Language-based software security, sixth lecture

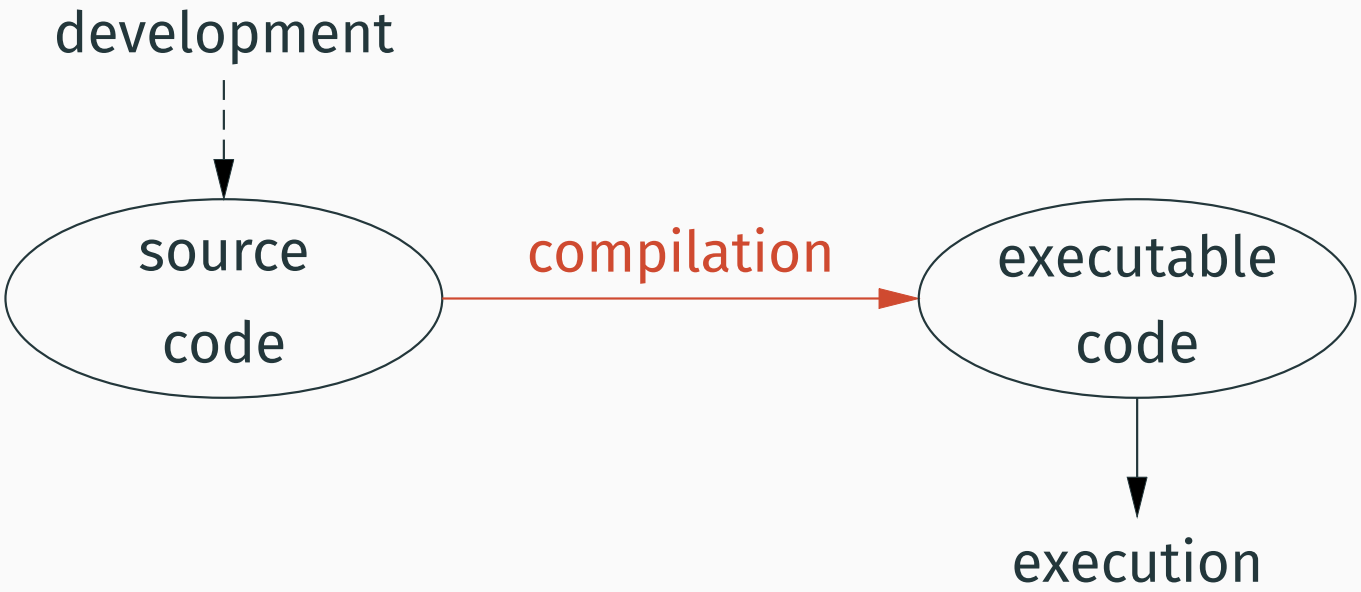
Compilation and security

Xavier Leroy

2022-04-14

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr

In this lecture



What does compilation contribute (positively or negatively) to the security of the compiled code (the code that actually runs)?

Compilation that implements security measures

Checking bounds for array accesses

A crucial check to guarantee safe execution and to prevent buffer overflow attacks.

$$\texttt{x := A[i]} \quad \rightsquigarrow \quad \texttt{assert(0} \leq \texttt{i} < \texttt{size(A)); x := *(A + i)}$$

$$\texttt{A[i] := x} \quad \rightsquigarrow \quad \texttt{assert(0} \leq \texttt{i} < \texttt{size(A)); *(A + i) := x}$$

`assert`(b) aborts execution or raises an exception if b is false.

`*(A + i)` stands for a direct memory read or write, without bounds checking.

Checking bounds for array accesses

A crucial check to guarantee safe execution and to prevent buffer overflow attacks.

$$x := \texttt{A}[i] \quad \rightsquigarrow \quad \texttt{assert}(0 \leq i < \texttt{size}(\texttt{A})); \; x := *(\texttt{A} + i)$$

$$\texttt{A}[i] := x \quad \rightsquigarrow \quad \texttt{assert}(0 \leq i < \texttt{size}(\texttt{A})); \; *(\texttt{A} + i) := x$$

`size(A)` is the size of array `A`, determined

- either statically from the declaration of `A`, as in `int A[10];`
- or dynamically from the in-memory representation of `A`.

Avoiding useless bounds checks

Sometimes, the context of the array access guarantees that it is within bounds:

```
for (int i = 0; i < size(A); i++)  x += A[i];
```

We can recognize this code pattern and compile it without adding an `assert`.

It is safer and more general to systematically generate the dynamic test (`assert`), then eliminate it when possible, via optimizations exploiting the results of static analyses.

(→ lectures of 19/12/2019 and 16/01/2020)

Eliminating redundant assertions

No need to check the same assertion twice in a row!
(An instance of Common Subexpression Elimination.)

Source code:

```
t = A[i]; A[i] = t + 1;
```

Adding bounds checks:

```
assert(0 ≤ i < size(A));   t = *(A + i);
assert(0 ≤ i < size(A));   *(A + i) = t + 1;
```

Eliminating a redundant check:

```
assert(0 ≤ i < size(A));   t = *(A + i);
~~assert(0 ≤ i < size(A));~~   *(A + i) = t + 1;
```

Optimizations based on interval analysis

The static analysis infers properties of the shape $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: program variable; a, b constants inferred by the analyzer)

Source code:

```
int A[10];
for (int i = 2; i < 8; i++)  {
    x += A[i];
}
```

Optimizations based on interval analysis

The static analysis infers properties of the shape $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: program variable; a, b constants inferred by the analyzer)

Adding bounds checks:

```
int A[10];
for (int i = 2; i < 8; i++)  {
    assert(0 ≤ i < 10);
    x += *(A + i);
}
```

Optimizations based on interval analysis

The static analysis infers properties of the shape $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: program variable; a, b constants inferred by the analyzer)

Interval analysis:

```
int A[10];
for (int i = 2; i < 8; i++)  {
    // invariant: 2 ≤ i ≤ 7
    assert(0 ≤ i < 10);
    x += *(A + i);
}
```

Optimizations based on interval analysis

The static analysis infers properties of the shape $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: program variable; a, b constants inferred by the analyzer)

Exploiting interval information:

```
int A[10];
for (int i = 2; i < 8; i++)  {
    // invariant: 2 ≤ i ≤ 7
    assert(true);
    x += *(A + i);
}
```

Optimizations based on interval analysis

The static analysis infers properties of the shape $a \leq \mathtt{x} \leq b$
($\mathtt{x}$: program variable; a, b constants inferred by the analyzer)

Removing trivial assertions:

```
int A[10];
for (int i = 2; i < 8; i++)  {
    // invariant: 2 ≤ i ≤ 7
    assert(true);
    x += *(A + i);
}
```

To go further: relational analyses
(polyhedra $a\mathtt{x} + b\mathtt{y} \leq c$, octagons $\pm\mathtt{x} \pm \mathtt{y} \leq c$, etc)

Moving assertions outside loops

(An instance of Loop-Invariant Code Motion.)

