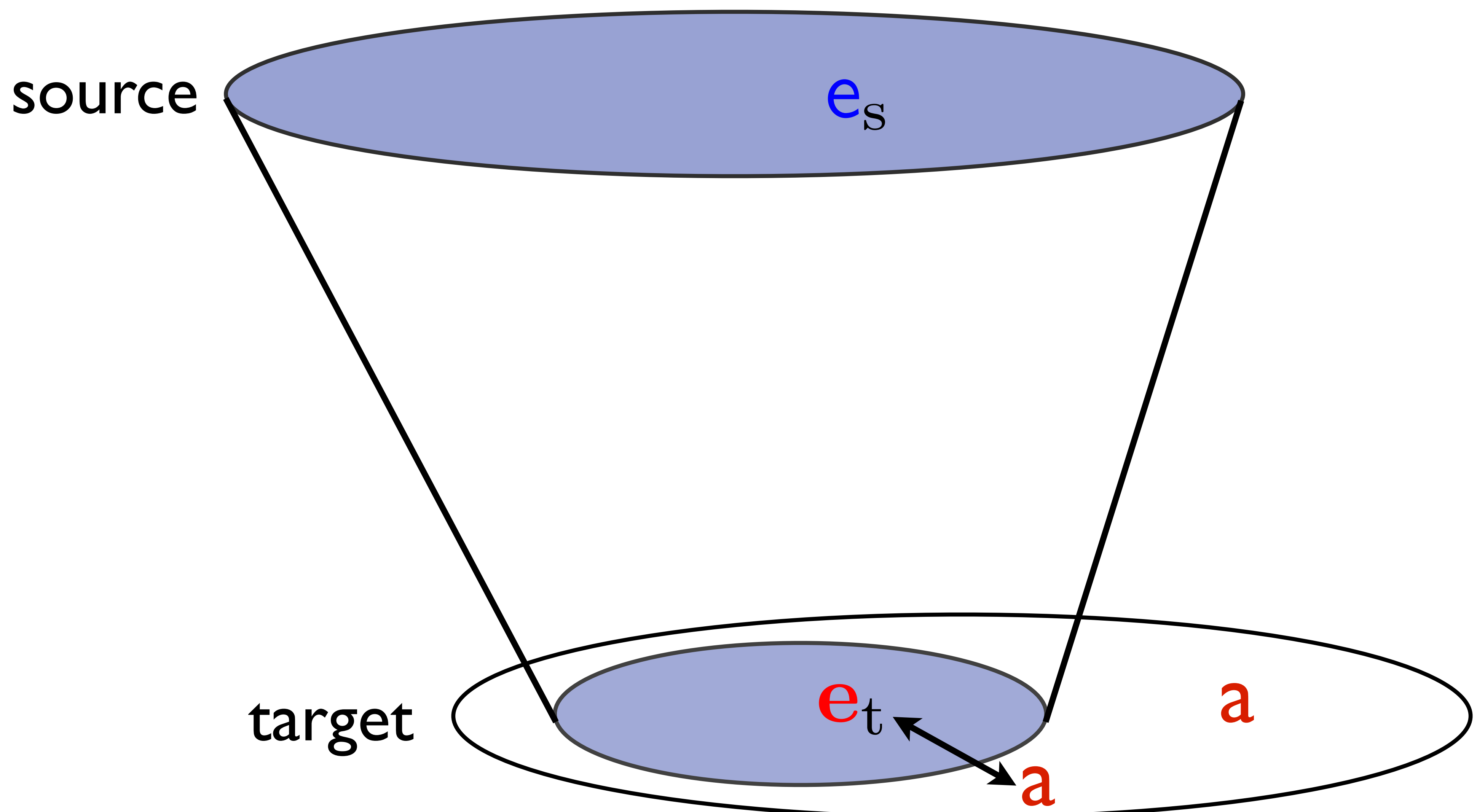
Must ensure that any a we link with behaves like some source context

Ensuring Full Abstr. / Secure Comp.



