Program Equivalences and Fully Abstract Compilation

summer semester 18-19, block

Marco Patrignani[1,2]

Stanford University

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

What is a Compiler

What is a Compiler

https://en.wikipedia.org/wiki/Compiler

What is a Compiler

https://en.wikipedia.org/wiki/Compiler

In this course:

- only care about the code generation phase

What is a Compiler

https://en.wikipedia.org/wiki/Compiler

In this course:

- only care about the code generation phase
- takes programs written in a
  source language S

What is a Compiler

https://en.wikipedia.org/wiki/Compiler

In this course:

- only care about the code generation phase
- takes programs written in a source language S
- output programs written in a target language T

What is a Compiler

https://en.wikipedia.org/wiki/Compiler

In this course:

- only care about the code generation phase
- takes programs written in a source language S
- output programs written in a **target language T**
- it is a function from S to $\mathbf{T}$: $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$

What is a Compiler

https://en.wikipedia.org/wiki/Compiler

In this course:

- only care about the code generation phase
- t
  s
- c
  t
- it is a function from S to $\mathbf{T}$: $[\![\cdot]\!]^{S}_{\mathbf{T}}$

Gross simplification:

- PL perspective on this subject (will remain for the whole course)

Example: Insecure Compilation

```
public class Account
  private int balance = 0;

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

Java
source

Example: Insecure Compilation

```java
public class Account
  private int balance = 0;

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

Java
source

No access to `balance` from outside `Account`

Example: Insecure Compilation

```java
public class Account
  private int balance = 0;

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

Java
source

No access to `balance` from outside `Account`

enforced by the language

Example: Insecure Compilation

```java
public class Account
  private int balance = 0;

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

Java

source

```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;
}
```

C

target

Example: Insecure Compilation

```
publi
  priv

  publ
    th
```

Pointer arithmetic in `C` leads to security violation: undesired access to `balance`

Security is not preserved.

```
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;
}
```

Secure Compilation

- **Q:** what does it mean to preserve security properties across compilation?

Secure Compilation

- **Q:** what does it mean to preserve security properties across compilation?
- long standing question

Secure Compilation

- **Q:** what does it mean to preserve security properties across compilation?

- long standing question
- many anwers have been given, we focus on the formal ones

Secure Compilation

- **Q:** what does it mean to preserve security properties across compilation?

- long standing question
- many anwers have been given, we focus on the formal ones
- conceptually:

  “take what was secure in the source and make it as secure in the target”

Secure Compilation

- **Q:** what does it mean to preserve security properties across compilation?

Even more questions!

- how do we identify (or specify) what is secure in the source?
- how do we preserve the meaning of a security property?

Example: Confidentiality

Confidential: adjective

> *spoken, written, acted on, etc., in strict privacy or secrecy; secret:*

Example: Confidentiality

Confidential: adjective

> *spoken, written, acted on, etc., in strict privacy or secrecy; secret:*

```
private secret : Int = 0;

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

Java
source

Example: Confidentiality

Confidential: adjective

- **Q:** how do we know that `secret` is confidential?

```
priva

publi
  secr
  retu
}
```

Example: Confidentiality

Confidential: adjective

```
priva
publi
  secr
  retu
}
```

- **Q:** how do we know that `secret` is confidential?

- Type annotations
- Program verification
- ...
- Behaviour analysis
- Program equivalences

Program Equivalence

- a possible way to know what is secure in a program

Program Equivalence

- a possible way to know what is secure in a program
- useful tool to answer many questions posed about programming languages

Quiz: Are these Equivalent Programs?

```
public Bool getTrue( x : Bool )
  return true;
```

```
public Bool getTrue( x : Bool )
  return x or true;
```

```
public Bool getTrue( x : Bool )
  return x and false;
```

```
public Bool getTrue( x : Bool )
  return false;
```

```
public Bool getFalse( x : Bool )
  return x and true;
```

- P1
- P2
- P3
- P4
- P5

Quiz: Are these Equivalent Programs?

