# Enforcing Security through Execution Monitoring


Úlfar Erlingsson

Microsoft Research, Silicon Valley



Summer School on Software Security

University of Oregon, June 2004

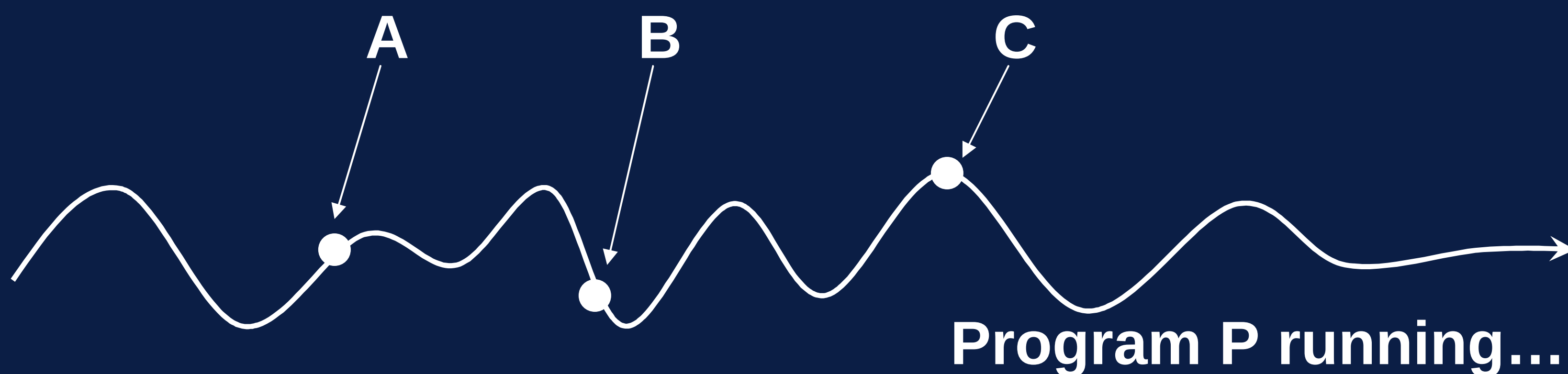

Microsoft Research

# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

# Execution Monitoring: Observe program execution

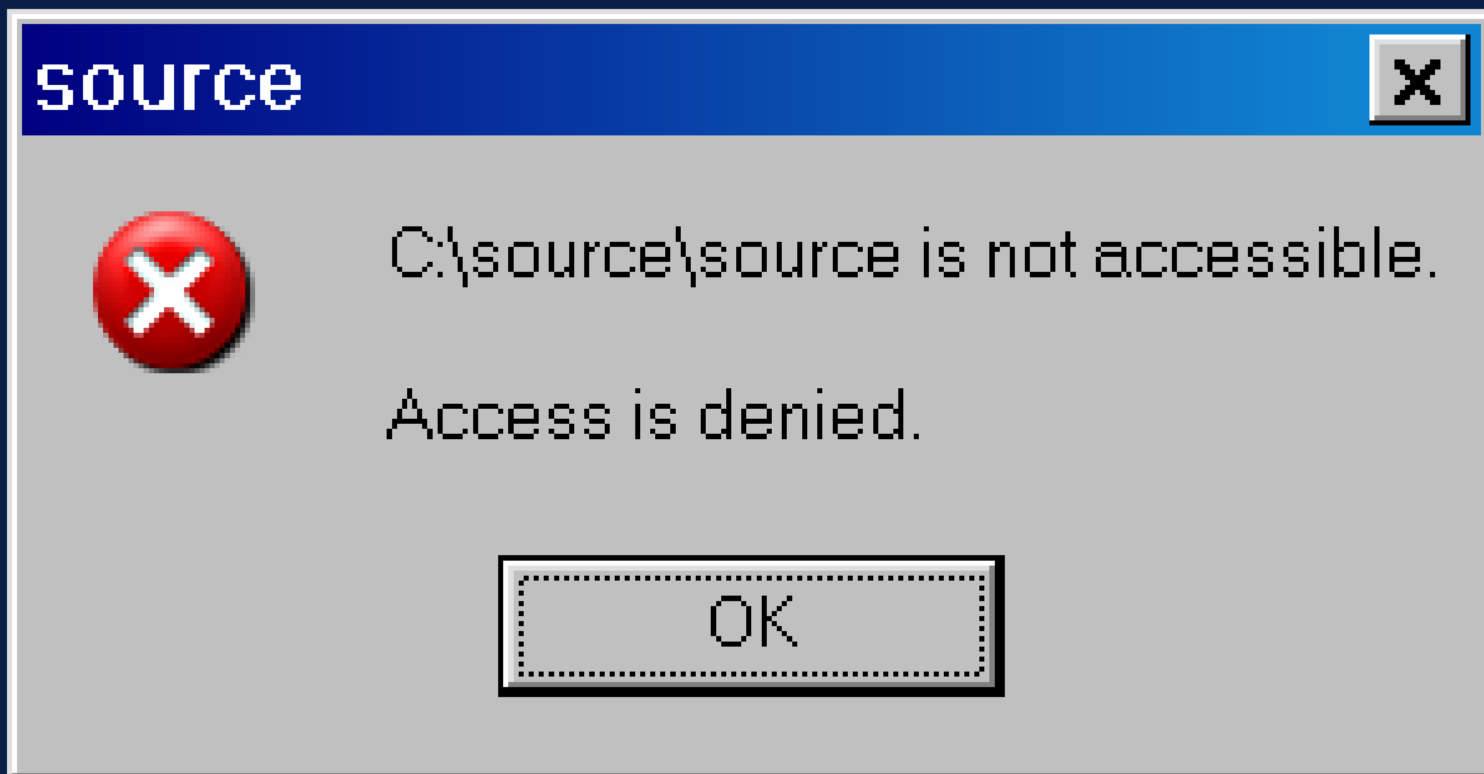
- Look at a program's execution on a given input as a sequence of runtime events (e.g., the A, B, and C below)
- Possibly do "something" on each event



# What is EM good for?

- Debugging, tracing, breakpoints, etc.
- Auditing and Logging
- Software testing: memory leaks, out-of-bounds array accesses, race conditions, atomicity, etc.
- Security (aka sandboxing, babysitting) like buffer overflow prevention etc.
- …

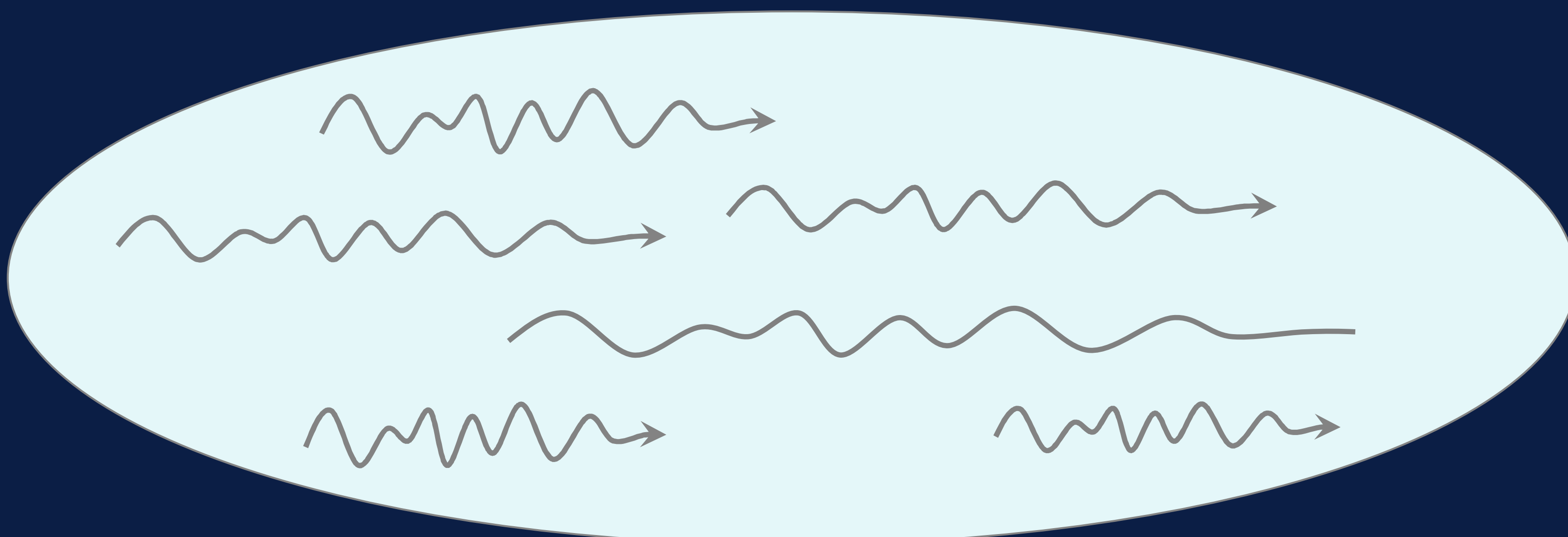
# In particular…



# Programs as Sets of Execution Traces

- View a program as defining an (infinite) set of (possibly infinite) execution traces
- All executions on all possible inputs + powercut

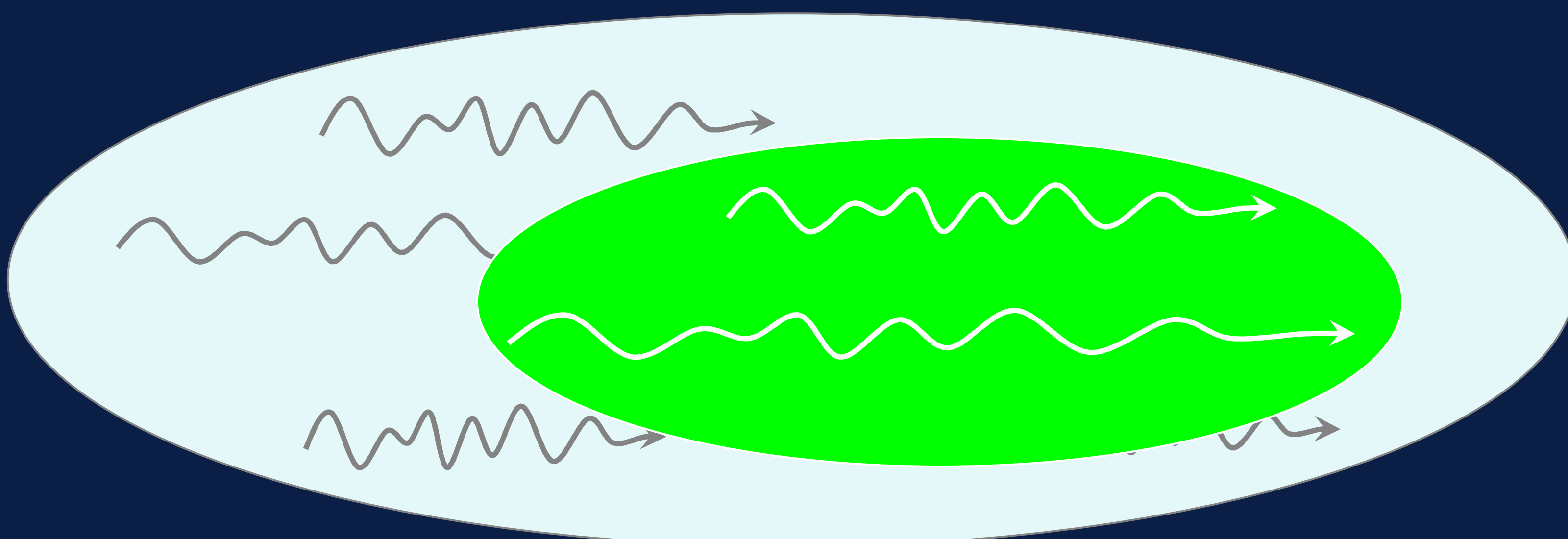
**Set of all possible executions of program P**



# Security Policies as Traces

- Define security policies as a subset of possible program execution traces
- Security policy set defines a predicate S

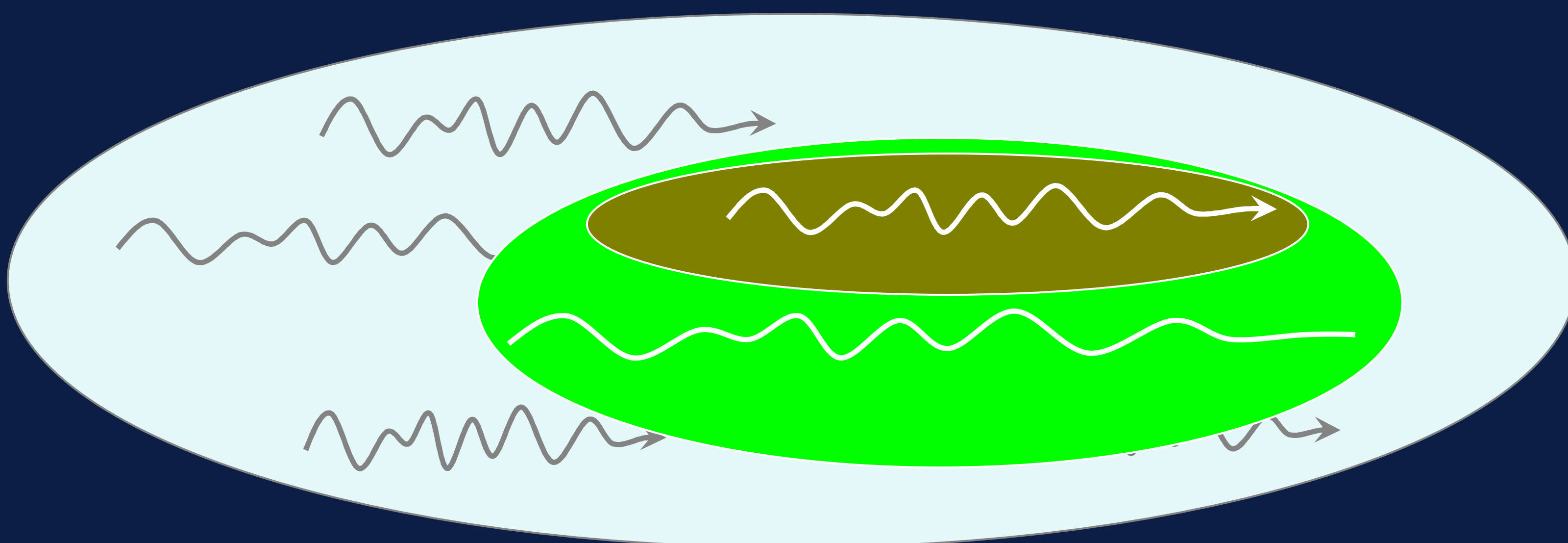
**Subset of executions of P that satisfy security policy S**



# Enforcing Security Policies

- Allows some traces that satisfy security policy
- Enforcement mechanism M is a concrete implementation that defines a subset of S

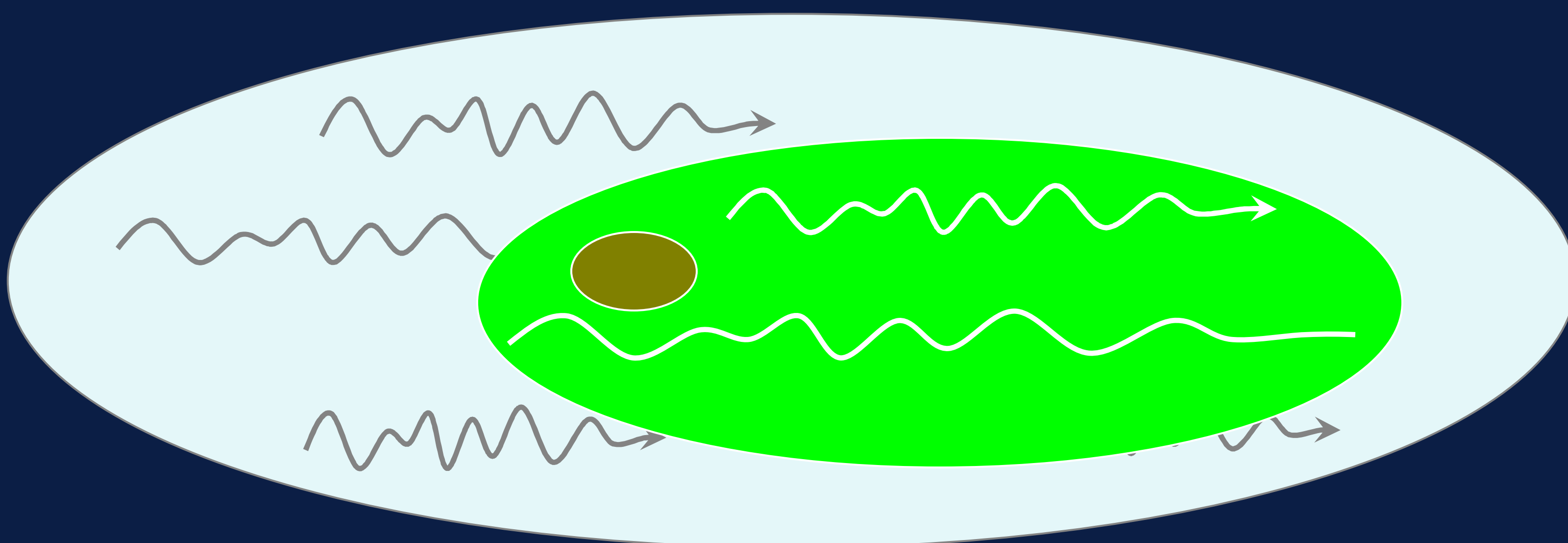
**Executions of P that enforcement mechanism M says satisfies policy S**



# Desirable Security Mechanisms

- Don't want enforcement to be vacuous (e.g. defining the empty set or disallowing all)
- Want enforcement to be exact (M == S)

**Vacuous subset that mechanism M enforces for security policy S**

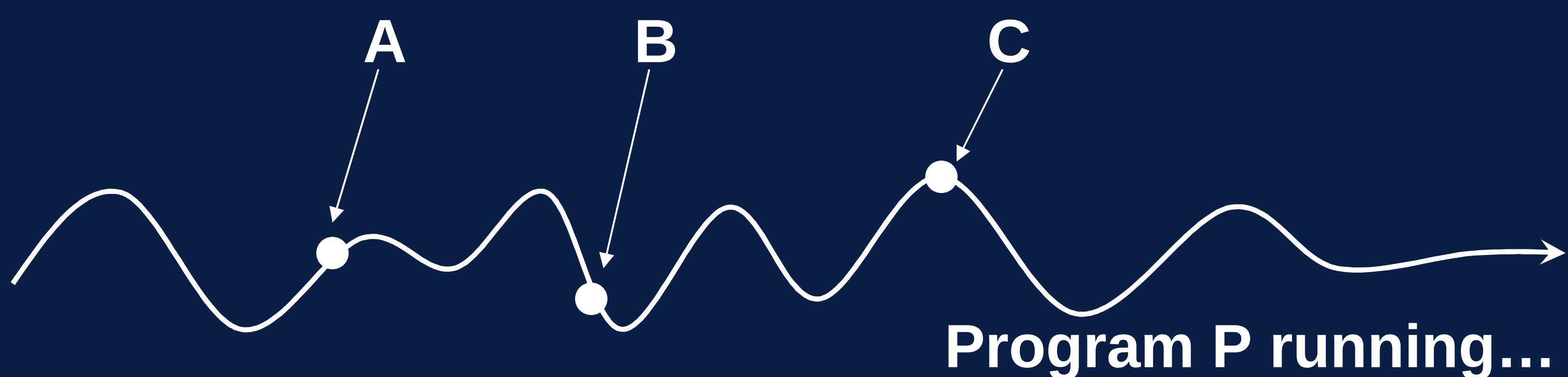


# “Hard” to Enforce Policies

- The design of M depends on policies, e.g., subsets/prefixes maybe insufficient
- Prefixes: “Pulling the plug” or halt
  - Liveness or “good things must happen”
    - E.g., if A happens, then B must follow
- Subsets: traces can be interdependent
  - Information flow: a subset may reduce uncertainty, hence pass information
    - E.g. trace of P “return 1” w/restricted input

# Execution Monitoring: Focusing on one Execution Trace

- Easy to do (just observe and constrain)
- EM can often approximate desired policy
- EM closely related to safety properties, so policies compose nicely, etc.



