Lecture 1: What is Secure Compilation?

Secure Compilation Seminar

Marco Patrignani
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 $T$: $S \rightarrow T$
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 T: \([\cdot]_T^S\)
Example #1: Insecure Compilation

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

    public void deposit(int amount) {
        this.balance += amount;
    }
}
```
Example #1: 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 #1: 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 #1: Insecure Compilation

Java source

public class Account
private int balance = 0;

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

C target

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 #1: Insecure Compilation

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

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

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;
}
```

**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 answers 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 preserve it in the target"
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 preserve it in the target"

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?
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?

answers provided in this seminar
Example #2: Confidentiality

Confidential: adjective

spoken, written, acted on, etc., in strict privacy or secrecy; secret:
**Example #2: 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 #2: 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;
}
```

• **Q**: how do we **know** that secret is confidential?
Program Equivalence

• a possible answer to the questions before
Program Equivalence

• a possible answer to the questions before
• and to many more questions posed about programming languages
Quiz #1: Are these Equivalent Programs?

1. P1
   ```java
   public Bool getTrue( x : Bool )
   return true;
   ```

2. P2
   ```java
   public Bool getTrue( x : Bool )
   return x or true;
   ```

3. P3
   ```java
   public Bool getTrue( x : Bool )
   return x and false;
   ```

4. P4
   ```java
   public Bool getTrue( x : Bool )
   return false;
   ```

5. P5
   ```java
   public Bool getFalse( x : Bool )
   return x and true;
   ```

Program equivalences (generally) are:
- reflexive
- transitive
- symmetric
aka: relations
Quiz #1: Are these Equivalent Programs?

Program equivalences (generally) are:

• reflexive
• transitive
• symmetric
aka: relations
Quiz #1: 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.
• **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: to find two programs that, albeit syntactically different, both behave in a way that respects the property, no matter how they are used.
Example #4: 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 #4: 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;
}
```

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 #4: 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.
```
Example #4: Confidentiality as P.Eq.

```
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 #5: Integrity as P.Eq.

```java
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 #5: 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.

Maintenance of invariants
Example #5: Integrity as P.Eq.

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

```
public proxy( callback : Unit \rightarrow Unit ) : Int {
    var secret = 0;
    callback();
    return 0;
}
```
Example #6: Memory Allocation as P.Eq.

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

```java
public newObjects( ) : Object {
    var x = new Object();
    var y = new Object();
    return y;
}
```
Example #6: 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 …
Expressing Program Equivalence

This \( (\simeq_{ctx}) \) is called

Contextual Equivalence
Expressing Program Equivalence

This $\equiv_{ctx}$ is called

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.

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

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

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

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

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

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

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

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

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

- It is a program, written in the same language as $P_1$ and $P_2$.
- It is the same program $P$ interacting with both $P_1$ and $P_2$ in two different runs.
- So it cannot express out of language attacks (e.g., side channels).
- Often we write it $C$ or $\mathbb{C}$.

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

Often we write it $\mathbb{C}[P_1]$. 

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.
- 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
• distinguishing means: \textit{terminate with different values}
• the observer basically asks the question: \textit{is this program }$P_1$\textit{?}
• if the observer can find a way to distinguish $P_1$ from $P_2$, it will return true, otherwise false
• often we use \textit{divergence} and \textit{termination} as opposed to this boolean termination
Example #7: 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;
}
```
Example #7: 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 #8: CEQ

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

• if the programs are not equivalent ($\neq_{ctx}$) then the intended security property is violated

• this hardly happens as source languages are high level
Inequivalences as Security Violations

• if the programs are not equivalent \(\text{(*uni2244 ctx*)}\) then the intended security property is violated
• this hardly happens as source languages are high level

When does inequivalences escape the programmer’s reasoning?

• if languages have complex features
• if there are more languages involved (e.g., multiple target languages)
Inequivalences as Security Violations

When does inequivalences escape the programmer’s reasoning?

1. if languages have complex features
Inequivalences as Security Violations

When does *inequivalences* escape the programmer’s reasoning?

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

\[ J \cdot K \]

\( S \) is \( \text{FAC/numbersign.osf/one.osf} \)

\[ \text{def} = \forall P_1, P_2 \]

\( \text{if } P_1 \equiv ctx P_2 \text{ then } J P_1 K S T /\text{uni2243 ctx } J P_2 K S T /\text{seven.osf} \)

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

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

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

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

\[
\llbracket \cdot \rrbracket^S_T \text{ is } \text{FAC#1} \overset{\text{def}}{=} \forall P_1, P_2
\]

\[
\text{if } P_1 \sim_{ctx} P_2
\]

\[
\text{then } \llbracket P_1 \rrbracket^S_T \sim_{ctx} \llbracket P_2 \rrbracket^S_T
\]
A compiler is secure if, given source equivalent programs, it compiles them into equivalent target programs.

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

\[
\llbracket \cdot \rrbracket^S_T \text{ is FAC#1} \overset{\text{def}}{=} \forall P_1, P_2 \quad \text{if} \quad P_1 \sim_{ctx} P_2 \quad \text{then} \quad \llbracket P_1 \rrbracket^S_T \sim_{ctx} \llbracket P_2 \rrbracket^S_T
\]

Right?
Wrong.

\[ J \cdot K \cdot S \cdot T \text{ is FAC def } \forall P_1, P_2 \]
Wrong.

An empty translation would fit FAC#1!

\[ J \cdot K \overset{\text{S\ T}}{\Rightarrow} \]

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

\(^1\text{Abadi '99}\)
Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct. 

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

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

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

\(^1\)Abadi ’99
Wrong.

An empty translation would fit FAC#1!

We need the compiler also to be correct. 

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

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

$$P_1 \sim_{ctx} P_2 \iff \llbracket P_1 \rrbracket^S_T \sim_{ctx} \llbracket P_2 \rrbracket^S_T$$

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

\(^1\)Abadi ’99
Remarks on Fully Abstract Compilation

- widely adopted since 1999
Remarks on Fully Abstract Compilation

- widely adopted since 1999
- only preserves security property expressed as program equivalence
Remarks on Fully Abstract Compilation

- widely adopted since 1999
- only preserves security property expressed as program equivalence
- **not the silver bullet**: the papers will present shortcomings of fully abstract compilation
Conclusion

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

Short(ish) and high-level(ish):

• Martin Abadi. 1999. Protection in programming-language translations. In Secure Internet programming