Source code:

```
int A[10];
for (int i = 0; i < 10; i++)  {
    B[j] = B[j] + A[i];
}
```

Moving assertions outside loops

(An instance of Loop-Invariant Code Motion.)

Adding bounds checks:

```
int A[10];
for (int i = 0; i < 10; i++)  {
    assert(0 ≤ i < 10);
    assert(0 ≤ j < size(B));
    assert(0 ≤ j < size(B));
    *(B + j) = *(B + j) + *(A + i);
}
```

Moving assertions outside loops

(An instance of Loop-Invariant Code Motion.)

Applying the previous optimizations:

```
int A[10];
for (int i = 0; i < 10; i++)  {
    assert(0 ≤ i < 10);
    assert(0 ≤ j < size(B));
    assert(0 ≤ j < size(B));
    *(B + j) = *(B + j) + *(A + i);
}
```

Moving assertions outside loops

(An instance of Loop-Invariant Code Motion.)

Lifting the assertion before the loop:

```
int A[10];
assert(0 ≤ j < size(B));
for (int i = 0; i < 10; i++)  {
    ~~assert(0 ≤ i < 10);~~
    ~~assert(0 ≤ j < size(B));~~
    ~~assert(0 ≤ j < size(B));~~
    *(B + j) = *(B + j) + *(A + i);
}
```

Function inlining

Inlining (expanding) a function at its call site can allow the compiler to remove more bounds checks:

```
int f(int A[], int i) {
    assert (0 ≤ i < size(A));
    return *(A + i) + 1;
}
int B[10];
int g(void) { return f(B, 2); }
```

After inlining `f` in `g`:

```
int B[10];
int g(void) {
    assert (0 ≤ 2 < size(B));
    return *(B + 2) + 1;
}
```

Protecting against the Spectre v1 attack

Ensure that the speculative execution of an out-of-bounds access cannot access memory "far away" from the array.

$$\texttt{x := A[i]} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{assert(0} \leq \texttt{i} < \texttt{size(A));} \\ \texttt{x := *(A +} \ \texttt{clip(i, size(A)))} \end{array}$$

$$\texttt{A[i] := x} \quad \rightsquigarrow \quad \begin{array}{l} \texttt{assert(0} \leq \texttt{i} < \texttt{size(A));} \\ \texttt{*(A +} \ \texttt{clip(i, size(A))) := x} \end{array}$$

where $\texttt{clip}(i, n)$ is a branchless expression such that

$$\texttt{clip}(i, n) = \begin{cases} i & \text{if } 0 \leq i < n \\ 0 & \textit{otherwise} \end{cases}$$

x86 implementation: `cmp Ri, Rn; sbb Rc, Rc; and Rc, Ri`.

Other examples

Efficient implementation of dynamic type-checking.

Protections against hardware faults. ($\rightarrow$ K. Heydemann's seminar)

Protections against other attacks on speculative execution.
($\rightarrow$ F. Piessens's seminar)

Code obfuscation. ($\rightarrow$ S. Blazy's seminar)

Compilation that removes security measures

The compiler's contract

The compiler shall produce machine code, as efficient as possible, that computes "the same thing" as the source program.

Two major hypotheses:

- The machine behaves as described in its ISA manual.
  (No faults; timing and speculative executions are not observable; …)
- The source program is free of undefined behaviors.
  (E.g. because it passed static type-checking, or because the programmer swears it.)

“Optimizing” protections against fault attacks

(See K. Heydemann’s seminar on April 7.)

The programmer writes a redundant test:

```
if (cond) {
    assert (cond);
    ...
} else {
    assert (! cond);
    ...
}
```

A trivial static analysis shows that `cond` is true in the `then` branch and false in the `else` branch

$\rightarrow$ the compiler removes both assertions.

“Optimizing” protections against fault attacks

Redundancy between control flow and data:

```
t = 0;
if (cond) {
    t |= 1; ...
} else {
    t |= 2; ...
}
assert (t != 0 && t != 3);
```

Interval analysis shows that $t \in [1, 2]$ at the point of the assertion.
Therefore, the assertion is always true and gets removed.

“Optimizing” the protection against Spectre v1

As outlined in the 4th lecture:

$$x := \texttt{A}[i] \quad \rightsquigarrow \quad \texttt{assert}(0 \leq i < \texttt{size}(\texttt{A}));\ x := *(\texttt{A} + \texttt{sel}(0 \leq i < \texttt{size}(\texttt{A}), \texttt{i}, 0))$$

A trivial static analysis “knows” that after $\texttt{assert}(b)$, condition b is true. Therefore, the access can be rewritten into

$$x := *(\texttt{A} + \texttt{sel}(\texttt{true}, \texttt{i}, 0))$$

then simplified into

$$x := *(\texttt{A} + \texttt{i})$$

Hence the need to use $*(\texttt{A} + \texttt{clip}(i, \texttt{size}(\texttt{A})))$ with an “opaque” $\texttt{clip}$ operator that cannot be optimized away.

“Optimizing” constant-time codes

The compiler is allowed to use conditional branches or memory accesses (tabulation) to implement source code that contains neither.