# EM Security Policies [Schneider00]

Define acceptable/unacceptable execution

- Execution Monitoring (EM) is one class
- EM observes execution (and truncates it)
- EM-enforceable part of safety properties

**Safety property**
- access control
- integrity
- D-availability

**Not Safety Property**
- information flow
- liveness
- availability

# Definitions

Security policy $P$: predicate on sets of executions

Target system $S$: set $\Sigma$ of executions

$S$ satisfies $P$: $P(\Sigma) = \text{true}$

Warning: For general security policy predicates

$$( \Pi \subseteq \Sigma \text{ and } P(\Sigma) ) \quad \text{implies} \quad P(\Pi)$$

does not hold !

# What can EM enforce?

Uses the single (current) execution

$$P(\Pi): \quad \left(\forall \sigma \in \Pi: \; p(\sigma)\right)$$

Only “properties” are EM enforceable

- Information flow is not a property [McLean94]
- Information flow is not EM enforceable (in an exact fashion)

# Properties

[Schneider00]:

"In Alpern and Schneider [1985] and the literature on linear-time concurrent program verification, a set of executions is called a *property* if set membership is determined by each element alone and not by other members of the set. Using that terminology, we conclude from (1) that a security policy must be a property in order for that policy to have an enforcement mechanism in EM."

# EM means Punctuality

Must truncate execution as soon as prefix violates policy:

$$\neg p(\tau) \Rightarrow \left(\forall \sigma : \neg p(\tau \sigma)\right)$$

where $\tau$ is a finite trace, $\sigma$ a trace, and the juxtaposition operator extends one trace with another

# EM means Finite Time

Must detect violations after a finite time.

$$\neg p(\sigma) \Rightarrow \left(\exists i : \neg p(\sigma[..i])\right)$$

where the [..$i$] postfix operator denotes a prefix of a given trace that is $i$ steps long

# Characteristics of EM

Any EM enforcement mechanism …

- Analyzes the single (current) execution.

$$P(\Pi): \quad \left(\forall \sigma \in \Pi: \ p(\sigma)\right)$$

- Must truncate execution as soon as prefix violates policy: $\neg p(\tau) \Rightarrow \left(\forall \sigma : \neg p(\tau \sigma)\right)$

- Must detect violations after a finite time:

$$\neg p(\sigma) \Rightarrow \left(\exists i : \neg p(\sigma[..i])\right)$$

Enforceable policy <u>implies</u> safety property

# Properties of EM Policies

- Prefix closed
  - If trace $\sigma$ is OK all prefixes of $\sigma$ are
- Subset closed
  - If S satisfies policy, then subset(S) does
- EM security policies compose nicely
  - Composed policy is intersection of sets

# Safety and Liveness Redux

[Alpern Schneider 87]

- Characterize safety and liveness
- Properties defined as **Buchi automata**
  - 1st-order predicates on transitions
  - Accepting & non-accepting states
    - Accept infinite traces iff accept infinitely often
  - Reject if unable to make transition

# Example: Total Correctness

A *Buchi automaton* [Eilenberg 74] $m$ accepts the sequences of program states that are in $L(m)$, the property it specifies. Figure 2.1 is a Buchi automaton $m_{tc}$ that accepts (i) all infinite sequences in which the first state satisfies a predicate $\neg Pre$ and (ii) all infinite sequences consisting of a state satisfying $Pre$, followed by a (possibly empty) sequence of states satisfying $\neg Done$, followed by an infinite sequence of states satisfying $Done \wedge Post$. Thus, $m_{tc}$ specifies *Total Correctness* with precondition $Pre$ and postcondition $Post$, where $Done$ holds if and only if the program has terminated.

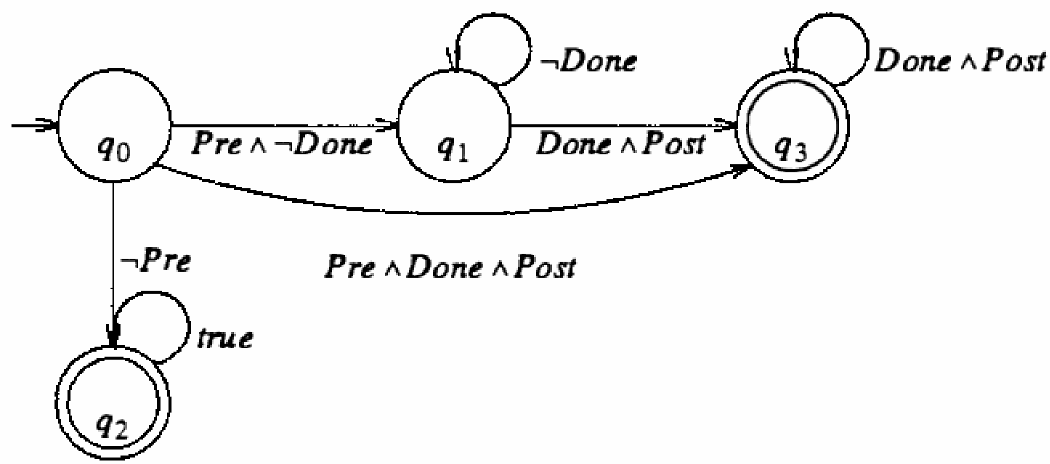



Figure 2.1. $m_{tc}$

# Formalizing the Automata

A Buchi automaton $m$ for a property of a program $\pi$ is a five-tuple :

$\langle S, Q, Q_0, Q_\infty, \delta \rangle$, where

- $S$ is the set of program states of $\pi$,
- $Q$ is the set of automaton states of $m$,
- $Q_0 \subseteq Q$ is the set of start states of $m$,
- $Q_\infty \subseteq Q$ is the set of accepting states of $m$,
- $\delta \in (Q \times S) \rightarrow 2^Q$ is the *transition function* of $m$.

# Formalizing the Infinite

Adding some notation for traces, so

sequence $\sigma = s_0 s_1 \ldots$ ,

$$\sigma[i] \equiv s_i$$
$$\sigma[..i] \equiv s_0 s_1 \ldots s_i$$
$$\sigma[i..] \equiv s_i s_{i+1} \ldots$$
$$|\sigma| \equiv \text{the length of } \sigma \text{ } (\omega \text{ if } \sigma \text{ is infinite}).$$

then can extend δ to finite sequences by

$$\delta^*(q, \sigma) = \begin{cases} \{q\} & \text{if } |\sigma| = 0 \\ \{q' \mid q'' \in \delta(q, \sigma[0]) \wedge q' \in \delta^*(q'', \sigma[1..])\} & \text{if } 0 < |\sigma| < \omega \end{cases}$$

# Formalizing Progress

To process a sequence $s_1 s_2 \ldots$ of input symbols the automaton starts with its current state set Q′ equal to $Q_0$ and reading the sequence one symbol at a time changes its current state set Q′ to the set Q″ where

$$Q'' = \bigcup_{q \in Q'} \delta(q, s_i).$$

# Making the Transition

We can encode δ as transition predicates by:

If $p_{ij}$ denotes the predicate for the transition from automaton state $q_i$ to automaton state $q_j$, then the security automaton, upon reading an input symbol $s$ with current state set Q′, changes it's current state set to Q″

$$Q'' = \{q_j \mid q_i \in Q' \wedge s \models p_{ij}\}.$$

# Accepting and Rejecting

- **Rejecting** is easy:

  Reject if $Q''$ is ever becomes empty

- **Accepting** is slightly harder:

  For trace $\sigma$, $INF_m(\sigma)$ is set of automaton states that appear infinitely often in $Q''$,

  then accept if $INF_m(\sigma) \cap Q_\infty \neq \emptyset.$

# Final Results

Can exactly characterize all properties

- Safety Properties
  exactly automata w/all states accepting
- Liveness Properties
  exactly automata w/non-accepting state(s)
- Properties
  conjunction of safety & liveness automata

# Trace Questions

- Why InfoFlow is not a property
  - **Language-based intuition:** We must prove if program P returns “low” value X on inputs S, then for all possibilities for the “high” portion of S, we have that P still returns X
- What do Buchi automata with no accepting states define?
  - The empty set of traces, no matter how many states there are, or what the transition predicates say

# Back to the Future

OK…

done with Safety & Liveness

Already saw EM $\subset$ safety properties

# Why EM ≠ Safety Properties

- EM can only use bounded memory
  - Can't buy unbounded machines (yet)
  - Safety properties can use infinite state
- EM must be able to control the system
  - May not be the case (turn flames off)
  - In particular, EM can't stop time
    - Can't do "gets service every X seconds"
    - ***Can*** do "gets service every Y steps"



Microsoft Research

# What EM can & can’t do

- EM ***can*** do access control
  - Whether DAC, MAC, MLS, …
- EM can’t do information flow
  - InfoFlow is not a property [McLean94]
    - Depends on other traces
    - Can’t have sets correlate High/Low
- EM can’t do Liveness/Availability
  - But ***can*** do D-availability

# EM Policies that Work

- **Integrity**: “More valuable data never overwritten by less valuable data”
  - Works, as just a simple comparison on writes
- **D-Availability**: “Y must follow X within D steps”
  - Works, but failure truncates without any Y
- **Foo and Bar happen as a pair, in order**:
  - ***Doesn’t work***, because EM can’t preclude prefixes with only a Foo and no Bar
  - A property: intersection of safety and liveness

# Better than EM: Security via Static Analysis

- Static analysis can make statements about all program execution traces:
  - B always follows A in all traces
  - Return value independent of input
- But hard to prove program properties…
- One way: Type-safe languages
  - Write program in a way that facilitates proving certain properties about it

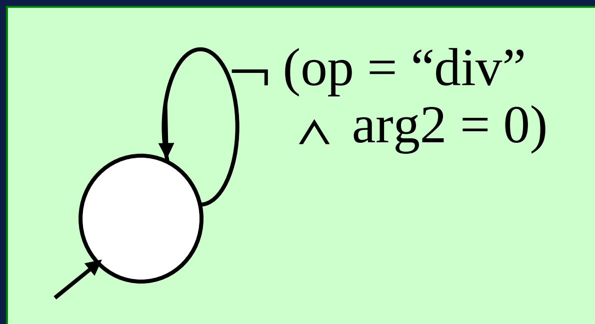
# Can we do it dynamically?

- Execution monitoring does it at runtime
- Easy to do “proofs”
  - Check sorted array after sort routine
- But need good runtime failure model
  - Can’t “un-launch” the missile
  - But might stop the train
- Security forms a special category
  - Usually OK to halt (turn attack into DoS)
  - Shows safety-critical ≠ security ?

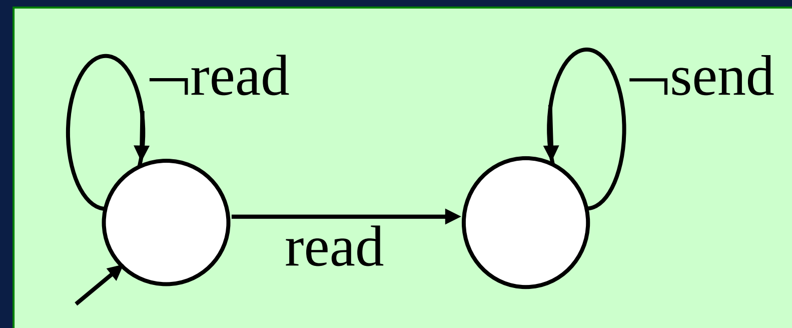
# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

# Specifying Security Policies




No division by zero




No network send after file read

One way is as **Security Automata**

- Formalism expresses the right properties
  - SA $\equiv$ safety properties $\supset$ EM-enforceable
- Simple to specify, interpret, and compile
- Good for analysis, emulation, testing

# Security Automata: The Hidden Truth

Security Automata are just regular Buchi Automata w/all states accepting

So we already know them !

PS: Because all states are accepting, we can use standard “Dragon book” techniques to make Security Automata deterministic… sweet!

# Simple Access Control


- Works for both discretionary and mandatory simple access control

# Why not just Single State?

- Easy to construct vacuous single-state security automata that push all the work into the predicate
- Good reasons to use more SA states
  - SA states can maintain security-relevant data outside the monitored program
  - SA states can encode program history
  - SA states can help “synthesize” higher-level security-relevant program events

# More Interesting…

[Brewer Nash 89]

```
STATE {
    Assume Categories maps each company to a category.
    Let usedCategories be an empty set.
    Let seenCompanies be an empty set.
}
EVENT Any application-level operation
CONDITION The operation involves an access to a company
UPDATE {
    Let company be the company being accessed.
    Let category be company's category in Categories.
    If category ∈ usedCategories and company ∉ seenCompanies {
        REJECT the operation.
    } Else {
        Add category to usedCategories.
        Add company to seenCompanies.
        ALLOW the operation.
    }
}
```

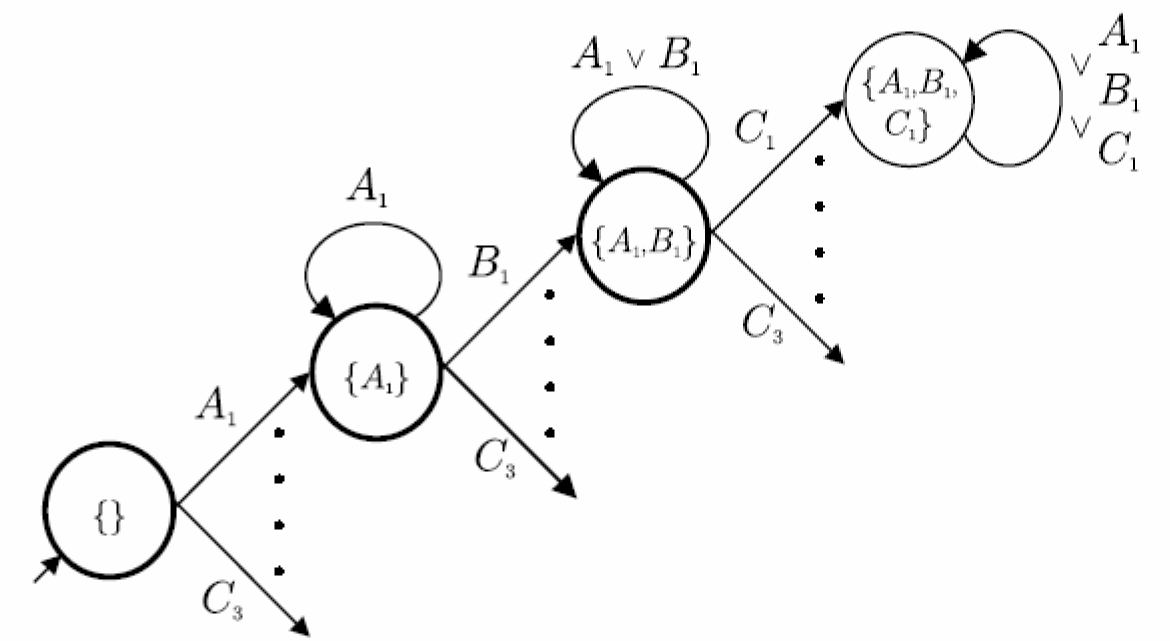
Example 2.4: The Chinese Wall security policy.

# First, some Assumptions…

Let’s assume a

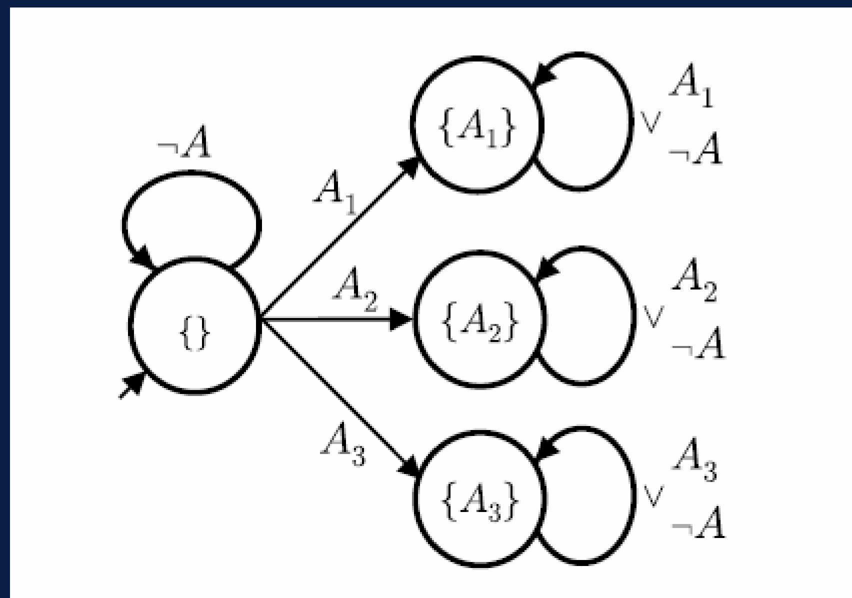
Chinese Wall security policy with a fixed number of companies and categories. In particular, the security policy is assumed to have three company categories, named $A$, $B$, and $C$, with three companies in each category. Within a category $X$, companies are named $X_1$ through $X_3$. (E.g., companies in category $A$ are $A_1$, $A_2$, and $A_3$.) The only way a user can access data concerning a company $X_i$ is assumed to be through invoking the command read($X_i$), which may, e.g., display information about company $X_i$ on the user’s screen. Further, users are assumed to be perpetually subject to the security policy—i.e., each user has a copy of the security state, and that security state is never reset.

# One Big Security Automaton




$$\text{states}(N,k) = \sum_{i=0}^{N} \binom{N}{i} k^i, \qquad \text{transitions}(N,k) = \sum_{i=0}^{N} \binom{N}{i} k^i \left((N-i)k+1\right).$$

# One SA per Category

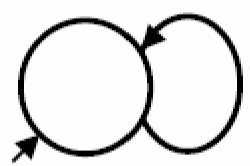



- Scales much better (linear in size)
- Can we tie SAs to program abstractions?



