Using Memory Errors to Attack a Virtual Machine

Sudhakar Govindavajhala

and

Andrew W. Appel

Princeton University

16 July 2003

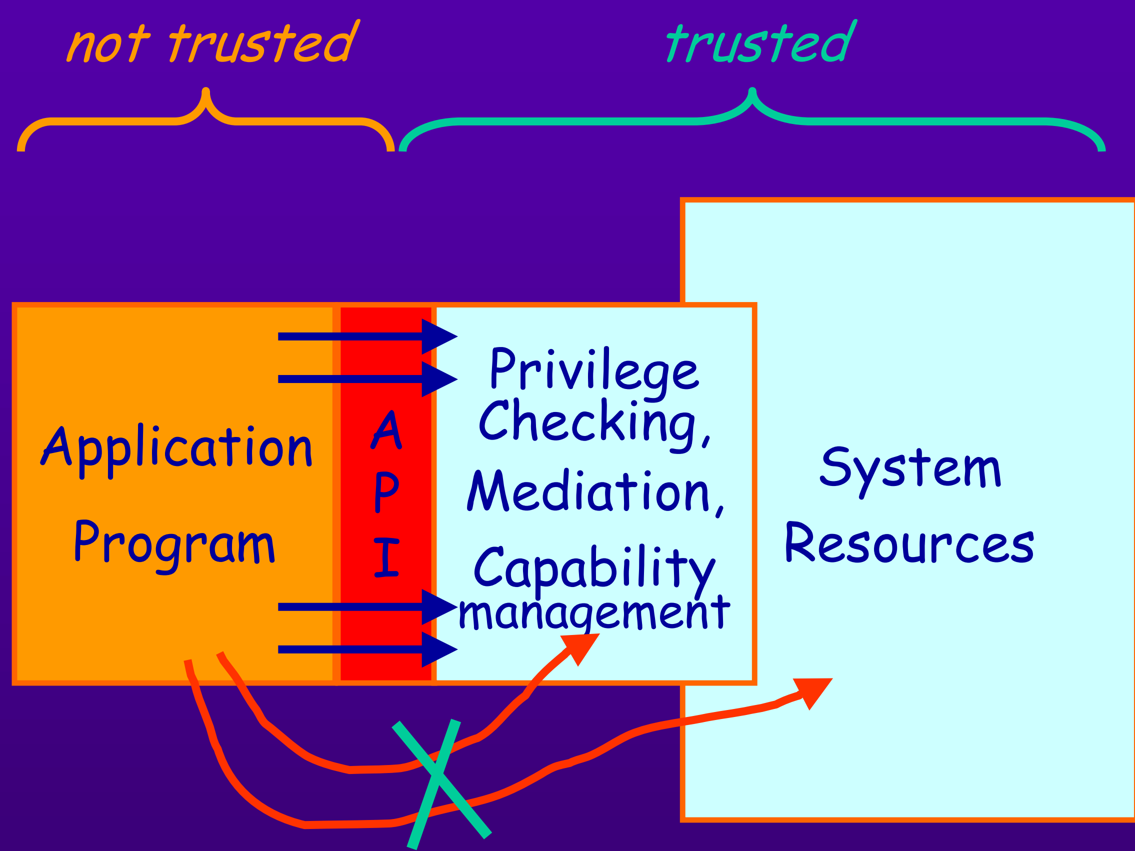
Introduction

- Java and .NET use static checking for security
- Proved sound
- We use natural memory errors to defeat them
- We execute arbitrary code and obtain complete access to the VM