```
public Bool getTrue( x : Bool )
  return true;
```

```
public Bool getTrue( x : Bool )
  return x or true;
```

```
public Bool getTrue( x : Bool )
  return x and false;
```

```
public Bool getTrue( x : Bool )
  return false;
```

```
public Bool getFalse( x : Bool )
  return x and true;
```

- P1
- P2

=

- P3
- P4

=

- P5

Quiz: Are these Equivalent Programs?

```
public Bool getTrue( x : Bool )
  return
```

```
public
  return
```

```
public
  return
```

Program equivalences (generally) are:

- reflexive
- transitive
- symmetric

aka: relations

```
public Bool getTrue( x : Bool )
  return false;
```

```
public Bool getFalse( x : Bool )
  return x and true;
```

P1

P4

P5

Program Equivalence

- **Q:** When are two programs equivalent?

Program Equivalence

- **Q:** When are two programs equivalent?

- When they behave the same

Program Equivalence

- **Q:** When are two programs equivalent?
- When they behave the same even if they are different

Program Equivalence

- **Q:** When are two programs equivalent?

- When they behave the same even if they are different
- Semantics (behaviour) VS Syntax (outlook)

Program Equivalence

Defining a security property using program equivalence:

Program Equivalence

Defining a security property using program equivalence:
to find two programs that, albeit syntactically different, both behave in a way that respects the property, no matter how they are used.

Example: Confidentiality as P.Eq.

Example: Confidentiality as P.Eq.

```
private secret : Int = 0;

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

Example: Confidentiality as P.Eq.

```
private secret : Int = 0;

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

```
private secret : Int = 0;

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

Example: Confidentiality as P.Eq.

```
private secret : Int = 0;

publi
  secr
  retu
}
```

```
priva

publi
  secr
  retu
}
```

With a Java-like semantics, `secret` is never accessed from outside.
With a C-like semantics, `secret` can be accessed from outisde.

Example: Confidentiality as P.Eq.

```
private secret : Int = 0;

publi
  secr
  retu
}
```

```
priva

publi
  secr
  retu
}
```

With a Java-like semantics, `secret` is never accessed from outside.

With a C-like semantics, `secret` can be accessed from outisde.

The Language defines how to reason

(it's what programmers already do!)

Example: Integrity as P.Eq.

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

Integrity: internal consistency or lack of corruption in data.

```
publi
  var secret = 0;
  callback();
  return 0;
}
```

Example: Integrity as P.Eq.

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

```
publi
  var secret = 0;
  callback();
  return 0;
}
```

Integrity: internal consistency or lack of corruption in data.
Maintenance of invariants

Example: Integrity as P.Eq.

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

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

Example: Memory Allocation as P.Eq.

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

Example: Memory Allocation as P.Eq.

```
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;
}
```

Example: Memory Allocation as P.Eq.

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

```
publi
  var
  var
  return y;
}
```

Guessing addresses in memory leads to common exploits: ROP, return to libc, violation of ASLR …

Example: Memory Size as P.Eq.

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

Example: Memory Size as P.Eq.

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

```
public kernel( n : Int, callback : Unit → Unit ) :
    Int {



  callback();
  return 0;
}
```

Expressing Program Equivalence

Contextual Equivalence

Expressing Program Equivalence

Contextual Equivalence

(also, observational equivalence)

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

$$P_1 \simeq_{ctx} P_2 \overset{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}[P_1]\downarrow \iff \mathfrak{C}[P_2]\downarrow$$

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}[P_1]\downarrow \iff \mathfrak{C}[P_2]\downarrow$$

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}[P_1]\downarrow \iff \mathfrak{C}[P_2]\downarrow$$

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

$$P_1 \simeq_{ctx} P_2 \overset{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}\left[P_1\right]\downarrow \iff \mathfrak{C}\left[P_2\right]\downarrow$$