Microsoft Research

# Of course, can do it in One



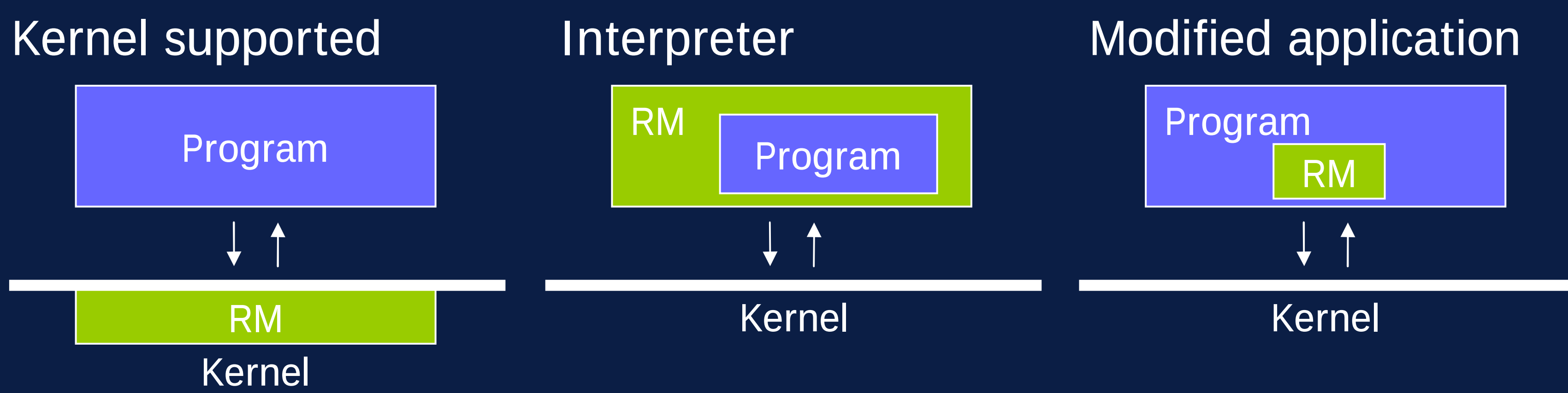

- Relies heavily on existing program
  - Program maintains state
  - Program implements two predictes
- Perhaps realistic…
  - At least for traditional OS access control

# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

# Reference Monitors [Anderson72]

- Execution monitor that forwards events to security-policy-specific validity checks
- Implementing RMs
  - Capture *all* policy-relevant events
  - Protect RM from subversion

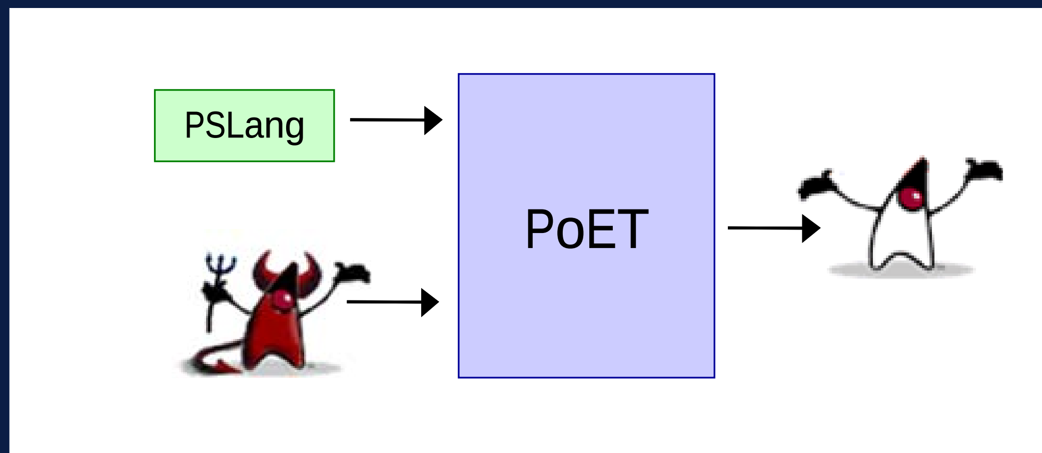


# Validity Checks

- Triggered by RM on each event
- Encodes the security policy
- Perform arbitrary computation to decide whether to allow event or halt
  - Can have side effects?  (Not if EM)
  - Can change program flow? (Not if EM)
- (PS: RM+Validity checks sometimes called the Security Kernel)

# Inlined Reference Monitors

[Erlingsson Schneider 99]

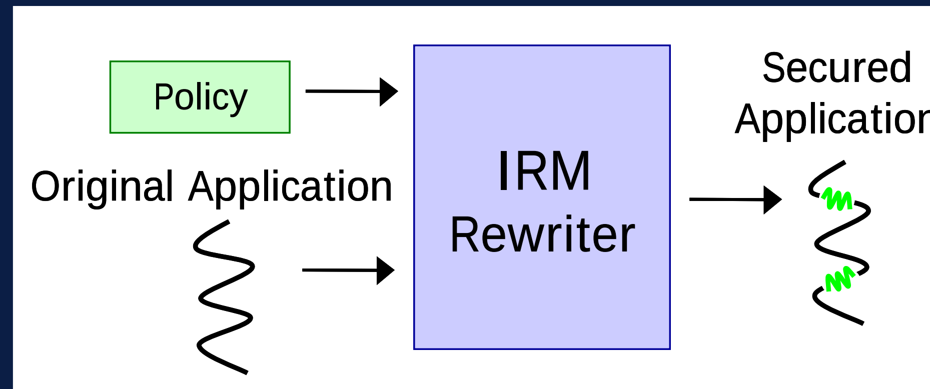



IDEA: Use 3rd type of RM implementations

- Use Security Automata to specify security policy
- Policy specifies both RM and Validity Checks
- Permanently embed security into application

# IRM Implementation

**Implement RMs by program modification**



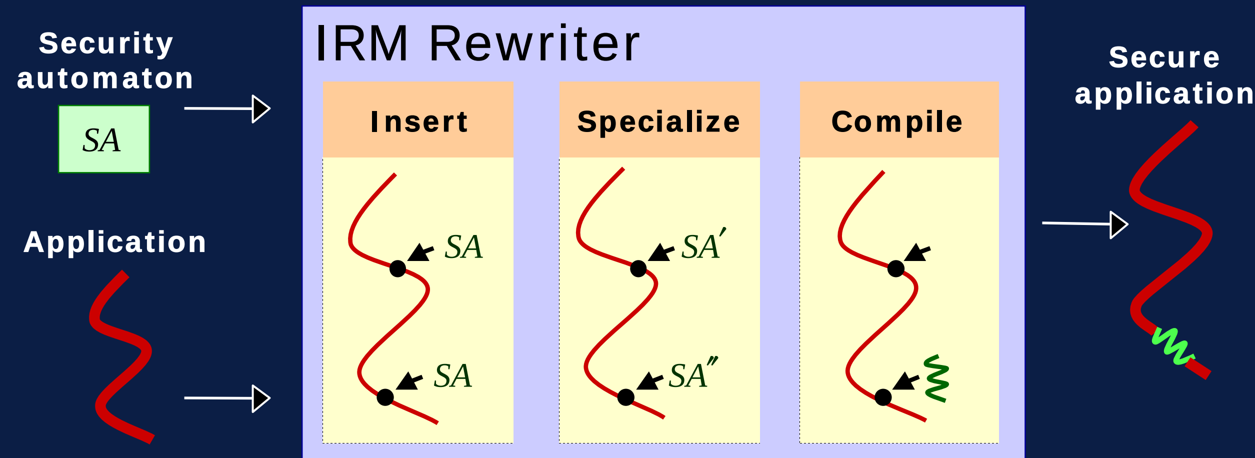

- IRMs have access to program abstractions
  - Capture all potentially security-relevant events
  - Rewriter works on machine language programs
- Issues: How to capture all relevant events
  Prevent application subverting inserted RM
  Preserve application behavior

# IRM Enforcement Advantages

- Can enforce policies on application abstractions
  - E.g., Restrict MSWord macros and documents
- Each application can have a distinct policy
  - Enforcement overhead determined by policy
  - Mechanism customized to the policy
- Mechanism is simple and efficient and travels
  - Rewrites machine code
  - Kernel is unaware of security enforcement
  - No enforcement overhead from context switches

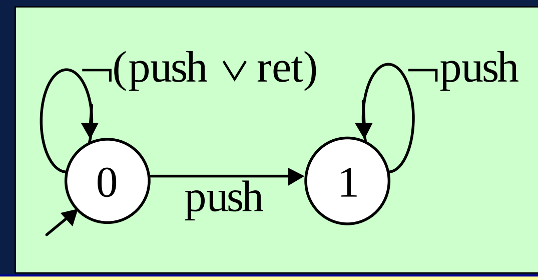
# Efficient IRM Enforcement

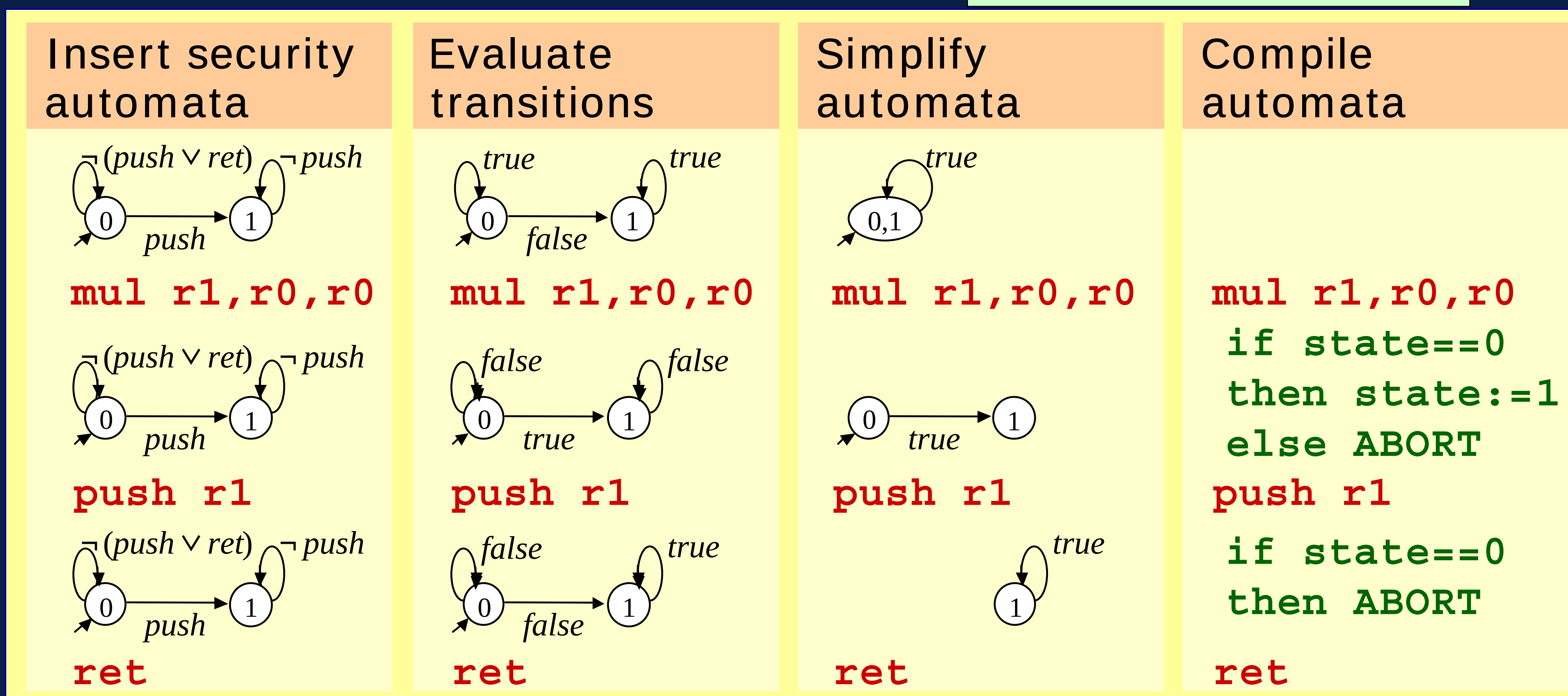
- Evaluate SA policy at every point in program
- Often no need to check at a machine instruction
  - "No div zero": Only check before "div" instructions
- Simplify SA by partial evaluation
  - Insert security policy checking code before every instruction
  - Use static knowledge of insertion point to simplify the check



# Example IRM Rewriting

Policy: Push exactly once before returning






Microsoft Research

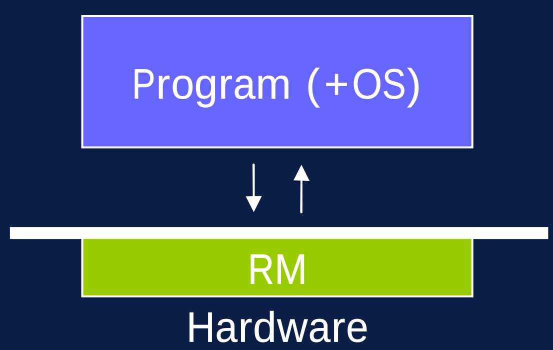
# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

# Traditional hardware protection

- Hardware execution monitor that forwards events to validity checks
  - Easily captures all events
  - Easily protected from subversion
  - Validity checks use page tables etc.

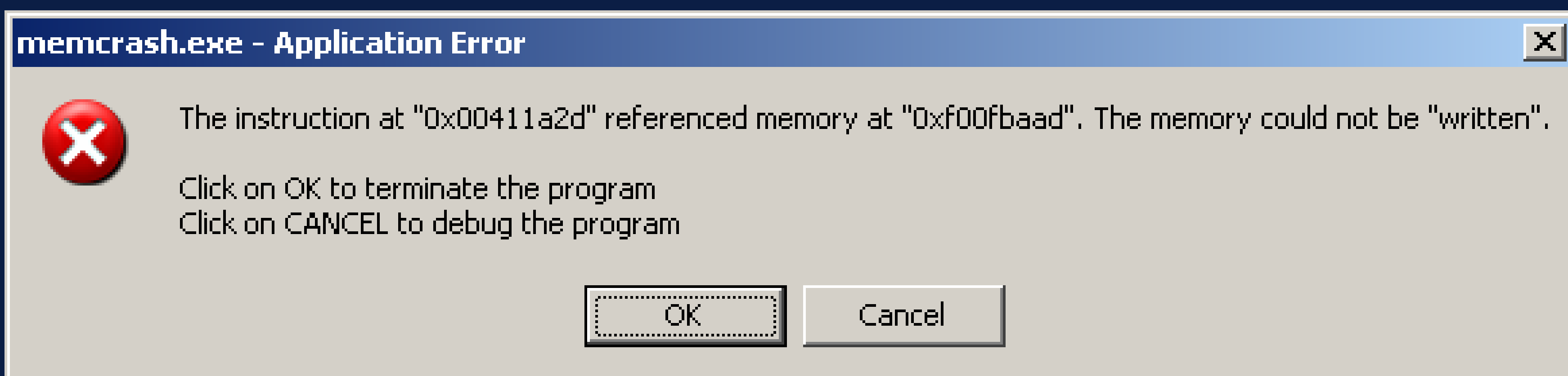
Hardware supported



# Memory Protection Policy

- MMU hardware is RM for memory accesses
- Access subject to validity checks on page tables

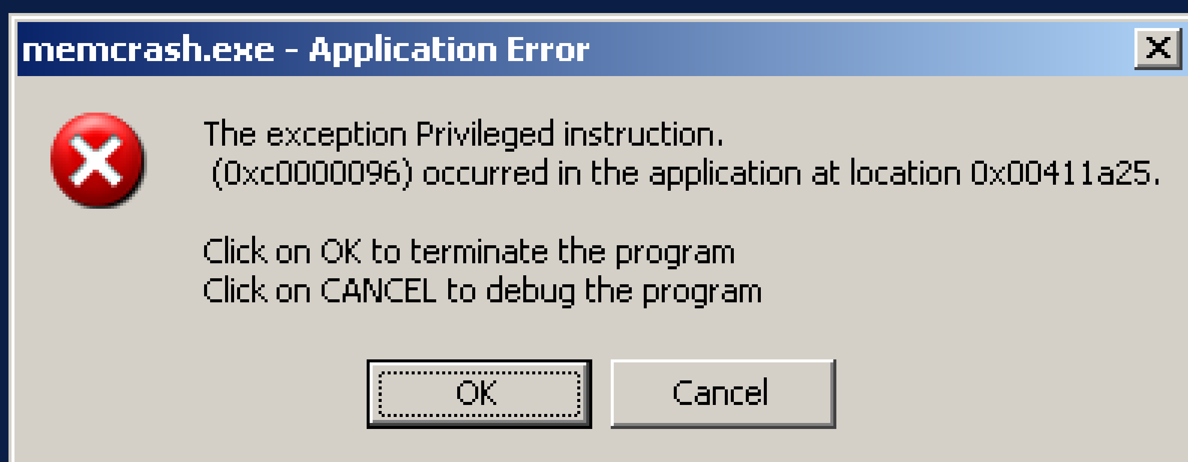
```
void main()
{
    char* badPtr = (char*)0xF00FBAAD;
    *badPtr = (char)"crash the program";
}
```



# Hardware Privilege Policy

- Hardware instruction decoder disallows certain operations in non-privileged mode

```
void main()
{
    __asm cli;/* try to disable interrupts */
              /* this crashes the program  */
}
```



# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

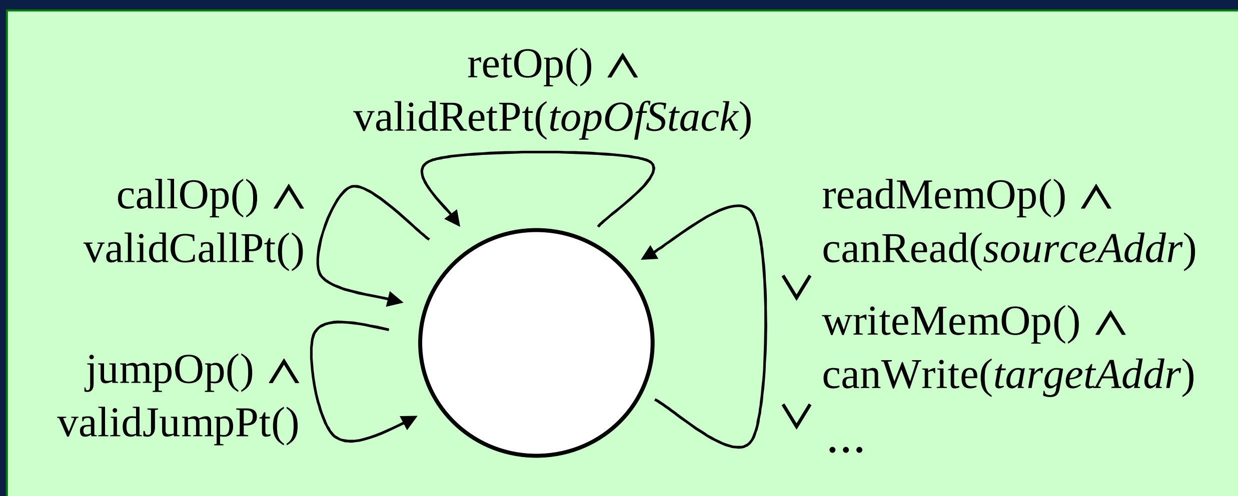
# Enforcing Hardware Policies in Software (aka SFI)

[Wahbe Lucco Anderson Graham 93]

