NANYANG TECHNOLOGICAL UNIVERSITY

Lecture 12: Type Safety

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

Motivation

- Taint analysis tools (and other security analysis tools) are applied after vulnerable code has been written
- Why not write bug-free code in the first place?
- Sentiment after *Smashing the stack for fun and profit*: blame yourself for writing code in C/C++; use a type safe language like Java and your problems go away
- This lecture will examine what is in the claim that type safe languages will guarantee secure software

Type Safety

- Guarantees the absence of untrapped errors
- The precise meaning of type safety thus depends on the definition of error
  - A language is unsafe if you are not told when an error has occurred during execution
- Luca Cardelli: practitioners who invented type safety often meant just "memory integrity", while theoreticians always meant "execution integrity", and it is the latter that seems more relevant now
- Languages do not have to be typed to be safe: LISP

Memory and Type Safety

Memory safety: each dereference can only access "intended target"

- Spatial access errors (e.g., out-of-bounds array access)
  - E.g., if `i` is negative or is too large
- Temporal access errors (e.g., stale pointer dereference)
  - E.g., if a had been freed prior to `*(p+i)`

```
p = &a        // p's valid target is a
*(p+i)        // valid only if it accesses a
```

Memory and Type Safety

- Type safety: operations on the object are always compatible with the object's type
  - Each object is ascribed a type (`int`, pointer to `int`, pointer to function)
  - Type safe programs do not "go wrong" at runtime
- Type safety is stronger than memory safety

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
// memory safe but not type safe
cmp = (int (*)(char*,char*))p;
cmp("hello","bye");        // crash!
```

Type Safe Languages

- Useful for improving quality of software
- Safety guaranteed by static checks and by runtime checks
- Language design: trade-off between efficiency and flexibility
  - If too general, incurs a high runtime overhead to enforce
  - If too restrictive, limits expressiveness and utility of language
- Examples: Java, Ada, C#, Rust, Go, Perl, Python, etc.
  - Dynamically typed languages, like Python, which do not require declarations that identify types, can be viewed as type safe as well
  - Each operation on a dynamic object is permitted, but may be unimplemented
  - In this case, it throws an exception (unfortunate, but still well-defined)

Type Safety – Limitations

- Typical enforcement of type safety is expensive
  - Garbage collection avoids temporal violations
  - Bounds and null-pointer checks avoid spatial violations
  - Hiding representation (abstract types) may inhibit optimization
    - Many C-style casts, pointer arithmetic, & operator, are not allowed
- Type system may not deal with all security problems
  - E.g., race conditions are possible in Java code
  - Erasure of sensitive data: consider a login program accepting a user password; how to make sure that you know where copies of the password are held and that all copies are erased once the login process has finished?

Execution Integrity

Execution Integrity

- Execution integrity is at the core of software security
- Separate data from control information; we have seen examples before
- System-environment integrity: certain aspects of the processor, e.g., condition flags, cannot be changed
- Memory access: module can only access (read/write/execute) certain memory regions; e.g., module cannot overwrite its own code
- Interface: calls from a module only to predefined routines, returns only at defined entry points

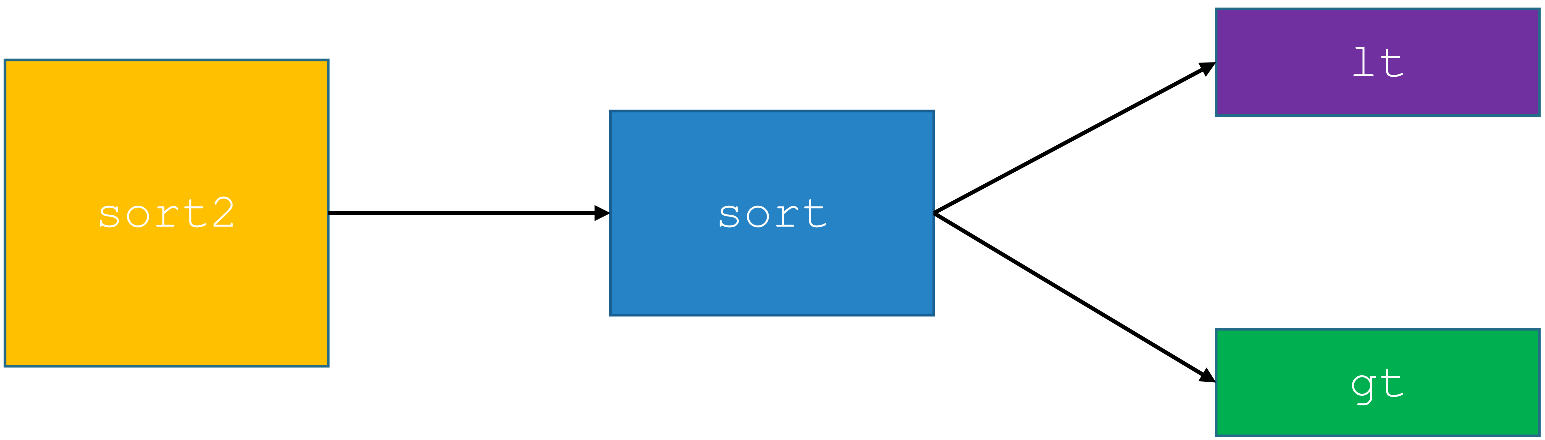
Control-Flow Integrity

- Control-flow integrity: execution must follow a control-flow graph (CFG) determined in advance
  - E.g., function must return to its caller
- CFG can be defined by analysis, by execution profiling, or by an explicit security policy, e.g., written as a security automaton
- Machine-code rewriting can be used to insert runtime checks that dynamically ensure that control flow adheres to a given CFG

Call Graph