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

$$P_1 \simeq_{ctx} P_2 \overset{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}[P_1]\downarrow \iff \mathfrak{C}[P_2]\downarrow$$

Contextual Equivalence (CEQ)

Two p... exter... obse... ...at

- the external observer $\mathfrak{C}$ is generally called context
- it is a program, written in the same language as P_1 and P_2
- it is the same program $\mathfrak{C}$ interacting with both P_1 and P_2 in two different runs
- so it cannot express out of language attacks (e.g., side channels)

Contextual Equivalence (CEQ)

Two programs are equivalent if no matter what exter… obse…

- interaction means link and run together (like a library)

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}.\ \mathfrak{C}[P_1]\downarrow \iff \mathfrak{C}[P_2]\downarrow$$

Contextual Equivalence (CEQ)

Two ... *at*
exter...
obse...

- distinguishing means: terminate with different values
- the observer basically asks the question: *is this program P_1?*
- if the observer can find a way to distinguish P_1 from P_2, it will return `true`, otherwise `false`
- often we use *divergence* and *termination* as opposed to this boolean termination

Example: CEQ

```
private secret : Int = 0; //P1
public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

Java

```
private secret : Int = 0; //P2
public setSecret( ) : Int {
  secret = 1;
  return 0;
}
```

Java

```
// Observer P in Java
public static isItP1( ) : Bool {
  Secret.getSecret();
  ...
}
```

Java

Example: CEQ

```
typedef struct secret { // P1
  int secret = 0;
  void ( *setSec ) ( struct Secret* ) = setSec;
} Secret;
void setSec( Secret* s ) { s→secret = 0; return; }
```

C

```
typedef struct secret { // P2
  int secret = 0;
  void ( *setSec ) ( struct Secret* ) = setSec;
} Secret;
void setSec( Secret* s ) { s→secret = 1; return; }
```

C

