Using Memory Errors to Attack a Virtual Machine

Sudhakar Govindavajhala & Andrew W. Appel
at Princeton University
IEEE S&P in 2003
review by Jerome Brechbühl

About the Paper

The paper addresses a **security breach in type checking systems**:

It shows how to **exploit memory errors** (unwanted bitflips in memory) by writing an **attack program** to take over the system

Besides the **analysis** and **successful practical tests** of the program, it also provides **methods to inject memory errors**

At last it **states what mechanisms** one could apply to guard against similar attacks

Content

- Background
- Attack Program
  - Overview
  - Pointer as Security Breach
  - Attack Program
  - Enforcing Memory Errors
- Performance
- Protective Measures
- Strengths & Weaknesses
- Discussion & Questions

Background

Security & programs

When loading an **untrusted program** into your system (and memory) you want to perform a **security check**. Two common mechanisms are:

Hardware **virtual memory** managed by operating system

or

as used in the Java Virtual Machine (and in the similar Microsoft .NET virtual machine), is **type checking**, done by a bytecode verifier

Background

Type checking

Each **object** is assigned a **type. Operations** on object have to be **compatible** with its **type “well typed”** (fmfp typing)

Assign **security classes** to objects and **ensure safety** (non-interference) by checking the **information flow** (variable with lower security class cannot influence variable with higher security class)

The **type checking** model is proven to be **sound** however does **not consider memory errors** (2003)

Background

Memory

Memory errors (unwanted bitflips in memory) are **known** to occur

Protection given by **parity** bits (check modulo values) & **ECC**’s (Error Correcting Codes)

However these **protective mechanisms** cause a memory overhead that makes machines less competitive therefore **tend to be left out**

Content

- Background
- Attack Program
    - Overview
    - Pointer as Security Breach
    - Attack Program
    - Enforcing Memory Errors
- Analysis & Performance
- Protective Measures
- Strengths & Weaknesses
- Discussion & Questions

Thread Model

Attack a **virtual machine** that uses **type checking** as its protective mechanism

Basic knowledge about the machine (memory size, etc…)

Granted **physical access** to **hardware**

Load the **program into memory** with **security check** and **run** it

No control over **data memory** of **program**

Attack Program

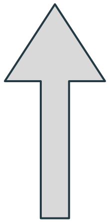
Overview

Take over system that uses **type-checking** as basic **protection mechanism**

by **injecting memory errors**

on the **attack program**

to **circumvent** the **type system** with aid of pointers



Attack Program

Overview

Take over system that uses **type-checking** as basic **protection mechanism**

by **injecting memory errors**

on the **attack program**

to **circumvent** the **type system** with aid of pointers

Pointer & Security breach

Assume we have two pointer of **different type** that **point to the same location**:

type A pointer **p** & **type B** pointer **q**

```
A p;
B q;   //p & q point to same location (Obj A) due to memory error
int offset = 6 * 4;  //offset of the integer field in A
void write(int address, int value) {   //write value at location address
        p.i = address - offset;      // write address into integer field
        q.a6.i = value ;             //overwrite the target
}
```

Classes defined as:

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

Types & Pointers

This method can write at **arbitrary locations** and take over the system

By writing **machine code** and overwriting a **virtual method table** with address to the machine code.

or

By overwriting the **security manager**

Attack Program

Overview

Take over system that uses **type-checking** as basic **protection mechanism**

by **injecting memory errors**

on the **attack program**

to **circumvent** the **type system** with aid of pointers

Attack Program

Overview

Take over system that uses **type-checking** as basic **protection mechanism**

by **injecting memory errors**

on the **attack program**

to **circumvent** the **type system** with aid of pointers

Attack Program

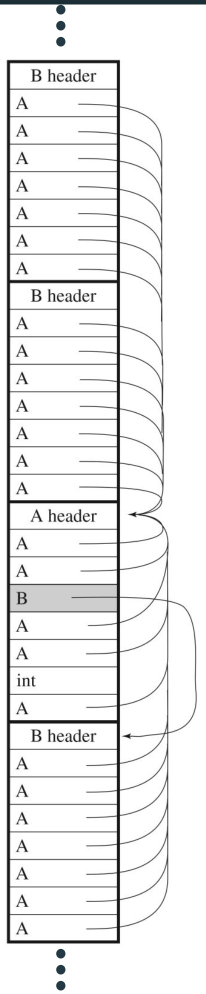
The attack program is built like this:

One object A, many objects B

All A's inside all objects point to only A object

B inside the A object points to an arbitrary B

Now a memory error occurs!