Success ratio: $10^{-9} \longrightarrow 0.70$

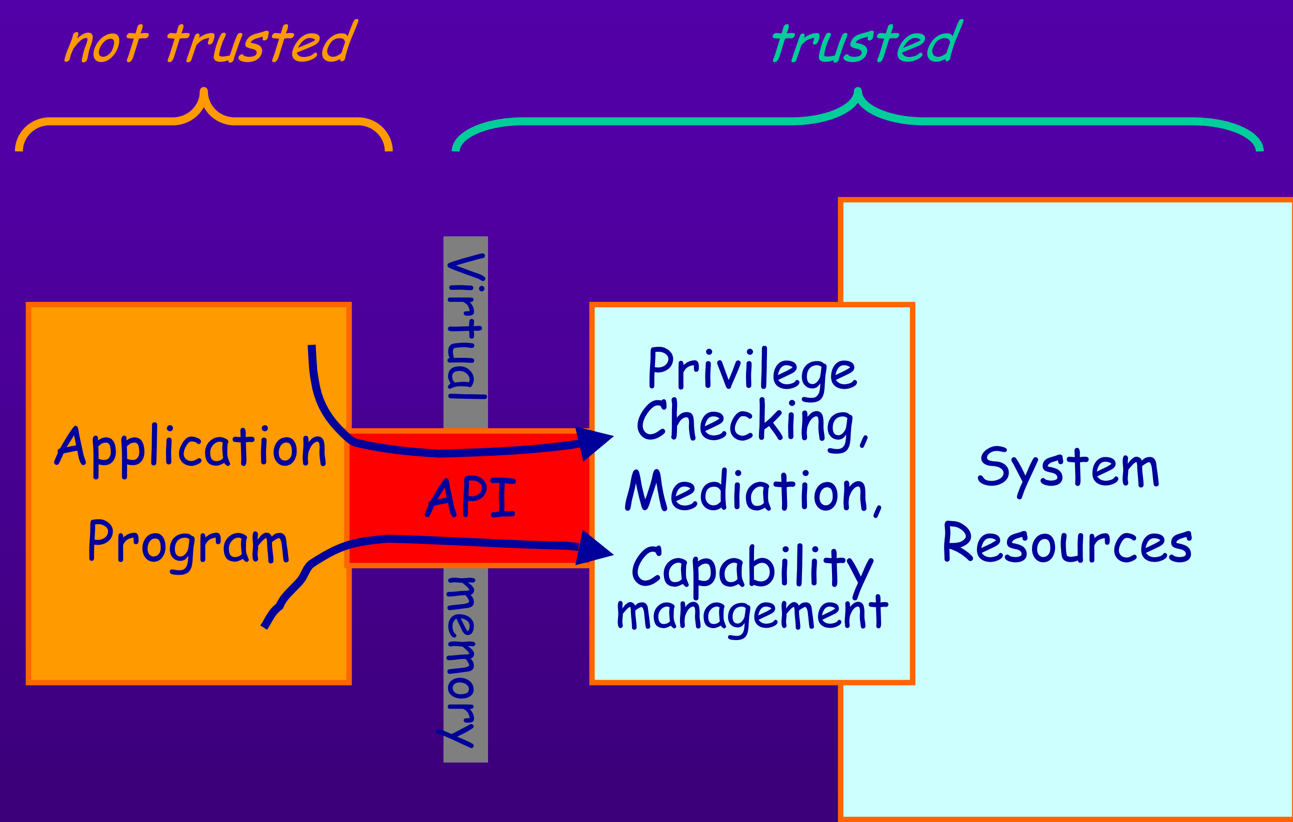
Typical system architecture

Access to system resources mediated by API



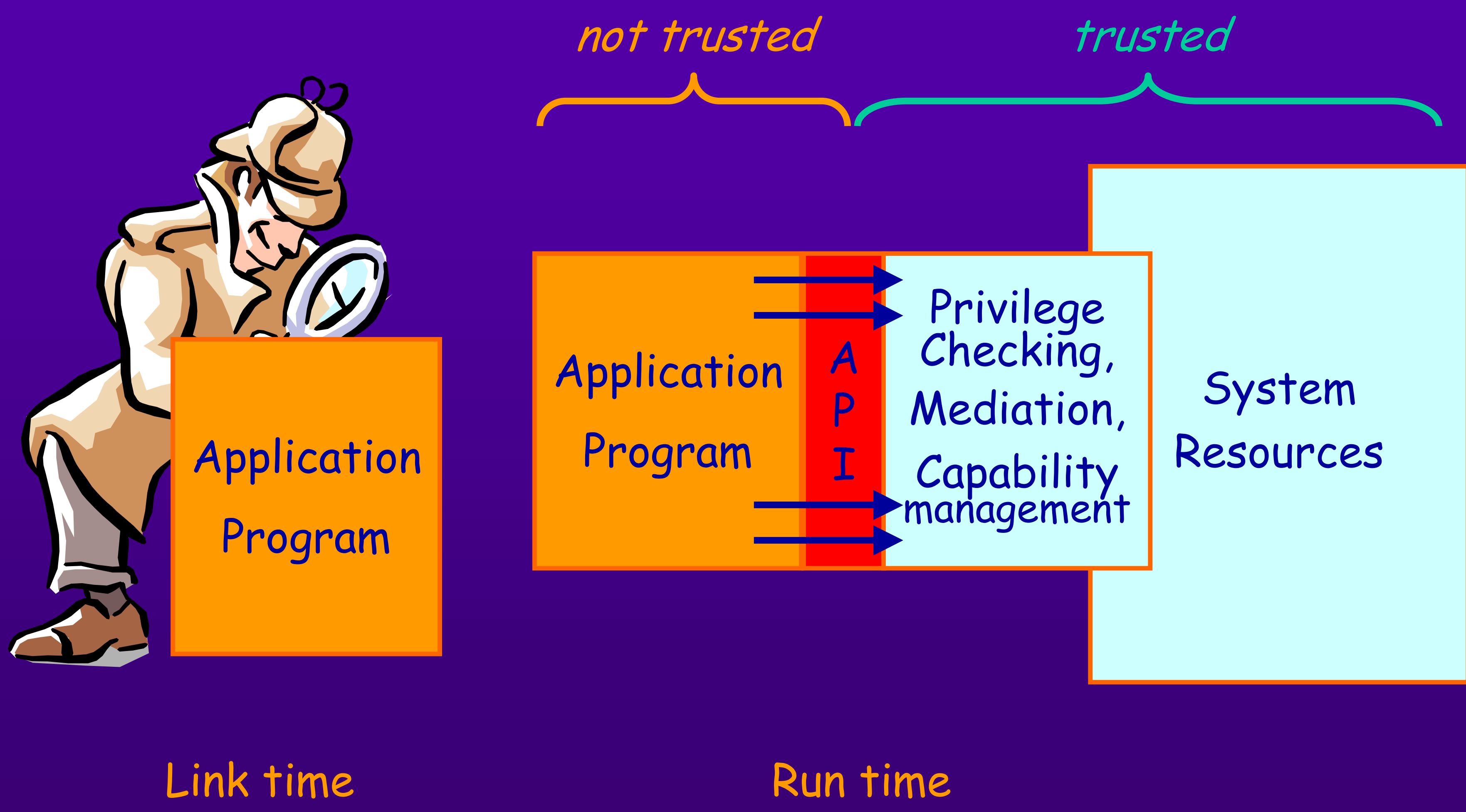
Must prevent capability management from being bypassed

Operating systems, virtual memory



Crossing virtual-memory boundary is slow and clumsy

Static checking



Java

- Efficiently compiled to machine code
- Portable
- Byte code verifier guarantees safety of program
- After verification, trusted and untrusted components run in the same address space
- Proved type sound; subsets machine checked.

Overview

- Security is premised on type safety

Single memory error

↓

Type-safety violation

↓

Memory-safety violation

↓

Executing arbitrary code

```
class SecurityManager {
    .
    .
    private   bool allowFileAccess;
    .
    .
}
```