```
// Observer P in C
int isItP1( ){
  struct Secret x;
  sec = &x + sizeof(int);
  if *sec == 0 then return true else return false
}
```

C

Inequivalences as Security Violations

- if the target programs are not equivalent ($\nsimeq_{ctx}$) then the intended security property is violated

Inequivalences as Security Violations

- if the target programs are not equivalent ($\not\simeq_{ctx}$) then the intended security property is violated

When does inequivalences escape the (compiler) programmer's reasoning?

Inequivalences as Security Violations

- if the target programs are not equivalent ($\nsimeq_{ctx}$) then the intended security property is violated

When does inequivalences escape the (compiler) programmer's reasoning?

1. if languages have complex features

Inequivalences as Security Violations

- if the target programs are not equivalent ($\nsimeq_{ctx}$) then the intended security property is violated

When does inequivalences escape the (compiler) programmer's reasoning?

1. if languages have complex features
2. if there are more languages involved (e.g., multiple target languages)

Preserving Equivalences in Compilation

Back to our question …

- **Q:** what does it mean to preserve security properties across compilation?

Preserving Equivalences in Compilation

Back to our question …

- **Q:** what does it mean to preserve security properties across compilation?

A possible answer:

- Given source equivalent programs (which have a security property), compile them into equivalent target programs

Preserving Equivalences in Compilation

Back

- Assumption 1: the security property is captured in the source by program equivalence

A pos

Preserving Equivalences in Compilation

- Assumption 1: the security property is captured in the source by program equivalence
- **Crucial**: being equivalent in the target means contextual equivalence w.r.t. target observers (i.e., target programs)

Preserving Equivalences in Compilation

- Assumption 1: the security property is captured in the source by program equivalence
- **Crucial**: being equivalent in the target means contextual equivalence w.r.t. target observers (i.e., target programs)
- These are the attackers in the secure compilation setting

Fully Abstract Compilation

Fully Abstract Compilation

A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs

$$
\begin{aligned}
\llbracket\cdot\rrbracket^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC\#1} \overset{\text{def}}{=} & \ \forall \mathsf{P}_1, \mathsf{P}_2 \\
\text{if } & \ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \\
\text{then } & \ \llbracket\mathsf{P}_1\rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket\mathsf{P}_2\rrbracket^{\mathsf{S}}_{\mathbf{T}}
\end{aligned}
$$

Fully Abstract Compilation

A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs

$$\llbracket\cdot\rrbracket^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC\#1} \overset{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\text{if } \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$$

$$\text{then } \llbracket\mathsf{P}_1\rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket\mathsf{P}_2\rrbracket^{\mathsf{S}}_{\mathbf{T}}$$

Fully Abstract Compilation

A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs

$$\llbracket\cdot\rrbracket^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC\#1} \overset{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\text{if } \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$$

$$\text{then } \llbracket\mathsf{P}_1\rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket\mathsf{P}_2\rrbracket^{\mathsf{S}}_{\mathbf{T}}$$

Fully Abstract Compilation

A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs

$$\llbracket \cdot \rrbracket^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC\#1} \stackrel{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\text{if } \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$$

$$\text{then } \llbracket \mathsf{P}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket \mathsf{P}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}$$

Fully Abstract Compilation

A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs

$$
\begin{aligned}
[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC\#1} \stackrel{\text{def}}{=}\ & \forall \mathsf{P}_1, \mathsf{P}_2 \\
\text{if } & \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \\
\text{then } & [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}
\end{aligned}
$$

Fully Abstract Compilation

A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs

$$
\begin{aligned}
\llbracket \cdot \rrbracket^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC\#1} \overset{\text{def}}{=}\ & \forall \mathsf{P}_1, \mathsf{P}_2 \\
\text{if}\ \ & \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \\
\text{then}\ \ & \llbracket \mathsf{P}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket \mathsf{P}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}
\end{aligned}
$$

Right?