```
sort2(int a[], int b[], int len){
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y){
  return x < y;
}
bool gt(int x, int y){
  return x > y;
}
```

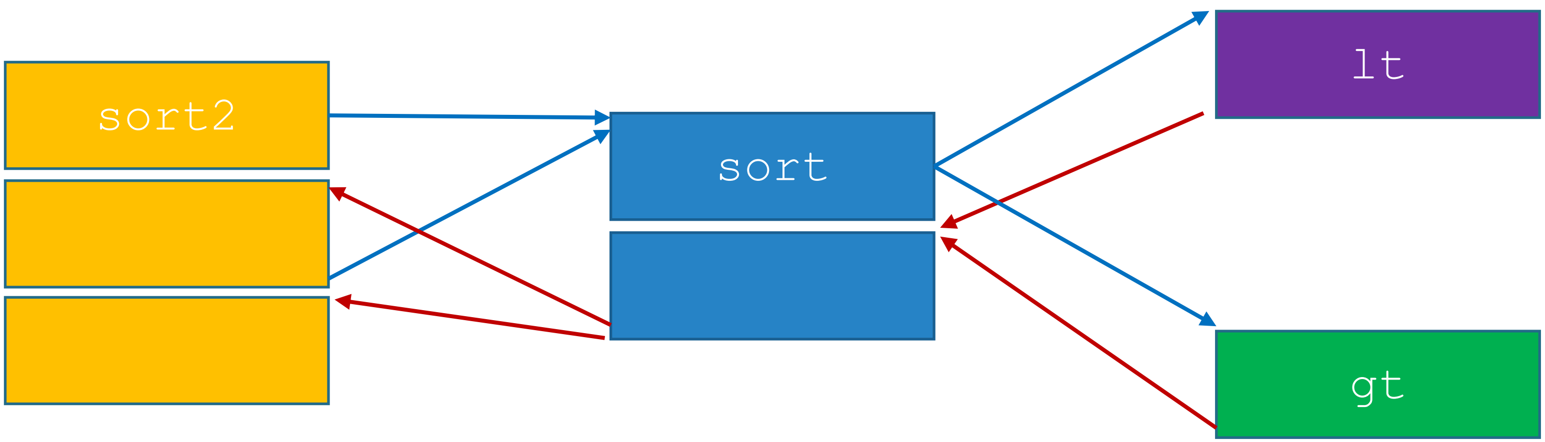


Which functions call other functions

Control Flow Graph

```
sort2(int a[], int b[], int len){
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y){
  return x < y;
}
bool gt(int x, int y){
  return x > y;
}
```



Break into basic blocks

Distinguish calls from return

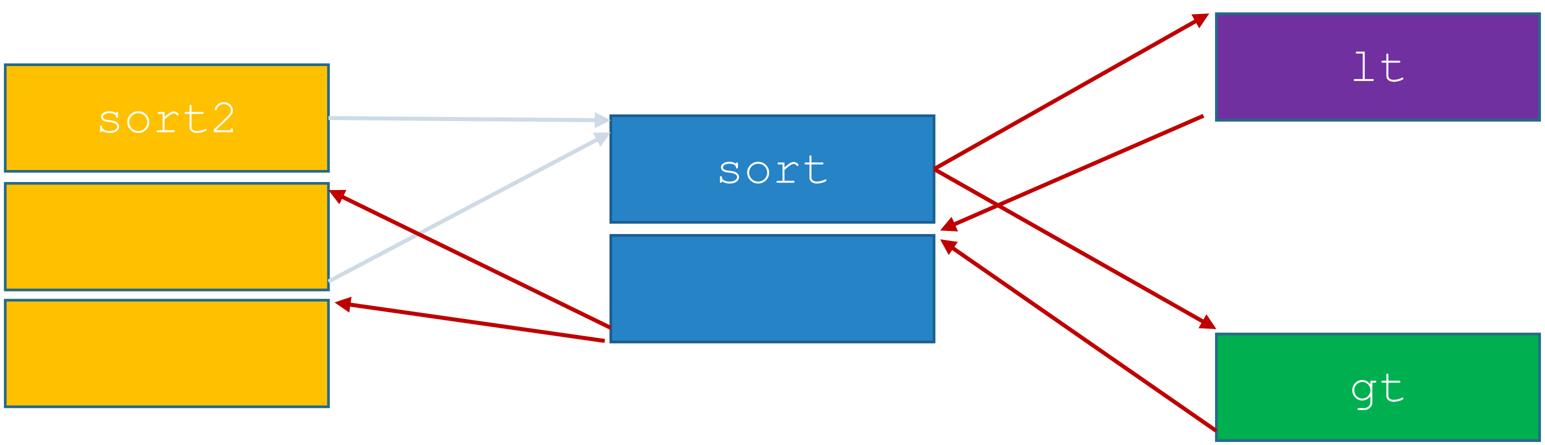
Control-Flow Integrity

- Observation: direct calls need not be monitored
  - Assuming the code is immutable, the target address cannot be changed
- Therefore, monitor only indirect calls
  - `jmp`, `call`, `ret` via registers
- Forward-edge CFI for indirect calls and jump instructions
- Backward-edge CFI for return instructions
  - Shadow stack for return instructions: separate isolated stack for storing return addresses
  - Enforces natural semantics of call/return: return is supposed to transfer the control flow to the next instruction after the call instruction that invoked the current function

Control Flow Graph