- Attack possible due to hardware error

The class definitions

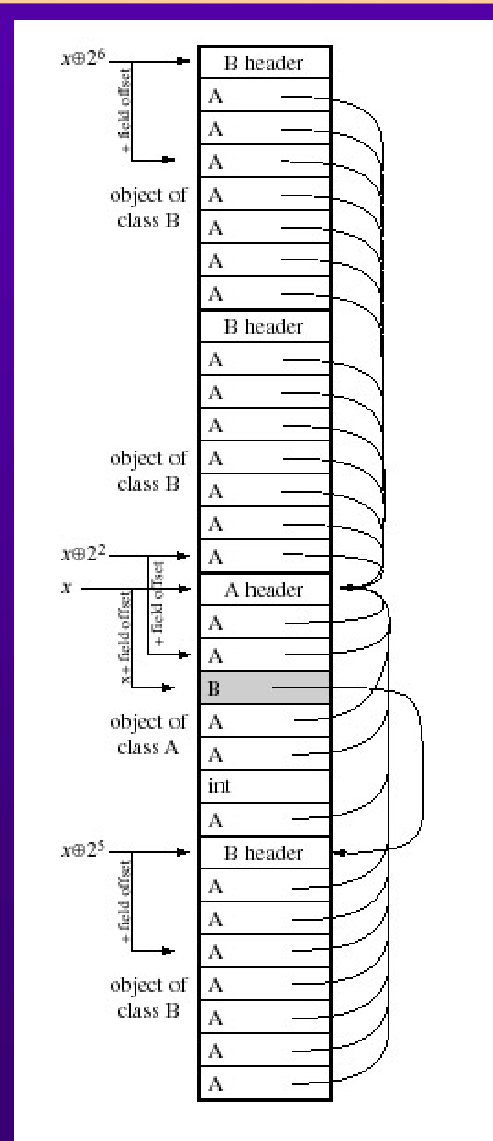
```
class A {
        A a1;
        A a2;
        B b;
        int i;
        A a5;
        A a6;
        A a7;
}
```

```
class B {
        A a1;
        A a2;
        A a3;
        A a4;
        A a5;
        A a6;
        A a7;
}
```

Object : 32 bytes, field : 4 bytes, header : 4 bytes.

The attack applet

- Inviting to attack type safety.
  - Segment size : Data >> text
- Attack applet allocates a lot of memory like this
- Applet waits for a memory error
- Random pointer dereference will fetch from an A field.
- Type safety violated
  - Pointer of type B points to an A object

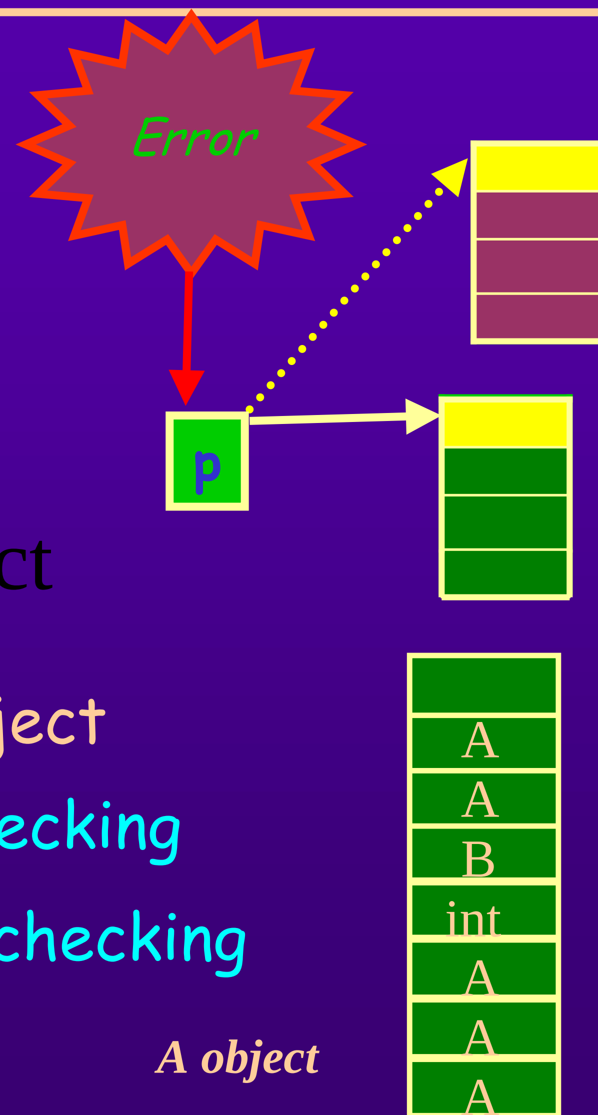


Princeton University

Detecting and exploiting the bit flip

- Wait for a flip; detect it
  - for each pointer p of type A,
    - if (p != a) …
    - Now p.b points to a A object

- VM thinks p.b type is B
- But, we can typecast p.b to an A object
  - Object r = p.b;   Static checking
  - A q = (A) r;   Dynamic checking
- Result : undetected type system violation