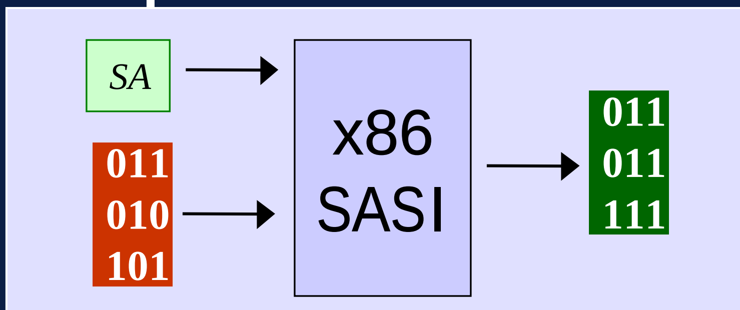
- Hardware policies can be enforced using any interpreter-based RM.
- Idea: Use a regular interpreter
  - For each instruction perform checks in software—SFI: software fault isolation
  - Make the interpreter very efficient
- Benefits
  - Don’t need to rely on MMU hardware
  - More flexibility in policy (validity checks)

# SFI using Security Automata

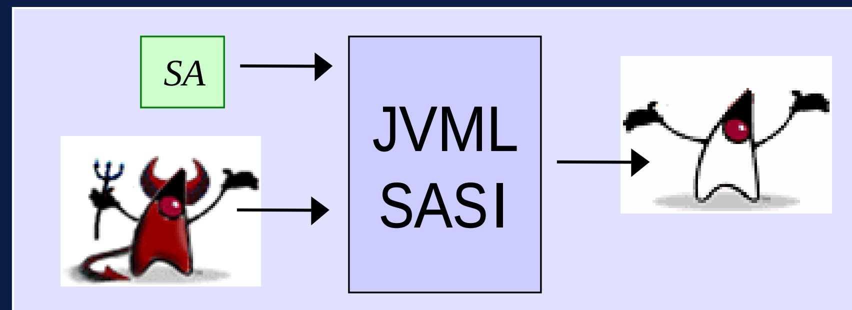
- SA receives as input the currently executed instruction and current program state
- Need to synthesize "instruction" event
  - Restrict memory access & control-flow



# SASI Prototypes




x86 SASI




JVML SASI

- SASI: Security Automata SFI Implementation
  - Inputs: SAL security policy and target application
  - Output: Application modified according to policy
- Expand TCB by at least SAL and rewriter
  - SAL ≈ 4.2K lines; rewriter ≈ 1K lines

# Example SFI'd code

```
pushl %ebx
leal  dirty(,%eax,4),%ebx
andl  offsetMask, %ebx
orl   writeSegment, %ebx
movl  %edx, (%ebx)
popl  %ebx
```

MiSFIT [Small97]

```
pushl  %ebx
leal   dirty(,%eax,4), %ebx
andl   segmentMask, %ebx
cmpl   writeSegment, %ebx
jne    .FAIL
popl   %ebx
movl   %edx, dirty(,%eax,4)
```

SASI x86 SFI

```
movl  %edx, dirty(, %eax, 4)
```

# Efficient SFI: Inlined Checks

- Very efficient interpreter:
  - RM: Identify all relevant instructions in code
  - Validity checks: Insert into code at instructions
- Do what checks the hardware would do
  - Address checks on memory access
  - HALT on illegal instructions (should run?)
- Ensure validity checks cannot be subverted
  - Control flow checks disallow circumvention
  - Deal with self-mod code, signals, etc…
  - On x86: find set of runtime instructions
- Get inductive proof of enforcement

# Preventing Circumvention

A proof outline…

An informal proof that the transformation suffices proceeds by contradiction, along the following lines. Only branch, call, return, and write instructions can subvert the security automaton simulation. Let $i$ be the first instruction that accomplishes the subversion. Before each branch, call, return, and write instruction, code to check that instruction's operand is added by x86 SASI for the policy in Figure 5. Thus, such checking code must immediately precede instruction $i$. Since, by assumption, $i$ is the first instruction that accomplishes the subversion, the checking code that precedes it must have been reached and executed. And since the transition predicates are, by construction, accurate, the checking code that precedes $i$ will prevent instruction $i$ from executing. The assumption that $i$ is able to execute and subvert the security automaton simulation is thus contradicted.

# x86 SASI: Implementing SFI

- x86 SASI: Modified `gcc` assembly
- Must protect RM inserted by x86 SASI
  - Can use x86 SASI to enforce SFI
  - Use SFI guarantees to protect RM
- Good performance vs. SFI tool MiSFIT

| Benchmark | MiSFIT | x86 SASI SFI |
|---|---|---|
| Hotlist | 2.38 | 3.64 |
| LFS simulate | 1.58 | 1.65 |
| MD5 | 1.33 | 1.36 |

Execution-time slowdown relative to SFI-free code

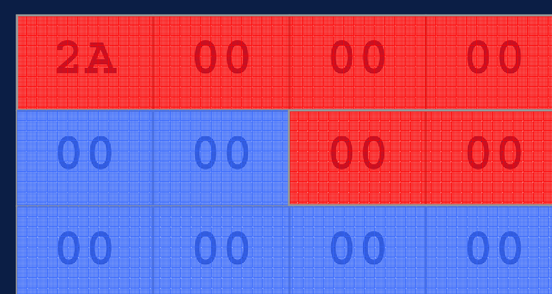
# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

# What is a buffer overrun?

- The ability to arbitrarily corrupt memory
- Overflows lead to arbitrary code
- Underflows lead to denial of service
- Problem is usually isolated to C and C++

```
int x = 42;
char zip[6];
strcpy(zip, userinput);
printf("x = %i\n", x);
```



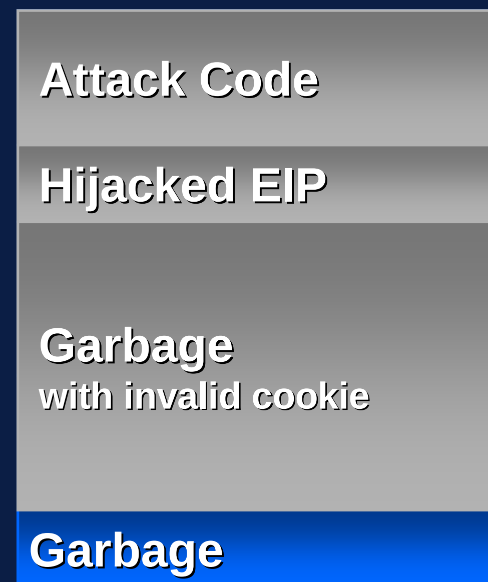
# Anatomy of the stack

| |
|---|
| **Previous function's stack frame** |
| **Function arguments** |
| **Return address** |
| **Frame pointer** |
| **EH frame** |
| **Local variables and locally declared buffers** |
| **Callee save registers** |
| **Garbage** |

- x86 stacks grow downward
- A buffer overrun on the stack can always rewrite the:
  - Return address
  - Frame pointer
  - EH frame

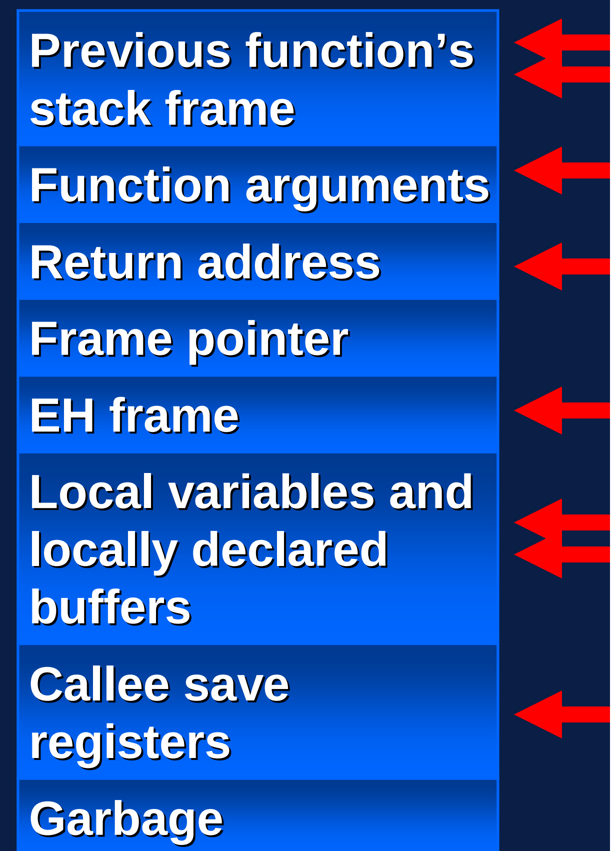
# Stack smashing

```
#define BUFLEN 4

void vulnerable(void) {
  wchar_t buf[BUFLEN];
  int val;

  val = MultiByteToWideChar(
    CP_ACP, 0, "1234567",
    -1, buf, sizeof(buf));
  printf("%d\n", val);
}
```



# Types of exploits

- Stack smashing
- Register hijacking
- Local pointer subterfuge
- V-Table hijacking
- C++ EH clobbering
- SEH clobbering
- Multistage attacks
- Parameter pointer subterfuge



# Unsafe APIs

- Many historical APIs of the C standard library are bad
  - **`strcpy`** does not know the array size
  - **`strncpy`** cannot validate the array size
  - Many more unsafe APIs exist
- Static analysis tools are helpful
- ~~Impossible~~ to guarantee a safe API

  Challenge

  [Jones Kelly 97]
  [Ruwase Lam 04]

# Stack layout in VC++ .NET

Function prolog:

```
sub    esp,24h
mov    eax,dword ptr
  [___security_cookie (408040h)]
xor    eax,dword ptr [esp+24h]
mov    dword ptr [esp+20h],eax
```

Function epilog:

```
mov    ecx,dword ptr [esp+20h]
xor    ecx,dword ptr [esp+24h]
add    esp,24h
jmp    __security_check_cookie (4010B2h)
```

- **Previous function's stack frame**
- **Function arguments**
- **Return address**
- **Frame pointer**
- **Cookie**
- **EH frame**
- **Local variables and locally declared buffers**
- **Callee save registers**
- **Garbage**

# Stack layout in VC++ 2003

Function prolog:

```
sub    esp,24h
mov    eax,dword ptr
  [__security_cookie (408040h)]
mov    dword ptr [esp+20h],eax
```

Function epilog:

```
mov    ecx,dword ptr [esp+20h]
add    esp,24h
jmp    __security_check_cookie (4010B2h)
```

- Previous function's stack frame
- Function arguments
- Return address
- Frame pointer
- Cookie
- EH frame
- Locally declared buffers
- Local variables
- Callee save registers
- Garbage

# What is this cookie?

[Cowan et al 98]

- Generated by the function **`__security_init_cookie`**
- Original stored in the variable **`__security_cookie`**
- Cookie is random (at least 20 bits)
- Cookie is per image and generated at load time
- Cookie is the size of a pointer

# Performance impact

- Expect less than a 2% degradation
- Most application did not notice anything
- With both VC7 and VC7.1 improvements in optimization make up for these security checks
- Each security check is nine instructions

**"The perf hit hasn't shown up for us. There was no test hit associated with the change. The only cost we've had associated with this is getting ourselves to build with /GS.**

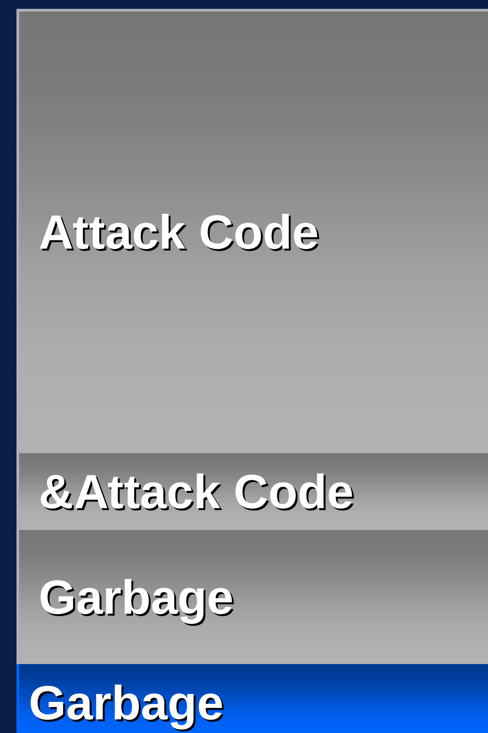
**– IIS6 Developer**

# V-Table hijacking

```
class Vulnerable {
public:
  int value;
  Vulnerable() {value=0;}
  virtual ~Vulnerable()
    {value=-1;}
};

void vulnerable(char* str) {
  Vulnerable vuln;
  char buf[20];
  strcpy(buf, str);
}
```

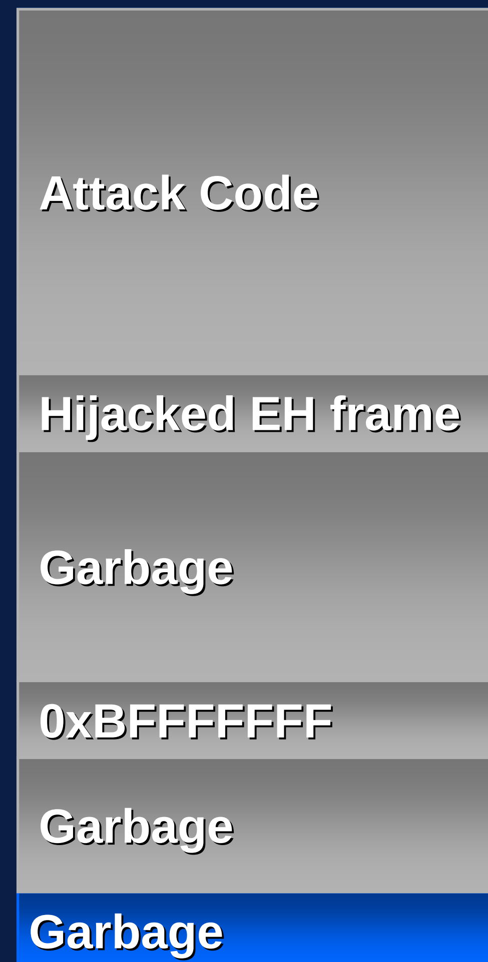
# Pointer subterfuge

```
void vulnerable(
  char* buf, int cb)
{
  char name[8];
  void (*func)() = foo;

  memcpy(name, buf, cb);
  (func)();
}
```



# EH clobbering

```
int vulnerable(char* str) {
  char buf[8];
  char* pch = str;
  strcpy(buf, str);
  return *pch == '\0';
}

int main(
  int argc, char* argv[]) {
  __try {
    vulnerable(argv[1]);
  } __except(2) { return 1; }
  return 0;
}
```



# Exploits possible despite /GS

- Parameter pointer subterfuge
- Two stage attacks
- Local objects with buffers
- Heap attacks
- …
- …
- …

# Class of x86 injection attacks

- Attacker controls victim behavior by getting x86 machine code of their choice to execute in victim's environment
- Subverts execution trace
  - Different x86 machine instructions execute

- Not the only attack: e.g., scripts

# NX: x86 Code Lockdown

- Distinguish between code and data
  - Harvard architecture?
- Prevent data from being executed as x86 machine code
- Slight modification to hardware RM validity checks

# Implementing NX

- Piece of cake with SFI (albeit slow)
- Hardware support on many CPUs
  - IA-64 (more realistically on amd64)
- Breaks lots of software:
  - Most Win32 GUI apps, CLR (and JITs)
- Can synthesize on IA-32 chips
  - Mark all data pages non-touchable
  - On trap, temporarily mark read/write, touch with MOV (which loads D-TLB), revert the page back to being untouchable

# Circumventing NX (aka jump-to-libc)

- Don't introduce new code (at first)
- Script existing code!
- E.g.

```
Victim's arg 2
Victim's arg 1
Victim's return addr
```

→

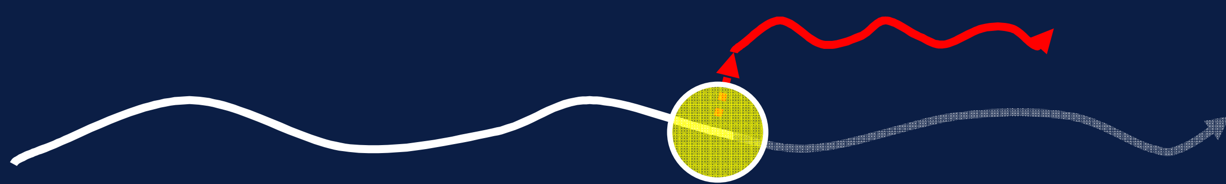
```
4th Function Args
4th Function Args
garbage (4th func is end)
3rd Function Args
3rd Function Args
4th Function Address
2nd Function Args
2nd Function Args
3rd Function Address
1st Function Args
1st Function Args
2nd Function Address
garbage
garbage
1st Function Address
```

- VirtualAlloc exec page, then InterlockedExch, then memcpy, then jump to alloc'd page

# Address-space Randomization

[PaX]

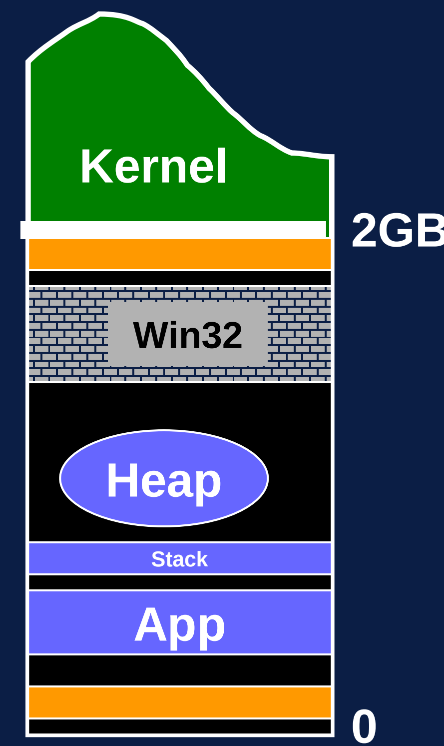
- x86 code injection attack must target the last “good” machine instruction
  - What happens just before exploit starts ?



- It’s control flow to an absolute address
  - Call [EAX],  Jmp [EBX],  Ret (implicit [ESP])
- Attacker must know where to go !!!