Fully Abstract Compilation

Wrong.

Fully Abstract Compilation

Wrong.

An empty translation would fit FAC#1!

Fully Abstract Compilation

Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct.

Roughly, turn $\Rightarrow$ into a $\iff$:

$$[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC} \overset{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

Fully Abstract Compilation

Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct.
Roughly, turn $\Rightarrow$ into a $\iff$:

$$[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC} \stackrel{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

Note: $\Leftarrow$ does not mean compiler correctness in the general sense, but it's a consequence

Fully Abstract Compilation

Wrong.

An empty translation would fit FAC#1!

We n

Roug

Criteria need to be precise and general.

$[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ is FAC $\overset{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

Note: $\Leftarrow$ does not mean compiler correctness in the general sense, but it's a consequence

Remarks on Fully Abstract Compilation

- widely adopted since 1999

Remarks on Fully Abstract Compilation

- widely adopted since 1999
- intuition that was circulating for 10 years

Remarks on Fully Abstract Compilation

- widely adopted since 1999
- intuition that was circulating for 10 years
- only preserves security property expressed as program equivalence

Remarks on Fully Abstract Compilation

- widely adopted since 1999
- intuition that was circulating for 10 years
- only preserves security property expressed as program equivalence
- not the silver bullet: we will see shortcomings of fully abstract compilation

Fully Abstract Compilation

$$[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC} \stackrel{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

Fully Abstract Compilation

$$[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC} \stackrel{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

- break the $\iff$:
  1. $\Rightarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$
  2. $\Leftarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$
- point 2 (should) follow from compiler correctness

Fully Abstract Compilation

$\llbracket \cdot \rrbracket^{\mathsf{S}}_{\mathbf{T}}$ is FAC $\stackrel{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff \llbracket \mathsf{P}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket \mathsf{P}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}$$

- break the $\iff$:
  1. $\Rightarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow \llbracket \mathsf{P}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket \mathsf{P}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}$
  2. $\Leftarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ \llbracket \mathsf{P}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket \mathsf{P}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$
- point 2 (should) follow from compiler correctness
- point 1 is tricky, because of $\simeq_{ctx}$ and its $\forall \mathfrak{C}$

Fully Abstract Compilation

$[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ is FAC $\stackrel{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$

- break the $\iff$:
    1. $\Rightarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$
    2. $\Leftarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$
- point 2 (should) follow from compiler correctness
- point 1 is tricky, because of $\simeq_{ctx}$ and its $\forall \mathfrak{C}$

This structure is called a backtranslation

Backtranslations

- Context-based: relies on the structure of the context
- Trace-based: relies on trace semantics

Backtranslations

- Context-based: relies on the structure of the context
  when source and **target** contexts are similar
- Trace-based: relies on trace semantics

Backtranslations

- Context-based: relies on the structure of the context
  when source and $\mathbf{target}$ contexts are similar
- Trace-based: relies on trace semantics
  when there is a large abstraction gap
  between source and $\mathbf{target}$

Trace Semantics

- we replace $\simeq_{ctx}$ with something equivalent

Trace Semantics

- we replace $\simeq_{ctx}$ with something equivalent
- but simpler to reason about

Trace Semantics

- we replace $\simeq_{ctx}$ with something equivalent
- but simpler to reason about
- a semantics that abstracts from the context (observer)

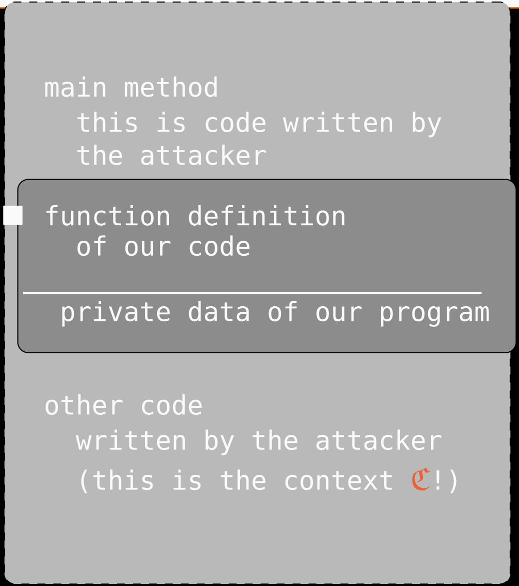
Trace Semantics

- we replace $\simeq_{ctx}$ with something equivalent
- but simpler to reason about
- a semantics that abstracts from the context (observer)
- and still describes the behaviour of a program precisely

Trace Semantics

- we replace $\simeq_{ctx}$ with something equivalent
- but simpler to reason about
- a semantics that abstracts from the context (observer)
- and still describes the behaviour of a program precisely
- a trace semantics

Traces for a program



- interest in the behaviour of our code (component)

Traces for a program

