Formal Approaches to Secure Compilation

Amal Ahmed
Northeastern University & Inria Paris
Secure Compilation

building compilers that ensure security properties of source programs are preserved in target programs
package Bank;

public class Account{
    private int balance = 0;
    
    public void deposit( int amount ) {
        this.balance += amount;
    }
}

Listing 1. Example of Java source code.

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

void deposit_f( Account ⇤ a, int amount ) {
    a!balance += amount;
    return;
}

Listing 2. C code obtained from compiling the Java code of Listing 1.

When the Java code in Listing 1 interacts with other Java code, the latter cannot access the contents of balance since it is a private field. However, when the Java code is compiled into the C code in Listing 2 and then interacts with arbitrary C code, the latter can access the contents of balance by doing simple pointer arithmetic. Given a pointer to a C Account struct, an attacker can add the size (in words) of an int to it and read the contents of balance, effectively violating a confidentiality property that the source program had.

This violation occurs because there is a discrepancy between what abstractions the source language offers and what abstraction the target language has. This discrepancy is both inevitable and dangerous. The inevitability stems from the fact that source languages provide powerful abstractions whose goal is allowing a programmer to write better code. The danger stems from the fact that source-level abstractions can be used to enforce security properties, but target languages that do not preserve such security properties are vulnerable to attacks. Unfortunately, most target languages cannot preserve the abstractions of their source-level counterparts [1].

In order to withstand the danger posed by exploitable target languages, secure compilation techniques can be adopted. Secure compilation is a discipline that studies compilers (or, more generally, compilation schemes) that preserve the security properties of source languages in their compiled, target-level counterparts. Secure compilation is concerned with the security of partial programs (also called components in this paper) since attackers are modelled as the environment such programs interact with. Partial programs are programs that do not implement all the functionality they require to operate. Instead, they are linked together with an environment (often also called a context) that provides the missing functionality in order to create a runnable whole program.

An open environment is used to model possible attackers to the component, which is not possible when whole programs are considered.

Secure compilers use a variety of techniques to protect compiled components from attacker contexts. Once devised, such compilers must be proven to be secure, i.e., they must be proven to conform to some criterion that implies secure compilation. As we discuss later in this survey, a variety of formal statements can capture when a compiler is secure. One such formal criterion that has been widely adopted for secure compilation is compiler full abstraction [1].
Secure compilers use a variety of techniques to protect compiled components from attacker contexts. Once devised, such compilers must be proven to be secure, i.e., they must be proven to conform to some criterion that implies secure compilation. As we discuss later in this survey, a compiler full abstraction property that the source program had.

In order to withstand the danger posed by exploitable target languages, secure compilation generally, compilation schemes that preserve the security properties of source languages in their contexts. However, when the Java code is compiled into the C code in this paper, attackers are modelled as the environment such that does not preserve such security properties are vulnerable to attacks. Unfortunately, most target languages cannot preserve the abstractions of their source-level counterparts [6]. The inevitability stems from the fact that source languages provide powerful techniques can be adopted. Secure compilation is a discipline that studies compilers (or, more precisely, techniques that...
When the Java code in Listing 1 is compiled into C target code, the gap between source and target abstractions manifests in the way the balance field is handled. In the source code, the balance field is private and is only accessible through the deposit method. However, in the compiled C code, the balance field is public and can be directly manipulated. This discrepancy highlights the need for a mechanism to hide the balance field in the target language.

The inevitability of this discrepancy stems from the fact that source languages provide powerful abstractions, while target languages may not preserve such properties. In order to withstand the danger posed by exploitable target languages, secure compilation schemes must be developed. These schemes must not only preserve the security properties of source languages but also ensure that the balance field is hidden in the target language. This means developing a context that allows the compiler to preserve the security properties of the source code.

To address this challenge, secure compilers use a variety of techniques, including formal criteria that can capture when a compiler is secure. One such formal criterion is the compiler full abstraction, which specifies that the compiled code must behave exactly as the source code. This ensures that the security properties of the source code are preserved in the target code.

