Program Equivalences and Fully Abstract Compilation

summer semester 18-19, block

Marco Patrignani\textsuperscript{1,2}
What is a 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 $T$:

Gross simplification:

• PL perspective on this subject (will remain for the whole course)
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 $S$

Gross simplification:

• PL perspective on this subject (will remain for the whole course)
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 $T$
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 $T$: $[\cdot]^S_T$
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 $T$: $\mathbb{J} \cdot \mathbb{K}

Gross simplification:

PL perspective on this subject (will remain for the whole course)
Example: Insecure Compilation

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

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

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

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

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;
    }
}
```

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;}
}
```

```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;
}
```
Example: Insecure Compilation

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

Security is not preserved.
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 answers 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:

```java
private secret : Int = 0;

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

Confidential: adjective
spoken, written, acted on, etc., in strict privacy or secrecy; secret:

//one.osf
private
//two.osf
public
setSecret()
{ Int
//three.osf
secret = 0;
//four.osf
return
//five.osf
0;
//six.osf
}

Java source

Q: how do we know that secret is confidential?
Example: Confidentiality

Confidential: adjective
spoken, written, acted on, etc., in strict privacy or secrecy; secret:

```
private secret = 0;
```

Java source

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;
```

Program equivalences (generally) are:
- reflexive
- transitive
- symmetric
aka: relations
Quiz: Are these Equivalent Programs?

Program equivalences (generally) are:
• reflexive
• transitive
• symmetric
aka: relations

```
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;
```
Quiz: Are these Equivalent Programs?

Program equivalences (generally) are:
- reflexive
- transitive
- symmetric
aka: relations
Program Equivalence

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

  - When they behave the same even if they are different
  - Semantics (behaviour) VS Syntax (outlook)
  - We care about the former, not the latter!

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.
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

Q: When are two programs equivalent?
- When they behave the same even if they are different
- Semantics (behaviour) VS Syntax (outlook)
- We care about the former, not the latter!

Defining a security property using program equivalence:
Program Equivalence

• When are two programs equivalent?
  - When they behave the same even if they are different

• Semantics (behaviour) VS Syntax (outlook)
  - we care about the former, not the latter!

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.

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

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

The Language defines how to reason (it's what programmers already do!)
Example: Confidentiality as P.Eq.

```java
private secret : Int = 0;

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

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

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

The Language defines how to reason (it's what programmers already do!)
Example: Confidentiality as P.Eq.

```java
private secret : Int = 0;

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

```java
private secret : Int = 0;

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

With a Java-like semantics, `secret` is never accessed from outside. With a C-like semantics, `secret` can be accessed from outside. The Language defines how to reason (it's what programmers already do!)
Example: Confidentiality as P.Eq.

With a Java-like semantics, secret is never accessed from outside. With a C-like semantics, secret can be accessed from outside.
Example: Confidentiality as P.Eq.

```
private secret : Int = 0;

public setSecret( ):
    secret = 0;
    return 0;
```

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

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

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;
    }
    return 1;
}
```

**Integrity**: internal consistency or lack of corruption in data.
Example: Integrity as P.Eq.

**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.

```java
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.

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

Guessing addresses in memory leads to common exploits: ROP, return to libc, violation of ASLR ...
Example: Memory Size as P.Eq.

```java
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.

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

```java
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.

\[ P_1 \sim_{ctx} P_2 \overset{\text{def}}{=} \forall C. \ C[P_1] \downarrow \iff C[P_2] \downarrow \]
Contextual Equivalence (CEQ)

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

\[ P_1 \sim_{ctx} P_2 \overset{\text{def}}{=} \forall C. C[P_1] \downarrow \iff C[P_2] \downarrow \]
Contextual Equivalence (CEQ)

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

\[ P_1 \sim_{ctx} P_2 \overset{\text{def}}{=} \forall C. \; C[P_1] \downarrow \iff C[P_2] \downarrow \]
Contextual Equivalence (CEQ)

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

\[ P_1 \sim_{ctx} P_2 \overset{\text{def}}{=} \forall C. \ C[P_1] \downarrow \iff C[P_2] \downarrow \]
Two programs are equivalent if no matter what external observer interacts with them that observer cannot distinguish the programs.

\[ P_1 \sim_{ctx} P_2 \overset{\text{def}}{=} \forall C. C[P_1] \downarrow \iff C[P_2] \downarrow \]
Contextual Equivalence (CEQ)

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

\[ P_1 /\text{uni2243} \quad \text{ctx} \quad P_2 \quad \text{def} = \forall C. C[P_1] \downarrow \Leftrightarrow C[P_2] \downarrow \]

- the external observer \( \mathcal{C} \) is generally called context
- it is a program, written in the same language as \( P_1 \) and \( P_2 \)
- it is the same program \( \mathcal{C} \) 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 external observer interacts with them, the observer cannot distinguish the programs.

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

Interaction means **link and run together** (like a library).
Contextual Equivalence (CEQ)

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

\[ P_1 /ctx P_2 \text{ def } \forall C. C[P_1] \downarrow \iff C[P_2] \downarrow \]

- 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