```
sort2(int a[], int b[], int len){
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y){
  return x < y;
}
bool gt(int x, int y){
  return x > y;
}
```

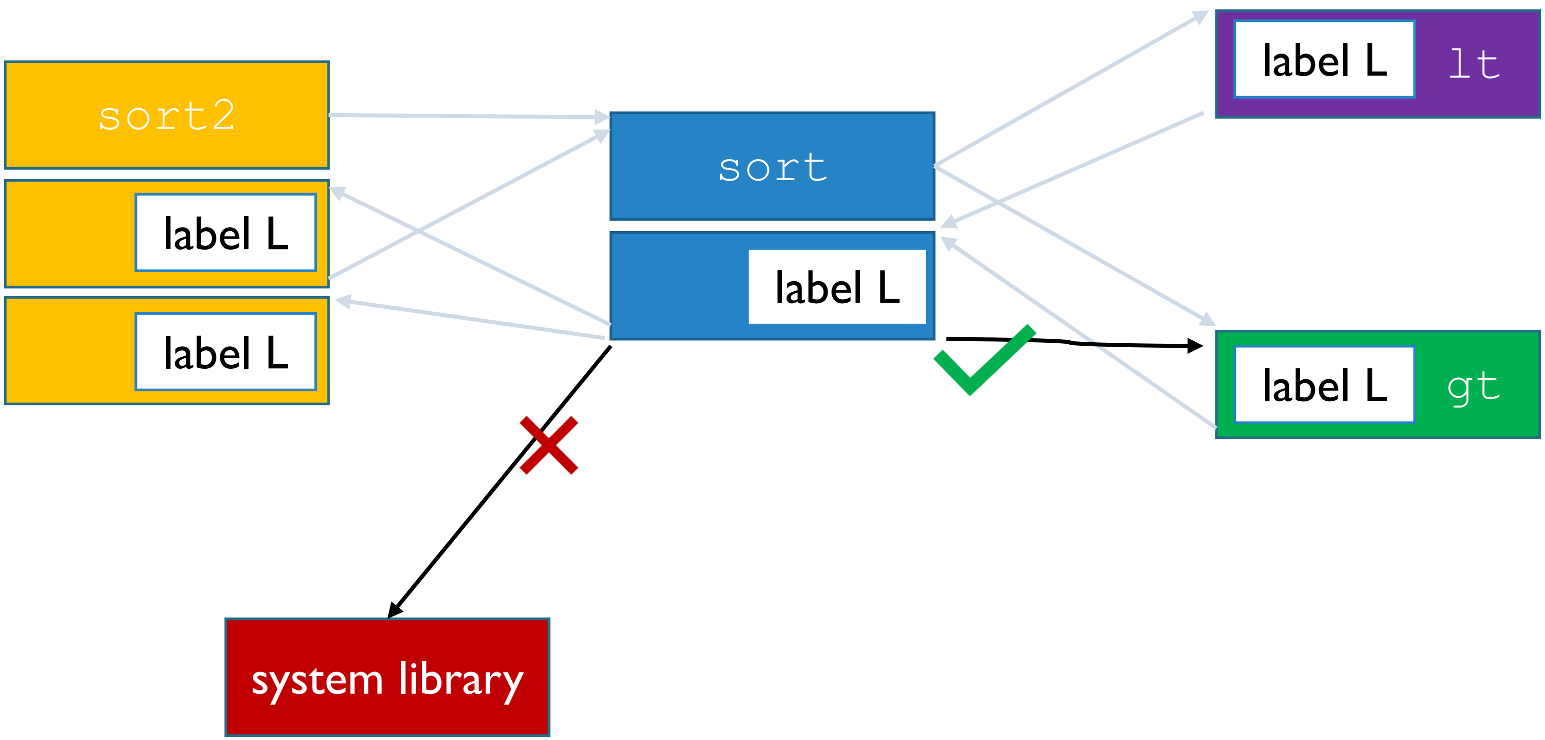


Indirect transfer (`call` via register, or `ret`)

Control-Flow Integrity

- During execution, each machine instruction that transfers control must target a valid destination, as determined by the given CFG
- For instructions that target a constant destination, this can be done by a static check
- Computed control-flow transfer needs dynamic check
- At each destination, insert an ID (label) identifying an equivalence class of destinations
  - Design challenge: how to define equivalence?
- Before each source, insert an ID-check that validates that the destination has the right ID

Control-Flow Integrity

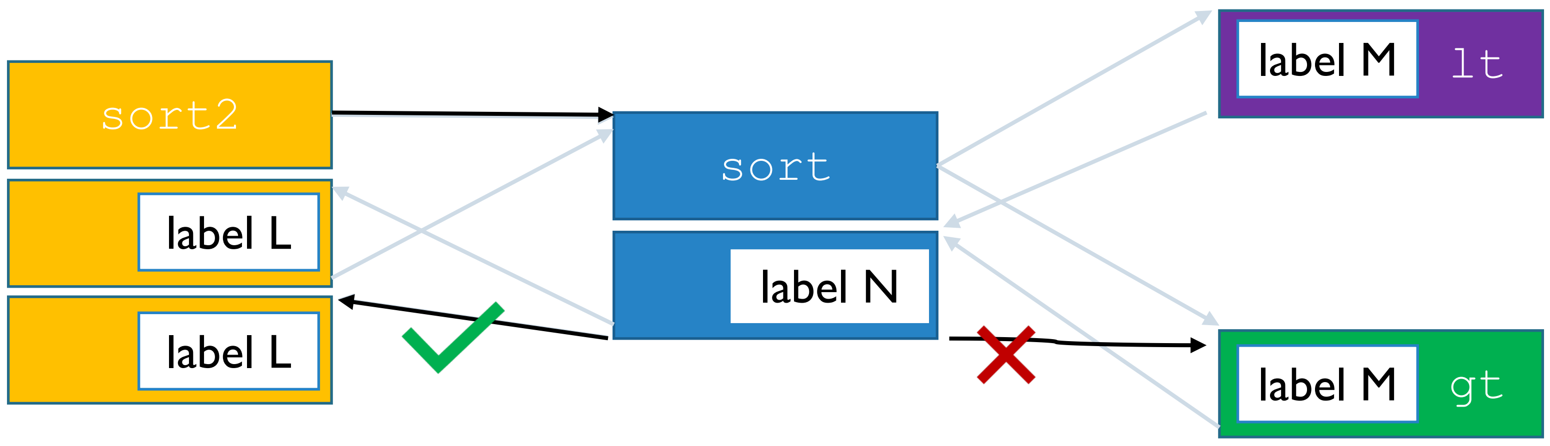


Equivalence class 1: use the same label at all targets

Control-Flow Integrity

- Design question: how many different IDs?
  - Unique IDs?
  - IDs chosen from a given set and reused?
- Attacker who does not know the IDs has to be lucky when deviating from the CFG to reach a destination with a valid ID
