Imperial College London

Safe C Compilers

Cristian Cadar
c.cadar@imperial.ac.uk

Software Reliability Course
Autumn 2016

Imperial College London

Defending against buffer overflows

- The root cause of many critical bugs and security attacks are buffer overflows!
- Can we prevent them from happening in the first place?

- First line of defence: off-line program analysis tools
  - static analysis, symbolic execution, model checking, etc.

- Can we prevent them at run-time?
  - sure, use memory-safe languages like Java!
  - but they are not a good fit for systems programming and performance can be an issue too
  - significant part of our computing base (operating systems, network servers, compilers, office utilities, etc.) is written in C/C++

TIOBE index: 2001-present

The C Programming Language

Some information about C:

Highest Position (since 2001): #1 in Mar 2015

Lowest Position (since 2001): #2 in Nov 2016

Language of the Year: 2008

Most network-facing security-critical code written in C!

TIOBE Index for C

Source: www.tiobe.com

Can we prevent buffer overflows in C at runtime?

- Safe C compilers
  - Instrument the program with dynamic checks to detect illegal memory accesses
  - When a buffer overflow is detected, program is terminated
- First attempts: *fat pointers* BCC [Usenix 1983], RTCC [SP&E 1992]
  - Disadvatages?

Fat pointers: disadvantages

1) Increases runtime performance
   - Note that pointers don’t fit into a single register anymore
2) Increases memory consumption
3) Breaks assumption about pointer size
4) Loses checks when converted to integers and back

```
void *p = &main;
long hash = (long) p | 3;
void *q = (void*)(hash & ~3);
```

Imperial College London

Fat pointers: interacting with uninstrumented code

5) Biggest showstopper in practice

- Would need to always link against instrumented libraries, so everyone would need to adopt this at once
- Would also need to worry about the interaction with OS and hardware (system calls, DMA controller, etc.)

Fat pointers: interacting with uninstrumented code


```
// instrumented
char* foo(char* p) {
   ...
}

// uninstrumented
char *a, *b;
a = foo(b);
```

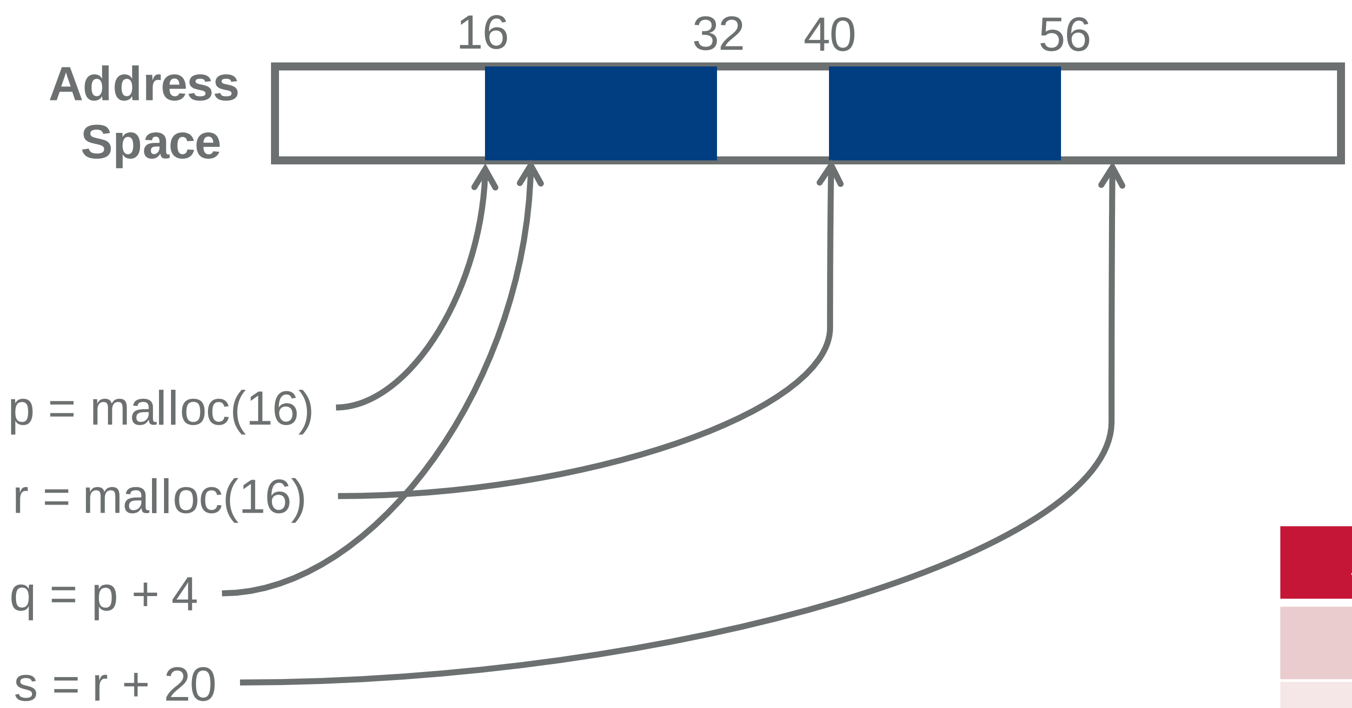
```
// uninstrumented
int bar(int *p, int* q) {
   ...
}

// instrumented
int *a, *b, r;
r = bar(a, b);
```

