CS 489 / 698: Software and Systems Security

Module: Defenses against Common Vulnerabilities

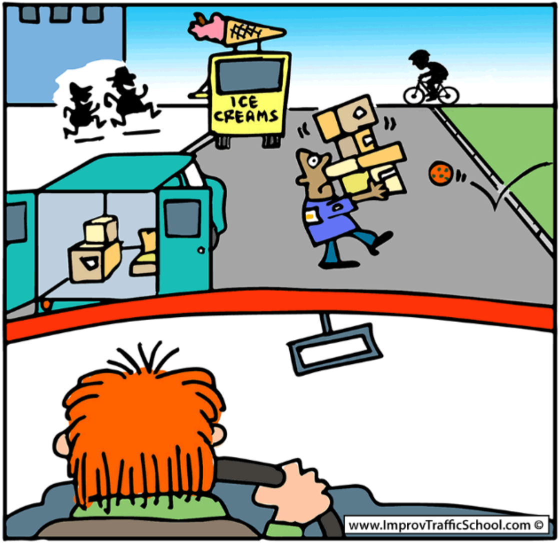
Lecture: runtime sanity checking

Meng Xu *(University of Waterloo)*

Fall 2024

Outline

Defensive driving



Credits / Trademark: `www.ImprovTafficSchool.com`

Defensive programming

Like defensive driving, defensive programming requires the developer to anticipate what might go wrong in the software and program defensively against these anticipated issues, potentially with the help of compiler, runtime, or even external auditors.

Defensive programming

Driving	Programming
Follow traffic rules Follow local customs	Follow typing rules Follow coding conventions

In normal paradigm: expect others to follow the rules
In defensive paradigm: expect others to ignore / by-pass the rules

Defensive programming

Driving	Programming
Follow traffic rules	Follow typing rules
Follow local customs	Follow coding conventions

In normal paradigm: expect others to follow the rules
In defensive paradigm: expect others to ignore / by-pass the rules

Apply defensive actions **at the cost of performance**

Outline

Paranoia

Defining **paranoia**:

a mental condition characterized by delusions of persecution, unwarranted jealousy, or exaggerated self-importance, typically elaborated into an organized system.

Example: NULL-check for every pointer access

```c
int foo_inner(int *ptr) {
  return *ptr;
}
```

```diff
 int foo_inner(int *ptr) {
+ if (ptr == NULL) {
+   abort("nullptr exception");
+ }
  return *ptr;
 }
```

```c
int foo_outer(int arg) {
  // guaranteed non-null
  return foo_inner(&arg);
}

int foo_inner(int *ptr) {
  return *ptr;
}
```

```diff
 int foo_outer(int arg) {
  // guaranteed non-null
  return foo_inner(&arg);
 }

 int foo_inner(int *ptr) {
+ if (ptr == NULL) {
+   abort("nullptr exception");
+ }
  return *ptr;
 }
```

Example: NULL-check for every pointer access

```c
int foo_inner(int *ptr) {
  return *ptr;
}
```

```c
int foo_inner(int *ptr) {
+ if (ptr == NULL) {
+   abort("nullptr exception");
+ }
  return *ptr;
}
```

```c
int foo_outer(int *ptr) {
  *ptr = 42;
  // guaranteed non-null
  return foo_inner(ptr);
}

int foo_inner(int *ptr) {
  return *ptr;
}
```

```c
int foo_outer(int *ptr) {
+ if (ptr == NULL) {
+   abort("nullptr exception");
+ }
  *ptr = 42;
  // guaranteed non-null
  return foo_inner(ptr);
}

int foo_inner(int *ptr) {
+ if (ptr == NULL) {
+   abort("nullptr exception");
+ }
  return *ptr;
}
```

Is this really paranoia?

Is this really paranoia?

This paranoid check is actually happening in Java / Python / . . . .

- therefore, this is not a stupid idea.

It helps to guard against a very subtle and implicit assumption:
what if `foo_inner()` is not an internal function anymore?

Undefined behavior sanitizer (UBSan)

NULL-pointer dereference is just one case of undefined behaviors, there are many other cases of undefined behaviors in C-like languages. UBSan in the LLVM compiler toolchain provides a comprehensive list of checkers.

Undefined behavior sanitizer (UBSan)

NULL-pointer dereference is just one case of undefined behaviors, there are many other cases of undefined behaviors in C-like languages. UBSan in the LLVM compiler toolchain provides a comprehensive list of checkers.