```java
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();
    ...
}
```
Example: CEQ

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

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

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

• if the target programs are not equivalent \( \not\equiv_{ctx} \) then the intended security property is violated
Inequivalences as Security Violations

- if the target programs are not equivalent \( \not\equiv_{ctx} \) 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 \( \not\equiv_{ctx} \) 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 \( \not\equiv_{ctx} \) 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 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.

• Assumption /one.osf: 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.
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
- Assumption: 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

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
  - Assumption /one.osf: 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
A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs:

\[ J \cdot S T \]

\[ \text{def} \quad \forall P_1, P_2 \quad \text{if} \quad P_1 \not\equiv ctx P_2 \quad \text{then} \quad J P_1 K S T \not\equiv ctx J P_2 K S T \]

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

\[
\langle \cdot \rangle^S_T \text{ is FAC\#1} \overset{\text{def}}{=} \forall P_1, P_2 \quad \text{if} \quad P_1 \approx_{ctx} P_2 \quad \text{then} \quad \langle P_1 \rangle^S_T \approx_{ctx} \langle P_2 \rangle^S_T
\]
A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs.

\[
\boxed{\cdot}_T^S \text{ is FAC#1} \overset{\text{def}}{=} \forall P_1, P_2 \text{ if } P_1 \sim_{ctx} P_2 \text{ then } [P_1]^S_T \sim_{ctx} [P_2]^S_T
\]
A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs.

\[ [\cdot]_T^S \text{ is FAC}\#1 \stackrel{\text{def}}{=} \forall P_1, P_2 \]

if \( P_1 \sim_{ctx} P_2 \)

then \( [P_1]_T^{S\sim_{ctx}} [P_2]_T^S \)
A compiler is secure if, given source equivalent programs, it *compiles them* into equivalent target programs.

\[
\begin{align*}
\llbracket \cdot \rrbracket^S_T \text{ is FAC#1 } & \overset{\text{def}}{=} \forall P_1, P_2 \\
\text{if } P_1 \sim_{ctx} P_2 \text{ then } & \llbracket P_1 \rrbracket^S_T \sim_{ctx} \llbracket P_2 \rrbracket^S_T
\end{align*}
\]
A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs.

\[
\begin{align*}
\text{[•]}^S_T \text{ is FAC#1} & \quad \overset{\text{def}}{=} \quad \forall P_1, P_2\\
\text{if} & \quad P_1 \sim_{ctx} P_2\\
\text{then} & \quad \text{[•]}^S_T P_1 \sim_{ctx} \text{[•]}^S_T P_2
\end{align*}
\]
A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs.

\[
\left[ \cdot \right]_T^S \text{ is } \text{FAC}#1 \overset{\text{def}}{=} \forall P_1, P_2 \\
\text{if } P_1 \sim_{ctx} P_2 \\
\text{then } \left[ P_1 \right]_T^S \sim_{ctx} \left[ P_2 \right]_T^S
\]

Right?
Wrong.

An empty translation would fit $\text{FAC}/\text{numbersign.osf}/\text{one.osf}$!

We need the compiler also to be correct. Roughly, turn $\Rightarrow$ into a $\iff$: $J \cdot K S T$ is FAC $\text{def} = \forall P_1, P_2 P_1/\text{uni2243 ctx} P_2 \iff J P_1 K S T /\text{uni2243 ctx} J P_2 K S T$

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

Criteria need to be precise and general.
Wrong.

An empty translation would fit FAC#1!
Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct.

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

$[[\cdot]]_T^S$ is FAC $\defeq \forall P_1, P_2$

$P_1 \simeq_{ctx} P_2 \iff 
[[P_1]]_T^S \simeq_{ctx} [[P_2]]_T^S$
Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct.

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

$$
\mathbb{P} \cdot \mathbb{K} \mathbb{S} \mathbb{T}\text{ is FAC} \overset{\text{def}}{=} \forall \mathbb{P}_1, \mathbb{P}_2
$$

$$
\mathbb{P}_1 \sim_{ctx} \mathbb{P}_2 \iff [\mathbb{P}_1]^S_T \sim_{ctx} [\mathbb{P}_2]^S_T
$$

**Note:** $\iff$ does not mean compiler correctness in the general sense, but it’s a consequence
Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct.