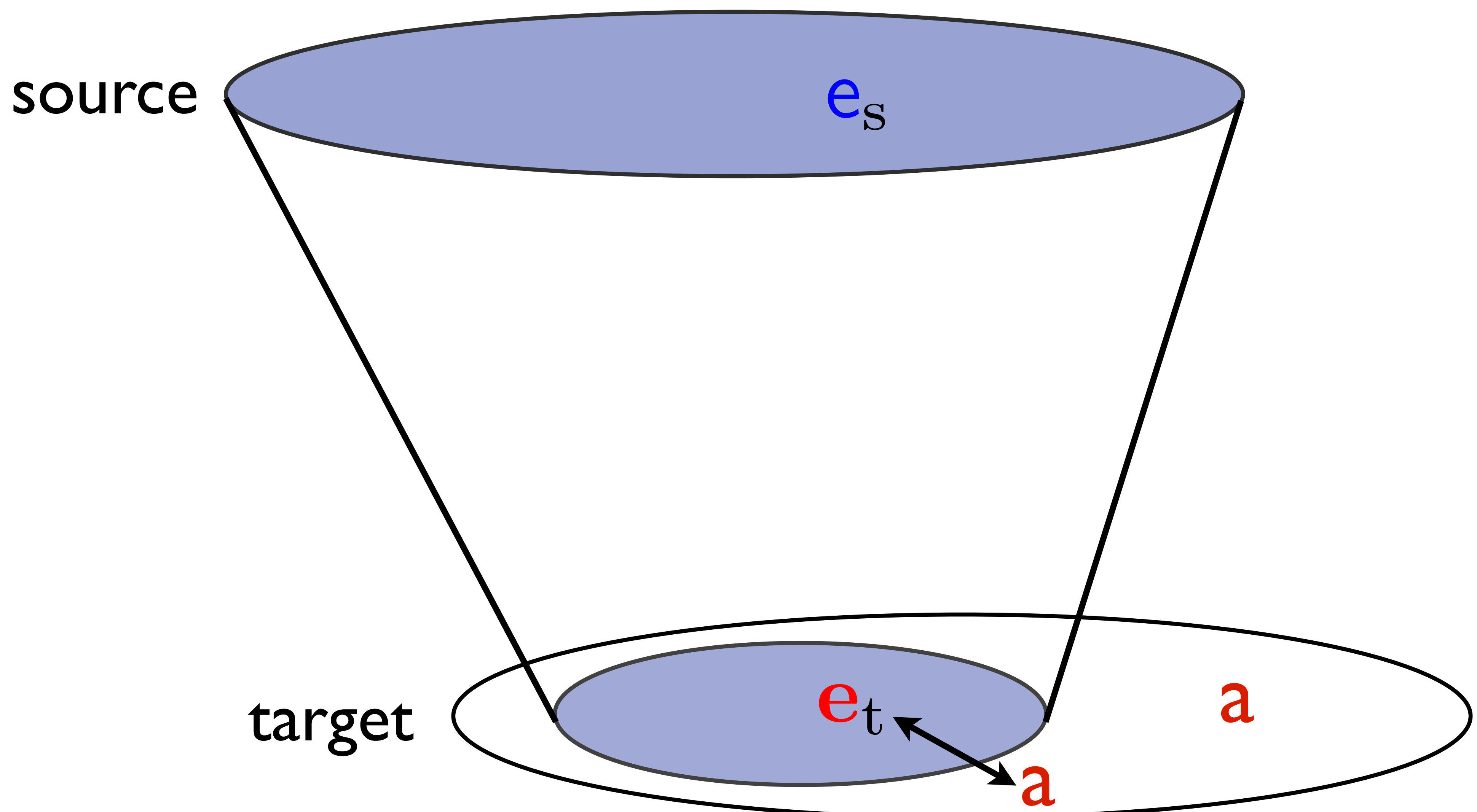
1. Add target features to the source language. **Bad!**

Ensuring Full Abstr. / Secure Comp.



1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act. **Performance cost**

Ensuring Full Abstr. / Secure Comp.