- `-fsanitize=bool`
  Load of a bool value which is neither true nor false.
- `-fsanitize=bounds`
  Out of bounds indexing, in cases where the bound is statically known
- `-fsanitize=function`
  Indirect call of a function through a pointer of the wrong type
- `-fsanitize=null`
- `-fsanitize=integer-divide-by-zero`
- `-fsanitize=integer-overflow`
- ...

Undefined behavior sanitizer (UBSan)

Q: What do the checks in UBSan have in common?

Undefined behavior sanitizer (UBSan)

Q: What do the checks in UBSan have in common?

A: They are stateless sanity checks, i.e., the execution can be considered as either valid or invalid by simply examining the statement / instruction and its operand.

As a consequence, sanity checks in UBSan are independent of each other (allows modularity), easy to instrument at compile time, and less expensive (performance-wise) to check at runtime. Typical runtime overhead of UBSan is 20%.

But UBSan is far from enough

- `-fsanitize=bounds`

  Out of bounds indexing, in cases where the bound is statically known

But UBSan is far from enough

- `-fsanitize=bounds`

  Out of bounds indexing, in cases where the bound is statically known

Q: What about cases where bounds cannot be statically determined?

```
long* mk_array(int len) {
  return malloc(sizeof(long) * len);
}
void set_value(long *arr, int idx, long val) {
    arr[idx] = val;
}
long get_value(long *arr, int idx) {
    return arr[idx];
}
```

Recall memory safety definition

At any point of time during the program execution,
for any object in memory, we know its
`(object_id, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we know:

- Memory read: `(object_id, offset [int], length [int])`
- Memory write: `(object_id, offset [int], length [int]), _)`
- Memory free: `(object_id)`

Violation of spatial safety:

- `offset + length >= size`
- `offset < 0`

Violation of temporal safety:

- Read: `status != init`
- Write: `status == dead`
- Free: `status == dead`

On the practicality of these checks

This full suite of memory safety checks is inpractical. The performance overhead is at least 200% if not more, making it impossible to be deployed in production systems [1].

[1] In fact, I am not aware of any tool that strictly follows the above definition. Practicality aside, such a tool is extremely valuable as a debugging tool that runs during testing time. Implementing such a tool does not seem to be very difficult in LLVM, so let me know if you are interested in this direction.

Outline

A typical technique in sanitizers

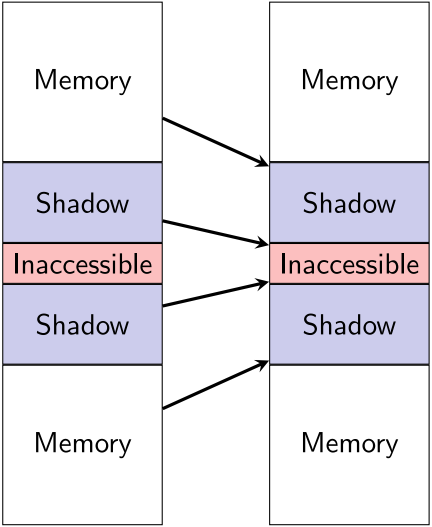
Credits / Trademark: `World Atlas`

Case study: AddressSanitizer (ASan)

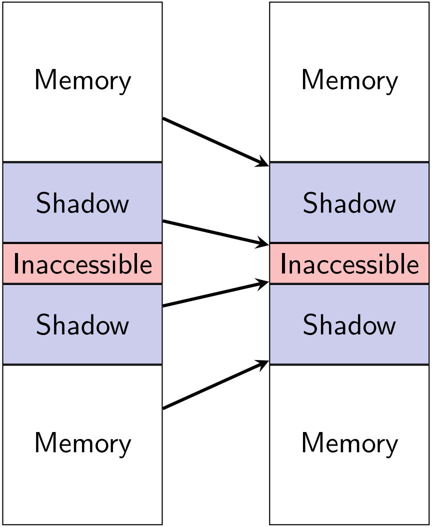
ASan is an efficient and industrial-grade implementation of memory error detector in both LLVM and GCC.

The alleged runtime overhead of ASan is 70% on average, making it almost suitable to run in a production environment. A series of follow-up work further improves the overhead situation.

