Lec 12: Execution Monitoring

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Execution Monitoring

Detection vs. Prevention

- Detection: detects a symptom.
- Prevention: Prevents a problem.

Detection or Prevention?

- Firewall.
- Encryption.
- Access controls.
- Antivirus.
- Canary.
- DEP.

Both Are Meaningful

- Prevention is not always feasible. So detection is needed.
- Detection of every possible symptom is not feasible, hence prevention (potentially in a limited scope) will help.

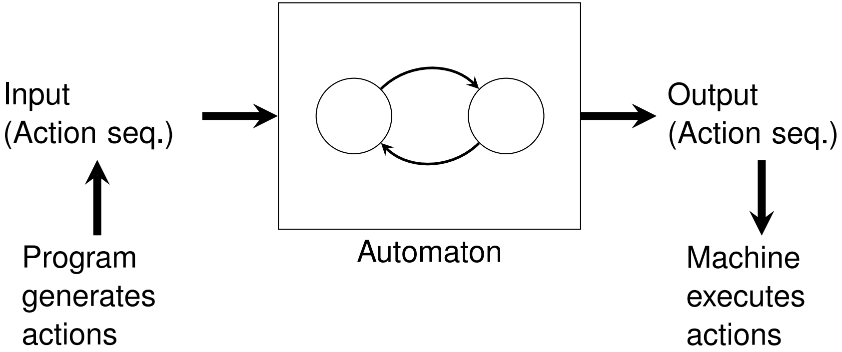
Execution Monitoring

We want to monitor executions, and ***detect unsafe symptoms*** at runtime.

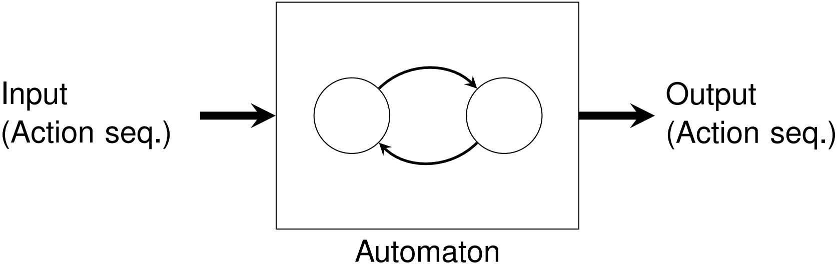
> What's the scope and limitation of execution monitoring? What kind of security policy is enforceable and at what cost?[1]

[1] Enforceable Security Policies, ***ACM TISSEC 2000***.

Security Automata



Security Automata



- σ: an execution (a sequence of actions).
- Ψ: universe of all possible sequences.
- Σ_S: subset of Ψ corresponding to the executions of target S.

Security Policy

A security policy is specified by giving a predicate on sets of executions. A target S satisfies security policy P if and only if $P(\Sigma_S)$ equals `true`.[2]

> Security policies rule out target executions that are deemed unacceptable.

Security Policy

A security policy is specified by giving a predicate on sets of executions. A target S satisfies security policy P if and only if $P(\Sigma_S)$ equals `true`.[2]

However,

Given sets of two executions $A \subset \Sigma_S$ and $B \subset \Sigma_S$, and a security policy P,
$P(A) = \texttt{true} \land B \subset A \not\Longrightarrow P(B) = \texttt{true}$.

One such example is ***information flow***!

[2] Enforceable Security Policies, ***ACM TISSEC 2000***.

Example: Information Flow

```
x = somefn();
if ( ... ) {
    ...
}
return y;
```

- Execution 1: `x = 1, y = 1`
- Execution 2: `x = 2, y = 2`

Example: Information Flow

```
x = somefn();
if ( ... ) {
    ...
}
return y;
```

- Execution 1: `x = 1, y = 1`
- Execution 2: `x = 2, y = 2`
- Execution 3: `x = 3, y = 1`

Policy vs. Property

- Policy: $P(\Sigma)$.
- Property: $P(\Sigma) : (\forall \sigma \in \Sigma : \hat{P}(\sigma))$,
  where $\hat{P}$ is a predicate on individual executions.
- A policy is a property if it can be defined by a predicate that holds on individual executions.

Not every security policy is a property!

EM-Enforceability (1)

A policy must be a property in order for that policy to be EM-enforceable.

EM-Enforceability (2)

Suppose σ' is the prefix of σ, where

$$\hat{P}(\sigma) = \texttt{true} \text{ and } \hat{P}(\sigma') = \texttt{false}.$$

Then, the execution might terminate before the execution is extended into σ.

EM cannot base decisions on possible future execution.

EM-Enforceability (2)

Suppose σ' is the prefix of σ, where

$$\hat{P}(\sigma) = \texttt{true} \text{ and } \hat{P}(\sigma') = \texttt{false}.$$

Then, the execution might terminate before the execution is extended into σ.

Let $\sigma[..i]$ be the prefix of σ involving its first i steps, and let $\tau'\sigma$ be execution τ' followed by execution σ. Then,

$$\forall \tau' \in \Psi^- : \neg\hat{P}(\tau') \implies (\forall \sigma \in \Psi : \neg\hat{P}(\tau'\sigma)).$$

Policy violation cannot be undone.