However, the danger lies in the fact that source-level abstractions can enforce security properties, but target languages may not preserve such security properties. This gap between source and target languages highlights the need for a mechanism to hide the balance field in the target language. The challenge is to develop a secure compiler that can prove security properties in whole programs, taking into account the contexts in which these programs interact.
Security Properties as Program Equivalences
Security Properties as Program Equivalences

**Example: Integrity**

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

```java
public proxy( callback : Unit → Unit )
    : Int {
    var secret = 0;
    callback();
    if ( secret == 0 ) {
        return 0;
    }
    return 1;
}
```
Security Properties as Program Equivalences

Example: Confidentiality

private secret : Int = 0;

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

private secret : Int = 0;

public setSecret() : Int {
    secret = 0;
    return 0;
}
Security Properties as Program Equivalences

Example: Unbounded vs. finite memory

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

```java
public kernel( n : Int, callback : Unit → Unit ) : Int {
    callback();
    // security-relevant code
    return 0;
}
```
Security Properties as Program Equivalences

**Example: Memory Allocation Order**

```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;
}
```
This Talk ...

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

1. Preserving security by preserving equivalence

2. Different compilation targets and threat models
   - is the target language typed or untyped?
   - what observations can the attacker make?

3. Different ways of enforcing secure compilation
   - static checking
   - dynamic checking (e.g., runtime monitoring, cryptographic & hardware enforcement)

4. Proof techniques
   - "back-translating" target attackers to source
Fully Abstract Compilation

Preserve and reflect contextual equivalence
Fully Abstract Compilation

Preserve contextual equivalence

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

i.e. target contexts (attackers a) can make no more observations about e than a source context can make about e
Ensuring Full Abstr. / Secure Comp.

Must ensure that any $a$ we link with behaves like some source context.
Ensuring Full Abstr. / Secure Comp.

1. Add target features to the source language. **Bad!**
Ensuring Full Abstr. / Secure Comp.

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

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

\[ e : \tau \sim e : \tau^+ \]
Type-Preserving Secure Compilation

Preserve well-typedness & equivalence

\[ e_1 \xrightarrow{\approx_{\text{ctx}}^S} e_2 \]

\[ \xrightarrow{\approx_{\text{ctx}}^T} \]

\[ : \tau \]

\[ : \tau^+ \]
Type-Preserving Compilation

\[ \mathit{e} : \tau \rightsquigarrow \mathit{e}^+ : \tau^+ \]

- System F to Typed Assembly Language
  \cite{Morrisett:POPL97,TOPLAS98}

- Typed compilation of Featherweight Java to F-omega, private fields to existential type \cite{League:TOPLAS02}

- FINE (F# with refinement & affine types) to DCIL (dependent CIL) \cite{Chen:PLDI10}

- Security-type-preserving compilation from WHILE lang. to stack-based TAL (both languages satisfy noninterference). Extended to concurrent setting with thread creation, secure scheduler \cite{Barthe+:2007,Barthe+:2010}
Type-Preserving Secure Compilation

Preserve well-typedness & equivalence

\( e_1 \approx_{ctx} e_2 \)

\( e_1 \approx_S e_2 \)

\( :: \tau \)

\( :: \tau^+ \)
Challenge: Proving Full Abstraction

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

$\Gamma \vdash e_1 \simeq_{\text{ctx}}^S e_2 : \tau$

$\Gamma^+ \vdash e_1 \simeq_{\text{ctx}}^T e_2 : \tau^+$
Challenge: Proving Full Abstraction

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

\[ \Gamma \vdash e_1 \approx_{\text{ctx}}^{S} e_2 : \tau \]

Given:
No $C_S$ can distinguish $e_1, e_2$

\[ \Gamma^+ \vdash e_1 \approx_{\text{ctx}}^{T} e_2 : \tau^+ \]
Challenge: Back-translation

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

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

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

2. If target is more expressive than source
   (a) Both terminating: use back-translation by partial evaluation
      • Equivalence-preserving CPS from STLC to System F
        [Ahmed-Blume ICFP'11]
      • Noninterference for Free (DCC to $F\omega$
        [Bowman-Ahmed ICFP'15]
   (b) Both nonterminating: use ??
       back-trans by partial evaluation is not well-founded!