ASan: shadow memory



ASan: shadow memory

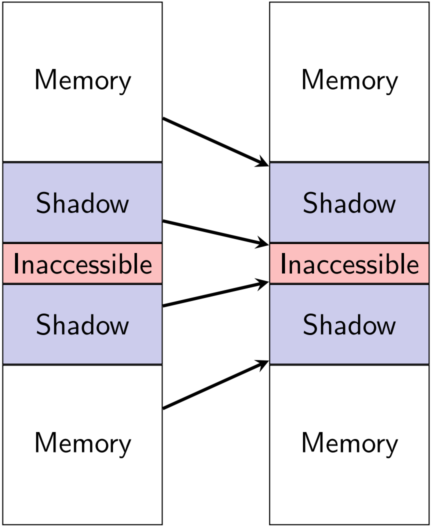


Fact 1: fast shadow translation

`Shadow = (Mem >> 3) + 0x7fff8000;`

[0x10007fff8000, 0x7fffffffffff]	HighMem
[0x02008fff7000, 0x10007fff7fff]	HighShadow
[0x00008fff7000, 0x02008fff6fff]	ShadowGap
[0x00007fff8000, 0x00008fff6fff]	LowShadow
[0x000000000000, 0x00007fff7fff]	LowMem

ASan: shadow memory



Fact 1: fast shadow translation

```
Shadow = (Mem >> 3) + 0x7fff8000;
```

`[0x10007fff8000, 0x7fffffffffff]`	`HighMem`
`[0x02008fff7000, 0x10007fff7fff]`	`HighShadow`
`[0x00008fff7000, 0x02008fff6fff]`	`ShadowGap`
`[0x00007fff8000, 0x00008fff6fff]`	`LowShadow`
`[0x000000000000, 0x00007fff7fff]`	`LowMem`

Fact 2: compact representation

By default, ASan maps 8 bytes of the application memory into 1 byte of the shadow memory *(1 bit per byte)*.

ASan: instrumentation for shadow memory

```
void foo() {
  char a[8];
  ...
  return;
}
```

```
void foo() {
  // instrumentation around a stack object
  char redzone1[32];  // 32-byte aligned
  char a[8];
  char redzone2[24];  // 32-byte aligned

  // instrumentation before return address
  char redzone3[32];  // 32-byte aligned
  int  *shadow_base = MemToShadow(redzone1);

  // poison redzone1
  shadow_base[0] = 0xffffffff;
  // poison redzone2, unpoison 'a'
  shadow_base[1] = 0xffffff00;
  // poison redzone3
  shadow_base[2] = 0xffffffff;

  ...

  // unpoison all
  shadow_base[0] = shadow_base[1] = shadow_base[2] = 0;
  return;
}
```

ASan: instrumentation for sanity check

Before:

```
*address = ...; // or:  ...  = *address;
```

After:

```
if (*MemToShadow(address) != 0) {
  ReportError(address, ...);
}
*address = ...; // or:  ...  = *address;
```

ASan: instrumentation for temporal rules

```c
void f() {
  int *p;
  if (b) {
    int x[10];
    p = x;
  }
  *p = 1;
}
```

```
  void f() {
    int *p;
    if (b) {
+     __asan_unpoison_stack_memory(x);
      int x[10];
      p = x;
+     __asan_poison_stack_memory(x);
    }
    *p = 1;
+   __asan_unpoison_stack_memory(frame);
  }
```

ASan: limitations

- Continuous overrun detection only
- Limited protection on use-after-free
- Incompatible with other security schemes (e.g., UBSan)
- Not suitable for library developers
  - It is not possible to use an application that is not using ASan with a library that has been compiled with ASan.

Bonus: why can Java do it efficiently?

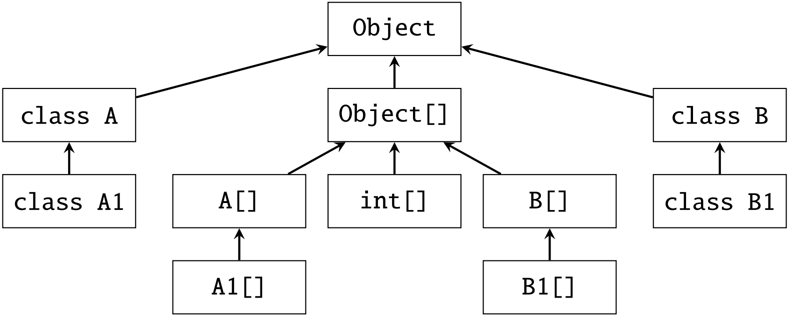
An example of the famous `ArrayIndexOutOfBoundsException`

```
String[] names = { "tom", "bob", "harry" };
for (int i = 0; i <= names.length; i++) {
    System.out.println(names[i]);
}
```

But we are never told that Java has a 70% overhead sanitizer running — *how is this possible?*

Bonus: why can Java do it efficiently?