- Attacker who knows the IDs may try to construct an attack that avoids detection
  - The more IDs are reused, the more options for attacks
- CFI enforcement is not a panacea: exploits within the bounds of the allowed CFG are not prevented

Control-Flow Integrity



Equivalence class 2:

- Return sites from calls to `sort` must share a label (L)
- Call targets `gt` and `lt` must share a label (M)
- Remaining label unconstrained (N)

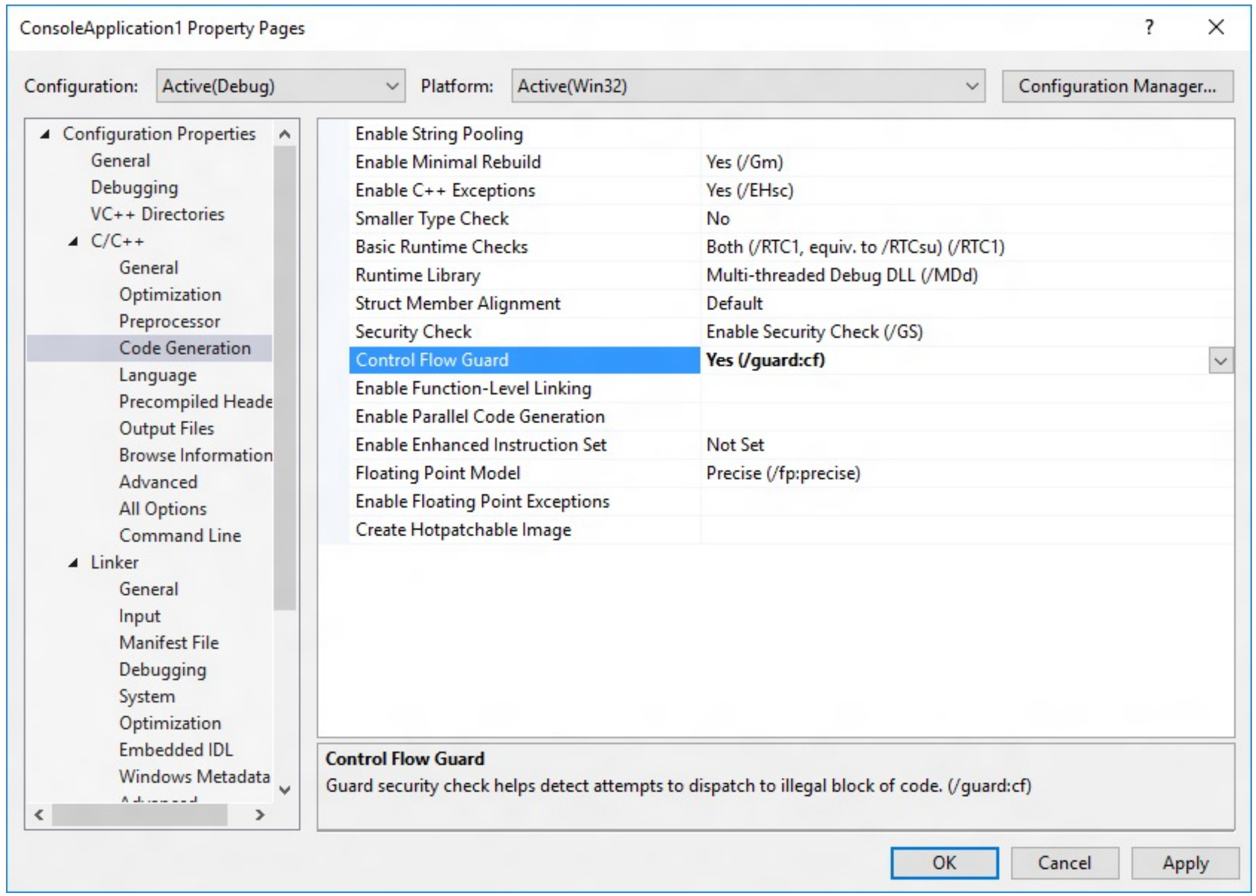
Still permits call from site A to return to site B

Backward-Edge CFI & Exceptions

- Note that there are benign cases in which the call/return semantic is broken
- E.g., if the currently executed function generates an exception, it need not be the calling function that catches the exception
- Another function in the call hierarchy or a default exception handler can be responsible
    - In such a case, the current function returns to the function in the call stack that implements the exception handler

Microsoft Control Flow Guard (CFG)

- CFG first released for Windows 8.1 Update 3 in 2014
- Compiler, operating system, user mode library, and kernel mode module are cooperating