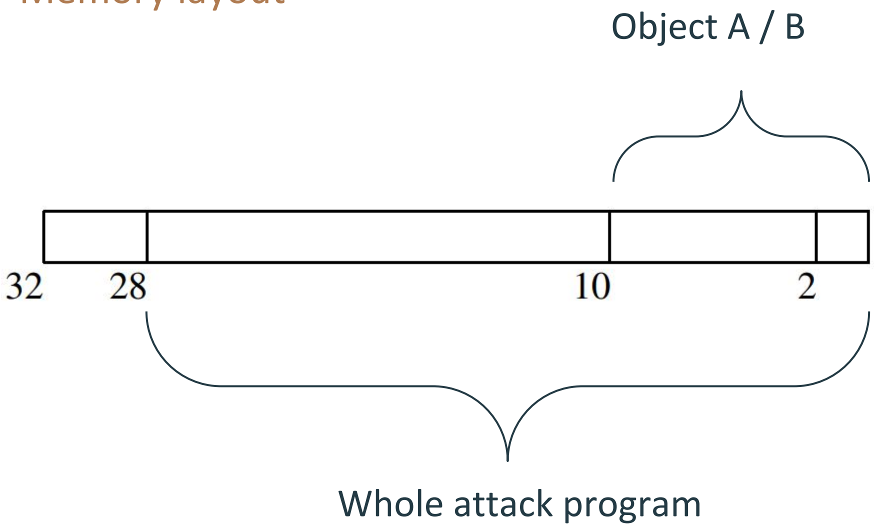



```
class A {
A a1;
A a2;
B b;
A a4;
A a5;
int i;
A a7;
 };
```