Example: if `b` has type `bool`,

```
x = b * a1 + (1 - b) * a0;
```

$\rightsquigarrow$ `if (b) x = a1; else x = a0;`

or even

$\rightsquigarrow$ `int t[2] = { a0, a1 }; x = t[b];`

“Optimizing” constant-time codes (recap from lecture #4)

(Simon, Chisnall, Anderson, *What you get is what you C: controlling side effects in mainstream C compilers*, 2018).

Experiment: 4 implementations of `sel` in portable C, compiled for x86-32 by various versions of Clang.

		VERSION_1		VERSION_2		VERSION_3		VERSION_4	
		inlined	library	inlined	library	inlined	library	inlined	library
Clang 3.0	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✗
	-O2	✓	✓	✓	✗	✗	✓	✓	✗
	-O3	✓	✓	✓	✗	✗	✓	✓	✗
Clang 3.3	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗
Clang 3.9	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗

✔ = constant-time code is generated.

✘ = a conditional branch is generated.

Undefined behaviors in C / C++

The ISO C 2018 standard lists over 200 undefined behaviors…

Some undefine behaviors, such as out-of-bounds array writes, can actually cause the program to do anything.

Example (from lecture #1): overflowing a stack-allocated buffer

```
int check(void) {
    char b[80]; int ok = 0;
    gets(b); ...; return ok;
}
```

If `gets` writes beyond the end of `b`, this can
– overwrite the value of `ok` → wrong result
– invalidate the return address → crash when `check` returns
– overwrite the return address → arbitrary code execution

Undefined behaviors in C / C++

Historically, many undefined behaviors are cases where different processors behave differently, and we wish to allow the compiler to "follow" the behavior of the machine. For example:

- `NULL` pointer dereference:
  a `load` or `store` processor instruction can either raise a *segmentation fault*, or access memory at address 0 normally.
- Overflow in a signed integer addition:
  the `add` processor instruction can take the result modulo 2^N, or saturate it to `INT_MAX` or `INT_MIN`, or raise a fault.

This is far from "doing anything" !

Compiling C/C++ in the presence of undefined behaviors

20th century interpretation: exploit the freedom left by undefined behaviors to translate the language operations to simple processor instructions, as long as they correctly implement the cases defined by the standard.

Operation	Compiled code
array indexing $t + i$	$\texttt{add}(t, \texttt{mul}(i, \texttt{sizeof}(\tau)))$
pointer dereference $*p$	$\texttt{load}(p)$ $\texttt{store}(p)$
signed integer addition $x + y$	$\texttt{add}(x, y)$

Compiling C/C++ in the presence of undefined behaviors

20th century interpretation: exploit the freedom left by undefined behaviors to translate the language operations to simple processor instructions, as long as they correctly implement the cases defined by the standard.

21st century interpretation: exploit the freedom left by undefined behavior to optimize under the assumption that there are no undefined behaviors, even if the compiled code is absurd if there are some.

Operation	Information usable for optimization
array indexing $t + i$	i is within the bounds of t
pointer dereference $*p$	$p \neq$ `NULL` and is a valid pointer
signed integer addition $x + y$	`INT_MIN` $\leq x + y \leq$ `INT_MAX`

“Optimizing” a division by zero

(Linux kernel, `lib/mpi/mpi-pow.c`. Example discussed in Wang et al, *Undefined behavior: what happened to my code?*, 2012.)

```
if (!msize)
    msize = 1 / msize;  /* provoke a signal */
```

The compiler removes the test and the division.

```
if (!msize) msize = 1 / msize; else skip;
                                        (constant propagation)
⇝ if (!msize) msize = 1 / 0; else skip;
                                (assumption: no undefined behaviors)
⇝ if (!msize) unreachable; else skip;
                                        (redundant branch elimination)
⇝ skip
```

“Optimizing” a null pointer test

(Linux kernel, `drivers/net/tun.c`)

```
unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk; // (1)
    if (!tun) return POLLERR; // (2)
    ...
}
```

The compiler removes line (2): since the `tun->sk` access on line (1) was defined, it must be the case that `tun` is not `NULL` …

Enables an attack by putting valid memory at address 0, using `mmap`.

“Optimizing” a test for integer overflow

(Linux kernel, `fs/open.c`; discussed in Wang et al.)

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0) return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

The compiler assumes `offset + len >= 0`, since `offset` and `len` are positive and the addition `offset + len` is assumed to not overflow.

“Optimizing” error reporting

Some compilers assume not only that no undefined behavior occurred in the past during execution, but also that none will occur in the future!

```
int safe_div(int x, int y)
{
    int res;
    if (y == 0) error("division by zero");  // (1)
    res = x / y;                            // (2)
    return res;
}
```

Some compilers take the liberty to schedule the division (line 2) before the test and the call to `error` (line 1). This defeats the purpose of this function, which is to log an error message and perhaps raise an exception *before* division by zero takes place.

The land mine

> *To me, this is deeply dissatisfying, partially because the compiler inevitably ends up getting blamed, but also because it means that huge bodies of C code are land mines just waiting to explode.*
>
> *(Chris Lattner, 2011)*

A compiler can reduce the security of a source program by eliminating security measures present in the source.

Yet, the compiler is not doing anything malicious! It just applies

- classic, obvious optimizations
  (constant propagation, simplification of conditionals, scheduling);
- the interpretation of “undefined behavior” as “I can produce any machine code of my choosing”.

Navigating the land mine

Static or dynamic analysis tools to detect undefined behaviors and unwanted optimizations.

Optimize less aggressively.

(Many `-fno-*` options in GCC and Clang to turn optimizations off.)

Define more behaviors.

(For example, CompCert C defines integer arithmetic modulo 2^N, evaluation orders, preserving behaviors up to the first undefined behavior, etc.)

Reconsider our choices of programming languages?

Security of compiled code executed in a hostile context

Security of compiled code

We expect the executable code produced by the compiler to be "as secure as" the source code, in the following sense:

> *Any attack on the compiled, executable code (leading to a violation of integrity, confidentiality, or availability) can also be conducted on the source code and explained in terms of the source language semantics.*

Simple case: whole program executed in isolation

The source code:

- a whole program, without free variables
- interacts via explicit input/output operations.

The compiled code:

- executes in isolation ($\rightarrow$ lecture #3)
- the only attack surface is to provide bad inputs or observe leaky outputs.