Backwards-Compatible Safe C compilers

- Introduced by Imperial's Jones and Kelly
- Does not change the pointer representation
  - Fully compatible with uninstrumented code

Backward-compatible bounds checking for arrays and pointers in C programs.
Richard Jones and Paul Kelly, International Workshop on Automated and Algorithmic Debugging (AADEBUG 1997)

Jones and Kelly compiler



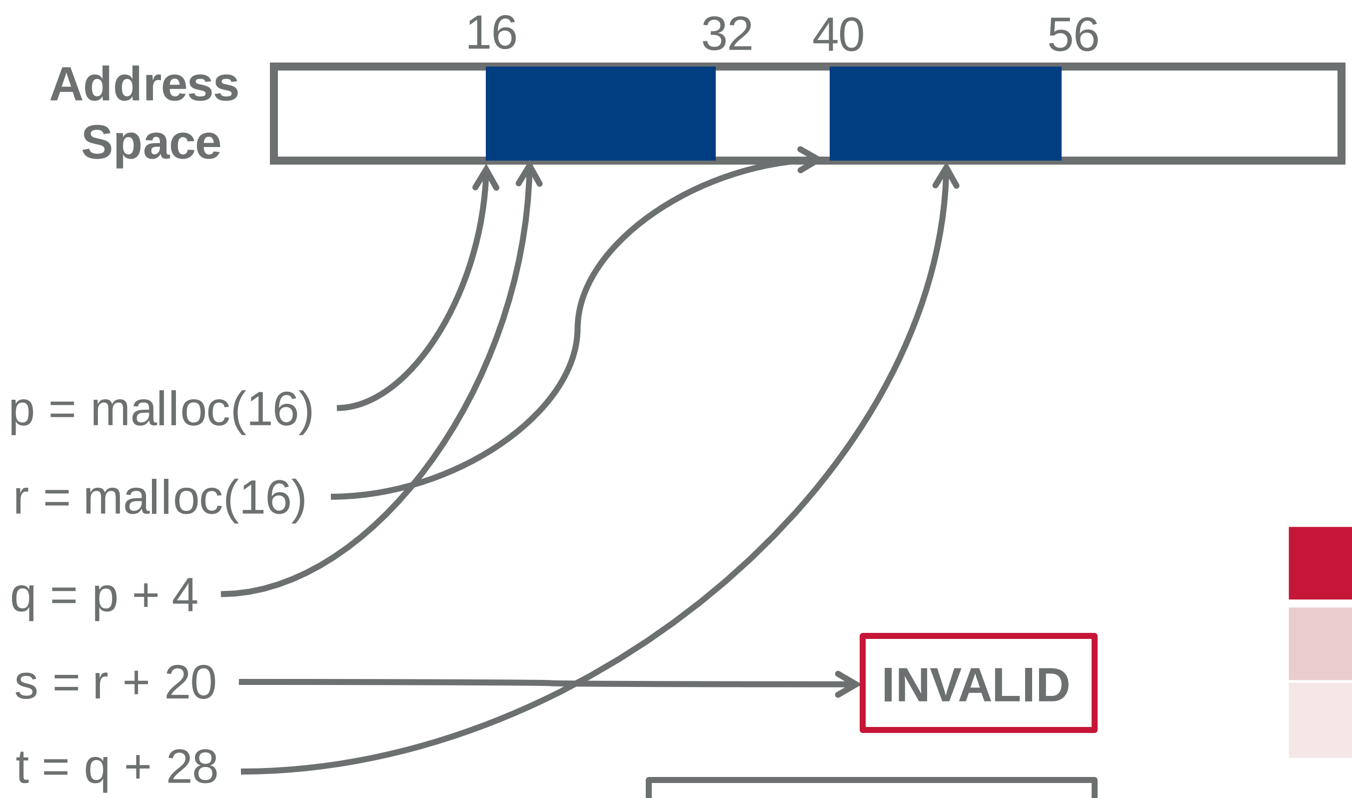
Object table

Start	End
16	32
40	56

Basic rule to check

The result of pointer arithmetic must preserve the referent object, and not overflow it!