Detecting : the code

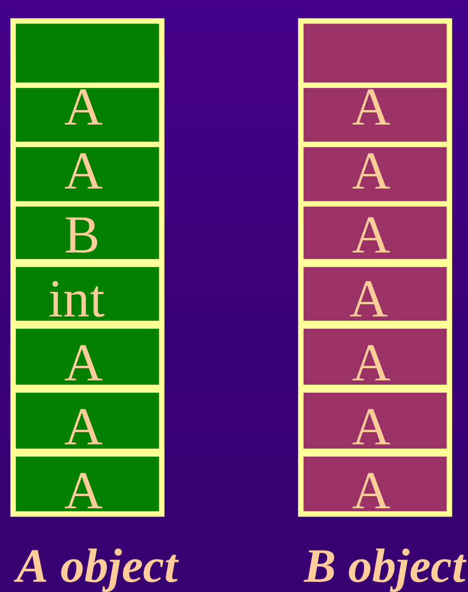
```
A p, q;
A ref_A; // the A object
while (true)
    for each pointer p of type A
        if (p != ref_A)  { // bit flipped
            Object r = p.b;
            q = (A)  r;
            .  // use p.b  and q, which contain same pointer, but are of diff
               types
            .
            .
        }
```

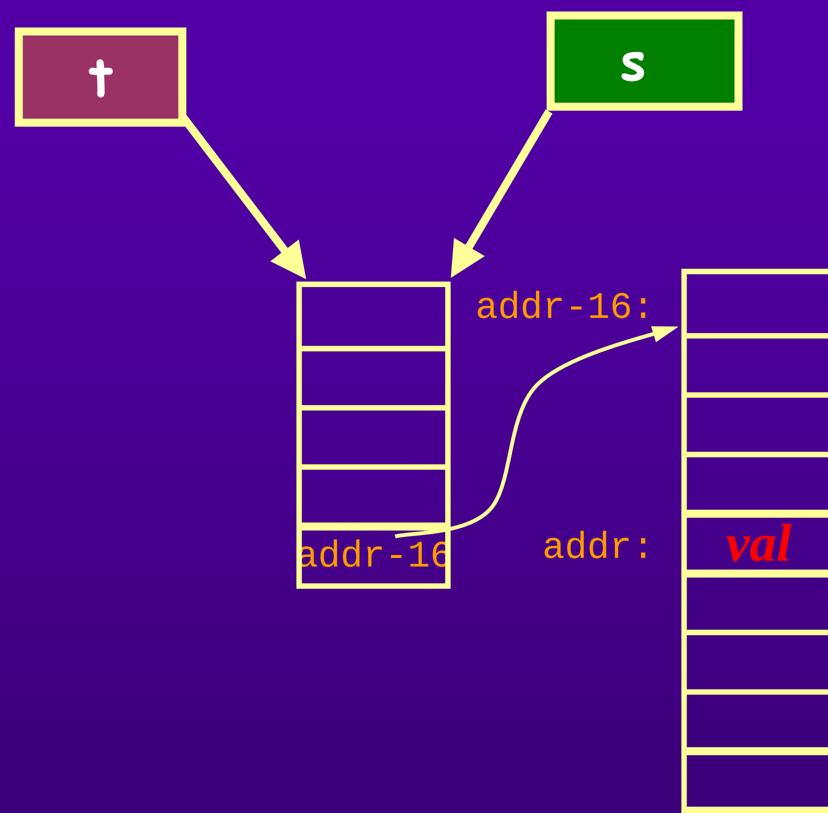
Obtaining memory safety violation

```
write (A s, B t, val, addr) {

 // s == t

  s . i = addr - 16;
  t.a4.i = val;
}
```



