Module 2: Program Security (Defenses)
runtime sanity checking

Meng Xu (University of Waterloo)
Spring 2023
Outline

1. Introduction
2. Paranoid runtime checking
3. Shadow execution
4. Reference monitor
5. Aspect-oriented programming
6. Control-flow integrity
Defensive driving
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

Follow traffic rules
Follow local customs

Programming

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

Follow traffic rules
Follow local customs

Programming

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

Apply defensive actions at the cost of performance
Outline

1. Introduction

2. Paranoid runtime checking

3. Shadow execution

4. Reference monitor

5. Aspect-oriented programming

6. Control-flow integrity
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
1 int foo_inner(int *ptr) {
2    return *ptr;
3 }

1 int foo_outer(int arg) {
2    // guaranteed non-null
3    return foo_inner(&arg);
4 }

1 int foo_outer(int arg) {
2    // guaranteed non-null
3    return foo_inner(&arg);
4 }

1 int foo_inner(int *ptr) {
2    if (ptr == NULL) {
3        abort("nullptr exception");
4    }
5    return *ptr;
6 }
```
Example: NULL-check for every pointer access

```c
1 int foo_inner(int *ptr) {
2   return *ptr;
3 }

1 int foo_outer(int *ptr) {
2   *ptr = 42;
3   // guaranteed non-null
4   return foo_inner(ptr);
5 }
6
7 int foo_inner(int *ptr) {
8   return *ptr;
9 }
```

```c
1 int foo_outer(int *ptr) {
2   + if (ptr == NULL) {
3     + abort("nullptr exception");
4   + }
5   return *ptr;
6 }
```

```c
1 int foo_outer(int *ptr) {
2   + if (ptr == NULL) {
3     + abort("nullptr exception");
4   + }
5   *ptr = 42;
6   // guaranteed non-null
7   return foo_inner(ptr);
8 }
9
10 int foo_inner(int *ptr) {
11   + if (ptr == NULL) {
12     + abort("nullptr exception");
13   + }
14   return *ptr;
15 }
```
Is this really a paranoia?
Is this really a 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`

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

```c
1 long* mk_array(int len) {
2     return malloc(sizeof(long) * len);
3 }
4 void set_value(long *arr, int idx, long val) {
5     arr[idx] = val;
6 }
7 long get_value(long *arr, int idx) {
8     return arr[idx];
9 }
```
Recall memory safety definition

At any point of time during the program execution, for any object in memory, we know its:

\((\text{object}\_\text{id}, \text{size} [\text{int}], \text{status} [\text{alloc}|\text{init}|\text{dead}])\)

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

- Memory read: \((\text{object}\_\text{id}, \text{offset} [\text{int}], \text{length} [\text{int}])\)
- Memory write: \((\text{object}\_\text{id}, \text{offset} [\text{int}], \text{length} [\text{int}]), _)\)
- Memory free: \((\text{object}\_\text{id})\)

Violation of spatial safety:
- \(\text{offset} + \text{length} \geq \text{size}\)
- \(\text{offset} < 0\)

Violation of temporal safety:
- Read: \(\text{status} \neq \text{init}\)
- Write: \(\text{status} = \text{dead}\)
- Free: \(\text{status} = \text{dead}\)
On the practicality of these checks

This full-suite of memory safety check 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

1. Introduction
2. Paranoid runtime checking
3. Shadow execution
4. Reference monitor
5. Aspect-oriented programming
6. Control-flow integrity
A typical technique in sanitizers
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 production environment. A series of follow-up work further improves the overhead situation.
ASan: shadow memory

Fact 1: fast shadow translation

\[
\text{Shadow} = (\text{Mem} >> 3) + 0x7fff8000;
\]

[0x10007fff8000, 0x7fffffffffff]

[0x02008fff7000, 0x10007fff7fff]

[0x00008fff7000, 0x02008fff6fff]

[0x00007fff8000, 0x00008fff6fff]

[0x000000000000, 0x00007fff7fff]

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).

19 / 49
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 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

```c
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] = 0xffffffff00;
  // poison redzone3
  shadow_base[2] = 0xffffffff;

  // unpoison all
  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;
}
```
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.
An example of the famous ArrayIndexOutOfBoundsException

```java
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 Java can 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

1. Introduction
2. Paranoid runtime checking
3. Shadow execution
4. Reference monitor
5. Aspect-oriented programming
6. Control-flow integrity
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.
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 (?)