Semantic preservation $\Rightarrow$ secure compilation

A compiler that preserves the semantics of programs (such as CompCert) guarantees that

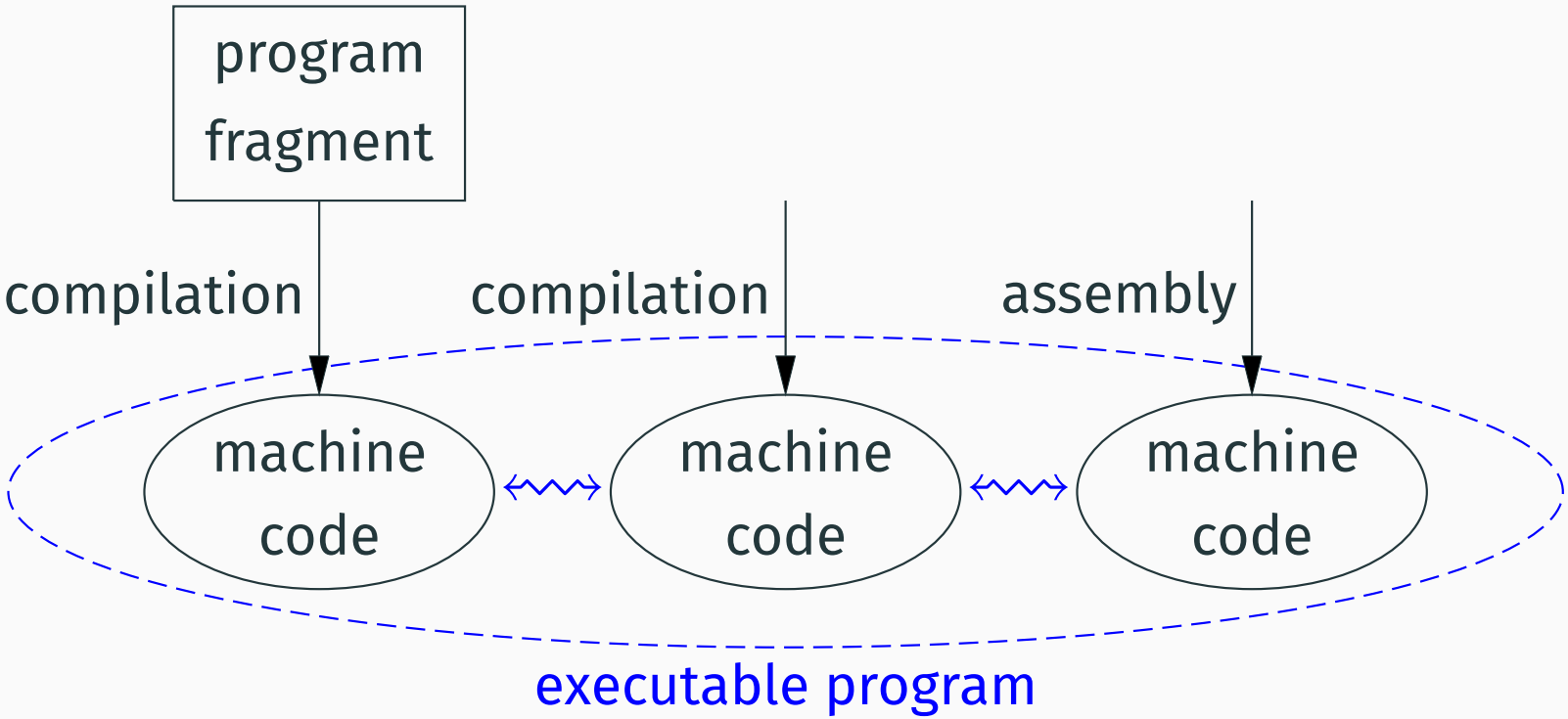
> the observable behavior of the compiled code (I/O traces)
> is one of the behaviors allowed by the semantics of the source.

An attack on the compiled code
= an I/O trace T that triggers an unwanted behavior.

The same trace T triggers the same unwanted behavior in the source code!

Therefore, compilation is secure.

Separate compilation and linking



A program fragment: function, class, abstract type, …
compiled to machine code,
then linked with other pieces of machine code
(compiled from the same language, or from another language, or hand-written).

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Given the address p of an instance of `Account`, machine code can access field `bal` directly (typically at address $p + 8$ or $p + 16$), bypassing the `private` protection $\Rightarrow$ *the invariant is violated.*

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Machine code can observe the contents of registers and stack after a call to `deposit`, and recover the value of `bal`.

$\Rightarrow$ *confidential information leakage.*

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Machine code can call `deposit` with a `this` argument that is the null pointer or a pointer to a different object

$\Rightarrow$ *crash or corruption of another object.*

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Machine code can call `deposit` with `amount` being a 32-bit integer larger than a `short`, bypassing the non-overflow test.

$\Rightarrow$ *the invariant is violated.*

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Machine code can jump directly to address `withdraw` $+ \delta$,
skipping over the validation of `amount`

$\Rightarrow$ *the invariant is violated.*

Machine-level attacks on a Java class

```
class Account {
    private short bal;  // always >= 0
    void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    void withdraw(short amount) {
        if (amount >= 0 && amount <= bal)
            bal -= amount;
    }
}
```

Machine code can call `deposit` with a return address and a call stack doctored so as to fool the stack inspection performed by the `SecurityManager`. $\Rightarrow$ *confused deputy attack.*

A JVM-level attack on a Java class

(M. Abadi, *Protection in programming-language translations*, 1998.)

The JVM has no notion of inner class. Java's inner classes are compiled to disjoint JVM classes, making it necessary to increase the visibility of some `private` fields.

Java source code:

```
class D {
    class E {
        private int y = x;
    }
    private int x;
    public void set_x(int v) { this.x = v; }
}
```

A JVM-level attack on a Java class

(M. Abadi, *Protection in programming-language translations*, 1998.)

The JVM has no notion of inner class. Java's inner classes are compiled to disjoint JVM classes, making it necessary to increase the visibility of some `private` fields.