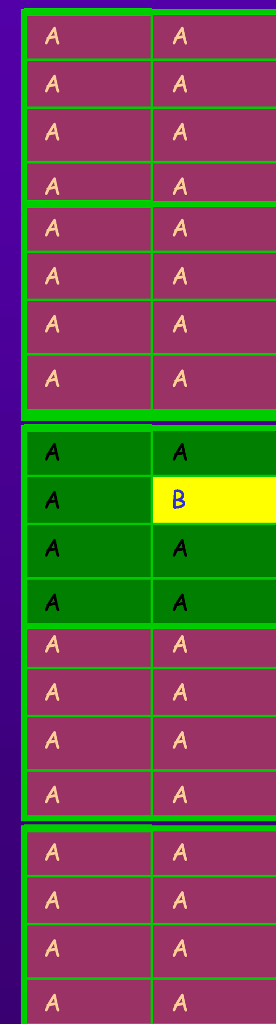


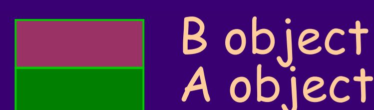
Using the memory safety violation

- We can write any data at any address
  - Fundamental safety property is violated
- Pointer forging — Clumsy ☹
  - Fill array with machine code
  - Forge function pointer table
- Security Manager — Elegant and portable ☺
  - Obtain the address of the security manager
  - Then turn off the security manager
  - Load and execute arbitrary code !
  - complete access to the VM !

Obtaining the layout

- Allocate a random number of B objects
- Allocate an A object
- Exhaust memory with B objects
- Set each A field of each object to the A object.
- Set the B field of the A object to any B object





Larger objects

- Object header overhead less
- More control on layout
- Maximum Hamming distance
- What if bits 2…9 flip?


```
class A {
        A a1;
        .
        A a125;
        B b
        A a127;
        .
        A a254;
}
```

Efficiency 25 % already!

Cousin number

- p, q are cousins if p, q differ in a bit
- x, y are cousins; different types; bit flip implies type error

- Cousin number(p) : number of cousins of p

- a is the single object of type *A*
- Cousin number(a) is large $\Rightarrow$ bit flip in large number of bits gives type error.

How much is cousin number ?

- N objects, each of size 1024 bytes

- Ideal case
  - Object size s, number of objects N is a power of 2
  - All objects are at contiguous locations
  - In this case, C(any object) = $\log_2 N$

- Relaxed case
  - Very likely that C(any object) is approximated well by $\log_2 N$

  - In particular, C(the A object) ~ $\log_2 N$

Measured distribution of cousin numbers

- Measured the cousin numbers in one particular run
- Object : 1024 bytes, word : 4 bytes
- N= 426,523; $\log_2 N = 18.7$
- Empirical measurement
  - mean cousin number = 17.56
- $\log_2 N$ is really a good predictor of cousin number!

Cousin number	# of objects
0,1	2
3	13
4	7
5	59
6	30
7-12	0
13	614
14	2,868
15	32
16	29,660
17	110,640
18	282,576

Number of exploitable bits

32 bit A-pointer

	$log_2 N$	8	2
32	28	10	2

0

- C(the A object) ~ $\log_2$ N
  - Flip in any $\log_2$ N bits in bits 31…10 usable
- flip in bits 9..2 is anyway usable
- flip in bits 1..0 results in an alignment error
- extreme high bits flip : core dump

We were able to use any flip in the bits 2 … 27

General Discussion

- This attack works on all virtual machines
  - Portable
  - Same code worked for IBM and Sun JVMs
- .NET attack similar

Experiment #1

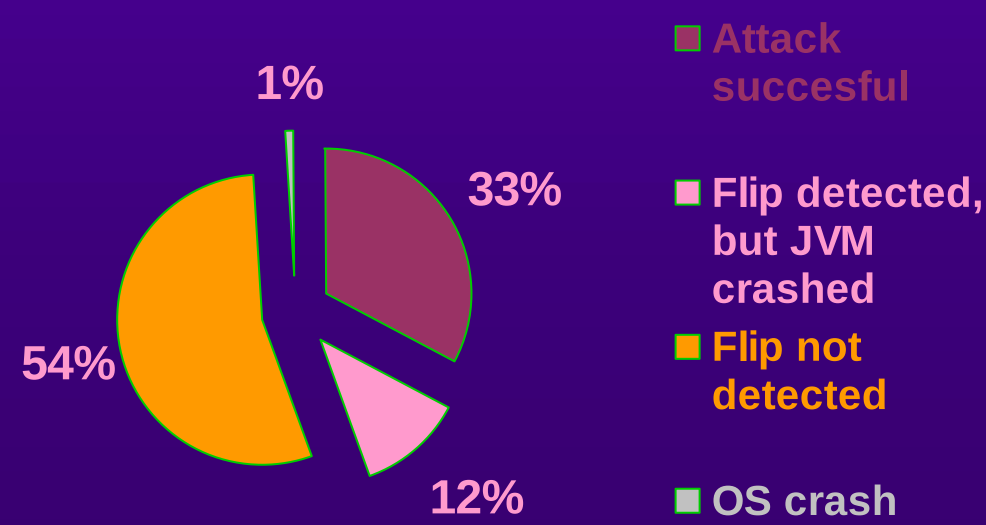
- Use a privileged process to inject single-bit "errors" in physical memory
  - Unix /dev/mem interface to physical memory
  - Measure the efficiency of our attack
  - 128 MB physical memory