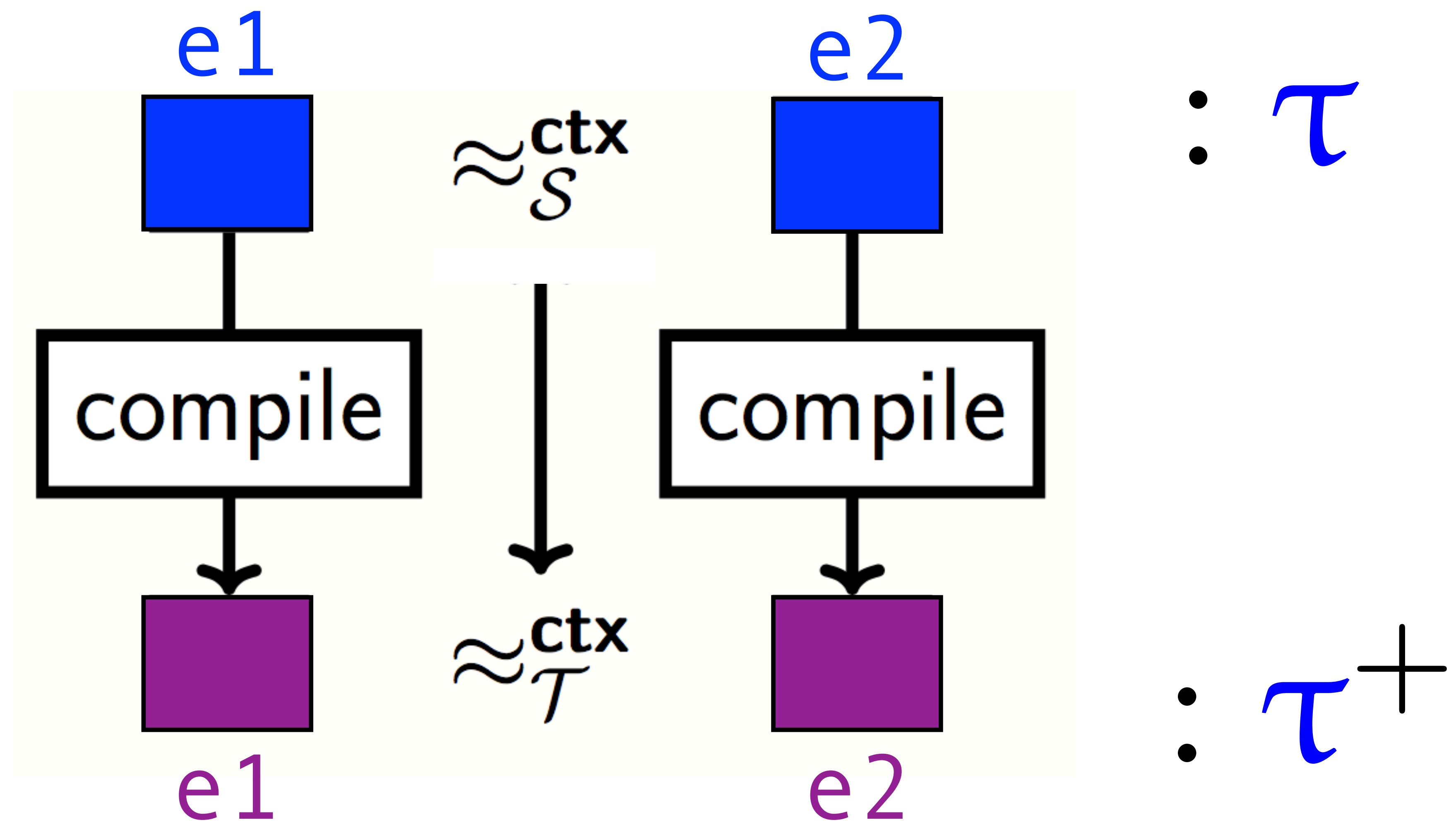
1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act. **Performance cost**
3. Static checks: rule our badly behaved code in the first place **Verification**