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

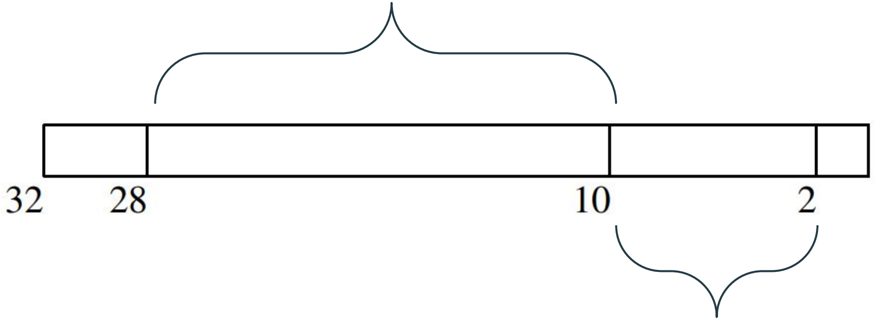
Attack Program

Memory layout

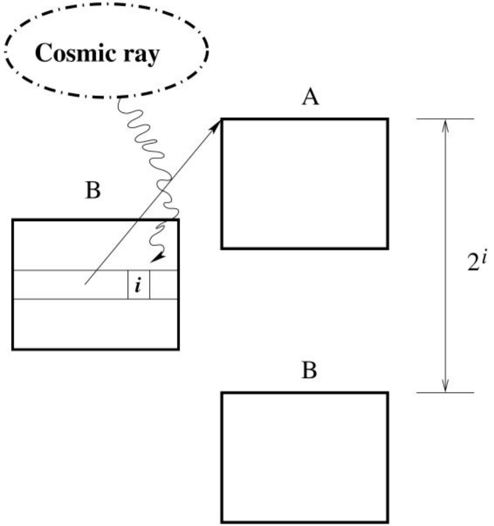


Attack Program

first case: bitflip
“outside” of object

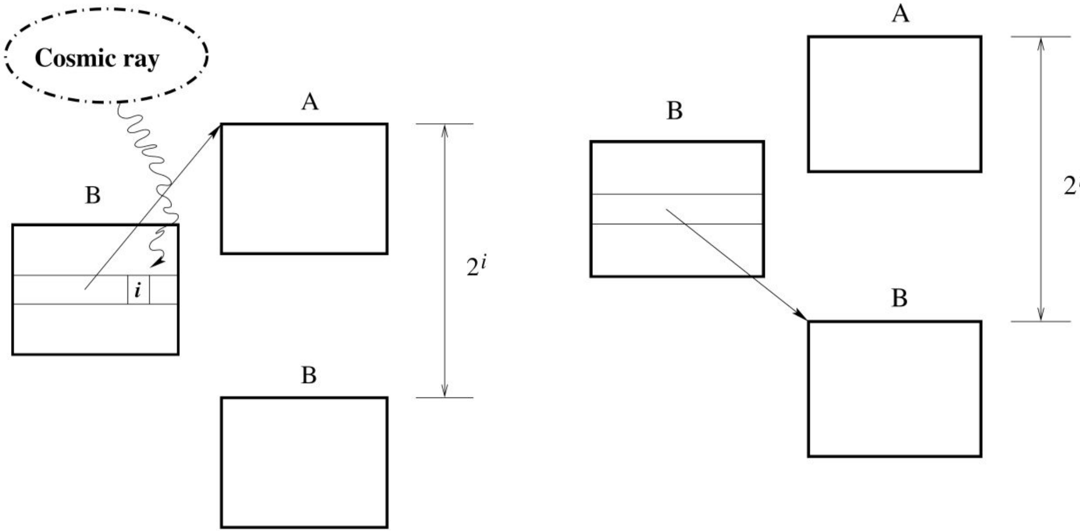


second case: bitflip
“inside” of object

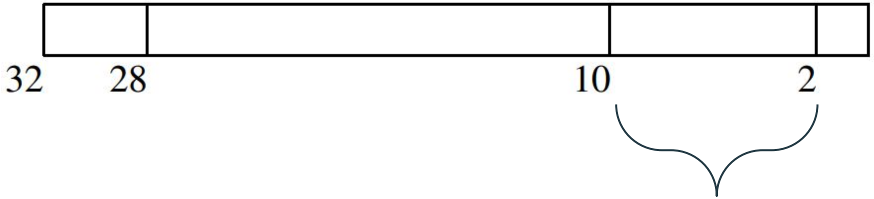


Attack Program

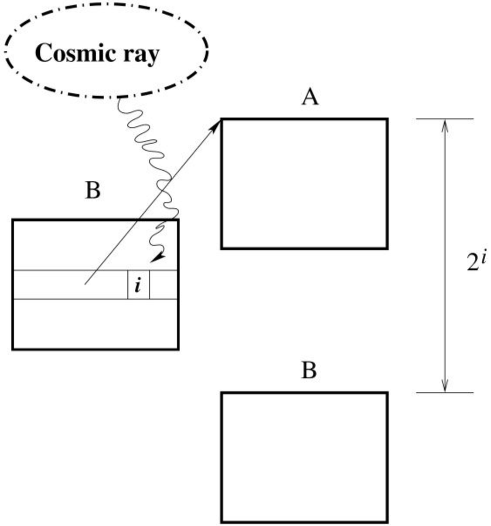
Instead of pointing to A a pointer inside the object B with a **flipped bit** (first case) **points** to an **object B**