- Linux

- IBM, Sun JVMs

Results : /dev/mem to inject errors.

- IBM JVM
- 128 MB memory
  - 57,753 objects
  - 3,035 trials
  - Efficiency : Expected 0.32   Actual 0.34
  - JVM allocates max of 60% of physical memory



Attack scenarios

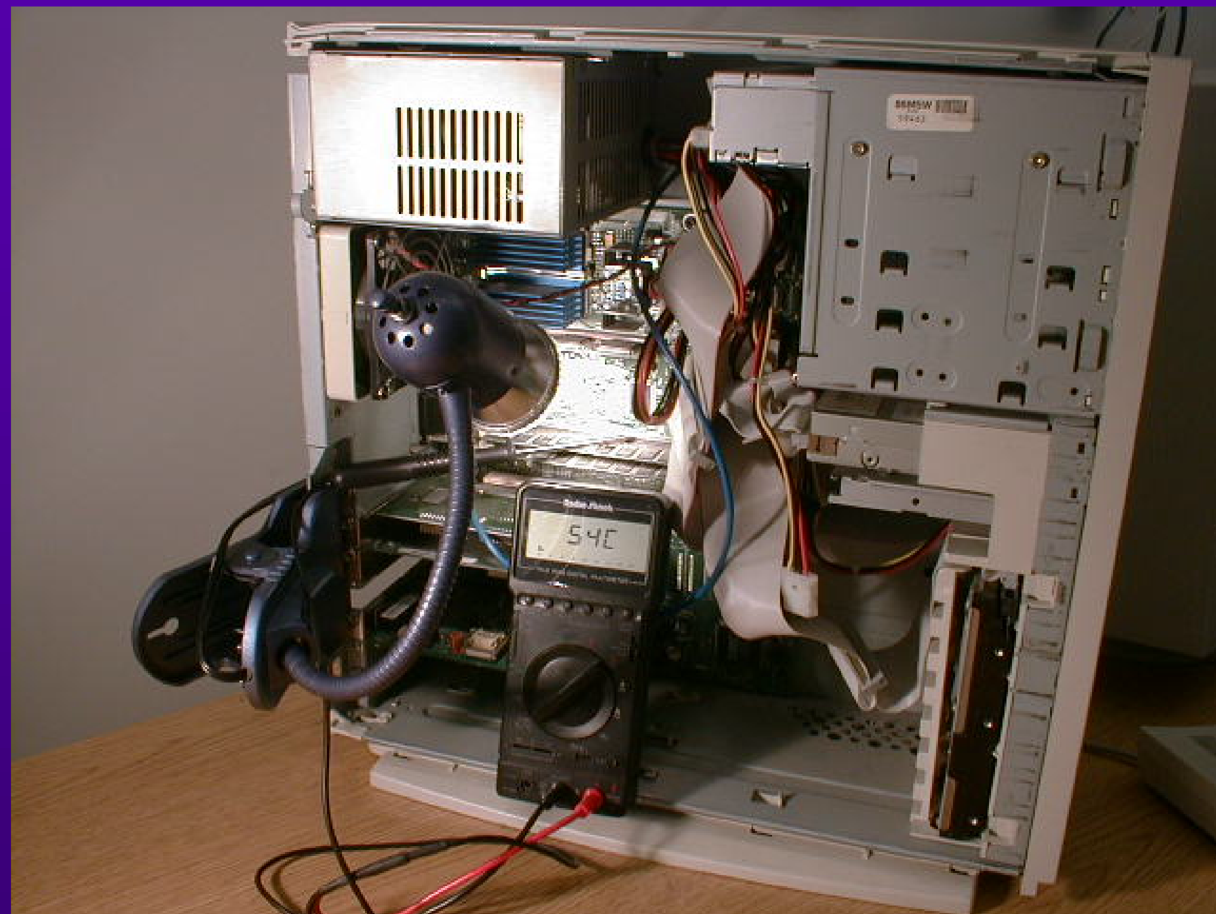
- No physical access
  - SETI @ HOME in Java
  - About one error per month (? Hard to find data)
  - Web browsers
- Complete physical access
  - Screwdriver to remove HDD
- Limited physical access
  - Supply program
  - Induce errors