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

Source: STLC + $\mu$ types

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

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

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

Proof technique: Universal Embedding

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

[New et al. ICFP’16]
Fully Abstract Closure Conversion

Source: STLC + μ types
Target: System F + ∃ types + μ types + exceptions

Equivalent source terms, inequivalent in lang. with exceptions:

\[ e_1 = \lambda f. (f \text{ true}; f \text{ false}; \langle \rangle) \quad e_2 = \lambda f. (f \text{ false}; f \text{ true}; \langle \rangle) \]

\[ C = \text{catch } y = ([.] (\lambda x. \text{raise } x)) \text{ in } y \]

\[ C[e_1] \downarrow \text{true} \quad C[e_2] \downarrow \text{false} \]

Idea: use modal type system at target to rule out linking with code that throws unhandled exceptions
Ensuring Full Abstraction via Types

\[ e_1 \approx^\text{ctx}_S e_2 : (\text{bool} \rightarrow 1) \rightarrow 1 \]

\[ (\text{bool} \rightarrow E\ 0\ 1) \rightarrow E\ 0\ 1 \neq \]

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

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

code snippet 1

\[ e_1 \]

compile

e_1

\[ e_2 \]

compile

e_2

\[ \equiv_{ctx}^S \]

\[ \equiv_{ctx}^T \]
Dynamic Secure Compilation

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

- STLC with recursion to untyped lambda-calc, proved fully abstract using *approximate back-translation*. Types erased and replaced w. dynamic checks. [Devriese et al. POPL’16]

- f* (STLC with refs, exceptions) to js* (encoding of JavaScript in f*). Defensive wrappers perform dynamic type checks on untyped js* [Fournet et al. POPL’13]

- Lambda-calc to VHDL digital circuits, run-time monitors check that external code respects expected communication protocol [Ghica and Al-Zobaidi ICE’12]
Dynamic Secure Compilation

3. Memory Protection Techniques
   (a) Address space layout randomization (ASLR)

   • STLC w. abstract memory, to target with concrete memory; show probabilistic full abstraction for large memory [Abadi-Plotkin TISSEC'12]

   • Added dynamic alloc, h.o. refs, call/cc, testing hash of reference, to target with probref to reverse hash [Jagadeesan et al. CSF'11]
3. Memory Protection Techniques

(b) Protected Module Architectures (PMAs) (e.g., Intel SGX) protected memory with code and data sections, and unprotected memory

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

3. Memory Protection Techniques

(c) **PUMP Machine** architecture tracks meta-data, registers and memory locations have tags, checked during execution

- Secure compartmentalizing compiler with mutually distrustful compartments that can be compromised by attacker. OO lang to RISC with micro policies

[Juglaret et al. 2015]
Dynamic Secure Compilation

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

1. Need languages / DSLs that allow programmers to easily express security intent.
   - Compilers need to know programmer intent so they can preserve that intent (e.g., FaCT, a DSL for constant-time programming [Cauligi et al. SecDev'17])

2. Performant secure compilers
   - Static enforcement avoids performance overhead, could run on stock hardware; need richly typed compiler IRs
   - Dynamic enforcement when code from static/dynamic and safe/unsafe languages interoperates (e.g., h/w support)
   - Better integration of static and dynamic enforcement...
Better integration of static and dynamic enforcement...

- C
- ML
- Rust
- Scheme
- Gallina

Gradually Typed IR

LLVM

- Intel SGX
- CHERI
Secure Compilation: Open Problems

3. Preserve (weaker) security properties than contextual equiv.
   - Full abstraction may preserve too many incidental/unimportant equivalences and has high overhead for dynamic enforcement

4. Security against side-channel attacks
   - Requires reasoning about side channels in source language, which is cumbersome. Can DSLs help?
   - Correctness-Security Gap in Compiler Optimizations [D'Silva et al. LangSec'15]. Make compilers aware of programmers' security intent to take into account for optimizations.
Secure Compilation: Open Problems

5. Cryptographically enforced secure compilation
   • e.g., Obliv-C ensures memory-trace obliviousness using garbled circuits, but no formal proof that it is secure

6. Concurrency (beyond message-passing, targeting untyped multi-threaded assembly)

7. Easier proof techniques and reusable proof frameworks (trace-based techniques, back-translation, logical relations, bisimulation)
Final Thoughts

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

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

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