Attack Program

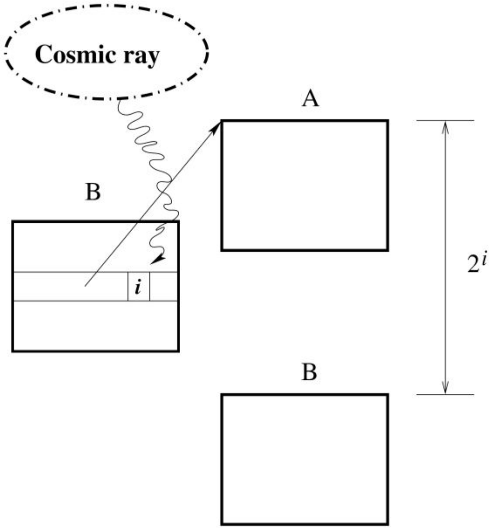


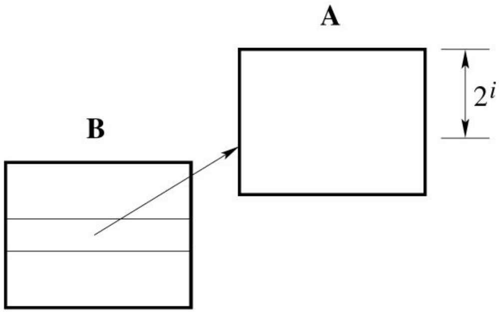
second case: bitflip
“inside” of object



Attack Program

Instead of pointing to A a pointer inside the object B with a **flipped bit** (second case) **points** to an **offset of A** (most likely type B)





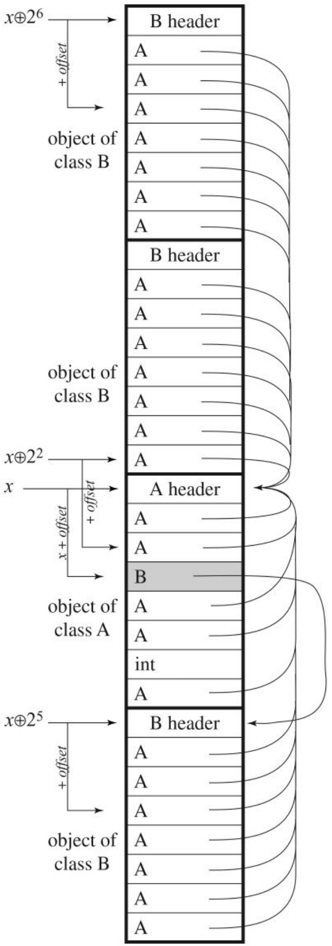
Attack Program

Create **pointer q** to B field of A

q has static **type B**

With bit flip **q** now points to Object **type A**

Take over system as explained



x: Address of the A object

⊕: Xor/flipped Bit location

offset: offset of the B field in the A object

Attack Program

Overview

Take over system that uses **type-checking** as basic **protection mechanism**

by **injecting memory errors**

on the **attack program**

to **circumvent** the **type system** with aid of pointers

Memory Errors

In general

Hard memory errors:

Permanent damage of the DRAM caused by **defects** in Silicon or metalisation

Soft memory errors:

“Natural” errors caused by **radiation**, **charged particles** or by **moving data**. Rewriting solves the error issue

For the attack to rely on natural memory errors it requires a **lot of space or time**

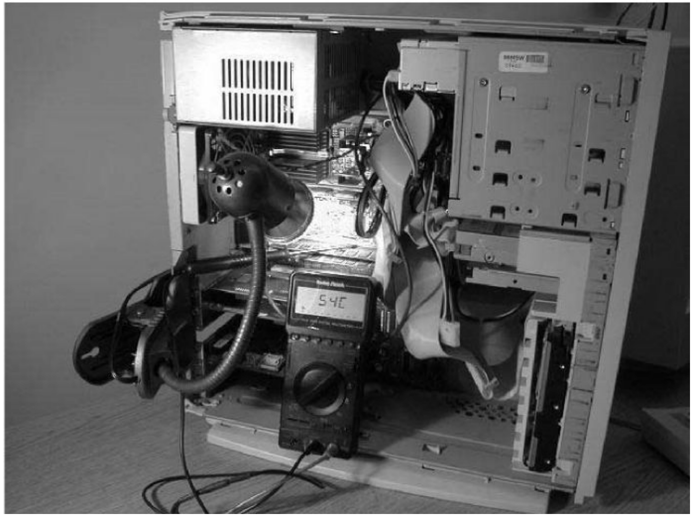
Enforcing Memory Errors

In the paper are **multiple ways** presented to **enforce memory errors**. The most **successful** ones are:

High-energy protons/neutrons

Infrared/heat

First option is very inaccessible therefore **heat** was the choice for the experiments in the paper.