Type-Preserving Compilation

$$e : \tau \leadsto e : \tau^{+}$$

Type-Preserving Secure Compilation

Preserve well-typedness & equivalence



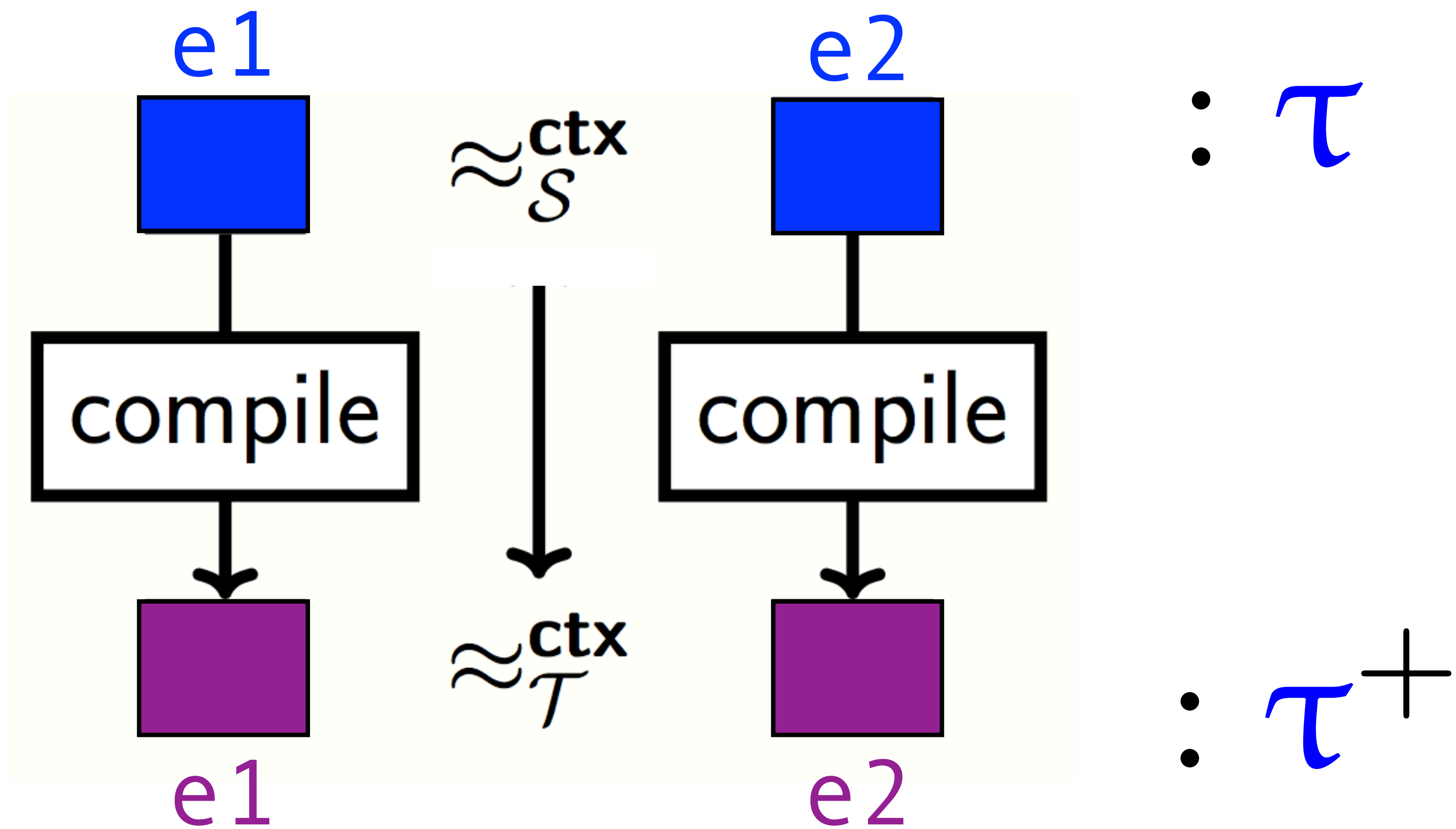
Type-Preserving Compilation

$$e : \tau \leadsto e : \tau^{+}$$

- System F to Typed Assembly Language *[Morrisett et al. POPL'97, TOPLAS'98]*
- Typed compilation of Featherweight Java to F-omega, private fields to existential type *[League et al. TOPLAS'02]*
- FINE (F# with refinement & affine types) to DCIL (dependent CIL) *[Chen et al. PLDI'10]*
- Security-type-preserving compilation from WHILE lang. to stack-based TAL (both languages satisfy noninterference). Extended to concurrent setting with thread creation, secure scheduler *[Barthe et al. 2007, 2010]*