Jones and Kelly compiler



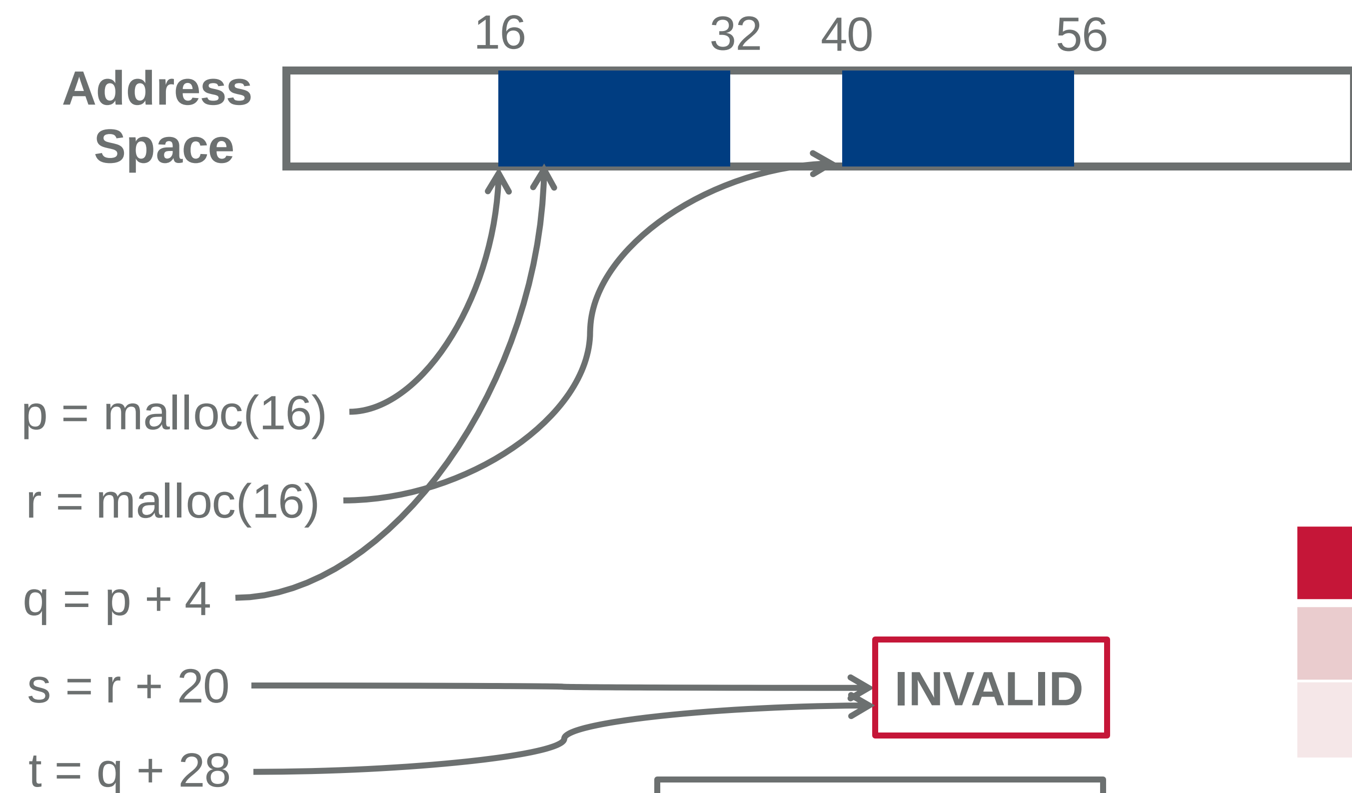
Object table

Start	End
16	32
40	56

Basic rule to check

The result of pointer arithmetic must preserve the referent object, and not overflow it!

Jones and Kelly compiler



*t = 1;

ERROR!

Object table

Start	End
16	32
40	56

Basic rule to check

The result of pointer arithmetic must preserve the referent object, and not overflow it!

Implementing the object table functionality

- Range checks need to be fast!
- Must exploit temporal and spatial locality of memory accesses
- J&K use a **splay tree**, a binary search tree with the property that recently accessed elements are quick to retrieve again

Passing pointers between instrumented and uninstrumented code

High-level design decision

Is it better to have false positives or false negatives?

False positives means valid programs stop with an error

False negatives means buffer overflow goes undetected

Passing pointers between instrumented and uninstrumented code (1)

- Passing pointer from instrumented to uninstrumented
  - Works seamlessly as pointer representation is not changed
  - Errors in uninstrumented code will be missed
    - Trusted libraries can be left unchecked for performance reasons (but see next)

Imperial College London

Passing pointers between instrumented and uninstrumented code (2)

- Passing pointer from uninstrumented to instrumented
  - Out-of-bounds pointer?
    - Points to unallocated space
      - Won't find it in the object table, so don't check