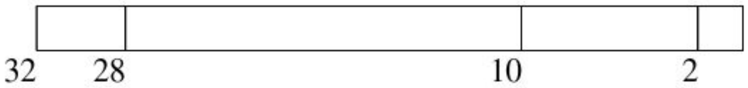


Content

Analysis

What errors can we exploit?

To keep it **simple** (for a 32-bit pointer value):



Bits **0-1, 28-31** bring the **risk** of OS crashing while garbage collecting or dereferencing

Bits **2-27 safely exploitable**

Overall for this example we can exploit 26 single bit errors and the success probability can be pushed to over 90%

Analysis

Estimate the **efficiency** by fraction of single bit errors that allow the attack so succeed. The exploitable number of bits in physical memory is given by:

$$\frac{N(s-h)(\log_2(Ns))}{8P}$$

N number of objects,

P bytes of physical memory on the computer,

s is the number of words in an object,

h is the number of words occupied by the header of each object

Performance

Practical tests

Attacks against **two JVM**’s both running on RedHat Linux 7.3:

IBM’s Java 2 Runtime Environment, Standard Edition (build 1.3.1)

Sun’s Java 2 Runtime Environment, Standard Edition (build 1.3.1_02-b02)

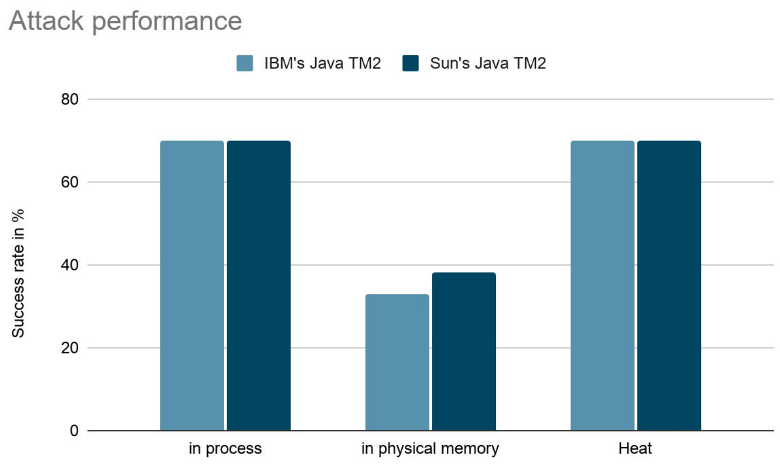
Performance

For each machine test in 3 different ways:

Software injected **in process** fault

Software injected **in physical memory** fault

Using heat to induce errors



Content

- Background
- Attack Program
  - Overview
  - Pointer as Security Breach
  - Attack Program
  - Enforcing Memory Errors
- Analysis & Performance
- Protective Measures
- Strengths & Weaknesses
- Discussion & Questions

Protective Measures

ECC (Error Correcting Code): 1, 2 bit errors: 72 bits required to represent 64 bit word, good but **not complete protection** (memory capacity overhead of 12.5%)

Error logging to detect patterns to diagnose a problem fast

Content

- Background
- Attack Program
  - Overview
  - Pointer as Security Breach
  - Attack Program
  - Enforcing Memory Errors
- Analysis & Performance
- Protective Measures
- Strengths & Weaknesses
- Discussion & Questions

Strengths

The paper **successfully performs attacks** on VM's and supports them with the **according analysis** and **tests**. It exposes a **practical problem** and also its **source. Gives simple solutions** to prevent further attacks.

Often **cited paper** as it is a **good example** for memory errors and exploits. It **builds** a solid **foundation** for further research.