Type-Preserving Secure Compilation

Preserve well-typedness & equivalence



Challenge: Proving Full Abstraction

Suppose $\Gamma \vdash e_1 : \tau \rightsquigarrow e_1$ and $\Gamma \vdash e_2 : \tau \rightsquigarrow e_2$

$$\Gamma \vdash e_1 \approx^{ctx}_{\mathrm{S}} e_2 : \tau$$

$$\downarrow$$

$$\Gamma^+ \vdash e_1 \approx^{ctx}_{\mathrm{T}} e_2 : \tau^+$$

Challenge: Proving Full Abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathsf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathsf{e}_2$

$\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau$

Given:

No C_{S} can distinguish e_1, e_2

$\downarrow$

$\Gamma^{+} \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathsf{e}_2 : \tau^{+}$

Challenge: Back-translation

1. If target is not more expressive than source, use the same language: back-translation can be avoided in lieu of *wrappers* between τ and τ^+

- Closure conversion: System F with recursive types *[Ahmed-Blume ICFP'08]*

- f* (STLC with refs, exceptions) to js* (encoding of JavaScript in f*) *[Fournet et al. POPL'13]*

Challenge: Back-translation

2. If target is more expressive than source

   (a) Both terminating: use back-translation by partial evaluation

   - Equivalence-preserving CPS from STLC to System F *[Ahmed-Blume ICFP'11]*
   - Noninterference for Free (DCC to Fω) *[Bowman-Ahmed ICFP'15]*

   (b) Both nonterminating: use ??
   back-trans by partial evaluation is not well-founded!

   Observation: if source lang. has recursive types, can write interpreter for target lang. in source lang.

Fully Abstract Closure Conversion

[New et al. ICFP'16]

Source: STLC + μ types

Target: System F + $\exists$ types + μ types + exceptions

First full abstraction result where target has exceptions but source does not.

Earlier work, due to lack of sufficiently powerful back-translation techniques, added target features to source.

Proof technique: **Universal Embedding**

- Untyped embedding of target in source
- Mediate between strongly typed source and untyped back-translation

Fully Abstract Closure Conversion

Source: STLC + μ types

[New et al. ICFP'16]

Target: System F + $\exists$ types + μ types + exceptions

Equivalent source terms, inequivalent in lang. with exceptions:

$$e_1 = \lambda f.\,(f\ \mathrm{true}; f\ \mathrm{false}; \langle\rangle) \qquad e_2 = \lambda f.\,(f\ \mathrm{false}; f\ \mathrm{true}; \langle\rangle)$$

$$C = \mathrm{catch}\ y = ([\cdot]\ (\lambda x.\,\mathrm{raise}\,x))\ \mathrm{in}\ y$$

$$C[e_1] \Downarrow \mathrm{true} \qquad C[e_2] \Downarrow \mathrm{false}$$

Idea: use modal type system at target to rule out linking with code that throws unhandled exceptions

Ensuring Full Abstraction via Types

[New et al. ICFP'16]