1. \( A(0) = 5.5 \)
2. \( A(1) = 5.545454545454546 \)
3. \( A(2) = 5.5901639344262435 \)
4. \( A(3) = 5.633431085044251 \)
5. \( A(4) = 5.674648620514802 \)
6. \( A(5) = 5.713329052462441 \)
7. \( A(6) = 5.74912092113604 \)
8. \( A(7) = 5.781810945409518 \)
9. \( A(8) = 5.81131466923334 \)
10. \( A(9) = 5.83766396240722 \)
11. \( A(10) = 5.861078484508624 \)
12. \( A(11) = 5.883542934069212 \)
13. \( A(12) = 5.935956716634138 \)
14. \( A(13) = 6.534421641135182 \)
15. \( A(14) = 15.413043180845833 \)
16. \( A(15) = 67.47239836474625 \)
17. \( A(16) = 97.13715118465481 \)
18. \( A(17) = 99.82469414672073 \)
19. \( A(18) = 99.98953968869486 \)
20. \( A(19) = 99.9993761416421 \)
21. \( A(20) = 99.99996275956511 \)
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.
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

Program

Thread 1  Thread 2

Specification

Verifier

Result
How to express the specification?

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

\[ \begin{array}{cccccccc}
& - & - & - & - & - & - & - \\
\text{start} & - & - & - & - & - & - & - \\
\end{array} \]
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:** $\land, \lor, \neg, \rightarrow$

- **Temporal connectives:**
  - $X\phi$: $\phi$ is true in the **next** state.
  - $G\phi$: $\phi$ is true **globally**, i.e., in current and all future states.
  - $F\phi$: $\phi$ is true in some **future** state.
  - $\phi U \gamma$: $\phi$ continues to hold true in future states **until** reaching a state where $\gamma$ starts to be true.
LTL examples

- **Temporal connectives:**
  - $X \phi$: $\phi$ is true in the **neXt** state.
  - $G \phi$: $\phi$ is true **Globally**, i.e., in current and all future states.
  - $F \phi$: $\phi$ is true in some **Future** state.
  - $\phi U \gamma$: $\phi$ continues to hold true in future states **Until** reaching a state where $\gamma$ starts to be true.

- **Examples:**
  - $\text{win\_lottery} \rightarrow |G| \text{rich}$
  - $\neg \text{homework} \land \text{party} \rightarrow |X| \neg \text{homework}$
  - $\text{start\_lecture} \rightarrow \text{talk} |U| \text{end\_lecture}$
  - $(\neg \text{passport} \lor \neg \text{ticket}) \rightarrow |F| \neg \text{board\_flight}$
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., $G(red \rightarrow F(green \land green \lor U yellow))$
Outline

1. Introduction
2. Paranoid runtime checking
3. Shadow execution
4. Reference monitor
5. Aspect-oriented programming
6. Control-flow integrity
Recap: general framework of runtime verification

- Program
- Specification
- Verifier
- Result

Thread 1
Thread 2
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) 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);
}
```

```java
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)

```java
1 aspect Logger {
2     Logger logger;
3
4     void Bank.transfer#entry(
5         Account from, Account into,
6         int amount,
7     ) {
8         logger.info("Transferring...");
9     }
10    void Bank.transfer#exit(
11         Account from, Account into,
12         int amount,
13     ) {
14         logger.info("Transaction done");
15     }
16    void User.isAuthorized#exit(
17         User user, Account acc,
18         boolean success,
19     ) {
20         if (!success)
21             logger.info("no permission");
22     }
23 }
```

```java
1 void transfer(
2     Account from,
3     Account into,
4     int amount,
5 )
6     throws Exception {
7         if (from.getBalance() < amount)
8             throw new InsufficientFunds();
9     from.withdraw(amount);
10     from.deposit(amount);
11 }
```
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

1. Introduction
2. Paranoid runtime checking
3. Shadow execution
4. Reference monitor
5. Aspect-oriented programming
6. Control-flow integrity
Control-Flow Integrity (CFI) is a classic example of runtime reference monitor in software security.
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

```c
1  void f1();
2  void f2();
3  void f3();
4  void f4(int, int);
5
6  void foo(int usr) {
7      void (*func)();
8
9      if (usr == MAGIC)
10         func = f1;
11    else
12       func = f2;
13
14    // forward edge CFI check
15    CHECK_CFI_FORWARD(func);
16    func();
17
18    // backward edge CFI check
19    CHECK_CFI_BACKWARD();
20  }
```
Basic use cases of CFI

Option 1: allow all functions
- f1, f2, f3, f4, foo, printf, system, ...

```c
1 void f1();
2 void f2();
3 void f3();
4 void f4(int, int);
5
6 void foo(int usr) {
7   void (*func)();
8
9   if (usr == MAGIC)
10      func = f1;
11   else
12      func = f2;
13
14   // forward edge CFI check
15   CHECK_CFI_FORWARD(func);
16   func();
17
18   // backward edge CFI check
19   CHECK_CFI_BACKWARD();
20 }
```
Basic use cases of CFI

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

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

```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();
}
```
Basic use cases of CFI

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

**SHADOW STACK (SS)**

SS delivers return address protection to defend against return-oriented programming (ROP) attack methods.
Security boundaries of CFI-protected programs

Figure from a paper published in ACM CCS’20
End