Motivates research to **improve** type checking safety constraints with **Control Flow Integrity** that guarantees security even against adversaries that have control over the data memory of the executing program. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.440.1407&rep=rep1&type=pdf)

Motivates research for **security** against **similar fault injecting hard- and software attacks** e.g. by light detectors, active shields, Execution randomizers, Execution redundancy. (https://www.hbarel.com/media/blogs/hagai-on-security/Sorcerers_Apprentice_Guide.pdf)

Weaknesses

A somewhat **dreamworld performance test** that does not reflect reality. Physical access for attacker, 60% of memory used and going undetected. Very **limiting attack conditions** for this very **general error concept**

Protective measures not satisfying

Memory encryption is not considered, or seen as irrelevant. **Errors** in the **encrypted word** also result in **errors** in the **decrypted word**.

As research progresses this paper **tends to be overshadowed** by **more recent papers** that cover a **similar subject in more detail** or with **advanced attacks**

Although the paper is clear and understandable it is **confusing when first reading it** as some information is scattered and repeated

Where are we today?

Ongoing research about errors and exploits. With every new generation of hardware and software new exploits can be found. Exploits get more and **more creative** as the latest attacks use software to control **voltage and frequency** of underlying hardware to **inject errors**:

Plundervolt (https://plundervolt.com/doc/plundervolt.pdf)(2020)

CLKscrew (https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-tang.pdf)(2017)

Thoughts & Ideas

Copy nature to find a protective measure for memory errors

Evolution optimizes processes in **nature** so all we need to do is finding a synonym for memory errors

Cancer cells provide **similar properties**. They scale with size, can be caused by radiation and can ultimately lead to death.

In fact we see organisms using **cells** to **check** the cells condition and destroy it if an “error” is detected. Does this represent a natural ECC? So are ECC’s the optimal solution to memory errors? (Discussion)

Questions & Discussion

Discussion

Can you think of more “modern” ways of enforcing a memory error?

Hint: Row + Hammer

What might be the problem with that in the year 2003?

Discussion

For a paper about security do you think it’s enough to outline a theoretically possible attack?

Should we take protective measures against theoretical attacks?

Discussion

Why bothering with type checking when we can just use the virtual memory (memory safety) to separate code? Or so to speak why don’t we just use C++ over Java?

Can you think of a hybrid model?

Discussion

What do you think about the statement: “A memory error in a computer is the equivalent to a cancer cell in the human body”?

Finding solutions to soft and hardware problems in Nature?

Thank you!

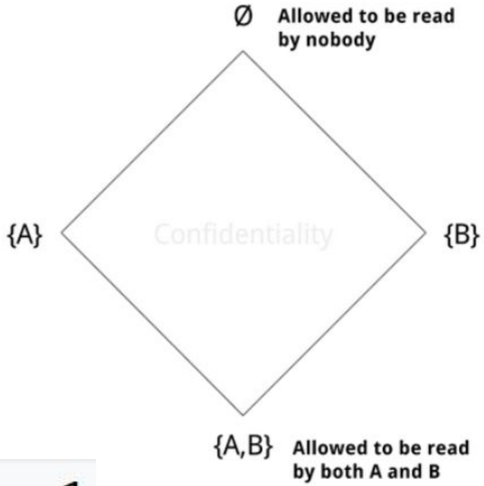
Backup Slides

Type checking - Information Flow

$SC = \{\emptyset, \{A\}, \{B\}, \{A,B\}\}$

$\rightarrow = \{(\{A\}, \{A\}), (\{B\}, \{B\}), (\{A,B\}, \{A,B\}), (\{A,B\}, \{A\}), (\{A,B\}, \{B\}), (\{A\}, \emptyset), (\{B\}, \emptyset), (\{A,B\}, \emptyset)\}$

$\text{if } y_{\{A\}} = 1 \text{ then } x_{\{A,B\}} := 0 \text{ else } x_{\{A,B\}} := 1$



source:https://en.wikipedia.org/wiki/Security_type_system