$$e_1 \approx_S^{ctx} e_2 : (\text{bool} \rightarrow 1) \rightarrow 1$$

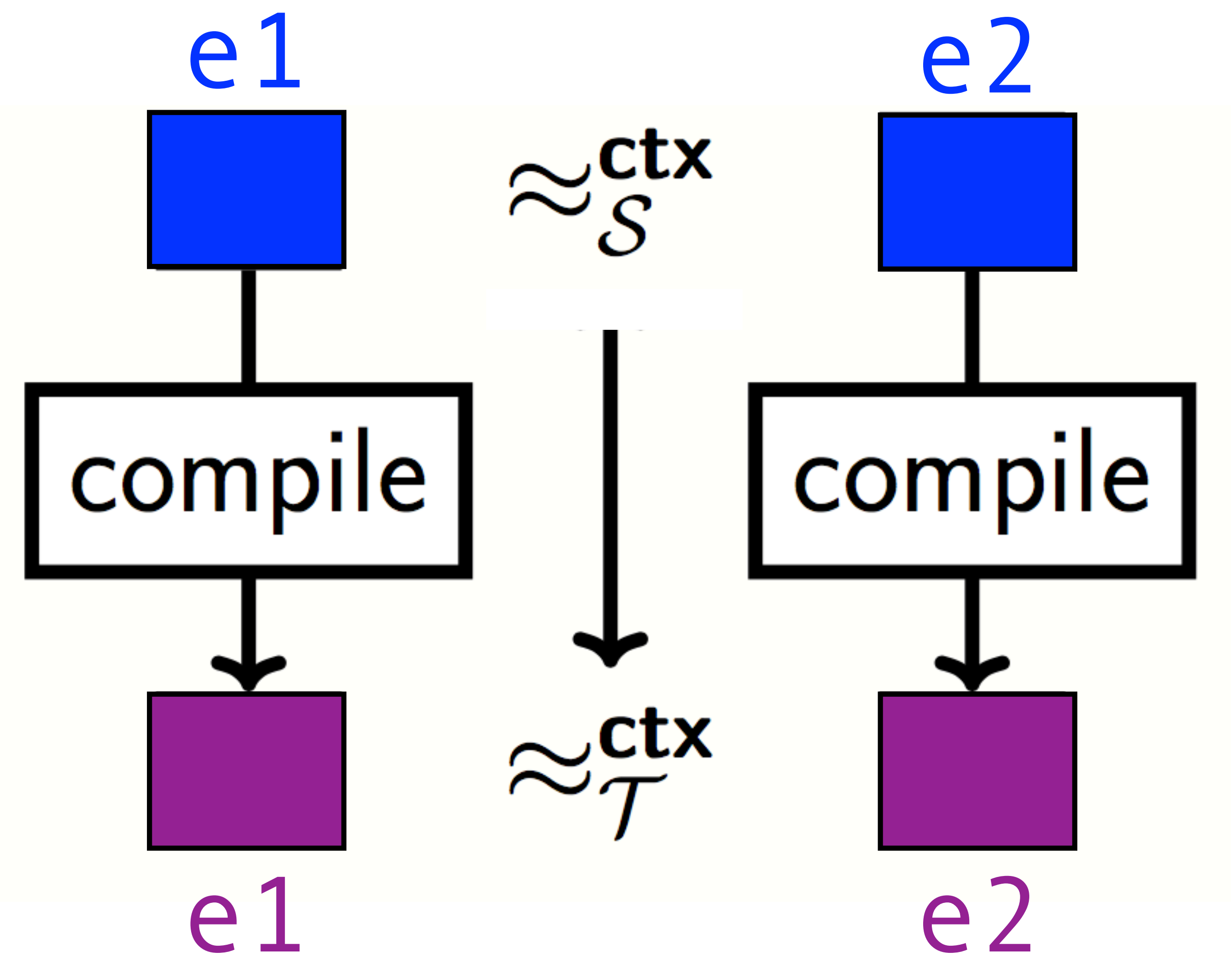
$$(\text{bool} \rightarrow \text{E } 0\ 1) \rightarrow \text{E } 0\ 1$$

$$\neq$$

$$C : (\text{bool} \rightarrow \text{E bool } 1) \rightarrow \text{E bool } 1$$

$$C = ([\cdot]\ (\lambda(x : \text{bool}).\ \text{raise } x)$$

Dynamic Secure Compilation



Dynamic Secure Compilation

1. Cryptographically enforced: concurrent, distributed langs.
   - Join calculus to Sjoin with crypto primitives, preserves and reflect weak bisimulation *[Abadi et al. S&P'99, POPL'00, I&C'02]*
   - Pi-calculus to Spi-calculus *[Bugliesi and Giunti, POPL'07]*
   - F# with session types to F# with crypto primitives *[Corin et al., J. Comp. Security'08]*
   - Distributed WHILE lang. with security levels to WHILE with crypto and distributed threads *[Fournet et al, CCS'09]*
   - *TINYLINKS* distributed language to *F7* (ML w. refinement types), preserves data and control integrity *[Baltopoulos and Gordon, TLDI'09]*