# What absolute addresses ?

- Attacker examines victim's address space on his/her machine:
  - For a version of Windows each address space is mostly the same (Win32 & friends)
  - Also, apps always lay out executables, stack, heap in the same way
- Attacker crafts exploit given above; waits for a vulnerability

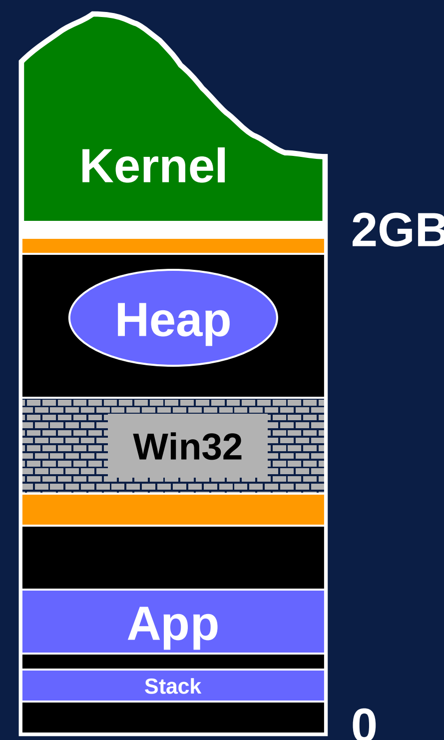





Microsoft Research

# Randomization / Rebasing

- Windows allows most things to be relocated
- Can do it
  - Dynamically @ load
  - Statically @ install
- Problems:
  - Most EXE files cannot be moved at all… etc.
- Example of Edit Automata



# Circumventing ASLR

- Learn the memory layout specifics of your target for attack [Durden 02]
  - Possible using format string attack etc.
  - May leak accidentally, e.g., via “nonce” in a protocol or Windows error reporting
  - Epidemic can automatically craft code
- Use brute force [Unpublished, Dan Boneh’s team 04]
  - Only 16 bits of shuffle on 32-bit machines
  - Easy to exhaust keyspace (automatically)

# A Quick Advertisement

- For students from (Northern) Europe

**NordSec 2004 Workshop**

4th - 5th November    Helsinki, Finland

- If you have some papers/work/TRs
  - Submit it and show up

# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

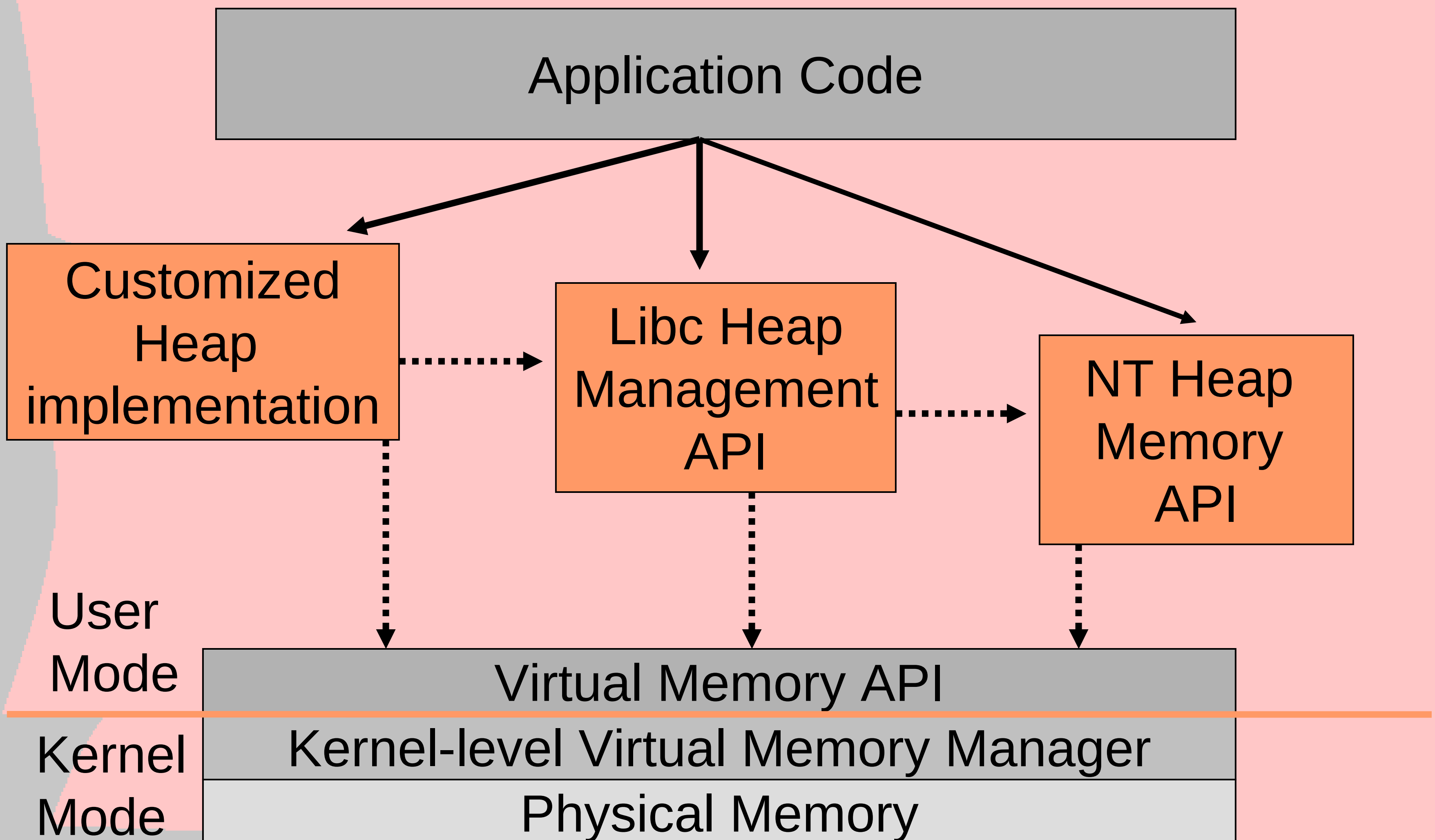
# Sample Vulnerability Discovery

[SolarDesigner01] [Slides, Halvar Flake 02]

- Surprisingly easy to find vulnerability
- Vulnerability types often translate
  - Heap manager problem in glibc.so
  - Heap problem in Windows 2000

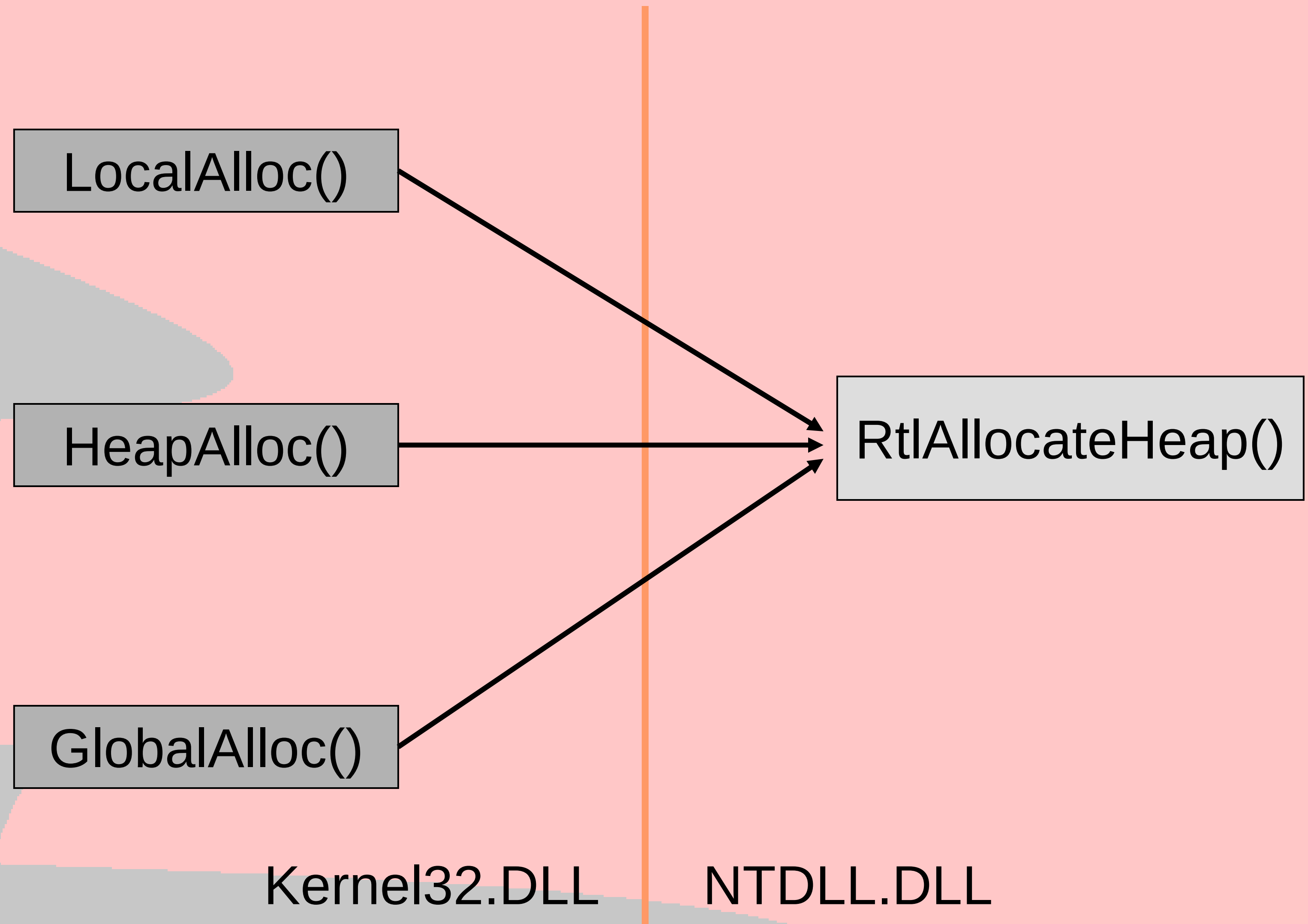
# Heap Structure Exploit Generalities

## Win32 heap management model

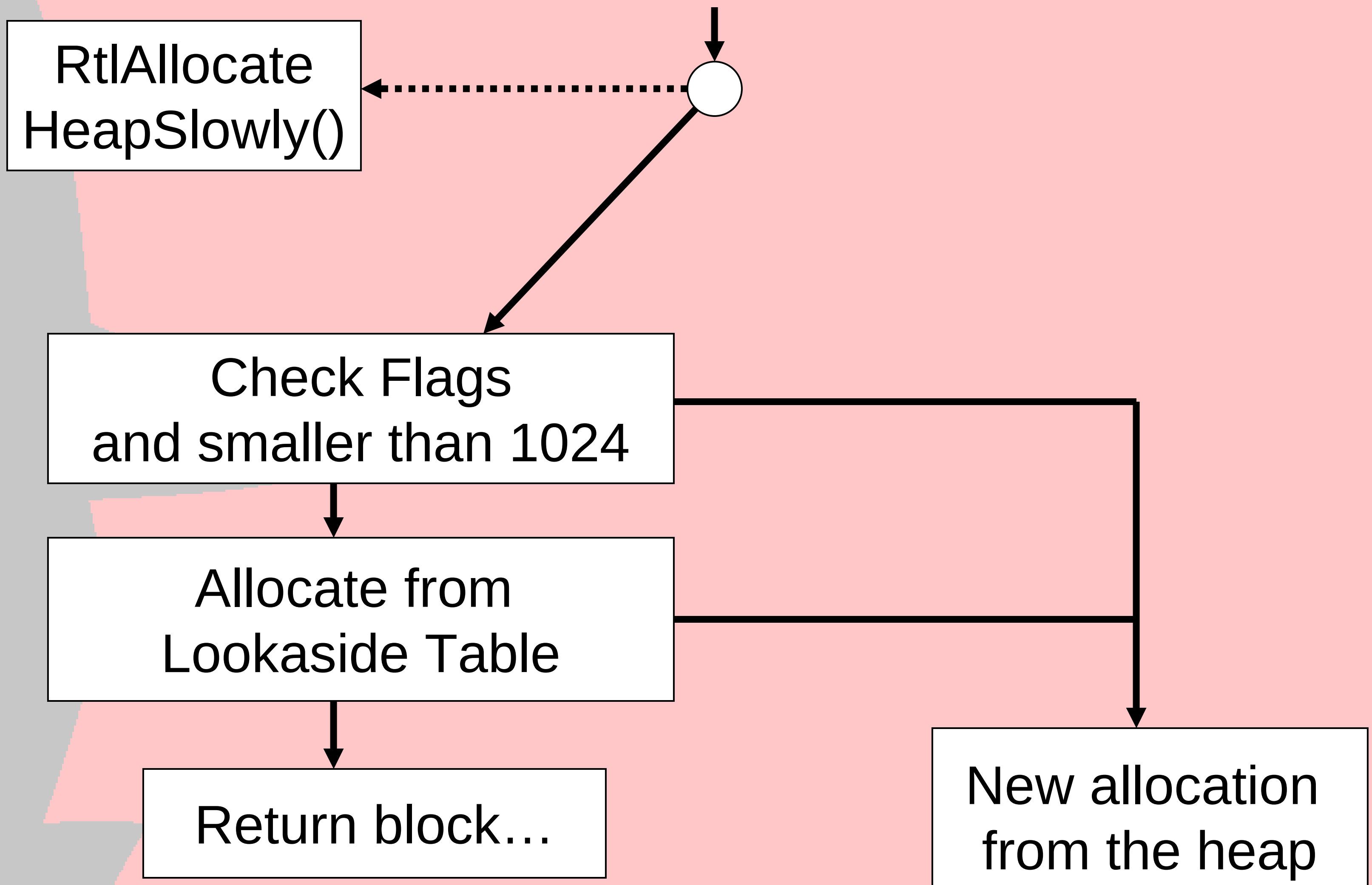


# Heap Structure Exploits

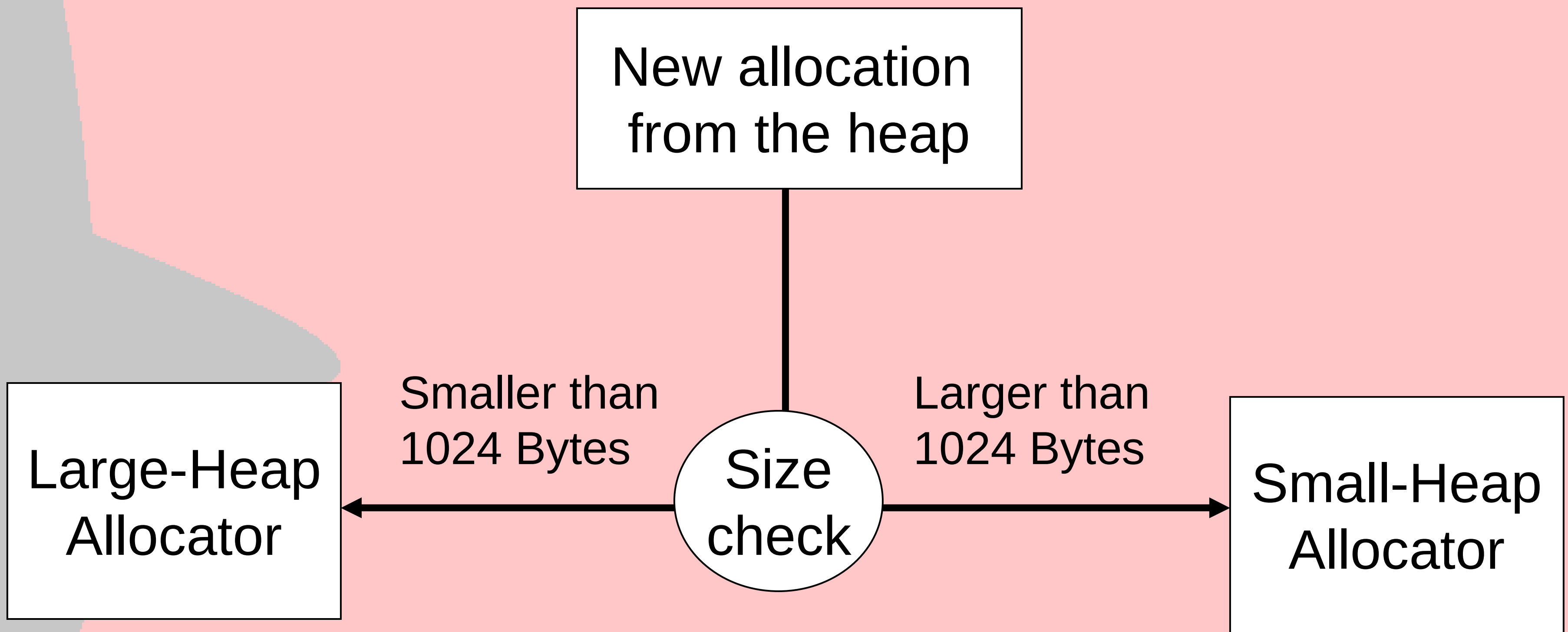
## Win2k Heap Manager (I)



# RtlAllocateHeap (I)



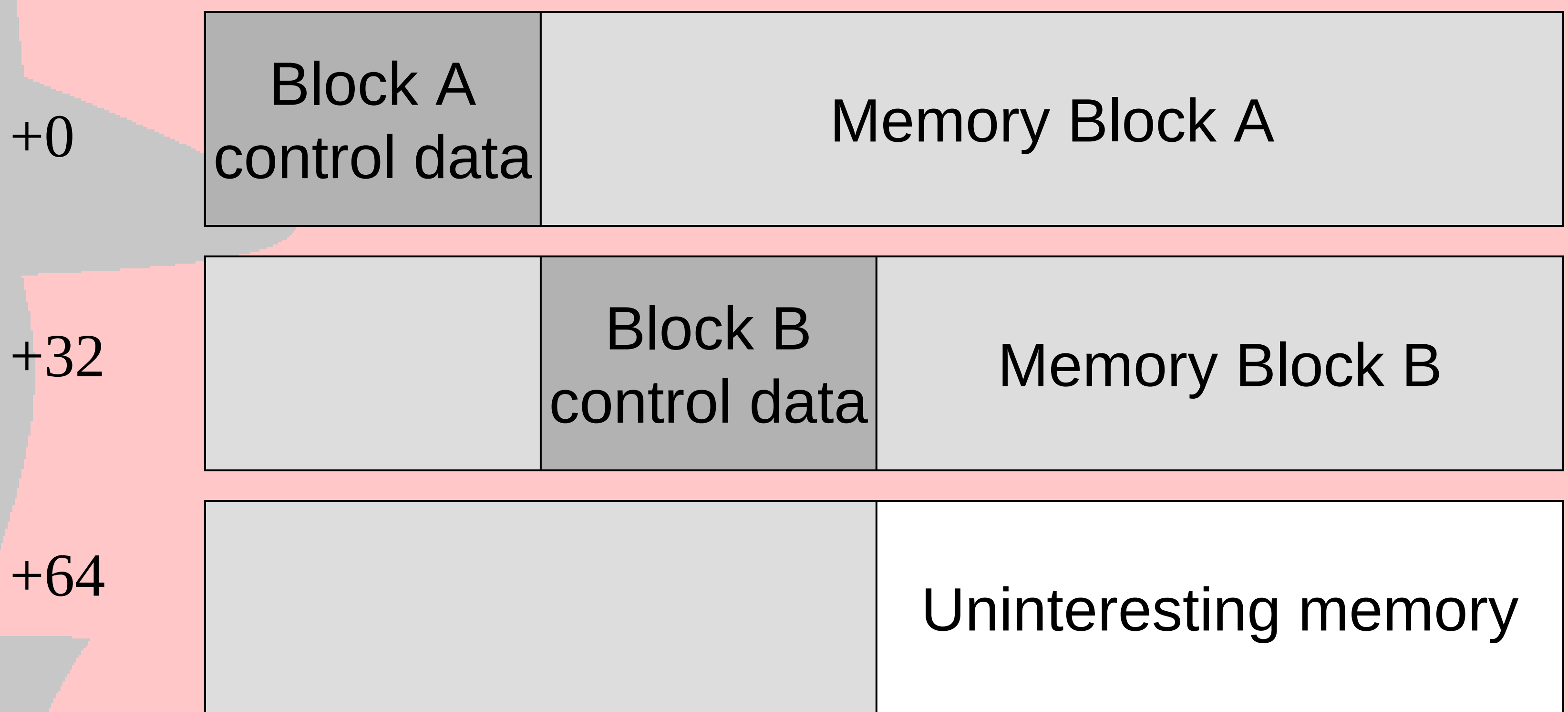
# RtlAllocateHeap (II)