EM-Enforceability (3)

$$\forall \sigma \in \Psi : \neg \hat{P}(\sigma) \implies (\exists i : \neg \hat{P}(\sigma[..i])).$$

Any execution rejected by an EM must be rejected after a finite period.

EM-Enforceable Security Policies

Should satisfy the following conditions:

1. $P(\Sigma) : (\forall \sigma \in \Sigma : \hat{P}(\sigma))$
2. $\forall \tau' \in \Psi^{-} : \neg \hat{P}(\tau') \implies (\forall \sigma \in \Psi : \neg \hat{P}(\tau' \sigma))$.
3. $\forall \sigma \in \Psi : \neg \hat{P}(\sigma) \implies (\exists i : \neg \hat{P}(\sigma[..i]))$.

EM-Enforceable or Not?

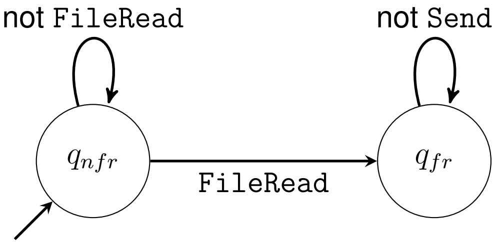
1. Access control.
2. Information flow.
3. Availability.

EM-Enforceable or Not?

1. Access control.
2. Information flow.
3. Availability.
   - If the availability is defined with a maximum limit?
   - If there's no limit?

Example

Prohibits execution of `Send` after `FileRead`.



Example

Runtime monitoring of Buffer Overflows.

- Goal: detect buffer overflows.
- Security policy: program should not access beyond the size of buffers.

What's the problem here?

Question

Is memory corruption EM-enforceable? Is there any counterexample?

Implementing EM

How to Monitor Program Execution?

- Attaching a debugger to a running process
  - GDB, LLDB, WinDbg, etc.
  - Single stepping: context switching for every single execution.
- Instrumentation
  - Modify code and inject code for monitoring!

Instrumentation

```
void somefn()
{
  char array[42];

  for (int i = 0; i < 42; i++) {

    array[i] = i;
  }
}
```

Instrumentation

```
void somefn()
{
  char array[42];
  printf("before loop\n");
  for (int i = 0; i < 42; i++) {
    printf("inner loop\n");
    array[i] = i;
  }
}
```

Instrumentation Tools Comparison

	Source-based	Binary-based
Dynamic	-	Pin (***PLDI 2005***) DynamoRIO (***CGO 2003***) Valgrind (***PLDI 2007***)
Static	LLVM (***CGO 2004***)	PEBIL (***ISPASS 2010***) DynInst (***HPCA 2000***) Diablo (***ISSPIT 2005***)

Binary Instrumentation

- Dynamic: emulate binary instructions and modify code at runtime.
- Static: rewrite binary prior to execution.

Dynamic vs. Static Instrumentation

- Dynamic
  - High overhead
  - Easy to instrument external libs.
  - Handles dynamically-generated code.
- Static
  - Fast
  - Difficult to instrument external libs.
  - Cannot handle dynamically-generated code.

Dynamic Binary Instrumentation: Valgrind

- Developed in 2003 by Nicholas Nethercote.
  - Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, ***PLDI 2007***
  - How to Shadow Every Byte of Memory Used by a Program, ***VEE 2007***
- Memcheck tool (implemented atop Valgrind) detects memory errors (only for dynamically allocated memory objects).

Memcheck

Memcheck uses shadow memory to store metadata for each memory cell.

- ***A bits***: every memory byte is shadowed with a single A bit, which indicates if the memory byte is accessible or not. (e.g., freed memory region is not accessible)
- ***V bits***: every register and memory byte is shadowed with eight V bits, which indicate if the value bits are initialized.

Detecting Dangling Pointers with Memcheck

- When accessing memory object whose V bits contain a zero.
- Delayed memory reuse specified by the argument `--freelist-vol`.

```
--freelist-vol=<number> [default: 1000000]

When the client program releases memory using free (in C) or delete (C++), that
memory is not immediately made available for re-allocation. Instead it is marked
inaccessible and placed in a queue of freed blocks. The purpose is to delay the
point at which freed-up memory comes back into circulation. This increases the
chance that Memcheck will be able to detect invalid accesses to blocks for some
significant period of time after they have been freed.

This flag specifies the maximum total size, in bytes, of the blocks in the
queue. The default value is one million bytes. Increasing this increases the
total amount of memory used by Memcheck but may detect invalid uses of freed
blocks which would otherwise go undetected.
```

[3]

[3] Excerpt from the manual.