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

$$J \cdot K S T \text{ is FAC } \overset{\text{def}}{=} \forall P_1, P_2$$

$$P_1 \simeq_{ctx} P_2 \iff [P_1]^S_T \simeq_{ctx} [P_2]^S_T$$

Note: $\iff$ 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

$\llbracket \cdot \rrbracket_T^S$ is FAC $^\text{def}$ $\forall P_1, P_2$

$P_1 \sim_{ctx} P_2 \iff [P_1]_T^S \sim_{ctx} [P_2]_T^S$
Fully Abstract Compilation

\[ \boxed{\cdot}^S_T \text{ is FAC} \overset{\text{def}}{=} \forall P_1, P_2 \]

\[ P_1 \sim_{ctx} P_2 \iff [P_1]^S_T \sim_{ctx} [P_2]^S_T \]

- break the \[\iff\]:
  1. \[\Rightarrow:\] \[\forall P_1, P_2. \ P_1 \sim_{ctx} P_2 \Rightarrow [P_1]^S_T \sim_{ctx} [P_2]^S_T \]
  2. \[\Leftarrow:\] \[\forall P_1, P_2. \ [P_1]^S_T \sim_{ctx} [P_2]^S_T \Rightarrow P_1 \sim_{ctx} P_2 \]

- point 2 (should) follow from compiler correctness
Fully Abstract Compilation

\[ \llbracket \cdot \rrbracket^S_T \text{ is FAC} \overset{\text{def}}{=} \forall P_1, P_2 \]

\[ P_1 \simeq_{ctx} P_2 \iff \llbracket P_1 \rrbracket^S_T \simeq_{ctx} \llbracket P_2 \rrbracket^S_T \]

- break the \iff:
  1. \( \Rightarrow: \forall P_1, P_2. P_1 \simeq_{ctx} P_2 \Rightarrow \llbracket P_1 \rrbracket^S_T \simeq_{ctx} \llbracket P_2 \rrbracket^S_T \)
  2. \( \Leftarrow: \forall P_1, P_2. \llbracket P_1 \rrbracket^S_T \simeq_{ctx} \llbracket P_2 \rrbracket^S_T \Rightarrow P_1 \simeq_{ctx} P_2 \)

- point 2 (should) follow from compiler correctness
- point 1 is tricky, because of \( \simeq_{ctx} \) and its \( \forall C \)
Fully Abstract Compilation

\[ [\cdot]^S_T \text{ is FAC} \overset{\text{def}}{=} \forall P_1, P_2
\]

\[ P_1 \sim_{ctx} P_2 \iff [P_1]^S_T \sim_{ctx} [P_2]^S_T \]

- break the \( \iff \):
  1. \( \Rightarrow \): \( \forall P_1, P_2. P_1 \sim_{ctx} P_2 \Rightarrow [P_1]^S_T \sim_{ctx} [P_2]^S_T \)
  2. \( \Leftarrow \): \( \forall P_1, P_2. [P_1]^S_T \sim_{ctx} [P_2]^S_T \Rightarrow P_1 \sim_{ctx} P_2 \)

- point 2 (should) follow from compiler correctness

- point 1 is tricky, because of \( \sim_{ctx} \) and its \( \forall C \)

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 target contexts are similar
- Trace-based: relies on trace semantics when there is a large abstraction gap between source and target
• we replace $\sim_{ctx}$ 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 $\sim_{ctx}$ 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 $\sim_{ctx}$ 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

- 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 🇨!)
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 😄!)

- interest in the behaviour of our code (component)

- need to consider the rest
Trace Semantics for Our Program

- disregard the rest

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!)
Trace Semantics for Our Program

- disregard the rest

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)
Trace Semantics for Our Program

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

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)
function definition
of our code

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

1. jump to an entry point

private data of our program

other code
written by the attacker

(this is the context C)

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

1. jump to an entry point
• 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)

[Mathematical expression]

\[ \text{TR}(C) = \{ \alpha : C \alpha \Rightarrow \_ \} \]
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

\[
\{ \overline{\alpha} \mid C \xrightarrow{\overline{\alpha}} \_ \}
\]
Trace Actions

Labels \( L ::= a | \epsilon \)

Observable actions \( \alpha ::= \sqrt{ | g? | g! } \)

Actions \( g ::= \text{call} \ f \ (v) | \text{ret} \ v \)
Traces for a program

We need to define:

• trace states (almost program states)
• labels that make traces
• rules for generating labels and traces...
• the traces of a component = …
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

\[ C_1 \Downarrow C_2 \]