# Heap Structure Exploits

## Win2k Heap Manager (II)

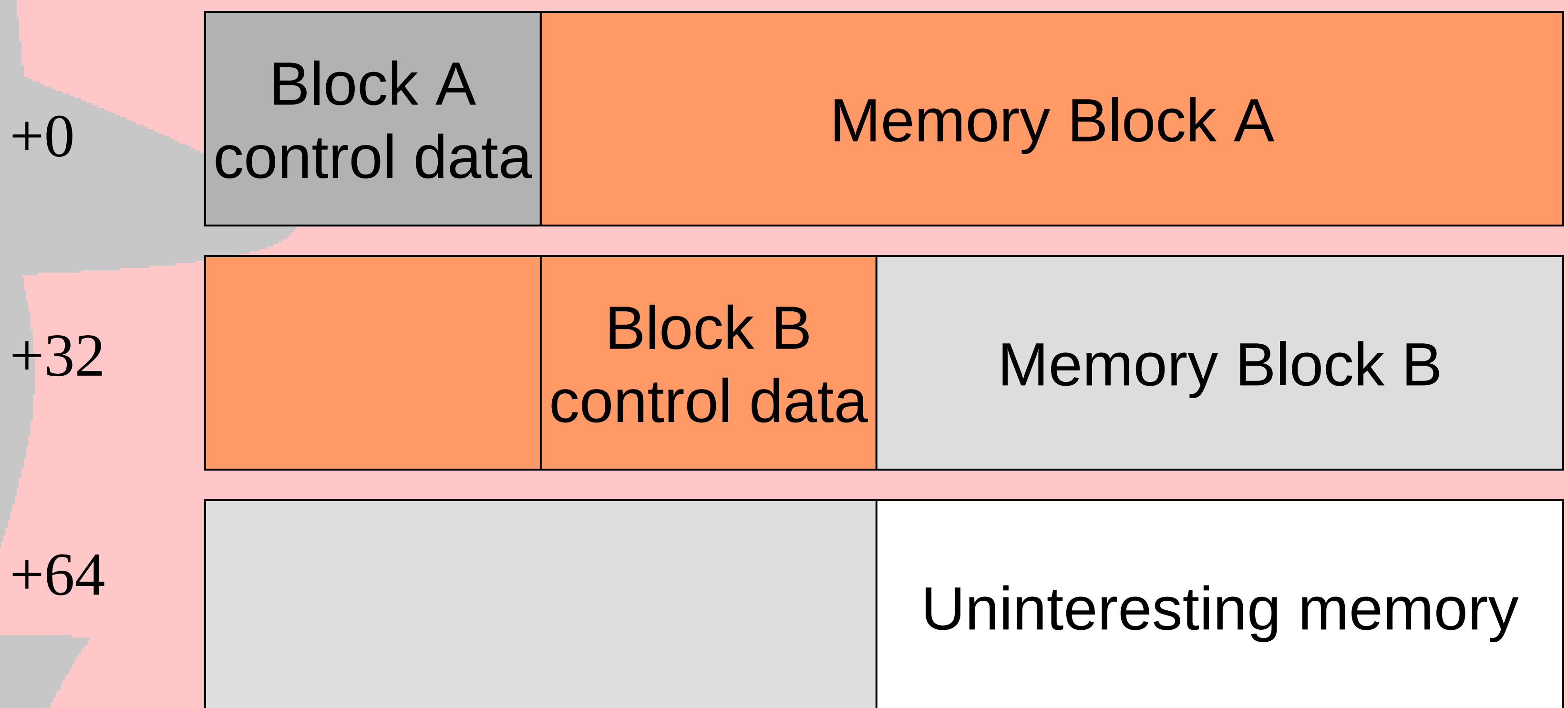
After two allocations of 32 bytes each our heap memory should look like this:



# Heap Structure Exploits

## Win2k Heap Manager (III)

Now we assume that we can overflow the first buffer so that we overwrite the *Block B control data*.



# Heap Structure Exploits

## Win2k Heap Manager (IV)

When Block B is being freed, an attacker has supplied the entire control block for it. Here is the rough layout:

<table>
<tr><td>+0</td><td colspan="2">Size of this Block divided by 8</td><td>Size of the previous Block divided by 8</td></tr>
<tr><td>+4</td><td>Field_4</td><td>8 bit for Flags</td><td></td></tr>
</table>

If we analyze the disassembly of _RtlHeapFree() in NTDLL, we can see that our supplied block needs to have a few properties in order to allow us to do anything evil.

# Heap Structure Exploits

## Win2k Heap Manager (V)

Properties our block must have:

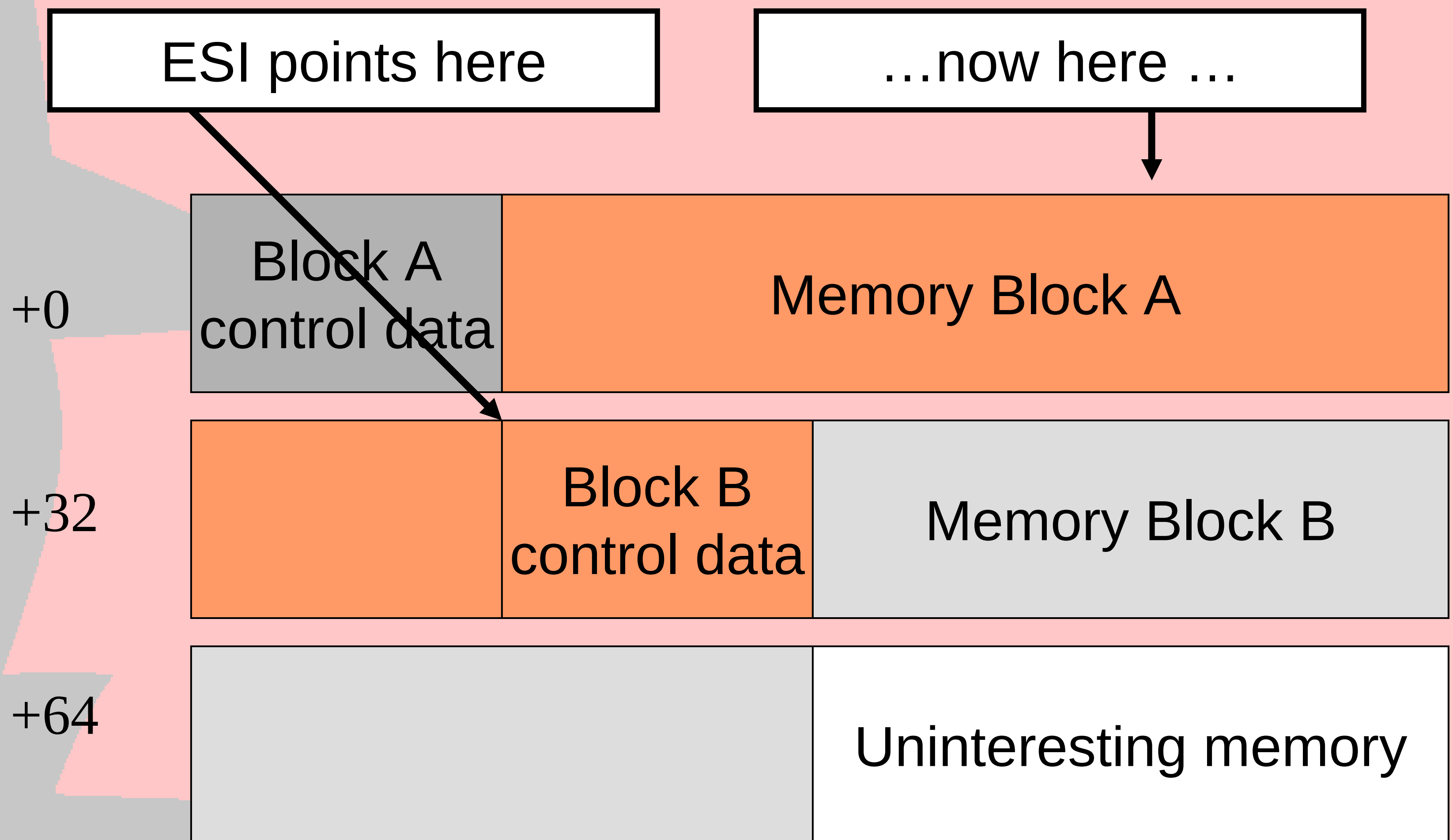
- Bit 0 of Flags must be set
- Bit 3 of Flags must be set
- Field_4 must be smaller than 0x40
- The first field (own size) must be larger than 0x80

The block 'XXXX99XX' meets all requirements.
We reach the following code now:

# Heap Structure Exploits

## Win2k Heap Manager (VI)

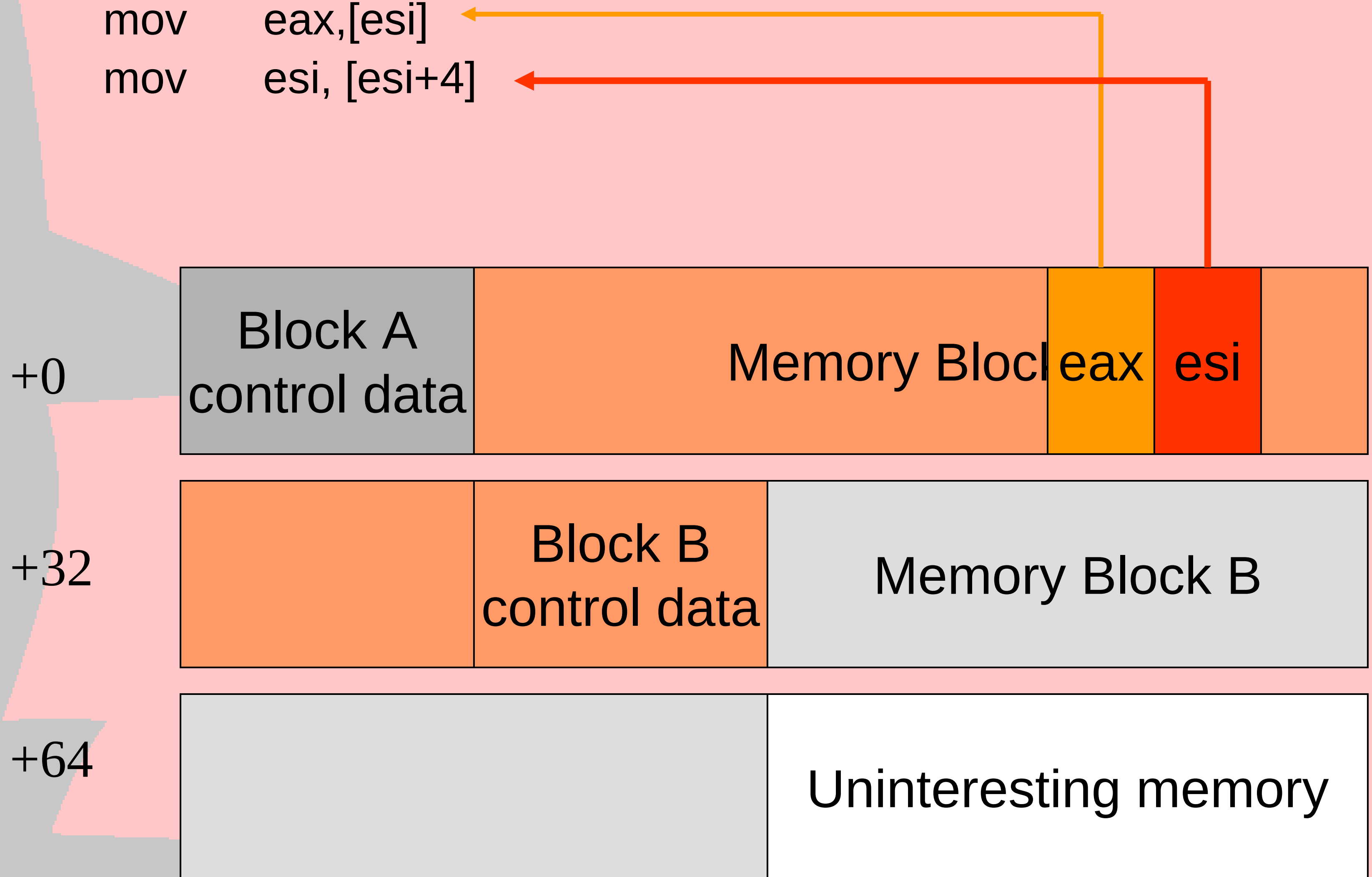
add esi, -24



# Heap Structure Exploit

## Win2k Heap Manager (VII)

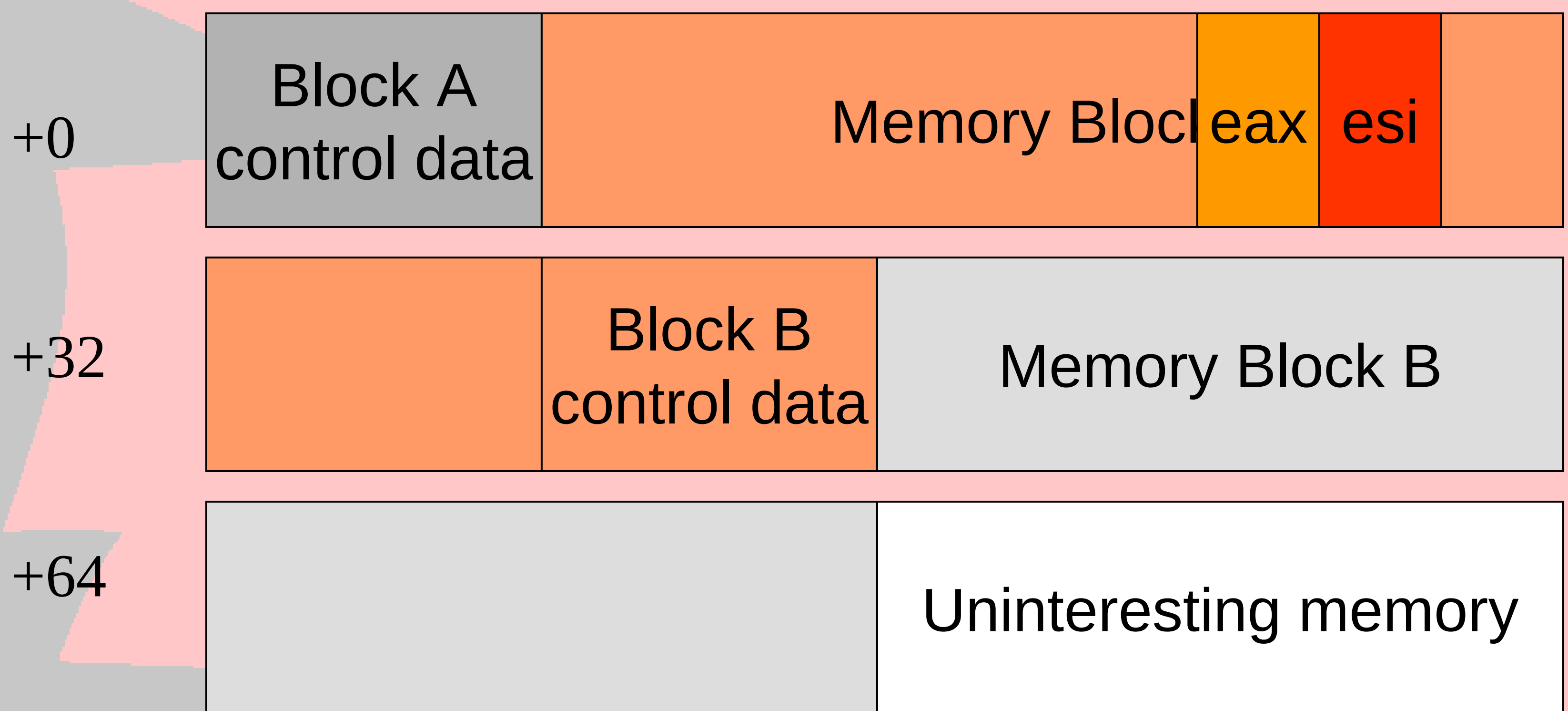
```
mov     eax,[esi]
mov     esi, [esi+4]
```



# Heap Structure Exploits

## Win2k Heap Manager (VIII)

```
mov     [esi], eax     ; Arbitrary memory overwrite
```



# Heap Structure Exploits

## Win2k Heap Manager (IX)

- If we can overwrite a complete control block (or at least 6 bytes of it) and have control over the data 24 bytes before that, we can easily write any value to any memory location.
- It should be noted that other ways of exploiting exist for smaller/different overruns – use your Disassembler and your imagination.

# Demo

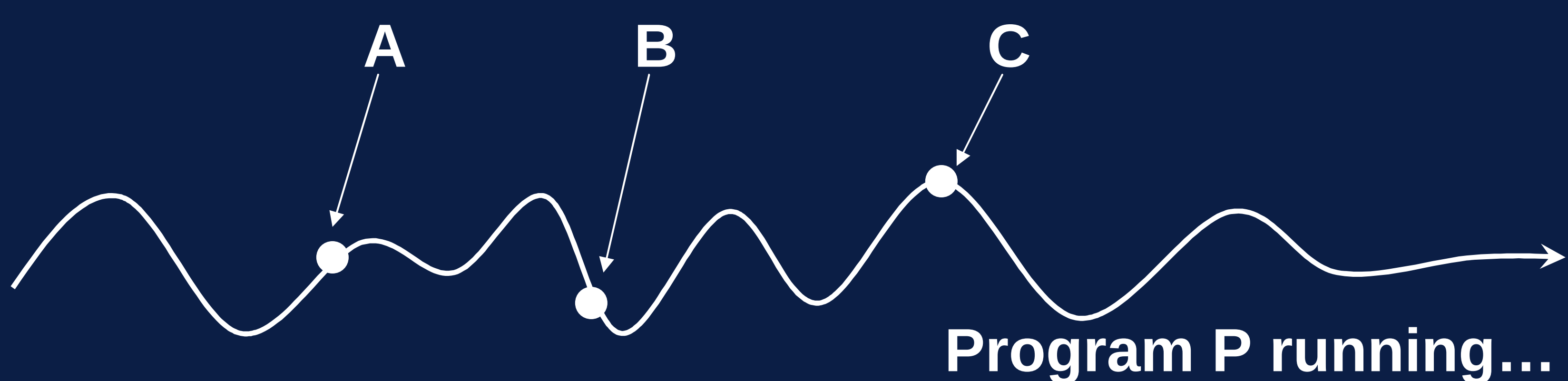
Fiddling with Machine Code

# Outline

1. Execution Monitoring Fundamentals
   - Programs and properties from traces
   - Security Policy as Security Automata
   - Introduction to Inlined Reference Monitors
2. Monitoring Machine Code Execution
   - Software Fault Isolation
   - Buffer Overflows and Mitigations
3. Advanced IRMs & future work
   - Low-level Actions and Event Synthesis
   - Static Analysis, Alternate Remedies, etc.