The key answer is: Java does not allow arbitrary casting.



- Upward cast is always allowed.
- Downward cast may be allowed.
- Re-interpret cast is never allowed.

Outline

A simple example

Compute the value of A_{20} given the following definition[2].

$$A_0 = \frac{11}{2}$$

$$A_1 = \frac{61}{11}$$

$$A_{n+2} = 111 - \frac{1130 - \frac{3000}{A_n}}{A_{n+1}}$$

[2] Example taken from Jose Ignacio Requeno's slides at TAROT 2022 summer school which further acknowledges Cesar Munoz (NASA, Langley) for the code.

Java implementation

```java
public class Mya {

    static double A(int n) {
        if (n == 0) {
            return 11 / 2.0;
        }
        if (n == 1) {
            return 61 / 11.0;
        }

        return 111 - (1130 - 3000 / A(n - 2)) / A(n - 1);
    }

    public static void main(String [] argv) {
        for (int i = 0; i <= 20; i ++) {
            System.out.println("A(" + i + ") = " + A(i));
        }
    }

}
```

The solution (?)

```
A(0) = 5.5
A(1) = 5.545454545454546
A(2) = 5.590163934426235
A(3) = 5.633431085044251
A(4) = 5.674648620514802
A(5) = 5.713329052462441
A(6) = 5.74912092113604
A(7) = 5.781810945409518
A(8) = 5.81131466923334
A(9) = 5.83766396240722
A(10) = 5.861078484508624
A(11) = 5.883542934069212
A(12) = 5.935956716634138
A(13) = 6.534421641135182
A(14) = 15.413043180845833
A(15) = 67.47239836474625
A(16) = 97.13715118465481
A(17) = 99.82469414672073
A(18) = 99.98953968869486
A(19) = 99.9993761416421
A(20) = 99.99996275956511
```

Should we trust the solution?

In fact, mathematically, for any $n \geq 0$, the value of A_n can be computed as following:

$$A_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n}$$

Where

$$\lim_{n \to \infty} A_n = 6$$

Therefore, we expect

$$A_{20} \approx 6$$

Runtime verification (RV)

Verification technique that allow for checking whether a specific run of a program under scrutiny satisfies or violates a given property.

Runtime verification (RV)

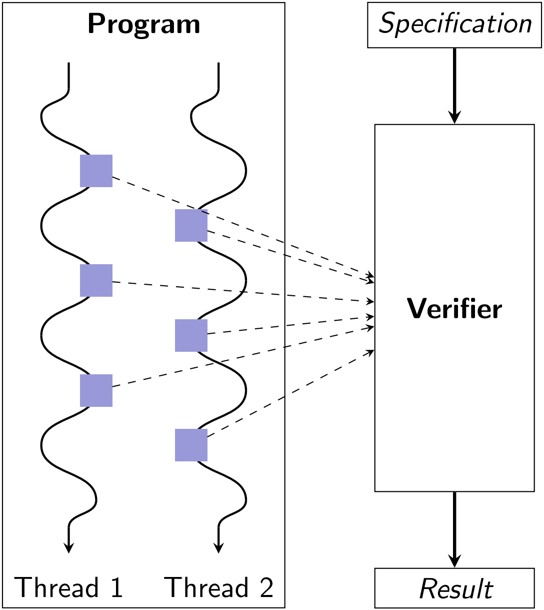
Verification technique that allow for checking whether a specific run of a program under scrutiny satisfies or violates a given property.

The word *"verification"* here is really misleading. It is not the same meaning as in formal verification. Instead, it is more like validation.

The following may help clarify the differences between validation (i.e., runtime verification) and verification (i.e., formal verification).

- Validation: "are we building the right product?"
- Verification: "are we building the product right?"