Type Confusion

Escaping out of the type system

Type Confusion

- Static type checking: check all possible executions of a program to see whether a type violation could occur
- Dynamic type checking: check the class tag when access is requested
- Type confusion attack: exploit vulnerability in type checking procedure to create two pointers to the same object with incompatible type tags
- Attack code passes static type checking, but actual access is to object of the wrong type

Type Checking in Java

- Type safety (memory safety): programs cannot access memory in inappropriate ways
- Java: each object has a class; only certain operations are allowed to manipulate objects of that class
- Every object in memory is labelled with a class tag
- When a Java program has a reference to an object, it has internally a pointer to the memory address storing the object
- The pointer can be thought of as tagged with a type that says what kind of object the pointer is pointing to

Type Confusion

- Assume the attacker manages to let two pointers point to the same location

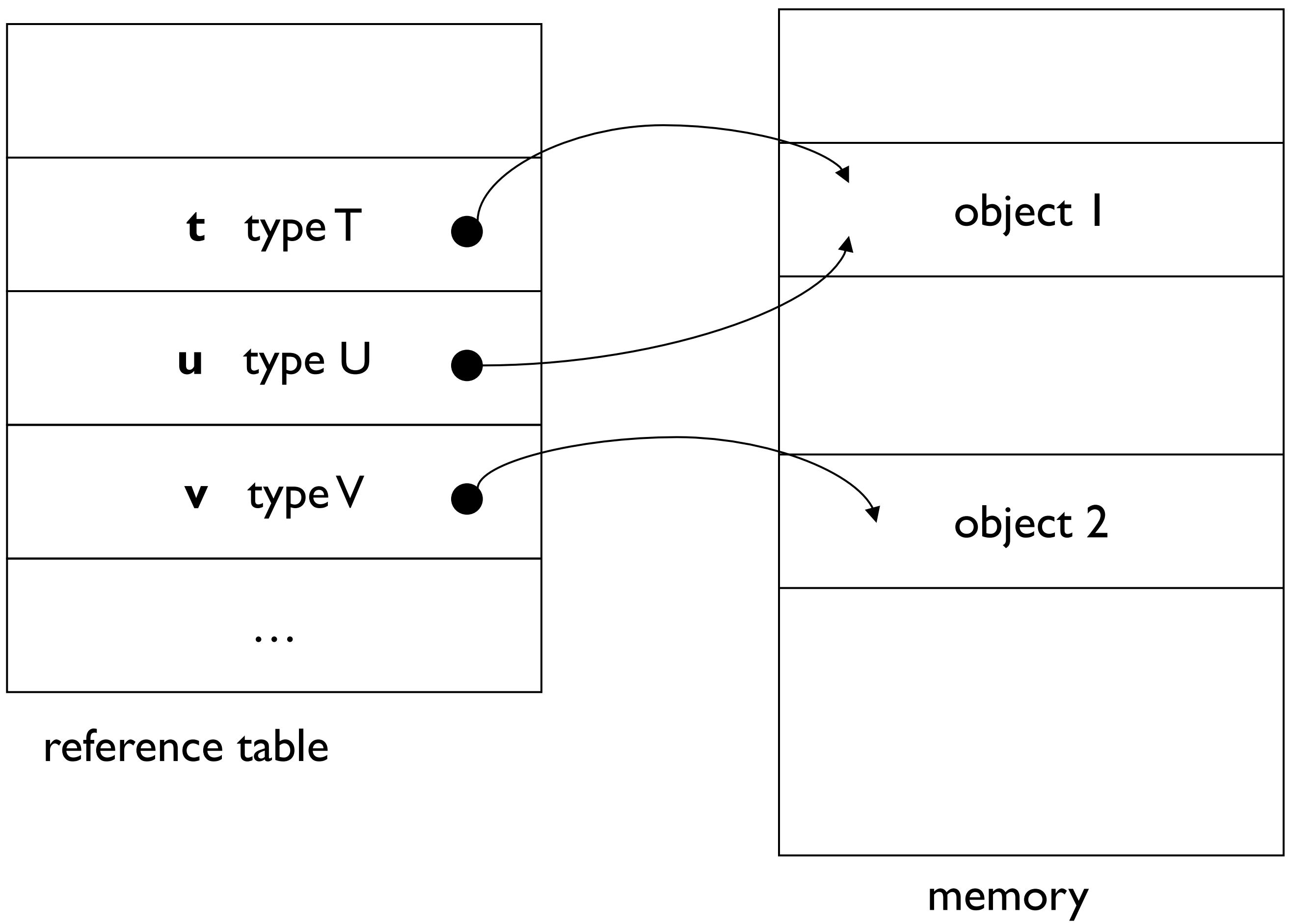
```
class T {
   SecurityManager x;
}