```
main method
  this is code written by
  the attacker

function definition
  of our code
_____________________________
 private data of our program

other code
  written by the attacker
  (this is the context ℭ!)
```

- interest in the behaviour of our code (component)
- need to consider the *rest*

Traces for a program

```
main method
  this is code written by
  the attacker

function definition
  of our code

 private data of our program

other code
  written by the attacker
  (this is the context ℭ!)
```

- interest in the behaviour of our code (component)
- need to consider the *rest*

Trace Semantics for Our Program

main method
this is code written by
the attacker

function definition
of our code

private data of our program

other code
written by the attacker
(this is the context $\mathfrak{C}$!)

- disregard the rest

Trace Semantics for Our Program

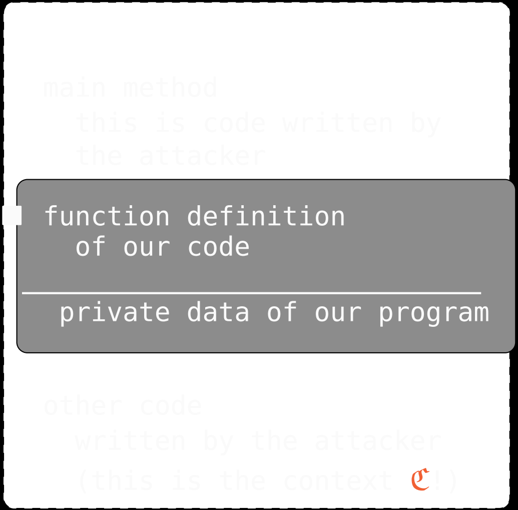
```
main method
  this is code written by
  the attacker