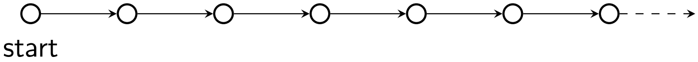
General framework



How to express the specification?

We are trying to specify behaviors of a program over time, i.e., over a sequence of states S_0, S_1, . . . , (potentially endless).

The corresponding mathematical construct we are looking at is called temporal logic, and in particular, concerning a single run of a program, the logic is linear temporal logic (LTL).



LTL specification

In LTL, the specifications are built from:

- Primitive properties of individual states.
  - e.g., "traffic light is green", "lock is acquired", "object is initialized"
- Propositional connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$
- Temporal connectives:
  - $X\phi$: ϕ is true in the **neXt** state.
  - $G\phi$: ϕ is true **Globally**, i.e., in current and all future states.
  - $F\phi$: ϕ is true in some **Future** state.
  - $\phi U\gamma$: ϕ continues to hold true in future states **Until** reaching a state where γ starts to be true.

LTL examples

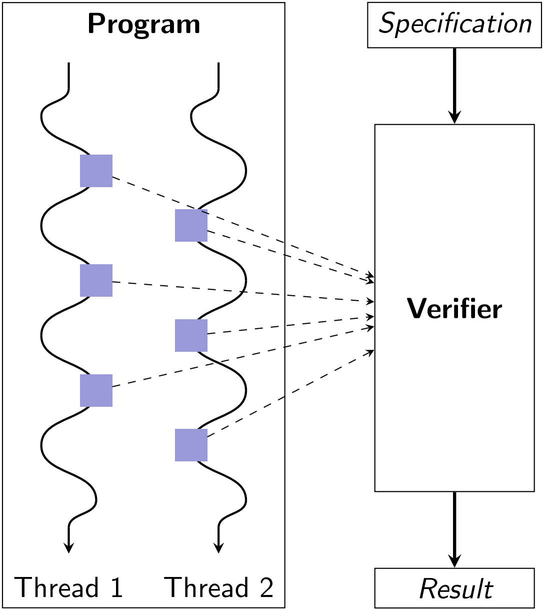
- Temporal connectives:
  - $X\phi$: ϕ is true in the **neXt** state.
  - $G\phi$: ϕ is true **Globally**, i.e., in current and all future states.
  - $F\phi$: ϕ is true in some **Future** state.
  - $\phi U\gamma$: ϕ continues to hold true in future states **Until** reaching a state where γ starts to be true.
- Examples:
  - $win\_lottery \rightarrow |G| rich$
  - $\neg homework \wedge party \rightarrow |X| \neg homework$
  - $start\_lecture \rightarrow talk |U| end\_lecture$
  - $(\neg passport \vee \neg ticket) \rightarrow |F| \neg board\_flight$

Type of properties

- Safety property: something bad will not happen
  - e.g., $|G|(green \rightarrow \neg|X|red)$
- Liveness property: something good will eventually happen
  - e.g., $|G|(|F|green)$
  - e.g., $|G|(red \rightarrow |F|(green \wedge green|U|yellow))$

Outline

Recap: general framework of runtime verification



Information collection

While the temporal logic is a good abstraction of specification writing in runtime verification, we still have the problem of how to collect information at runtime, especially in cases where compiler cannot provide any assistance.

Aspect-oriented programming (AOP)

Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.

It does so by adding behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification.

This allows behaviors that are not central to the business logic (such as logging for runtime verification) to be added to a program without cluttering the code core to the functionality.

AOP example (with intrusive instrumentation)

```java
void transfer(
  Account from,
  Account into,
  int amount,
)
throws Exception {
  if (from.getBalance() < amount)
    throw new InsufficientFunds();

  from.withdraw(amount);
  from.deposit(amount);
}
```

```
void transfer(
  Account from, Account into,
  int amount,
+ User user,
+ Logger logger,
)
throws Exception {
+ logger.info("Transferring...");

+ if (!user.isAuthorised(from)) {
+   logger.info("no permission");
+   throw new Unauthorised();
+ }

  if (from.getBalance() < amount)
    throw new InsufficientFunds();

  from.withdraw(amount);
  from.deposit(amount);

+ logger.info("Transaction done");
}
```