class U {
   MyObject x;
}
```

class definitions

```
T t = the pointer tagged T;
U u = the pointer tagged U;
t.x = System.getSecurity();
MyObject m = u.x;
```

malign applet

Type Confusion



Type Confusion

- The `SecurityManager` field can now also be manipulated from `MyObject`
- We sketch a type confusion attack in Netscape Navigator 3.0β5 (discovered by Drew Dean), fixed in version 3.0β6
- Source: Gary McGraw & Edward W. Felten: Java Security, John Wiley & Sons, 1997

Netscape Vulnerability

- Java allows a program that uses type `T` also to use type *array of* `T`
- Array types defined by the virtual machine for internal use; array names begin with the character `[`
  - Array of float: `[F`
  - Array of array of double: `[[D`
  - Array of objects: `[Ljava/lang/Object`
  - A programmer defined classname is not allowed to start with this character; hence, there should be no danger of conflict
- However, a Java bytecode file could declare its own name to be a special array types name, thus redefining one of Java's array types
  - Attempting to load such a class would generate an error, but the Java VM would install the name in its internal table anyway
  - Access to the layer below: from Java code down to bytecode

SUN Security Bulletin #00218

- Relates to the byte code verifier of the Java Runtime Environment (JRE) [March 18, 2002]
- Cause: flaw in the checking of type casts
- Applets could exploit this flaw to increase their privileges and get out of the sandbox
- If the user loads an applet designed to exploit this weakness, the attacker can execute arbitrary code on the local machine with the user's privileges

Summary

- Type confusion attacks are an example for attacks below the level that performs access control
- Type confusion attacks often result in complete system penetration
- It does not seem to be that easy to spot such vulnerabilities; relatively few are reported
- It is not easy either to prove that a system is type safe, or to achieve type safety first time round when a new system is being designed

Breaking Type Safety from Hardware

* this will not be tested *

Breaking Type-Safety with a Torch

- Type-confusion attack using random memory error
- Create a data structure where a random memory error is likely to change a reference so that it points to an object of the wrong type
- Then exploit the fact that pointers of different types point to the same object in memory

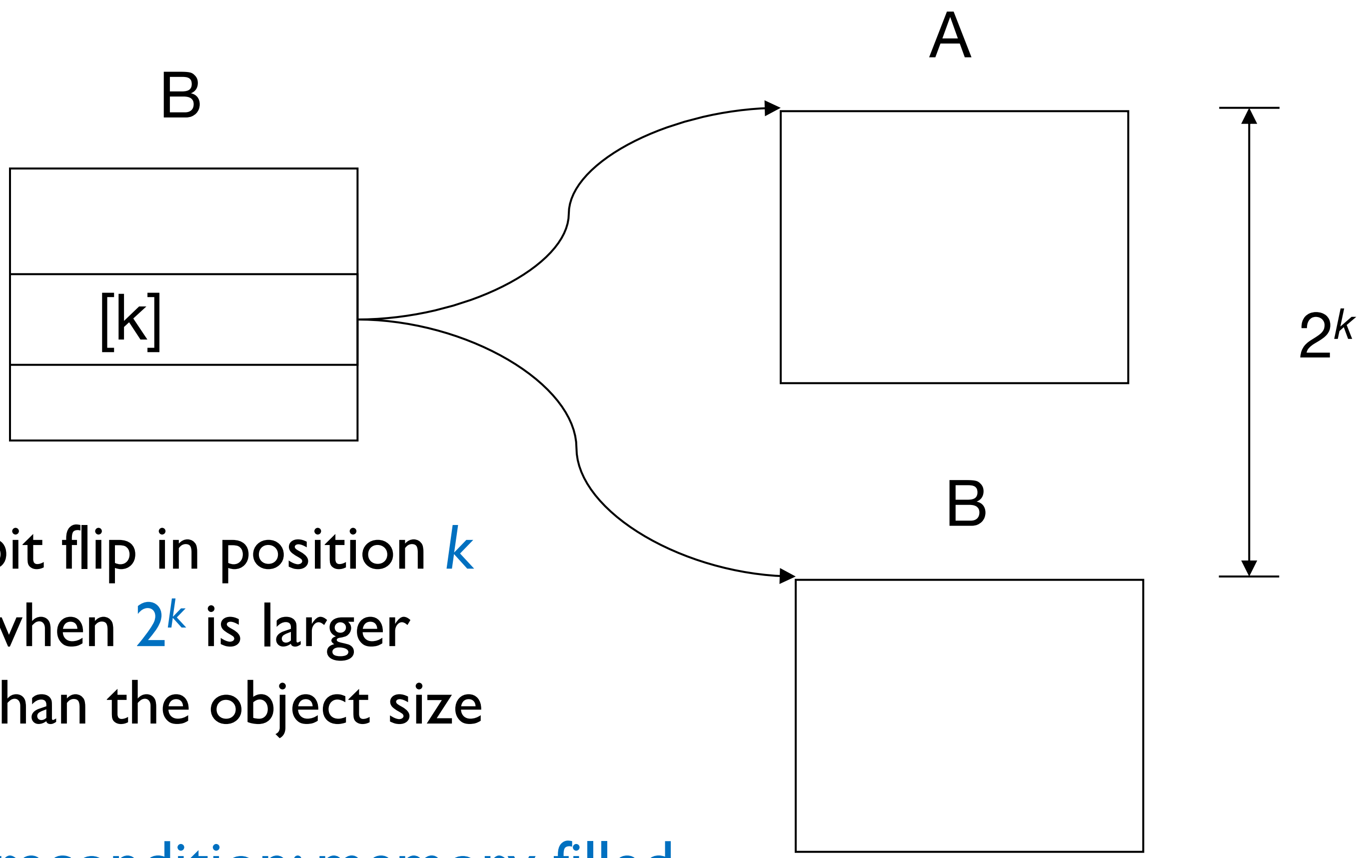
The Attack: Classes