The compiled JVM code, in pseudo-Java:

```
class D {
  private int x;
  public void set_x(int v)
    { this.x = v; }
  static int access$000(D d)
    { return d.x; }
}
```

```
class D$E {
  final D this$0;
  private int y;
  D$E(D d)
  { this$0 = d;
    y = D.access$000(d); }
}
```

Hand-written JVM code can read `x` by invoking `D.access$000`.

Low-level attacks and observational equivalences

Abadi (1998) proposes to study these low-level attacks on compiled code in terms of observational equivalences and their (non-)preservation during compilation.

Rough idea:

a low-level attack'
= two source code fragments F_1, F_2
indistinguishable at the source language level
whose compiled codes $\mathcal{C}(F_1)$, $\mathcal{C}(F_2)$
can be distinguished at the target language level.

Example where observational equivalence is not preserved

```
class D {
    class E {
        private int y = x;
    }
    private int x;
    public void set_x(int v)
    { this.x = v; }
}
```

```
class D {
    class E {
        private int y = 0;
    }
    private int x;
    public void set_x(int v)
    { this.x = v; }
}
```

The two `D` classes cannot be distinguished by Java code,
but their JVM codes can be distinguished by JVM code
(the class on the left has a method `access$000` that reveals the value of `x`, but not the class on the right).

Observational equivalence and full abstraction

Observational equivalence

Two code fragments (functions, classes, abstract types, libraries) F_1 and F_2 are observationally equivalent if, for every context C (= program with a hole), the programs $C[F_1]$ and $C[F_2]$ behave identically w.r.t. termination:

$$F_1 \approx F_2 \quad \stackrel{def}{=} \quad \forall C,\ C[F_1] \text{ terminates} \iff C[F_2] \text{ terminates}$$

Example: the two Java classes `D` shown previously are observationally equivalent.

Examples of observational equivalences for a strict, typed functional language

At base types, equivalence is equality:

$$0 \approx 0 \qquad 0 \not\approx 1$$

(The context $\texttt{if}\ [\,] = 0\ \texttt{then}\ \Omega\ \texttt{else}\ ()$ distinguishes 0 from 1.)

For functions over base types, equivalence is extensional equality (same arguments give same results):

$$(\lambda x : \texttt{int}.\, x + x) \approx (\lambda x : \texttt{int}.\, x \times 2)$$

$$\texttt{succ} \not\approx \texttt{pred}$$

$$(\lambda x : \texttt{unit}.\, x) \approx (\lambda x : \texttt{unit}.\, ())$$

$$(\lambda x : \texttt{bool}.\, x) \approx (\lambda x : \texttt{bool}.\, \texttt{if}\ x\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false})$$

($\texttt{if}\ [\,]\, 0 = 1\ \texttt{then}\ \Omega\ \texttt{else}\ ()$ distinguishes succ from pred.)

Examples of observational equivalences

For higher-order functions, equivalence reveals how these functions use their arguments.

$$(\lambda f : \texttt{unit} \to \texttt{unit}.\, f\,()) \approx (\lambda f : \texttt{unit} \to \texttt{unit}.\, f\,();f\,())$$

$$(\lambda f : \texttt{unit} \to \texttt{unit}.\, f\,()) \not\approx (\lambda f : \texttt{unit} \to \texttt{unit}.\, ())$$

(The context $[\,]\ (\lambda x.\Omega)$ distinguishes the bottom two functions.)

"Left sequential or" and "right sequential or" are not equivalent:

$$\texttt{or}_{\mathit{left}} \stackrel{\mathit{def}}{=} \lambda x, y : \texttt{unit} \to \texttt{bool}.\ \texttt{if}\ x()\ \texttt{then true else}\ y()$$

$$\texttt{or}_{\mathit{right}} \stackrel{\mathit{def}}{=} \lambda x, y : \texttt{unit} \to \texttt{bool}.\ \texttt{if}\ y()\ \texttt{then true else}\ x()$$

(They are distinguished by the context $[\,]\ (\lambda x.\Omega)\ (\lambda x.\texttt{true})$)

Observational equivalence and security properties

Several basic security properties of a programming language can be characterized by observational equivalences.

Integrity of local variables:

$$\begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad \mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in} \\ \quad f(); !x \end{array} \quad \approx \quad \begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad f(); 0 \end{array}$$

Confidentiality of local variables:

$$\begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad \mathtt{let}\ x = 0\ \mathtt{in}\ f() \end{array} \quad \approx \quad \begin{array}{l} \lambda f : \mathtt{unit} \to \mathtt{unit}. \\ \quad \mathtt{let}\ x = 1\ \mathtt{in}\ f() \end{array}$$

Observational equivalence and security properties

Indifference with respect to allocation order:

$$\begin{array}{lcl} \texttt{let } x = \texttt{ref } 0 \texttt{ in} & \approx & \texttt{let } x = \texttt{ref } 0 \texttt{ in} \\ \texttt{let } y = \texttt{ref } 0 \texttt{ in} & & \texttt{let } y = \texttt{ref } 0 \texttt{ in} \\ x & & y \end{array}$$

Procedural encapsulation:

$$\begin{array}{lcl} \texttt{let } c = \texttt{ref } 0 \texttt{ in} & \approx & \texttt{let } c = \texttt{ref } 0 \texttt{ in} \\ \lambda().\, \texttt{incr } c;\ !c & & \lambda().\, \texttt{decr } c;\ 0 - !c \end{array}$$

(The internal state c goes $0, 1, 2, \ldots$ for the first function and $0, -1, -2, \ldots$ for the second function, but this is not observable outside of the functions.)

Observational equivalence and security properties

Type abstraction:

```
struct
  type t = permission list
  let init () = [P0;P1;P2]
  let allowed = List.mem
  let drop = List.remove
end : Capa
```

$\approx$

```
struct
  type t = int
  let mask = function
    P0 -> 1 | P1 -> 2 | P2 -> 4
  let init () = 7
  let allowed p c =
    c land mask p <> 0
  let drop p c =
    c land lnot (mask p)
end : Capa
```

(Constructing a logical relation is an effective way to prove observational equivalence.)