AOP example (with aspects)

```
void transfer(
  Account from,
  Account into,
  int amount,
)
throws Exception {
  if (from.getBalance() < amount)
    throw new InsufficientFunds();

  from.withdraw(amount);
  from.deposit(amount);
}
```

```
aspect Logger {
  Logger logger;

  void Bank.transfer#entry(
    Account from, Account into,
    int amount,
  ) {
    logger.info("Transferring...");
  }
  void Bank.transfer#exit(
    Account from, Account into,
    int amount,
  ) {
    logger.info("Transaction done");
  }
  void User.isAuthorized#exit(
    User user, Account acc,
    boolean success,
  ) {
    if (!success)
      logger.info("no permission");
  }
}
```

Criticism

The most basic criticism of the effect of AOP is that control flow is obscured. The obliviousness of application means that the advices applied are invisible, therefore,

> *one must, in general, have whole-program knowledge to reason about the dynamic execution of an aspect-oriented program.*

Based on Gary T. Leavens's report.

Outline

Introduction

Control-Flow Integrity (CFI) is a classic example of runtime reference monitor in software security.

Introduction

Control-Flow Integrity (CFI) is a classic example of runtime reference monitor in software security.

CFI is also sometimes referred to as program shepherding

monitoring control flow transfers during program execution to enforce a security policy — from a paper in USENIX Security'02.

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

Basic use cases of CFI

```c
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- `f1`, `f2`, `f3`, `f4`, `foo`, `printf`, `system`, ...

Option 2: allowed only functions defined in the current module

- `f1`, `f2`, `f3`, `f4`, `foo`

Option 3: allow functions with type signature `void (*)()`

- `f1`, `f2`, `f3`

Basic use cases of CFI

```
void f1();
void f2();
void f3();
void f4(int, int);

void foo(int usr) {
  void (*func)();

  if (usr == MAGIC)
    func = f1;
  else
    func = f2;

  // forward edge CFI check
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

Option 1: allow all functions

- f1, f2, f3, f4, foo, printf, system, ...

Option 2: allowed only functions defined in the current module

- f1, f2, f3, f4, foo

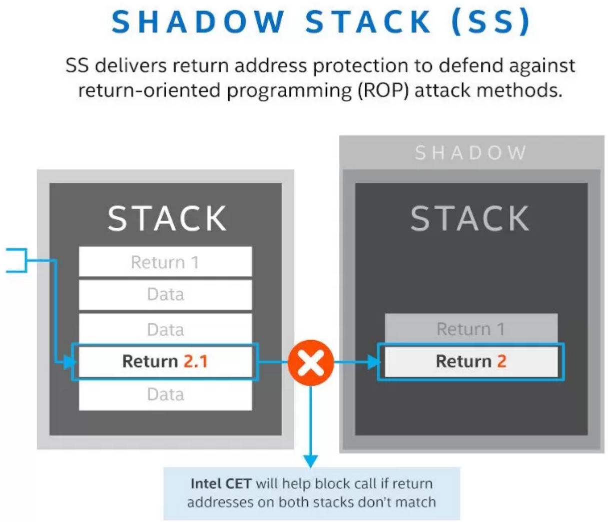
Option 3: allow functions with type signature `void (*)()`

- f1, f2, f3

Option 4: allow functions whose address are taken (e.g., assigned)

- f1, f2

Back-edge protection: shadow stack



Copyright: Intel

Security boundaries of CFI-protected programs

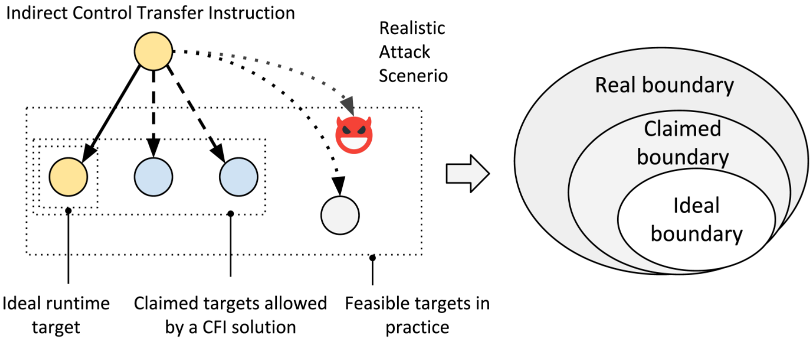


Figure from a paper published in ACM CCS'20

$\langle$ **End** $\rangle$