Dynamic Secure Compilation

2. Dynamic Checks / Runtime Monitoring
   - STLC with recursion to untyped lambda-calc, proved fully abstract using *approximate back-translation*. Types erased and replaced w. dynamic checks. *[Devriese et al. POPL'16]*
   - f* (STLC with refs, exceptions) to js* (encoding of JavaScript in f*). Defensive wrappers perform dynamic type checks on untyped js* *[Fournet et al. POPL'13]*
   - Lambda-calc to VHDL digital circuits, run-time monitors check that external code respects expected communication protocol *[Ghica and Al-Zobaidi ICE'12]*

Dynamic Secure Compilation

3. Memory Protection Techniques
   (a) Address space layout randomization (ASLR)
      - STLC w. abstract memory, to target with concrete memory; show probabilistic full abstraction for large memory *[Abadi-Plotkin TISSEC'12]*
      - Added dynamic alloc, h.o. refs, call/cc, testing hash of reference, to target with probref to reverse hash *[Jagadeesan et al. CSF'11]*

Dynamic Secure Compilation

3. Memory Protection Techniques

   (b) Protected Module Architectures (PMAs) (e.g., Intel SGX)

   protected memory with code and data sections, and unprotected memory

   - Secure compilation of an OO language (with dynamic allocation, exceptions, inner classes) to PMA; proved fully abstract using trace semantics. Objects allocated in secure memory partition *[Patrignani et al. TOPLAS'15]*

Dynamic Secure Compilation

3. Memory Protection Techniques
   (c) PUMP Machine architecture tracks meta-data, registers and memory locations have tags, checked during execution
      - Secure compartmentalizing compiler with mutually distrustful compartments that can be compromised by attacker. OO lang to RISC with micro policies *[Juglaret et al. 2015]*