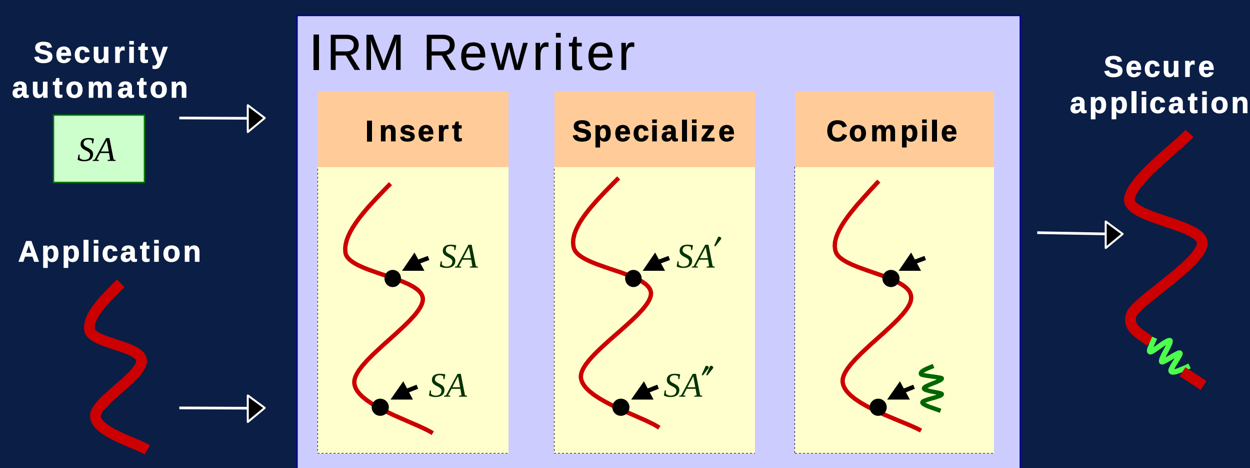
# General Execution Monitoring

- Not tied to any one type of event
  - But must be able to accurately identify events
- Validity checks can maintain state etc.
  - For instance: ensure execution order of A, B, C



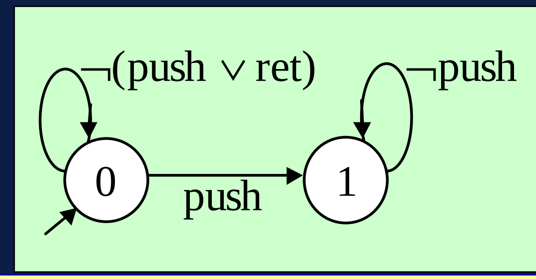
# Efficient IRM Enforcement

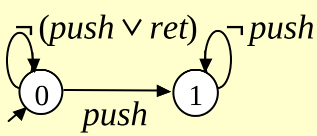
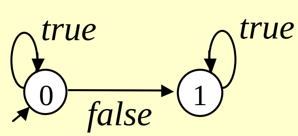
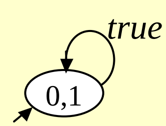
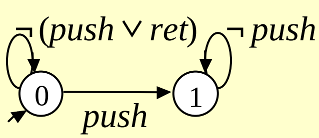
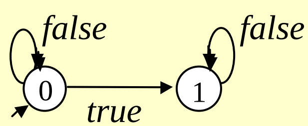
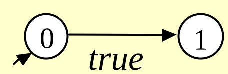
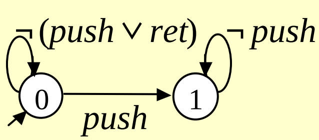
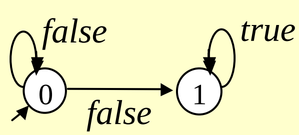
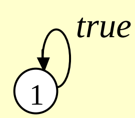
- Evaluate SA policy at every point in program
- Often no need to check at a machine instruction
  - "No div zero": Only check before "div" instructions
- Simplify SA by partial evaluation
  - Insert security policy checking code before every instruction
  - Use static knowledge of insertion point to simplify the check



# Example IRM Rewriting

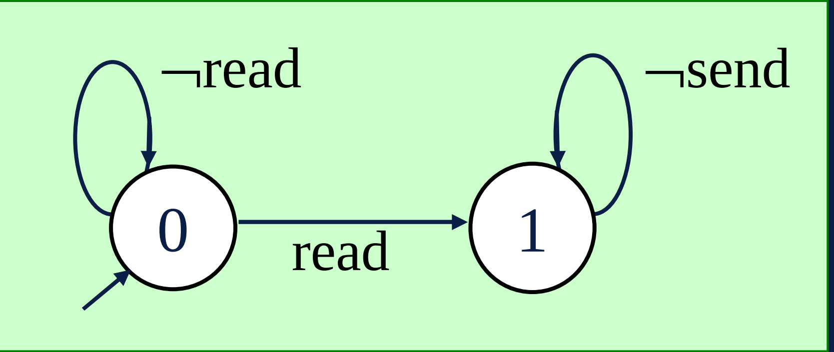
Policy: Push exactly once before returning




| Insert security automata | Evaluate transitions | Simplify automata | Compile automata |
|---|---|---|---|
|   |   |   | |
| `mul r1,r0,r0` | `mul r1,r0,r0` | `mul r1,r0,r0` | `mul r1,r0,r0` |
|   |   |   | `if state==0 then state:=1 else ABORT` |
| `push r1` | `push r1` | `push r1` | `push r1` |
|   |   |   | `if state==0 then ABORT` |
| `ret` | `ret` | `ret` | `ret` |

# Combining SFI and EM

No sending on the network after reading any files

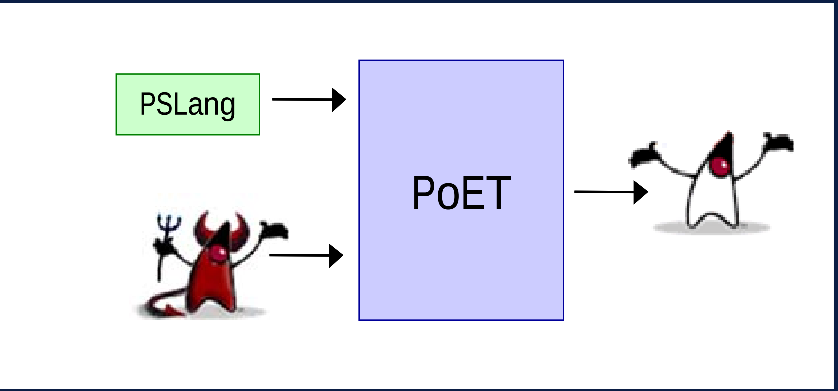



```
...
ldc 1
putstatic SASI.stateClass.state
invokevirtual java.io.FileInputStream.read()I
...
getstatic SASI.stateClass.state
ifeq SUCCEED
   invokestatic SASI.stateClass.FAIL()V
SUCCEED:
   invokevirtual java.net.SocketOutputStream.write(I)V
...
```

# Java IRM: PSLang & PoET

Java IRM Implemention

- Rewrite JVML classes
- Use guarantees given by the JVML verifier




- **PSLang:** Policy Specification Language
  - Exposes JVM abstractions: methods, classes, ...
- **PoET:** Policy Enforcement Toolkit
  - Captures JVM events: method calls, exceptions, ...
- Small addition to TCB: approx. 17.5K lines

# Elements of IRM Specification

- **Add Security State**
  - Rich set of data structures available
  - State either global or tied to program objects
  - Not visible to original program
- **Events trigger Security Updates**
  - Updates: Computation on security state
  - Any event may trigger an update
    - Begin/end of methods, instructions, ...
    - Both load-time and run-time system events
  - Updates can invoke HALT primitive

# Elements of PSLang

- Seperation of load- vs. run-time
  - Load-time synthesis of extended semantics
- Designed for partial evaluation
  - Run-time-constant data structures
  - Side-effect-free functions
- Global and context-local state
  - Local tied to classes or object instances
- Complete, modular, and extendable

# Writing IRM Security Policies

- IRMs allow arbitrarily tight constraints
  - Pin down app behavior, implementation details, even allowable input data
  - In the limit amounts to 2nd implementation
- Can enforce system-call security for any API
  - E.g., field access and calls to library methods
- Must know & trust all such library APIs
  - Implementation cannot betray this trust
    - Example: reading files via font-rendering
- Library policies are both reusable and important

# Event Synthesis Required

- Security Event Synthesis
  - Higher-level semantic information derived from computation on lower-level events
  - E.g., firewall stateful content inspection
- Most policies need additional semantics
  - "Push before Ret" and stack memory
  - "No-div-zero" and byte-aligned jumps
  - High-level API policy $\Rightarrow$ constrain low-level
- Mechanism shouldn't hard-code synthesis

# Examples:
# Limit open windows & Chinese Wall

```
IMPORT LIBRARY Lock;
ADD SECURITY STATE {
    int openWindows = 0;
    Object lock = Lock.create();
}

ON EVENT begin method
WHEN Event.is("Window.open()")
PERFORM SECURITY UPDATE {
    Lock.acquire(lock);
    if( openWindows == 10 ) {
      FAIL[ "Too many windows" ];
    }
    openWindows = openWindows+1;
    Lock.release(lock);
}

ON EVENT begin method
WHEN Event.is("Window.close()")
PERFORM SECURITY UPDATE {
    Lock.acquire(lock);
    openWindows = openWindows-1;
    Lock.release(lock);
}
```

```
IMPORT LIBRARY Map;
ADD SECURITY STATE {
    Object map = Map.create();
}

PROCEDURE Object getCategory(Object name) {
    if( name=="IBM"   ) {return "COMPUTERS";}
    if( name=="Apple" ) {return "COMPUTERS";}
    if( name=="GM"    ) {return "CARS";}
    if( name=="BMW"   ) {return "CARS";}
    FAIL[ "Unsupported company ",name ];
}

ON EVENT begin method
WHEN Event.methodNameIs("accessCompany")
PERFORM SECURITY UPDATE {
    Object C = State.get( "$methodArg1" );
    Object category = getCategory( C );
    Object oldC = Map.get( map, category );
    if( oldC == null ) {
        Map.put( map, category, C );
    } else {
        if( C != oldC ) {
          FAIL[ "Can't access company ", C ];
        }
    }
}
```

# Example IRM Tradeoff

Caller or callee instrumentation ?

- If callee: everybody pays price
  - Library callee used by multiple principals must fork codebase or do a runtime check
- If caller: must do a lot of work
  - Especially for calling object methods
  - Synthesis required for derived classes etc.
- IRM specification writer can choose

# Library policies

- Most IRM enforcement probably on libraries
  - Must know semantics to regulate use
  - Useful policies apply to more than one program
- Each API can be like a system-call interface
  - Includes objects, method calls and direct access
  - Library design affects potential policies
- High-level API policies might be subverted
  - Policies must either preclude lower-level access or synthesize that high-level operation occurred

# Racing issues

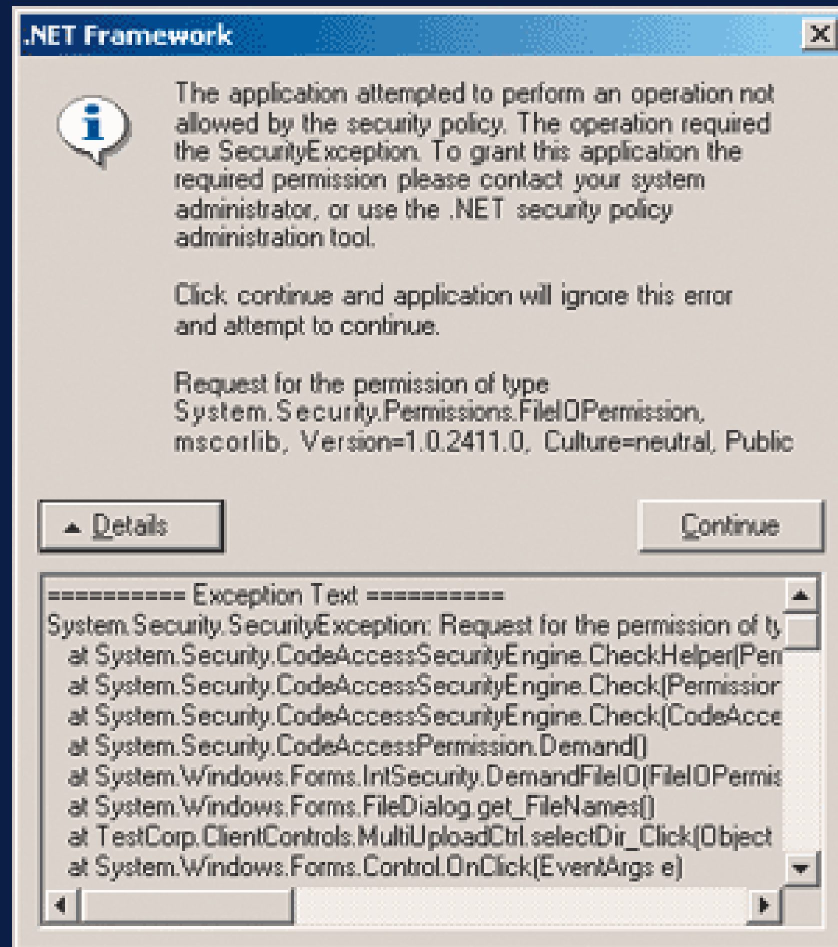
- Must enforce security policies on multi-threaded Java programs
- Must serialize check/event pairs
  - cobegin{Crd;Rd || Csnd;Snd}
    may run Csnd;Crd;Rd;Snd
- Time of check to time of use
  - Hard with complex history-based policies
  - Can sometimes emulate OS copy-in behavior
- PSLang offers synchronization mechanisms

# IRMs in Retrospect

- Writing good policies is hard
  - Extensive synthesis often required
- App-level policies tied to app semantics
  - Makes most sense for library code
- Environment agnostic
  - OS independent, can be added after-the-fact, and will travel
- IRMs can secure use of high-level APIs
  - With flexibility to make tradeoffs

# “Modern” Java/.NET Policies

- Modern policies may use a lot of properties and historical data to make access control decisions
- Java/MS’s CLR use the stack trace to implicitly constrain sets of permissions



# Java Stack Inspection

Two-second refresher course

- Enforcement based on runtime call stack
- Each stack frame is in a protection domain
- Each protection domain has set of premissions
- **`checkPermission:`** Stack has suitable permissions
- **`doPrivileged:`** Amplification of available permissions

| | Protection domain: **Untrusted Applet** | Protection domain: **GUI Library** | Protection domain: **File System** |
|---|---|---|---|
| **File access permissions:** | /home/ue/* | /fonts/* | <<ALL FILES>> |

**display:**
```
...
load(`thesis.txt')
use plain font
show on screen
...
```

**use plain font:**
```
...
doPrivileged {
  load(`Courier')
}
...
```

**load(file F):**
```
...
checkPermission(F,read)
access on disk(F)
...
```

# Implementing Stack Inspection

- How are the primitives actually used?

| Benchmark | Method calls | doPrivs | checkPerms | Thrds |
|---|---|---|---|---|
| Jigsaw | 2,476,731 | 1,002 | 5,333 (18,7) | 71 |
| javac | 1,456,970 | 0 | 1,067 (12,4) | 0 |
| tar | 19,580 | 0 | 6,509 (8,6) | 0 |
| MPEG_Play | 35,997,662 | 101 | 205 (5,7) | 201 |

- IRMs allow playing with the tradeoffs
  - Allows synthesis of security-relevant data...
  - ...or access to any interface that exposes it
- Can make an IRM as specific as wanted
  - ...to a particular app, or a particular policy

# Stack Inspection IRMs

[Erlingsson Schneider 00]

## IRM$_{SPS}$: The obvious first approach

- Maintain shadow call stack to consult in enforcement

**Method call**

Push/pop protection domains on shadow call stack

**doPriv { S }**

Push/pop **doPriv** token on shadow call stack, before/after **S**

**checkPerm( P )**

Scan shadow call backwards, check **P** for each domain, stop on **doPriv** or end

## IRM$_{Lazy}$: Optimize for the most common case

- Pry out JVM's call stack & compute enforcement data

**Method call**

Nothing

**doPriv { S }**

Get current call stack, push/pop its depth onto a seperate **privStack**

**checkPerm( P )**

Get current call stack, scan it backwards and check **P** for the domain for each frame, stop if reached the depth on the top of **privStack** or end

# Nitty-gritty details

```
//
// on doPrivileged, push the doPriv token onto the domainStack
//
SIDE-EFFECT-FREE FUNCTION boolean doPrivilegedCall(Object instr) {
    return Event.instructionIs("invokestatic")
        && JVML.strEq(Reflect.instrRefStr(instr),
                      JVML.strCat("java/security/AccessController/doPrivileged",
                            "(Ljava/security/PrivilegedAction;)Ljava/lang/Object;"));

}
ON EVENT at start of instruction
WITH doPrivilegedCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread, "java/lang/Thread/domainStack");
    Stack.push( stack, doPrivToken );
}
ON EVENT at finally completed instruction
WITH doPrivilegedCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread, "java/lang/Thread/domainStack");
    Object discard = Stack.pop( stack );
}
```

# IRM Performance

- **$IRM_{SPS}$**

| Method call | doPriv | checkPerm | New Thread |
|---|---|---|---|
| 1,00μs | 1.7μs | 7.7μs | 6.5μs |

- **$IRM_{Lazy}$**

| Method call | doPriv | checkPerm | New Thread |
|---|---|---|---|
| 0μs | 23.4μs | 22.4μs | 29.8μs |

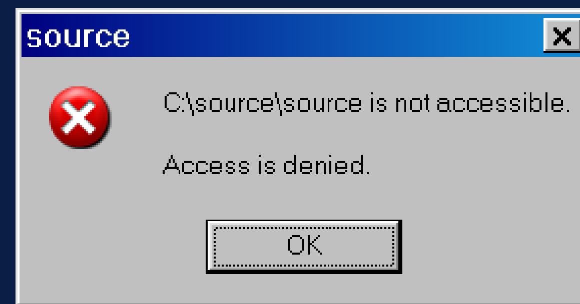
- End-to-end $IRM_{Lazy}$ performance competitive with Sun's JVM's built-in stack inspection

# Beyond EM

- IRMs more than safety properties
  - Can include static analysis
    - Load-time security updates already do this
  - On violation, truncation not only option
    - E.g., throw exception as remedial action
- However, harder to reason about
  - Composition problem even harder
- Subject of current study

# Virtualize or Modify Execution

- Allow execution monitors that change execution behavior without halting it




- Richer but more difficult to reason about
  - example info-flow: always return 1
  - can change return value in this case (more generally, can normalize all external behavior)
- Break out of “only security policies” box

# Example Problem

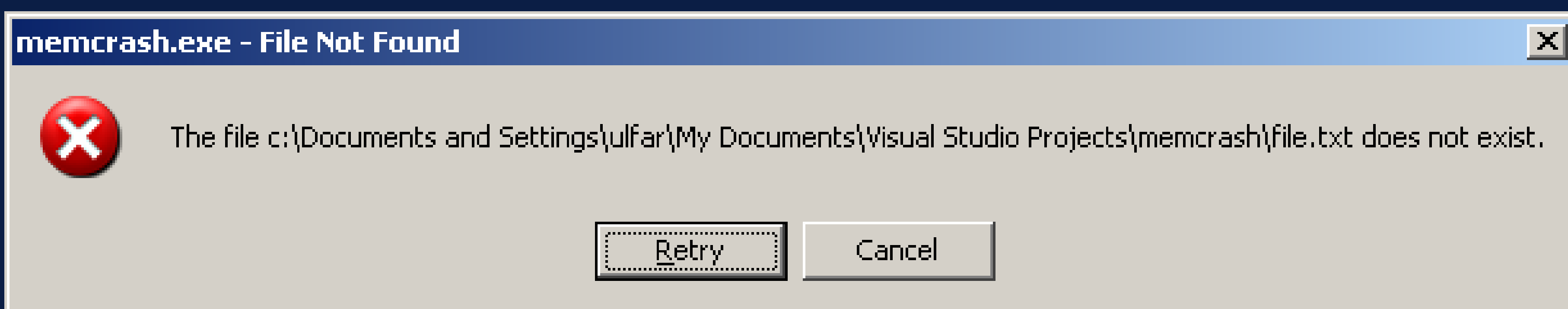
- Policies undo the effect of each other
- Composition may result in bad policy
  - Even so policy is *always* violated

To see this, consider two IRM security policies that both wish to prevent the occurrence of $F(1,1)$ and $F(-1,-1)$. Now, one of these policies might sandbox $F$ operations by negating the first argument, and the other might sandbox $F$ by negating the second argument. Then, however, the composition of these two security policies might turn $F(1,1)$ into $F(-1,-1)$, and vice versa, subverting the intent of both policies.