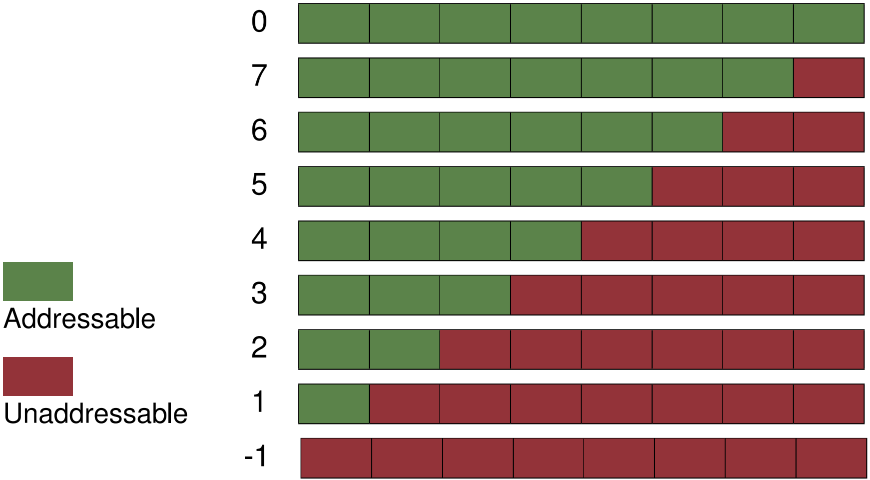
Address Sanitizer (ASan)

- Static source-level instrumentation using LLVM.
- Static instrumentation version of Memcheck.
- AddressSanitizer: A Fast Address Sanity Checker, ***USENIX ATC 2012***.

ASan’s Compact Shadow Memory

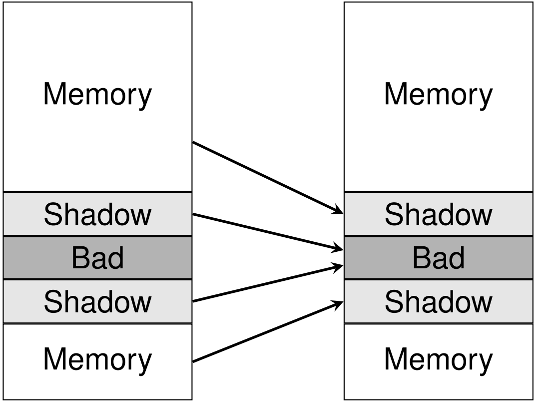
- Memcheck: byte-to-byte mapping.
- ASan: 8-byte-to-byte mapping.
- Key insight: heap memory is always 8-byte aligned.

9 States for 8-byte Aligned Memory



Mapping from Real to Shadow Memory

- Memcheck: address translation table.
- ASan: no table lookup
  - Reserve $\frac{1}{2^3}$ memory space
  - `Shadow = (Addr >> 3) + Offset`



Instrumentation: 8-byte Access

```
*Addr = 42;   // Original instruction
```

Instrumentation: 8-byte Access

```
// Instrumentation begins
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0) ReportAndCrash(Addr);
// Instrumentation ends
```

```
*Addr = 42;  // Original instruction
```

Instrumentation: 1-, 2-, or 4-byte Access

```
*Addr = 42;   // Original instruction
              // accessing (AccessSize) bytes
```

Instrumentation: 1-, 2-, or 4-byte Access

```
// Instrumentation begins
ShadowAddr = (Addr >> 3) + Offset;
k = *ShadowAddr;
if (k != 0 && ((Addr & 7) + AccessSize > k))
    ReportAndCrash(Addr);
// Instrumentation ends
```

```
*Addr = 42;   // Original instruction
              // accessing (AccessSize) bytes
```

Instrumenting Stack

```
void foo() {
  char arr[10];



  <function body>

}
```

Instrumenting Stack

```
void foo() {
  char rz1[32];
  char arr[10];
  char rz2[32-10+32];
  unsigned *shadow = (unsigned*)(((long)rz1>>3)+Offset);
  // poison the redzones around arr.
  shadow[0] = 0xffffffff; // rz1
  shadow[1] = 0xffff0200; // arr and rz2
  shadow[2] = 0xffffffff; // rz2
  <function body>
  // un-poison all.
  shadow[0] = shadow[1] = shadow[2] = 0;
}
```

Memory Alloc/Dealloc

- Insert red-zones around allocated memory objects.
- Freed page is set to be “red”.

ASan Has False Negatives

Example.

```
int *a = new int[2]; // 8-byte aligned
int *u = (int*)((char*) a + 6);
*u = 1; // access to range [6-9]
```

ASan Performance

- $1.73\times$ slowdown (RW).
- $1.26\times$ slowdown (W).

Comparison

	Valgrind	**ASan**
Heap out-of-bounds	Yes	Yes
Stack out-of-bounds	No	Yes
Global out-of-bounds	No	Yes
Use-after-free	(Yes)	(Yes)
Use-after-return	No	(Yes)
Uninitialized reads	Yes	No
Overhead	$10\times$ - $30\times$	$1.5\times$ - $3\times$

Question?

Further Readings

- Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation, ***USENIX ATC 2012***.
- StackArmor: Stopping Stack-based Memory Error exploits in Binaries, ***NDSS 2015***.
- Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing, ***NDSS 2018***.