Observational equivalence and denotational semantics

Consider a denotational semantics in the style of Milner (1978) (as seen in lecture #5):

$$\mathcal{D} : \mathit{Expr} \to \mathit{Env} \to V$$

where V is a Scott domain such that

$$V = (\mathit{Int} + \cdots + (V \to V) + \{\texttt{wrong}\})_\perp$$

The semantics is adequate if $\mathcal{D}(e_1) = \mathcal{D}(e_2) \Rightarrow e_1 \approx e_2$

The semantics is complete if $e_1 \approx e_2 \Rightarrow \mathcal{D}(e_1) = \mathcal{D}(e_2)$

The semantics is fully abstract if it is adequate and complete.

Adequacy

$$\mathcal{D}(e_1) = \mathcal{D}(e_2) \Rightarrow e_1 \approx e_2$$

Example of a non-adequate semantics:
a semantics that “doesn’t have enough values” and maps the constants `true` and `false` of the language to the same element of V. But this is a terrible semantics!

In practice, “good” denotational semantics are always adequate.

Adequacy

$$\mathcal{D}(e_1) = \mathcal{D}(e_2) \Rightarrow e_1 \approx e_2$$

Proof sketch:

Assume $e_1 \not\approx e_2$. Thus, we have a context C such that

$$C[e_1] \text{ diverges} \qquad C[e_2] \text{ terminates}$$

We expect the semantics to distinguish terminating terms from diverging terms: $\mathcal{D}(C[e_1]) = \bot \quad \mathcal{D}(C[e_2]) \neq \bot$

We expect the semantics to be compositional:
$\mathcal{D}(C[e])$ is a function of $\mathcal{D}(e)$.

Therefore, $\mathcal{D}(e_1) \neq \mathcal{D}(e_2)$, and the adequacy result follows by contraposition.

Completeness

$$e_1 \approx e_2 \Rightarrow \mathcal{D}(e_1) = \mathcal{D}(e_2)$$

Many “good” denotational semantics are incomplete, typically because the domain V contains semantic values that cannot be defined within the programming language.

Example: in Scott domains, the set of continuous functions $V \to V$ contains “parallel” functions that cannot be defined in a purely sequential language.

Parallel “or”

A lazy “or” function that returns `true` as soon as one of its arguments returns `true`.

$$\texttt{por}\ x\ y = \begin{cases} \texttt{true} & \text{if } x() = \texttt{true} \text{ (even if } y() \text{ diverge)} \quad (1) \\ \texttt{true} & \text{if } y() = \texttt{true} \text{ (even if } x() \text{ diverge)} \quad (2) \\ \texttt{false} & \text{if } x() = \texttt{false} \text{ and } y() = \texttt{false} \\ \bot & \text{otherwise} \end{cases}$$

It cannot be defined in a sequential language: the evaluation must start either by evaluating $x()$, invalidating (2), or by evaluating $y()$, invalidating (1).

Incompleteness of Scott domains

(G. Plotkin, *LCF considered as a programming language*, 1977.)

Define two "or-tasting functions" M_0 and M_1:

$$
\begin{aligned}
M_k \stackrel{def}{=}\ & \lambda f : (\mathtt{unit} \to \mathtt{bool}) \to (\mathtt{unit} \to \mathtt{bool}) \to \mathtt{bool}. \\
& \quad \mathtt{if}\ \ f\ (\lambda\_.\mathtt{true})\ (\lambda\_.\Omega) \\
& \quad \wedge\, f\ (\lambda\_.\Omega)\ (\lambda\_.\mathtt{true}) \\
& \quad \wedge\, \neg\, f\ (\lambda\_.\mathtt{false})\ (\lambda\_.\mathtt{false}) \\
& \quad \mathtt{then}\ \ k\ \ \mathtt{else}\ \ \Omega
\end{aligned}
$$

M_0 and M_1 do not have the same denotation, since

$$\mathcal{D}(M_0)\,\mathtt{por} = 0 \neq 1 = \mathcal{D}(M_1)\,\mathtt{por}$$

Yet, $M_0 \approx M_1$, since no function f definable in the PCF language satisfies the three conditions of the `if`.

Fully-abstract compilation

Fully-abstract compilation

Abadi (1998) defines full abstraction for a compiler as the preservation of observational equivalences in both directions:

$$F_1 \approx_{\text{source}} F_2 \quad \text{if and only if} \quad \mathcal{C}(F_1) \approx_{\text{compiled}} \mathcal{C}(F_2)$$

Such a fully-abstract compiler is interesting for security:

- All low-level attacks (by a context written in machine language) are also possible at the high level (by a context written in the source language, then compiled).
- Therefore, we can reason about the correctness and the security of a program entirely at the source language level.

“Adequacy”: compilation that preserves semantics

$$F_1 \not\approx_{\text{source}} F_2 \quad \Longrightarrow \quad \mathcal{C}(F_1) \not\approx_{\text{compiled}} \mathcal{C}(F_2)$$

This generally follows from the fact that a correct compiler preserves the semantics of source programs.

Let S be a source context such that $S[F_1]$ diverges but not $S[F_2]$.

By semantic preservation, the compiled code $\mathcal{C}(S[F_1])$ diverges but not the compiled code $\mathcal{C}(S[F_2])$.

If the compiler is compatible with separate compilation,
$\mathcal{C}(S[F_1]) = M[\mathcal{C}(F_1)]$ and $\mathcal{C}(S[F_2]) = M[\mathcal{C}(F_2)]$,
where M is a machine context obtained by translating S.

Therefore, $\mathcal{C}(F_1)$ et $\mathcal{C}(F_2)$ are not observationally equivalent.

“Completeness”: compilation that preserves equivalences

$$F_1 \approx_{\text{source}} F_2 \quad \Longrightarrow \quad \mathcal{C}(F_1) \approx_{\text{compiled}} \mathcal{C}(F_2)$$

This is the difficult part of fully-abstract compilation.

We saw several examples where a machine-level context M can, by direct inspection of memory and registers, distinguish two compiled codes $\mathcal{C}(F_1)$ and $\mathcal{C}(F_2)$.