      - But flag cases in which such a pointer is used to derive a pointer to a registered object
    - Points to another object
      - Some checks may pass
      - But an error issued if used to derive a pointer to a different object or unallocated space
  - In-bounds pointer?
    - Allocation site uninstrumented?
      - Don't check but flag cases in which the pointer is used to derive a pointer to a registered object
    - Allocation site instrumented? (malloc'ed sites always instrumented!)
      - Check as usual

Conversion between pointers and integers


```
void *p = &main;
long hash = (long) p | 3;
void *q = (void*)(hash & ~3);
```

Complication:

- Conversion between pointers and integers
- Similar to pointers coming from unchecked code
- May lose checks

One past the end...


```
char buf[100], *p;
    while (p < &buf[100])
        *p = 1;
```

Complication:

- It is legal in C to have a pointer one past the end of the array
- Change alloc behavior to add (at least) one-byte padding b/w objects
- What about function parameters?

  - No padding, for backward compatibility
  - Possible missed bugs
  - Possible false positives if pointer one past the end is brought back in-bounds
  - But rare to pass arrays, so minor concern in practice

One past the end...

- Deriving a pointer more than one past the end is undefined behaviour
  - And compilers could take advantage of this in arbitrary ways, as we have seen in previous lectures
- But a study by Ruwase and Lam in 2004 on 20 benchmarks, 1.2M LOC found that 60% of programs contain such violations
- Can this behaviour be supported?