```
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;
};
```

Size of the above classes A and B is a power of 2

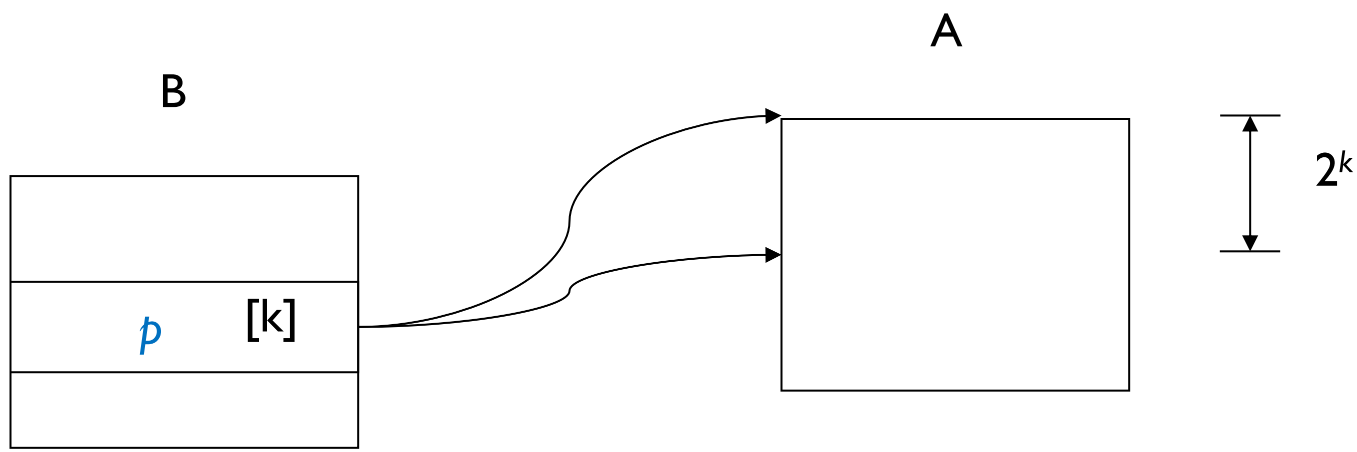
Attack: Flipping a Bit (Case 1)



bit flip in position k
when 2^k is larger
than the object size

Precondition: memory filled mainly with type B objects

Attack: Flipping a Bit (Case 2)



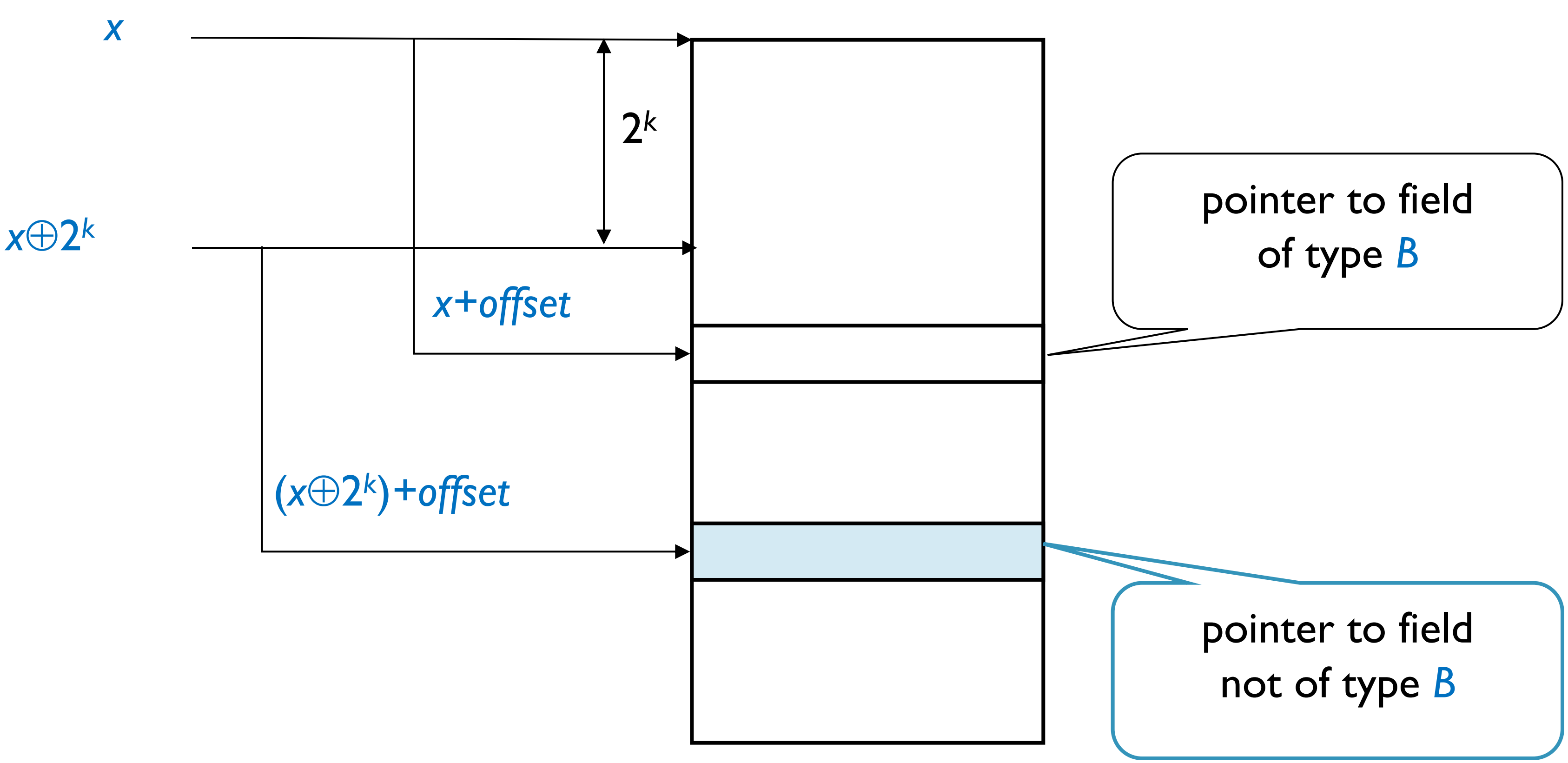
bit flip in position k when 2^k is smaller than the object size

p now likely to point to an object of type A, but start address of this object is misaligned

Misaligned Pointer References

- Assume a bit is flipped in a pointer variable *p* of class *A* at address *x*
- *offset:* distance between base of object and beginning of *b* field
- Dereferencing the *b* field of pointer *p* into a pointer *s* (of type *B*) fetches value of *s* from address $x + offset$
- When the *k*-th bit of *p* has flipped, the fetch would be from address $(x \oplus 2^k) + offset$
- Likely that *s* now points to an object of type *A*

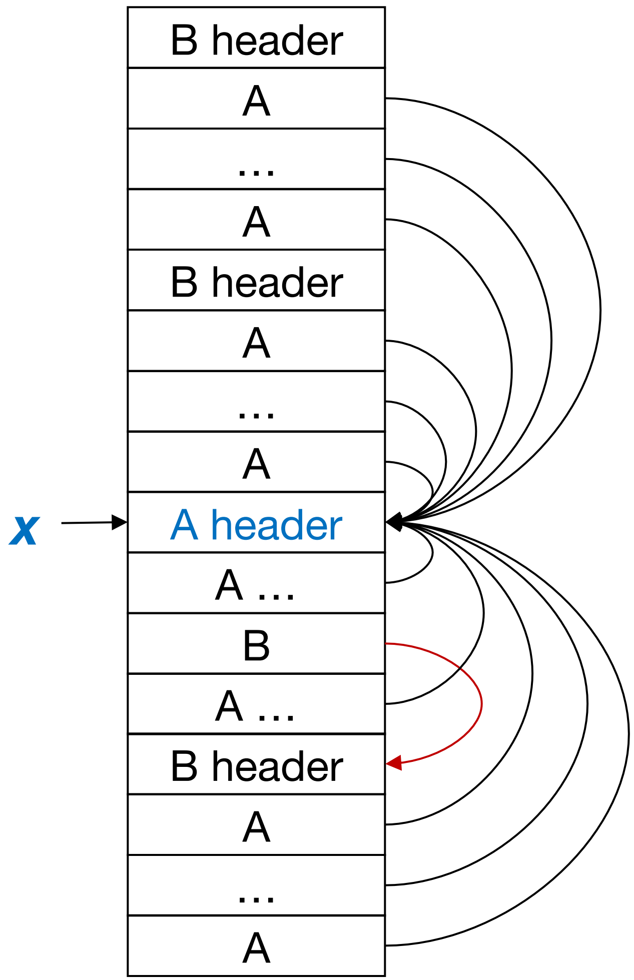
The Attack: Pointer References



Attack – Memory Layout

- Fill heap with lots of objects of type *B* and one object of type *A* at location *x*
- All *A* fields in the objects of type *B* point to this object (contain address *x*)
- The *B* field in the *A* object points to one of the objects of type *B*
- Induce – or wait for – a random bit error
- Keep checking the objects in memory until a position with a bit error has been found

Type-Confusion Attack: Data Structure



- Suppose a bit flips in a *B* object
- Original position contained an *A* object pointing to address *x*
- Modified pointer $x \oplus 2^k$ is likely to point to an *A* field
- When dereferencing its *b* field one is likely to hit a field of type *A*

Type Safe Attack Code