function definition
  of our code
_____________________________
 private data of our program

other code
  written by the attacker
  (this is the context ℭ!)
```

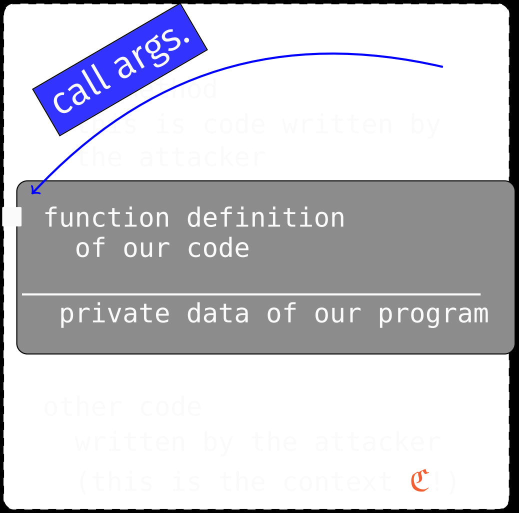
- disregard the rest

Trace Semantics for Our Program



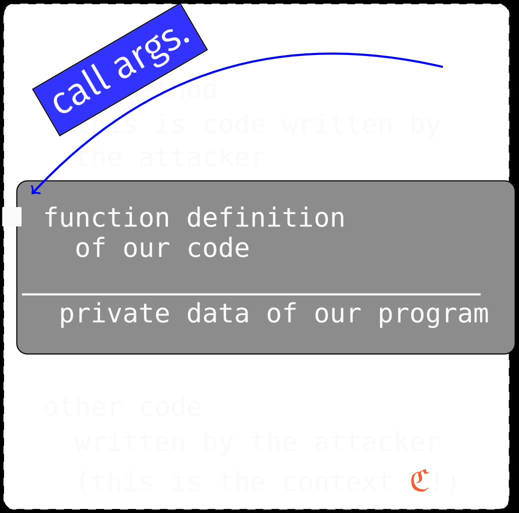
- disregard the rest
- abstract its behaviour from the component perspective:

Trace Semantics for Our Program



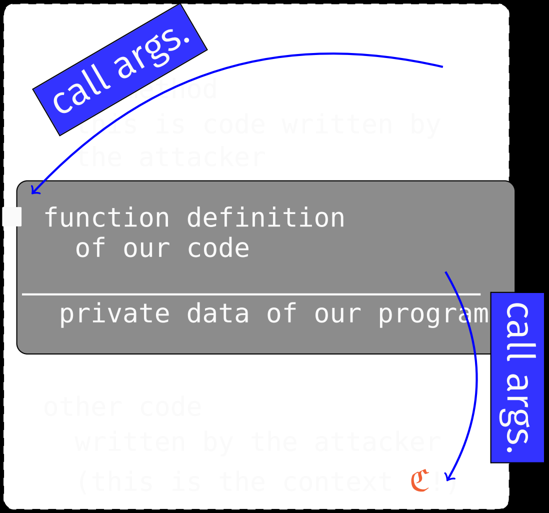
- disregard the rest
- abstract its behaviour from the component perspective:
  1. jump to an entry point ■

Trace Semantics for Our Program



- disregard the rest
- abstract its behaviour from the component perspective:
    1. jump to an entry point ■
- abstract the component behaviour from the rest perspective:

Trace Semantics for Our Program



- disregard the rest
- abstract its behaviour from the component perspective:
    1. jump to an entry point ■
- abstract the component behaviour from the rest perspective:
    1. call/return

Trace Semantics

- semantics for partial programs (component)

Trace Semantics

- semantics for partial programs (component)
- relies on the operational semantics

Trace Semantics

- semantics for partial programs (component)
- relies on the operational semantics
- denotational: describes the behaviour of a component as sets of traces

Trace Semantics

- semantics for partial programs (component)
- relies on the operational semantics
- denotational: describes the behaviour of a component as sets of traces
- a trace is (typically) a sequence of actions that describe how a component interacts with an observer

Trace Semantics

- semantics for partial programs (component)
- relies on the operational semantics
- denotational: describes the behaviour of a component as sets of traces
- a trace is (typically) a sequence of actions that describe how a component interacts with an observer
- without needing to specify the observer

Trace Semantics

- semantics for partial programs (component)
- relies on the operational semantics
- denotational: describes the behaviour of a component as sets of traces
- a trace is (typically) a sequence of actions that describe how a component interacts with an observer
- without needing to specify the observer
- indicated as $= \left\{ \overline{\alpha} \;\middle|\; C \stackrel{\overline{\alpha}}{\Longrightarrow} \_ \right\}$

Trace Actions

$$
\begin{aligned}
\textit{Labels} \quad & L ::= a \mid \epsilon \\
\textit{Observable actions} \quad & \alpha ::= \surd \mid g? \mid g! \\
\textit{Actions} \quad & g ::= \texttt{call}\ f\ (v) \mid \texttt{ret}\ v
\end{aligned}
$$

Traces for a program

We need to define:

- trace states (almost program states)
- labels that make traces
- rules for generating labels and traces $\cdots$
- the traces of a component $= \cdots$

Trace Equivalence

- all semantics yield a notion of equivalence

Trace Equivalence

- all semantics yield a notion of equivalence
- the operational semantics gives us contextual equivalence

$$C_1 \simeq_{ctx} C_2$$

Trace Equivalence

- all semantics yield a notion of equivalence
- the operational semantics gives us contextual equivalence

$$C_1 \simeq_{ctx} C_2$$

- trace semantics gives us trace equivalence

$$C_1 \stackrel{T}{=} C_2$$

Trace Equivalence

- all semantics yield a notion of equivalence
- the operational semantics gives us contextual equivalence

$$C_1 \simeq_{ctx} C_2$$

- trace semantics gives us trace equivalence

$$=$$

the traces of C_1 are the same of those of C_2

Trace Equivalence

- all semantics yield a notion of equivalence
- the operational semantics gives us contextual equivalence

$$C_1 \simeq_{ctx} C_2$$

- trace semantics gives us trace equivalence

$$\left\{ \overline{\alpha} \;\middle|\; C_1 \stackrel{\overline{\alpha}}{\Longrightarrow} \_ \right\} = \left\{ \overline{\alpha} \;\middle|\; C_2 \stackrel{\overline{\alpha}}{\Longrightarrow} \_ \right\}$$

the traces of C_1 are the same of those of C_2

Proofs about Trace Semantics

- any trace semantics won’t just work
- it needs to be
  correct and complete

Proofs about Trace Semantics

- any trace semantics won't just work
- it needs to be
  correct and complete

$$C_1 \simeq_{ctx} C_2 \iff C_1 \overset{\mathrm{T}}{=} C_2$$

Proofs about Trace Semantics