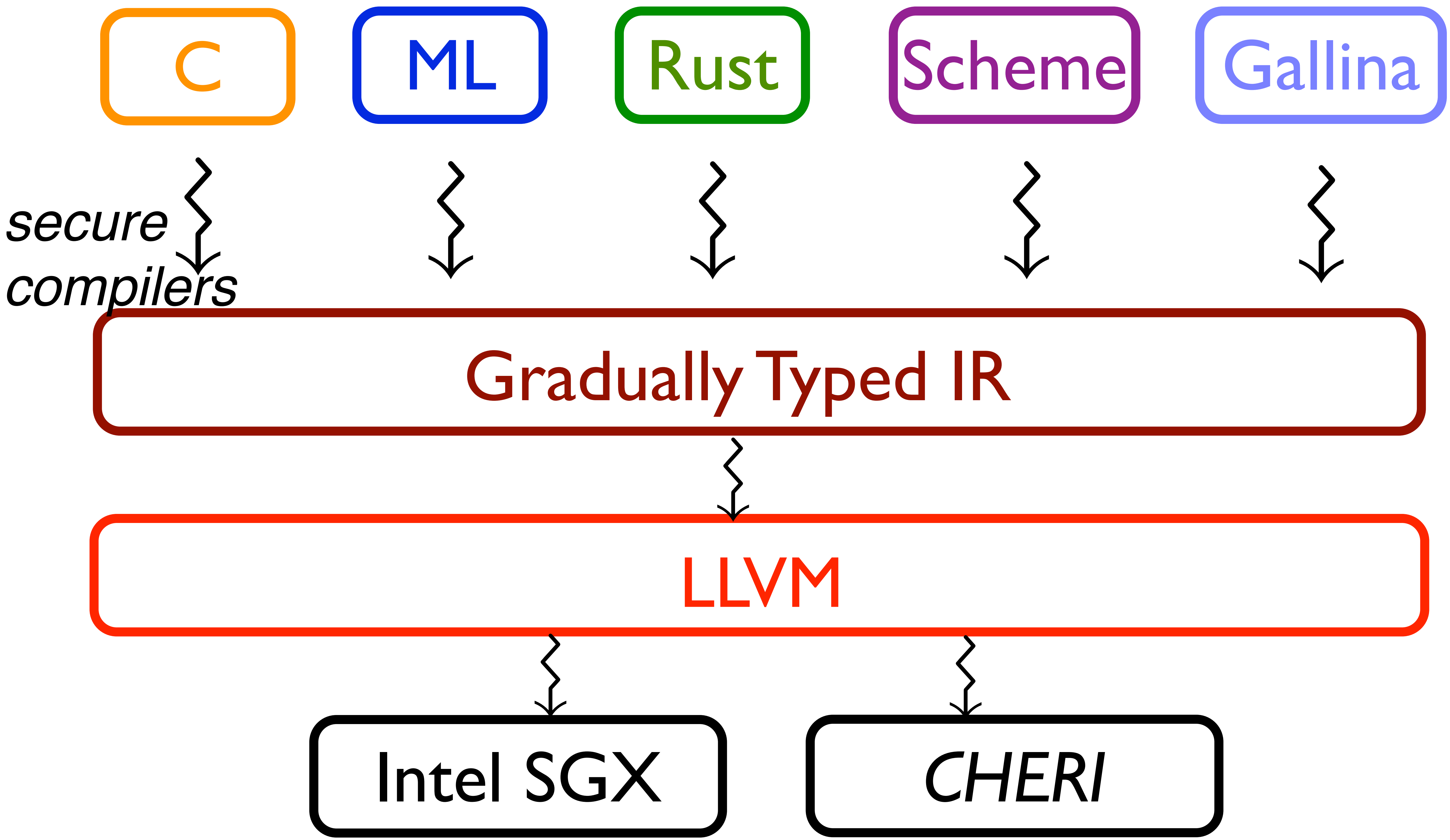
Dynamic Secure Compilation

4. Capability Machines
   - C to CHERI-like capability machine: give calling convention that enforces well-bracketed control-flow and encapsulation of stack frames using local capabilities; proved using logical relation *[Skorstengaard et al. ESOP'18]*

Secure Compilation: Open Problems

1. Need languages / DSLs that allow programmers to easily express security intent.
   - Compilers need to know programmer intent so they can *preserve* that intent (e.g., FaCT, a DSL for constant-time programming *[Cauligi et al. SecDev'17]*
2. Performant secure compilers
   - Static enforcement avoids performance overhead, could run on stock hardware; need richly typed compiler IRs
   - Dynamic enforcement when code from static/dynamic and safe/unsafe languages interoperates (e.g., h/w support)
   - Better integration of static and dynamic enforcement...

- Better integration of static and dynamic enforcement...



Secure Compilation: Open Problems

3. Preserve (weaker) security properties than contextual equiv.
    - Full abstraction may preserve too many incidental/ unimportant equivalences and has high overhead for dynamic enforcement
4. Security against side-channel attacks
    - Requires reasoning about side channels in source language, which is cumbersome. Can DSLs help?
    - *Correctness-Security Gap in Compiler Optimizations [D'Silva et al. LangSec'15].* Make compilers aware of programmers' security intent to take into account for optimizations.

Secure Compilation: Open Problems

5. Cryptographically enforced secure compilation
   - e.g., Obliv-C ensures memory-trace obliviousness using garbled circuits, but no formal proof that it is secure
6. Concurrency (beyond message-passing, targeting untyped multi-threaded assembly)
7. Easier proof techniques and reusable proof frameworks (trace-based techniques, back-translation, logical relations, bisimulation)

Final Thoughts

It's an exciting time to be working on secure compilation!

- Numerous advances in the last decade, in PL/formal methods and systems/security.

- For performant secure compilers, will need to integrate static and dynamic enforcement techniques, and provide programmers with better languages for communicating their security intent to compilers.