```
A p;
B q;
int offset = 6*4;
void write(int target,
  int value)
{
  p.i = target – offset;
  q.a6.i = value;
}
```

let *p* and *q* point to same memory location *x*

writes *target - offset* into field *q.a6*

writes *value* into location *target*

```
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;
};
```

Mounting the Attack

- How to induce a memory error?
- If you have physical access to the processor, shine a light bulb at it
- Demonstrated at the 2003 IEEE Symposium on Security & Privacy on IBM and Sun's JVM
  - Sudhakar Govindavajhala and Andrew W. Appel: Using Memory Errors to Attack a Virtual Machine, Proceedings of the 2003 IEEE Symposium on Security and Privacy
- Can overwrite arbitrary memory locations (as in the double free vulnerability)

Discussion

- Assessment of attack
  - To use a light bulb, you need physical access
  - You need to be able to fill a large potion of memory to increase the chance that a random memory flip has the desired effect
  - Summary: nice demo, but not particularly serious
- Countermeasures
  - Error-correcting memory would repair the bit flip
  - Error-detecting memory would flag the bit flip
  - Cost: a few bits of each memory word

Rowhammer

* this will not be tested *

DRAM – Coupling Effects

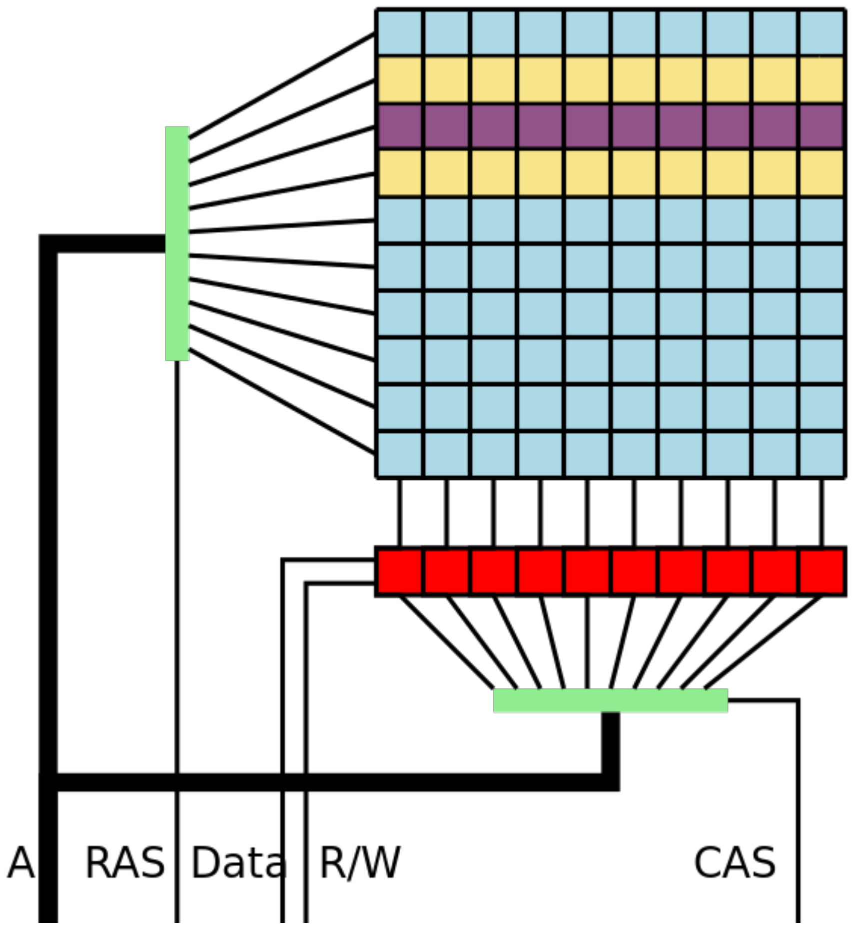
- Can we do better than relying on random bit flips
- Can we launch attacks remotely?
- DRAM – Dynamic Access Memory
- Electrical couplings in DRAM is not new
  - May cause disturbances during operation
  - Major hurdle in DRAM scaling
- Attack vector at physical layer: change in one memory cell can flip neighbouring bits

Rowhammer Attack

- It depends on the given DRAM chip which bits in which rows can be flipped by hammering a neighbour
  - These effects are repeatable!
- Arrange memory so that
  - Vulnerable row (victim) holds a data structure where a bit flip in the vulnerable position is of use to the attacker
  - Attacker has write access to a row (aggressor) next to victim
- Write to aggressor row repeatedly ("hammering")
- Induce an error in the victim row; the change in the data structure creates an exploitable vulnerability

https://www.extremetech.com/electronics/211655-rowhammer-a-new-javascript-attack-that-targets-dram

Rowhammer Attack



- Rowhammer: purple row hammers on yellow rows
- Double-sided rowhammer: yellow rows hammer on purple row
  - More effective at physical level
  - More of a challenge to get write access to the two rows next to your target

Rowhammer – Defences

- Change layout of DRAM
  - Addresses the issue at the level of physical structures
- Refresh memory cells more frequently
  - Addresses the issue at the "physical operational" level; increases power consumption
- Apply error correcting codes to memory words
  - Adds a defence in memory at a logical level
- Access counters to detect hammering
- After closing a row, open neighbouring rows with low probability (randomize allocation of memory rows)
  - Addresses the issue at a "logical operational" level

Drammer

- Rowhammer attack on Android devices
- Method: modify page table entries (PTEs)
  - PTEs map virtual memory to physical memory
  - Layer of indirection!
- Preparation (easily said, more difficult to do):
  - Find out which memory rows are susceptible to hammering on the given device
  - Arrange memory in a way that a PTE for a file where you have R/W access occupies a vulnerable memory row
  - Fill physical memory with PTEs for files with R/W access

Drammer – Execute Attack

- Hammer the aggressor row to induce a bit flip in victim row so that PTE points to another PTE in its own page table
- You thus get write access to a PTE and can scan kernel memory by writing new PTEs
- Once you find your credentials data structure you can overwrite it with zeros and become root

Drammer – Ressources

- https://www.vusec.net/projects/drammer/
- http://thehackernews.com/2016/10/root-android-phone-exploit.html
- https://vvdveen.com/publications/drammer.pdf
- https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf

Conclusion

Summary

- Type safety helps programmers avoiding certain classes of vulnerabilities
  - Programmers still have to catch and handle exceptions
- Memory integrity and control-flow integrity have raised the bar for attackers over the past 15 years
- Access to the layer below can undermine type safety
  - Direct manipulation of Java bytecode
  - Direct manipulation of memory
- You then need further defences at the layer below to preserve the security guarantees from a higher layer