- any trace semantics won't just work
- it needs to be
  correct ($\Leftarrow$) and complete ($\Rightarrow$)

$$C_1 \simeq_{ctx} C_2 \iff C_1 \overset{\mathrm{T}}{=} C_2$$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow \forall \mathfrak{C}.\ \mathfrak{C}\left[\llbracket \mathsf{C}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\!\downarrow \iff \mathfrak{C}\left[\llbracket \mathsf{C}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\!\downarrow$
- unfold $\simeq_{ctx}$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists \mathfrak{C}.\ \mathfrak{C}\left[\llbracket \mathsf{C_1} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\!\downarrow \;\not\Longleftrightarrow\; \mathfrak{C}\left[\llbracket \mathsf{C_2} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\!\downarrow \Rightarrow \mathsf{P_1} \not\simeq_{ctx} \mathsf{P_2}$
- unfold $\simeq_{ctx}$
- contrapositive

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists \mathfrak{C}.\ \mathfrak{C}\left[\llbracket \mathsf{C_1} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\downarrow \not\Leftrightarrow \mathfrak{C}\left[\llbracket \mathsf{C_2} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\downarrow \Rightarrow$
    $\exists \mathfrak{C}. \mathfrak{C}[\mathsf{C_2}]\downarrow \not\Leftrightarrow \mathfrak{C}[\mathsf{C_2}]\downarrow$
- unfold $\simeq_{ctx}$
- contrapositive
- unfold $\simeq_{ctx}$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists\mathfrak{C}.\ \mathfrak{C}\left[\llbracket \mathsf{C_1} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\downarrow \not\Longleftrightarrow \mathfrak{C}\left[\llbracket \mathsf{C_2} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\downarrow \Rightarrow$
    $\exists\mathfrak{C}.\mathfrak{C}[\mathsf{C_2}]\downarrow \not\Longleftrightarrow \mathfrak{C}[\mathsf{C_2}]\downarrow$
- unfold $\simeq_{ctx}$
- contrapositive
- unfold $\simeq_{ctx}$
- backtranslation!

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists\mathfrak{C}.\ \mathfrak{C}\left[\llbracket \mathsf{C}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\downarrow \not\Longleftrightarrow \mathfrak{C}\left[\llbracket \mathsf{C}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\downarrow \Rightarrow$
    $\exists\mathfrak{C}.\mathfrak{C}[\mathsf{C}_2]\downarrow \not\Longleftrightarrow \mathfrak{C}[\mathsf{C}_2]\downarrow$
- generate $\mathfrak{C}$ based on $\mathfrak{C}$

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $[\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \not\simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \exists \mathfrak{C}.\mathfrak{C}[\mathsf{C}_2]\downarrow \not\iff \mathfrak{C}[\mathsf{C}_2]\downarrow$
- generate $\mathfrak{C}$ based on $\mathfrak{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $[\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \not\simeq \ [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \exists \mathfrak{C}.\mathfrak{C}[\mathsf{C}_2]\!\downarrow \ \not\Longleftrightarrow \ \mathfrak{C}[\mathsf{C}_2]\!\downarrow$
- generate $\mathfrak{C}$ based on $\mathfrak{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\mathsf{TR}(\mathbf{C_1}) \neq \mathsf{TR}(\mathbf{C_2}) \Rightarrow \exists \mathfrak{C}. \mathfrak{C}[\mathsf{C_2}]\downarrow \not\Longleftrightarrow \mathfrak{C}[\mathsf{C_2}]\downarrow$
- generate $\mathfrak{C}$ based on $\mathfrak{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists \alpha \in \mathsf{TR}(\mathbf{C_1}), \alpha \notin \mathsf{TR}(\mathbf{C_2}) \Rightarrow$
    $\exists \mathfrak{C}. \mathfrak{C}[\mathsf{C_2}]{\downarrow} \not\Longleftrightarrow \mathfrak{C}[\mathsf{C_2}]{\downarrow}$
- generate $\mathfrak{C}$ based on $\mathfrak{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Conclusion

- program equivalences can be used to define security properties
- preserving (and reflecting) equivalences can be used to define a secure compiler

Further Reading

- Martin Abadi. 1999. Protection in programming-language translations.
- Andrew Kennedy. 2006. Securing the .NET Programming Model.
- Joachim Parrow. 2014. General conditions for Full Abstraction.
- Daniele Gorla and Uwe Nestman. 2014. Full Abstraction for Expressiveness: History, Myths and Facts.
- Patrignani, Ahmed, Clarke. 2019. Formal Approaches to Secure Compilation.