It is often impossible to construct a source context S (by “back translation” of M) that can distinguish F_1 and F_2 at the source language level.

An example of fully-abstract compilation: from a statically-typed language to a dynamically-typed language

Source language: simply-typed λ-calculus with Booleans, products, and sums.

Types: $\tau, \sigma ::= \texttt{unit} \mid \texttt{bool} \mid \sigma \to \tau$
$\mid \sigma \times \tau \mid \sigma + \tau$

Terms:

$a ::= x \mid \lambda x : \tau.\, a \mid a_1\, a_2 \mid \texttt{fix}$	functions (recursive)
$\mid \texttt{()} \mid \texttt{false} \mid \texttt{true}$	
$\mid \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3$	Booleans
$\mid (a_1, a_2) \mid \texttt{fst}(a) \mid \texttt{snd}(a)$	products
$\mid \texttt{inl}(a) \mid \texttt{inr}(a) \mid \texttt{match}\ldots$	sums

Target language: the same λ-calculus, but dynamically typed.

Compilation

(Devriese, Patrignani, Piessens, *Fully-abstract compilation by approximate back-translation*, 2016.)

Basic compilation: just erase the types! $\mathcal{C}(a) = \overline{a}$

However, this does not preserve equivalences…

$$\lambda x : \mathtt{unit}.\ x \quad \approx_{\text{typed}} \quad \lambda x : \mathtt{unit}.\ ()$$
$$\lambda x.\ x \quad \not\approx_{\text{untyped}} \quad \lambda x.\ ()$$

Secure compilation: erase types + protect the term by dynamically checking the values coming from the context (e.g. the argument x in $\lambda x : \mathtt{unit}.\ x$).

$$\mathcal{C}(a : \tau) = \mathtt{protect}_{\tau}(\overline{a})$$

Protection against ill-typed contexts

From the typed world to the untyped world:

$$\texttt{protect}_{\texttt{unit}} = \lambda x.x$$

$$\texttt{protect}_{\texttt{bool}} = \lambda x.x$$

$$\texttt{protect}_{\sigma\times\tau} = \lambda x.(\texttt{protect}_{\sigma}(\texttt{fst}(x)), \texttt{protect}_{\tau}(\texttt{snd}(x)))$$

$$\texttt{protect}_{\sigma\to\tau} = \lambda f.\lambda x.(\texttt{protect}_{\tau}(f\ (\texttt{confine}_{\sigma}(x))))$$

From the untyped world to the typed world:

$$\texttt{confine}_{\texttt{unit}} = \lambda x.()$$

$$\texttt{confine}_{\texttt{bool}} = \lambda x.\ \texttt{if}\ x\ \texttt{then true else false}$$

$$\texttt{confine}_{\sigma\times\tau} = \lambda x.(\texttt{confine}_{\sigma}(\texttt{fst}(x)), \texttt{confine}_{\tau}(\texttt{snd}(x)))$$

$$\texttt{confine}_{\sigma\to\tau} = \lambda f.\lambda x.(\texttt{confine}_{\tau}(f\ (\texttt{protect}_{\sigma}(x))))$$

Example of protection

$$
\begin{aligned}
&\mathcal{C}(\lambda x : \texttt{bool}.\, x) \\
&\quad = \quad \lambda x.\, \texttt{let } x = \texttt{if } x \texttt{ then true else false in } x \\
&\mathcal{C}(\lambda x : \texttt{bool}.\, \texttt{if } x \texttt{ then true else false}) \\
&\quad = \quad \lambda x.\, \texttt{let } x = \texttt{if } x \texttt{ then true else false in} \\
&\qquad\qquad \texttt{if } x \texttt{ then true else false}
\end{aligned}
$$

Observational equivalence holds even in untyped contexts: the two protected functions map `true` to `true`, `false` to `false`, and all other values to `wrong`.

Full abstraction by back-translation of contexts

Consider an untyped context M that distinguishes $\mathcal{C}(a_1 : \tau)$ and $\mathcal{C}(a_2 : \tau)$

$M\ (\texttt{protect}_\tau(\overline{a_1}))$ diverges $\qquad$ $M\ (\texttt{protect}_\tau(\overline{a_2}))$ terminates

Can we "back-translate" M to a typed context S that distinguishes a_1 from a_2?

Yes, relatively easily, if the source language contains a universal type `U` able to represent untyped terms:

```
type U = Wrong | Unit | Bool of bool
       | Pair of U * U | Fun of (U -> U)
```

We translate M in a typed context S_U where the hole has type `U`, then into S where the hole has type τ.

Full abstraction by back-translation of contexts

Consider an untyped context M that distinguishes $\mathcal{C}(a_1 : \tau)$ and $\mathcal{C}(a_2 : \tau)$

$M\ (\mathtt{protect}_\tau(\overline{a_1}))$ diverges $\qquad$ $M\ (\mathtt{protect}_\tau(\overline{a_2}))$ terminates

Can we "back-translate" M to a typed context S that distinguishes a_1 from a_2?

Yes, with much more work, even if the source language has no recursive types, using level-n approximations of type $\mathtt{U}$:

$$U_0 = \mathtt{unit} \qquad U_{n+1} = \mathtt{unit} + \mathtt{bool} + (U_n \times U_n) + (U_n \to U_n)$$

n is chosen large enough to observe the difference of behavior between a_1 and a_2.

(Devriese, Patrignani, Piessens, *Fully-abstract compilation by approximate back-translation*, 2016.)

Fully-abstract compilation of a statically-typed language to JavaScript

(Fournet, Swamy, Chen, Dagand, Strub, Livshits, *Fully Abstract Compilation to JavaScript*, 2013.)

Source language: functional and imperative, with static typing (originally a fragment of F*; in later work, TypeScript).

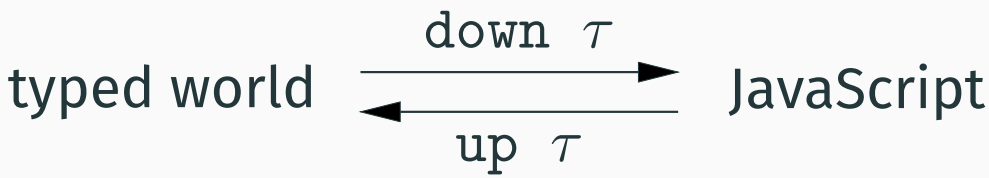
Target language: JavaScript.