# System Security Policies

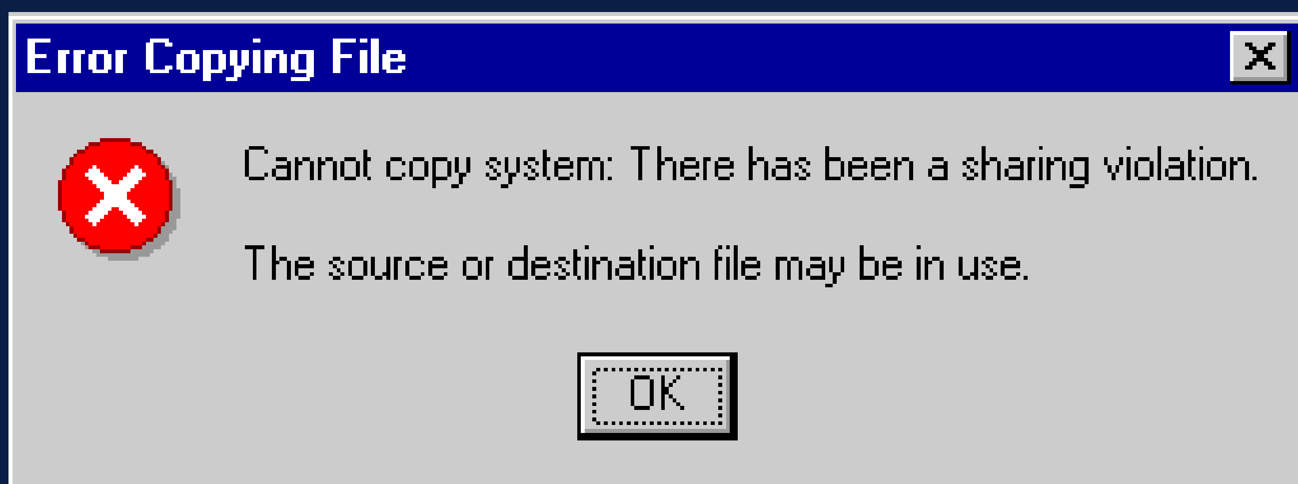
- Higher level operations (not just machine instructions)
- Kernel provides RM and validity checks

```
void main()
{
    OFSTRUCT ofs;
    HFILE f = OpenFile("file.txt",&ofs,OF_PROMPT);
}
```

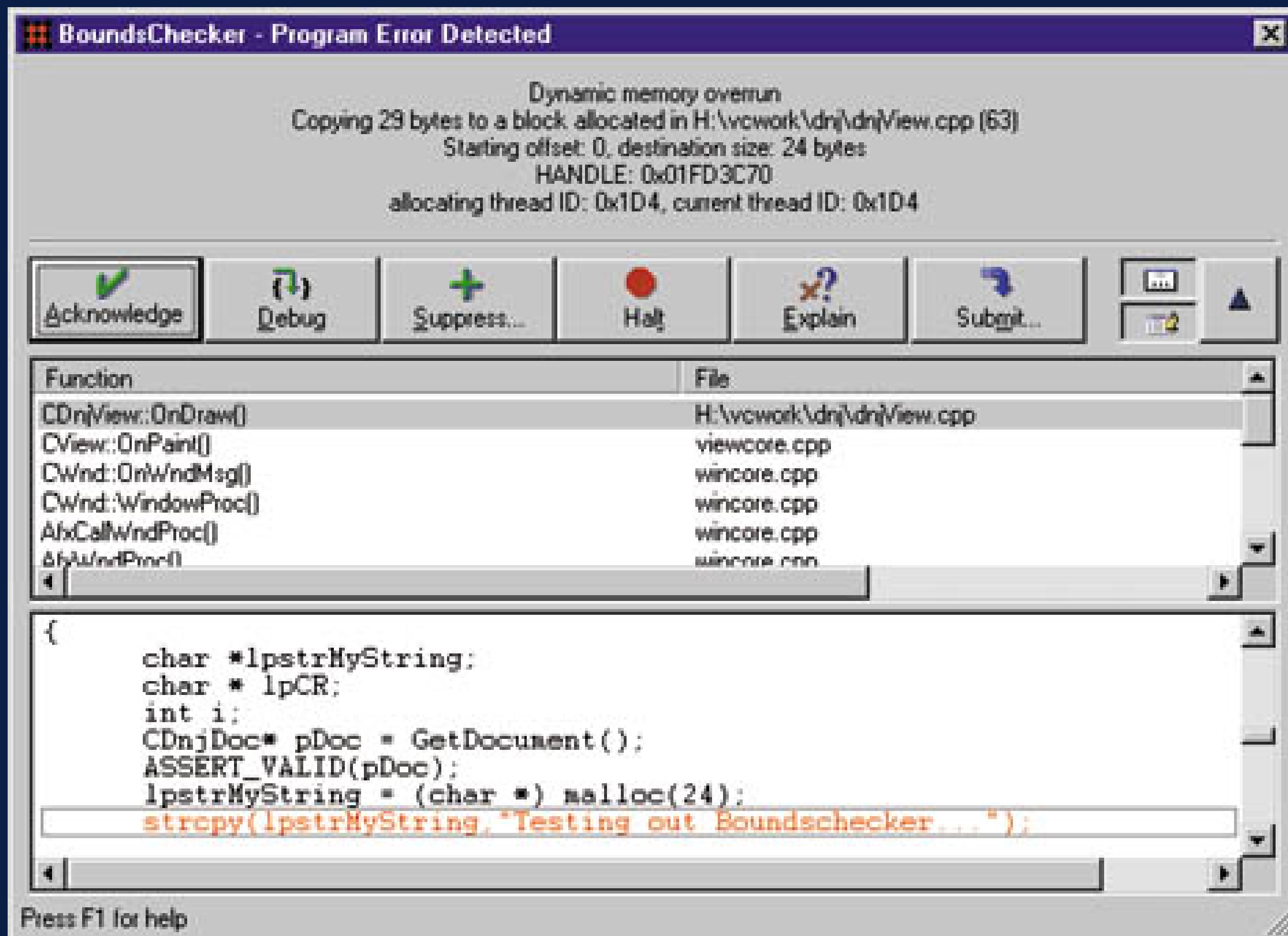


# File System Integrity Policy

- Execution monitoring used to enforce many properties by operating systems
- Apart from security, integrity of data structures etc.



# Buffer Overflow Policy



# Related work

- Lots of related work, old and new:
  - Dates back to SDS 940, at Berkeley in ’69
- Software Fault Isolation and Verifiable Code Certification: JVML verifiers, PCC, TAL, etc.
- Reference Monitor Literature is relevant
  - Application-specific security (e.g., Clark&Wilson)
  - History-based access control
- Program modification: ATOM and AspectJ
  - *Theory of Aspects* [Walker Zdancewic Ligatti 03]
- Also, Generic Software Wrappers, Naccio, etc.

# Type Encapsulation of State

[Walker 00]

- Sophisticated type system
  - Certifies (a la PCC or TAL) that an automaton policy is enforced
  - Types encodes passing of security state
  - Transformations and lemmas depend on particulars of the specific security automata(s) to be enforced
- Simpler notion of automata
  - Not 1st-order predicates on transitions



Microsoft Research

# More Efficient Rewriting

[Colcombet Fradet 00]

- Elaborate new theory and techniques
- Transform code according to policy
- Modified code propagates run-time encoding of security state
- State checked to block illegal actions
- Static analysis reasons about state
  - When analysis impossible, runtime check inserted (similar to cqual)

# Efficiency via Partial Evaluation

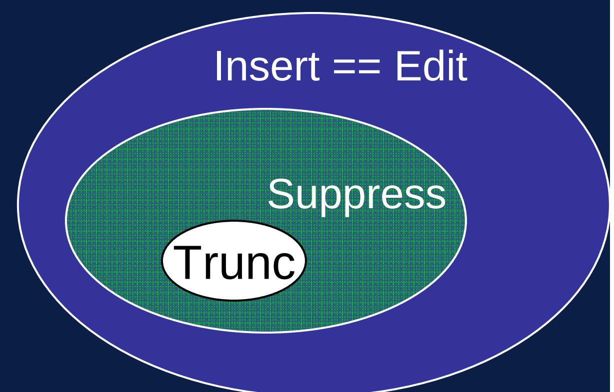
[Thiemann01, 04?]

- Standard specialization techniques
  - All work on partial evaluation applies
- Transform a monitoring interpreter into a non-standard (security) compiler
  - Get the IRM rewriter for free
- Nicely propagates check results etc.
  - Was exp.time with code duplication
  - Newer result: linear with no duplication

# More Enforceable Security

[Bauer Ligatti Walker 02, 04]

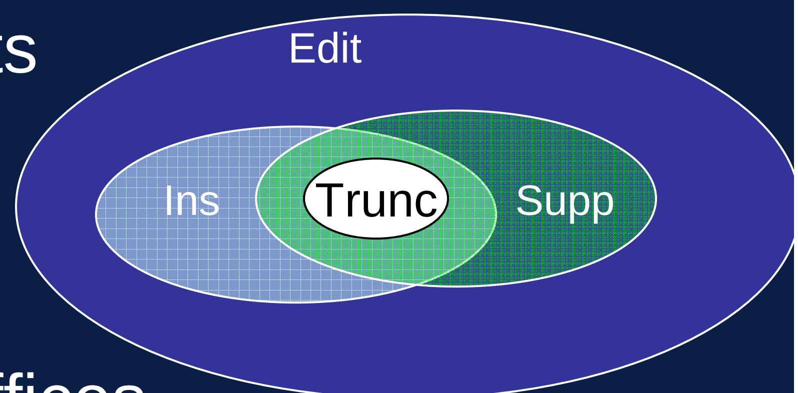
- Formal definition automata with side effects
- Uniformity and non-uniformity ($\Sigma \overset{?}{\subset} A^*$)
- Figure shows precise non-uniform
  - Insert equally powerful as Edit
  - Suppression strict subset
  - Truncation more restrictive
- If “precise uniform” then all three circles equal EM+truncation
- On non-uniform systems can do more than truncation automata



# Edit Automata

[Ligatti Bauer Walker 04]

- **Precise** means you have to accept good sequences in lockstep with their generation
  - so can't use the "edit tricks" on a good trace
- **Effective** means we can suppress actions and then later insert their (atomic) effects into trace
- **Transparency** accounts for semantic equality between input & output
- **Conservative** mean any good sequence suffices



# Calculus for Composing SP

[Bauer Ligatti Walker 04]

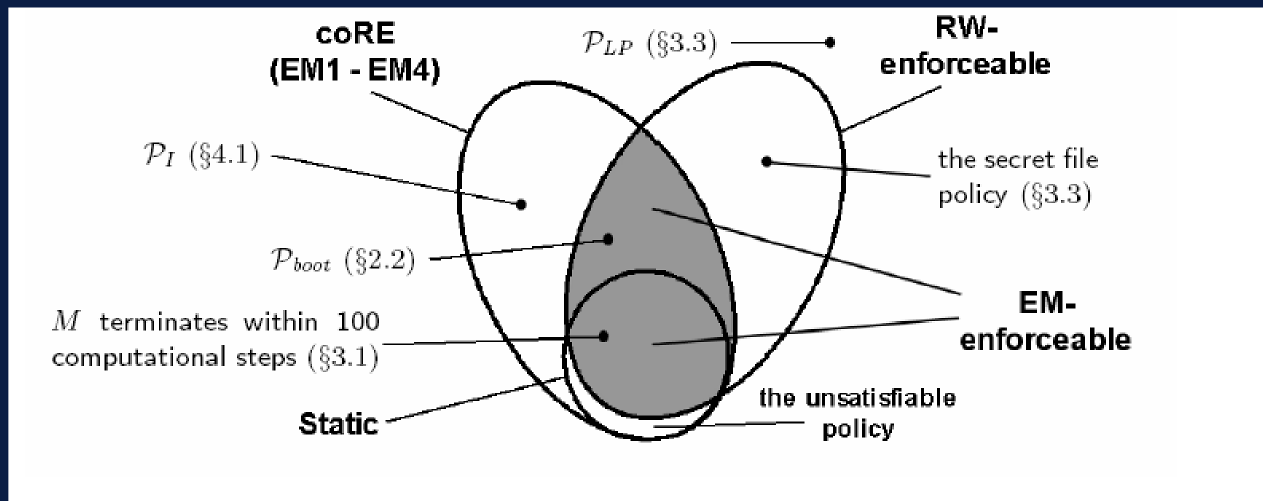
Types and Effects for Non-interfering EM [BLW02]

- Given a policy what are the actions (what can it suppress and insert)
- Make sure that the edit actions of two concurrently executing monitors don't affect the inputs relevant to each other
- Four combinators (two seq. two parallel)
  - seq. combinators can be affected by effects

Authors extend in later work to allow programmers to develop their own combinators (Polymer)

# EM Computability Classes

[Hamlen Morrisett Schneider 04] [Viswanathan00]




- Looks at detectors and complexity
  - Some detectors may reject “too early”
- Relates computability and enforcement

# Limiting the Security Automata

[Fong04]

- Constrain the capabilities of the Execution Monitor (aka SA)
- Restrict EM to track a shallow history
- Sufficient for
  - Chinese Wall
  - Low-water-mark
  - One-out-of-k authorization
  - etc.

# Another Extension to EM

- Introduce static analysis
  - can incorporate any deterministic finite-time decision procedure, as a step in monitoring mechanism
- Use as one step of security automaton
  - can do static analysis before execution (and get guarantees about all traces)
  - can do static analysis in the middle (are all suffixes of current state good?)
    - similar to partial evaluation of program
    - useful, say for locks, check at acquire that will it be released etc.

# More Lectures

- More information on Enforceable Security Policies, Software Fault Isolation, Java Stack Inspection, and Inlined Reference Monitors
  - http://www.cs.cornell.edu/html/cs513-sp99/03.lectsmry.html
  - http://www.cs.cornell.edu/Courses/cs513/2000sp/02.outline.html

# Bibliography 1 of 4


[Alpern Schneider 87] Recognizing safety and liveness. *Distributed Computing* 2, 3 (1987), 117--126. [TR 86-727]

[Anderson72] Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), October 1972.

[Bauer Ligatti Walker 02] More Enforceable Security Policies. *In the Workshop on Foundations of Computer Security (FCS 02)*. Copenhagen, July 2002.

[Bauer Ligatti Walker 04] Types and Effects for Non-interfering Program Monitors. International Symposium on Software Security. Tokyo, November, 2002. Revised for printing in Software Security -- Theory and Systems, LNCS 2609, Springer, pp 154--171. December 2002.

[Brewer Nash 89] The Chinese Wall Security Policy. *Proceedings of 1989 IEEE Symposium on Security and Privacy*,1989: 206-214

# Bibliography 2 of 4


[Colcombet Fradet 00] Enforcing Trace Properties by Program Transformation, *Proc. of Principles of Programming Languages, POPL'00*, ACM Press, pp. 54-66, Boston, January 2000.

[Cowan et al. 98] StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Published in the proceedings of the 7th USENIX Security Symposium, January 1998, San Antonio, TX.

[Durden 02] Bypassing PaX ASLR protection, Phrack, 2002-07-28

[Erlingsson Schneider 99] SASI enforcement of security policies: A retrospective. *Proceedings of the New Security Paradigms Workshop* (Caledon Hills, Ontario, Canada, September 1999), Association for Computing Machinery, 87--95.

[Erlingsson Schneider 00] IRM enforcement of Java stack inspection. *Proceedings 2000 IEEE Symposium on Security and Privacy* (Oakland, California, May 2000), 246--255. [TR 2000-1786]

[Fong04] Access Control By Tracking Shallow Execution History. *Proceedings 2004 IEEE Symposium on Security and Privacy* (Oakland, California, May 2004)

[Jones Kelly 97] Backwards-compatible bounds checking for arrays and pointers in C programs. *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, May 1997.

# Bibliography 3 of 4

[Hamlen Morrisett Schneider 04] Computability classes for enforcement mechanisms. Submitted for publication. Also available as Cornell Computer Science Department Technical Report TR 2003-1908, August 2003.

[McLean94] A General Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions. *Proceedings of 1994 IEEE Symposium on Research in Security and Privacy*, 1994. PostScript, PDF

[Ligatti Bauer Walker 04] Edit Automata: Enforcement Mechanisms for Run-time Security Policies. Submitted, December 2002; revised June 2003 for the International Journal of Information Security

[PaX] http://pax.grsecurity.net/

[Ruwase Lam 04] A Practical Dynamic Buffer Overflow Detector. *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004

# Bibliography 4 of 4

[Schneider00] Enforceable security policies. *ACM Transactions on Information and System Security* 3, 1 (February 2000), 30--50. [TR 99-1759]

[Small97] MiSFIT: A Tool for Constructing Safe Extensible C++ Systems. *Proceedings of the Third Usenix Conference on Object-Oriented Technologies*, Portland, OR, June 1997.

[Thiemann01] Enforcing Security Properties by Type Specialization. *In European Symposium on Programming (ESOP'01)*, volume ? of Lecture Notes in Computer Science, Genova, Italy, April 2001

[Wahbe Lucco Anderson Graham 93] Efficient Software-Based Fault Isolation. *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, December 1993.

[Walker 00] A Type System for Expressive Security Policies. *In the Twenty-Seventh ACM SIGPLAN Symposium on Principles of Programming Languages*. Boston, January 2000. A previous version of this paper appeared in the FLOC'99 Workshop on Run-time Result Verification, Trento, Italy, July 1999.

[Walker Zdancewic Ligatti 03] A Theory of Aspects. ACM SIGPLAN International Conference on Functional Programming, August 2003