Susceptibility of DRAM to faults

- Alpha particles
  - Significant in 1990, now insignificant
- Beta rays (high-energy electrons)
  - Don't penetrate packaging
- X-rays
  - Too low energy to affect a DRAM capacitor
- High-energy protons, neutrons (cosmic rays)
  - Yes, but need particle accelerator; available to a few large nation states only
- Thermal neutrons
  - Yes, but where to get a source? Oil drilling...
- Heat
  - Now we're talking!

Experiment #2 : using heat

- Old PC
- Clip-on lamp
- 50-watt spotlight bulb
- Variable AC power supply
- Digital thermometer



Result: between 80° - 100°C, memory starts to have a small number of few-bit-per-word errors. Attack applet is successful.

Defenses against this attack

- Error-Correcting-Code (ECC) memory
  - Not used in most desktop PCs because of cost
- ECC fault logging
  - When unusual numbers of errors seen, shut down
  - Difficult to create 3 bit errors without 1 bit errors
- Total datapath coverage for ECC
  - Old ECC coverage based on "natural fault" model, not a coordinated attack
  - Some new high-end x86 chips (Intel, AMD) have this
- More than 2-bit error detection
  - Need more than 72 bits to represent 64-bit word

Optimising for IBM JVM

hashCode

- Provides good hash function.
- Typically address based to reduce collisions.
- Don't want to expose address for security risks.

```
A = 2 * sp + clock()

B = 2 * sp + time() - 70

int hashCode (address) {
    t1 = address >> 3
    t2 = t1 ^ A
    t3 = (t2 << 15) | (t2 >> 17)
    t4 = t3 ^ B
    t5 = t4 >> 1
    return t5
}
```

Addresses differ in a bit iff hashcodes differ in a bit.

Cousin number can be computed !

Optimising for IBM JVM

- Allocate a large number of objects of type B.
- Compute cousin number of each object.
- Choose an object with maximum cousin number. Let address be x.
- Deallocate the object.   Address x is added to the front of the free list by the mark-and-sweep garbage collector. (Object size being 1024 is useful)
- Invoke System.GC().
- Allocate object of type A. Address x is reused and hence this new object has the max cousin number. Base the attack on this object.

IBM JVM : Defeating address obfuscation

hashCode

```
A = 2 * sp + clock()
B = 2 * sp + time() - 70
int hashCode (address) {
    t1 = address >> 3
    t2 = t1 ^ A
    t3 = (t2 << 15) | (t2 >> 17)
    t4 = t3 ^ B
    t5 = t4 >> 1
    return t5
}
```

- sp is predictable(#frames, argv, envp)
- clock depends on the time to initialise the JVM. Empirically, it is among 9 … 19
- time can be estimated by System.currentTimeMillis()
- XOR is reversible
- Given hashCode, address can be restricted to a small set

Conclusion

- Allowing attacker to choose the program alters many assumptions
- Attack possible as machine does not execute instructions correctly
- Especially the following are vulnerable:
  - Conventional Java virtual machines
  - Conventional .NET virtual machines
  - In fact, any system relying on static (type) checking to enforce security
- Always use ECC hardware with error logging
  - Shut down if too many errors
- Having a good heat sink is important!

Acknowledgments

Brent Waters, Crispin Cowan, David Fisch, Ed Felten, Eugene Normand, Gang Tan, Jim Roberts, Kartik Prasanna, Lujo Bauer, Michael Schuette, Peter Creath, Perry Cook, Tom van Vleck, Yefim Shuf.

Check out the paper and my commentary, available at my website, regarding the discussion of this work in slashdot.org!