Compilation: naive translation that erases types

$$[\![\lambda x : \tau.\ a]\!] = \texttt{function}(x)\{\texttt{var}\ \vec{y}; \texttt{return}\ [\![a]\!];\}$$
$$[\![a_1\ a_2]\!] = [\![a_1]\!]\ [\![a_2]\!]$$
$$[\![(a_1, a_2)]\!] = \{\texttt{tag} : \texttt{"Pair"};\ 0 : [\![a_1]\!];\ 1 : [\![a_2]\!]\}$$

followed by protection against the context.

Protection functions




```
function downunit(x) { return x;}
function upunit(x) { return undefined;}
function downbool(x) { return x;}
function upbool(z) { return (z ? true : false);}
function downstring(x) { return x;}
function upstring(x) { return (x + "");}

function downpair(dn_a, dn_b) {
  return function (p) {
    return {"tag":"Pair", "0":dn_a(p["0"]),"1":dn_b(p["1"])};}}
function uppair(up_a, up_b) {
  return function(z) {
    return {"tag":"Pair", "0":up_a(z["0"]),"1":up_b(z["1"])};}}
```

Protecting function values

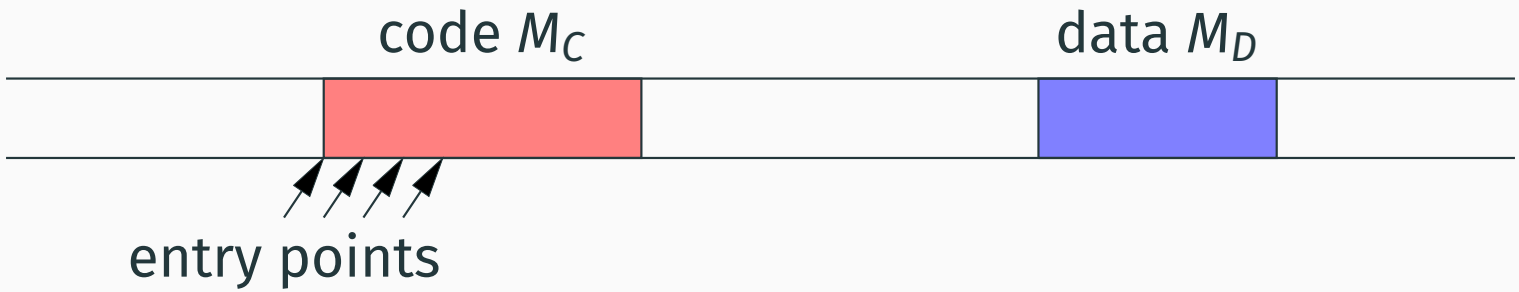
```
function downfun (up_a,down_b) {
  return function (f) {
    return function (z) { return (down_b (f (up_a(z)))); }}}

function upfun (down_a,up_b) {
  return function (f) {
    return function (x) {
      var z = down_a(x);
      var y = undefined;
      function stub(b) {
        if (b) { stub(false); } else { y = up_b(f(z)); } }
      stub(true); return y; };};}
```

All these tricks prevent `f` (which is provided by the possibly malicious context) to inspect the call stack and to interfere with the protection.

Using hardware protection mechanisms such as “enclaves”

(Agten, Strackx, Jacobs, Piessens, *Secure compilation to modern processors*, 2012).

Assume given two protected memory areas, a code area M_C and a data area M_D:



Code outside M_C can only jump to one of the entry points.

Only the code within M_C can access data in M_D or jump elsewhere within M_C.

(Examples: Intel SGX enclaves; Memory Access Controllers on smart cards.)

Source language (simplified)

Statically-allocated objects having private instance variables and public or private methods, taking and returning values of base types.

```
class Account {
    private short bal = 0;
    public void deposit(short amount) {
        if (amount >= 0 && bal + amount <= Short.MAX_VALUE)
            bal += amount;
    }
    public short balance() { return bal; }
}
```

Compilation (simplified)

Instance variables and a private stack are placed in the protected data area M_D.

Code is placed in the protected code area M_C.

Public methods are entry points.

On entry to each public method:

- switch to the private stack, save the return address;
- validate the arguments (e.g. `short` is in $[-2^{15}, 2^{15} - 1]$).

On exit:

- erase all registers except the one containing the result value
- reload the return address R; check that $R \notin M_C$
- switch to the caller's stack and jump to address R.

Main results

(Agten, Strackx, Jacobs, Piessens, *Secure compilation to modern processors*, 2012; Patrignani, Clarke, Piessens, *Secure compilation of object-oriented components to protected module architectures*, 2013.)

Extensions of the compilation schema:

- Callbacks from methods to functions (in non-protected code area) passed as arguments.
- Dynamic allocation of objects in the protected data area.

Semantics based on execution traces that capture function and method calls and returns.

A full abstraction result: if a machine context M in non-protected code area distinguishes $\mathcal{C}(O_1)$ from $\mathcal{C}(O_2)$, we know how to construct a source context S that distinguishes O_1 from O_2.
S is constructed from the traces of $M[\mathcal{C}(O_1)]$ and $M[\mathcal{C}(O_2)]$.

Summary

Principles of secure compilation

An active research area.

A characterization in terms of full abstraction that is elegant, but extraordinarily difficult to achieve.

Other characterizations were proposed, in terms of compilation that preserves (hyper-)properties:

- properties of one run of the program
  $\rightarrow$ preservation of integrity guarantees
- hyper-properties relating of two runs of the program
  $\rightarrow$ preservation of confidentiality guarantees.

(See the survey by Patrignani, Ahmed and Clarke, *Formal Approaches to Secure Compilation*, 2019.)

Practice of secure compilation

A tension (of a social nature) between

- aggressive optimization of codes (perceived as) having no security implications;
- respecting the intent of the source code, including security protections.

A nonobvious choice between

- adding security protections during compilation, or
- adding security protections to the source code and preserving them during compilation.