Trace Equivalence

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

\[ C_1 \sim_{ctx} C_2 \]

- trace semantics gives us trace equivalence

the traces of \( C_1 \) are the same as 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

\[ \left\{ \bar{\alpha} \mid C_1 \overset{\bar{\alpha}}{\rightarrow} \_ \right\} = \left\{ \bar{\alpha} \mid C_2 \overset{\bar{\alpha}}{\rightarrow} \_ \right\} \]

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

\[ C_1 \simeq_{ctx} C_2 \iff C_1 \vdash C_2 \]
Proofs about Trace Semantics

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

\[ C_1 \sim_{ctx} C_2 \iff C_1 \models C_2 \]
we have:

\( C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2) \)
• we have:
  • $C_1 \simeq_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2)$

• we need to prove
  • $P_1 \simeq_{ctx} P_2 \Rightarrow [P_1]^S_T \simeq_{ctx} [P_2]^S_T$
we have:

\[ C_1 \simeq_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \]

we need to prove

\[ P_1 \simeq_{ctx} P_2 \Rightarrow \forall C. \ e[[C_1^T]_S] \downarrow \iff e[[C_2^T]_S] \downarrow \]

unfold \( \simeq_{ctx} \)
we have:

\[ \mathsf{C}_1 \simeq_{ctx} \mathsf{C}_2 \iff \mathsf{TR}(\mathsf{C}_1) = \mathsf{TR}(\mathsf{C}_2) \]

we need to prove

\[ \exists \mathcal{E}. \mathcal{E}\left[ [\mathsf{C}_1]_T^S \right] \downarrow \iff \mathcal{E}\left[ [\mathsf{C}_2]_T^S \right] \downarrow \implies \mathsf{P}_1 \not\equiv_{ctx} \mathsf{P}_2 \]

unfold \ \simeq_{ctx}

contrapositive
• we have:
  - $C_1 \simeq_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2)$

• we need to prove
  - $\exists C. C[[C_1]_T^S] \downarrow \iff C[[C_2]_T^S] \downarrow \Rightarrow \exists C. C[C_2] \downarrow \iff C[C_2] \downarrow$

• unfold $\simeq_{ctx}$
• contrapositive
• unfold $\simeq_{ctx}$
we have:

\[ C_1 \approx_{ctx} C_2 \iff TR(C_1) = TR(C_2) \]

we need to prove

\[ \exists C. C[\llbracket C_1 \rrbracket_T^S] \downarrow \iff C[\llbracket C_2 \rrbracket_T^S] \downarrow \implies \exists C. C[\llbracket C_2 \rrbracket_T] \downarrow \iff C[\llbracket C_2 \rrbracket_T] \downarrow \]

unfold \( \approx_{ctx} \)

contrapositive

unfold \( \approx_{ctx} \)

backtranslation!
Fully Abstract Compilation & Target Traces

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

- we need to prove
  - $\exists \mathcal{C}. \mathcal{C}[\{C_1\}^S_T] \downarrow \iff \mathcal{C}[\{C_2\}^S_T] \downarrow \Rightarrow$

  - $\exists \mathcal{C}. \mathcal{C}[C_2] \downarrow \iff \mathcal{C}[C_2] \downarrow$

- generate $\mathcal{C}$ based on $\mathcal{C}$
Fully Abstract Compilation & Target Traces

• we have:
  • $C_1 \simeq_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2)$

• we need to prove
  • $[P_1]^S_T \not\xrightarrow{ctx} [P_2]^S_T \Rightarrow \exists \mathcal{C}. \mathcal{C}[C_2] \downarrow \iff \mathcal{C}[C_2] \downarrow$

• generate $\mathcal{C}$ based on $\mathcal{C}$
• if complex, apply Traces (folding $\simeq_{ctx}$)
we have:

\[ C_1 \sim_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \]

we need to prove

\[ [P_1]_T^S \neq [P_2]_T^S \Rightarrow \exists C. C[C_2] \downarrow \iff C[C_2] \downarrow \]

generate \( C \) based on \( \mathcal{C} \)

if complex, apply Traces (folding \( \sim_{ctx} \))
we have:

- \( C_1 \simeq_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \)

we need to prove

- \( \text{TR}(C_1) \neq \text{TR}(C_2) \Rightarrow \exists C. C[C_2] \downarrow \iff C[C_2] \downarrow \)

- generate \( C \) based on \( C \)
- if complex, apply Traces (folding \( \simeq_{ctx} \))
Fully Abstract Compilation & Target Traces

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

- we need to prove
  - $\exists \alpha \in \text{TR}(C_1), \alpha \notin \text{TR}(C_2) \Rightarrow$
    $\exists \mathcal{E}.\mathcal{E}[C_2] \downarrow \iff \mathcal{E}[C_2] \downarrow$

- generate $\mathcal{E}$ based on $\mathcal{E}$
- if complex, apply Traces (folding $\sim_{ctx}$)
Conclusion

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


• 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.