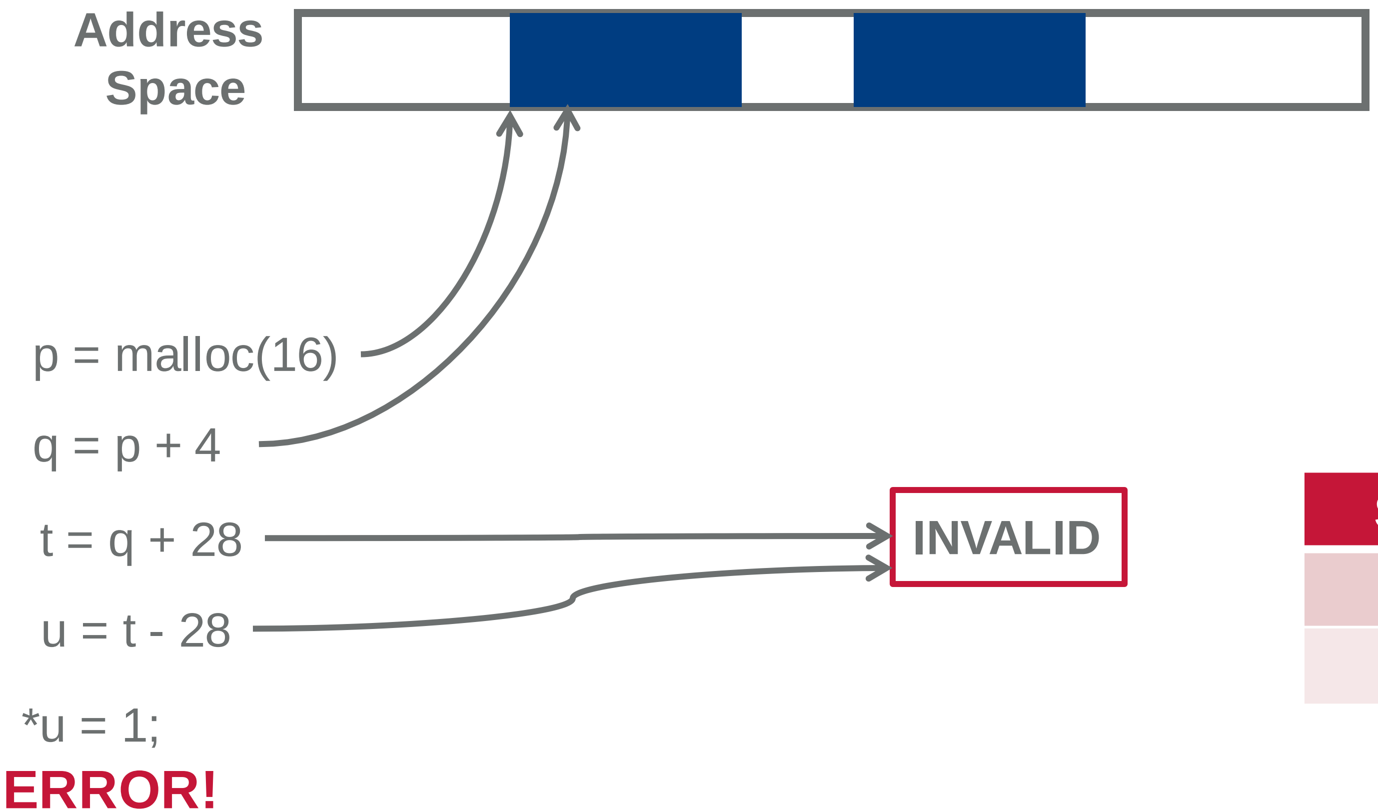
A practical dynamic buffer overflow compiler. Olatunji Ruwase and Monica Lam, Annual Network and Distributed System Security Symposium (NDSS 2004)

Compatibility experiment [CRED paper]

Program	Type	# Lines	Vuln.	Tests	JK	CRED
Apache-1.3.24	web server	73.6K	no	yes	fail	pass
binutils-2.13.2.1	binary tools	596.5K	no	yes	fail	pass
bison-1.875	parser generator	25.1K	no	yes	fail	pass
ccrypt-1.4	encryption utility	4.4K	no	yes	pass	pass
coreutils-5.0	file, shell, & text utilities	69.5K	no	yes	fail	pass
enscript-1.6.1	ascii to postscript converter	22.1K	no	yes	fail	pass
gawk-3.1.2	string manipulation tool	36.4K	yes	yes	fail	pass
gnupg-1.2.2	OpenPGP implementation	71.2K	no	yes	fail	pass
grep-2.5.1	pattern matching utility	20.8K	no	yes	fail	pass
gzip-1.2.4	compression utility	5.8K	yes	yes	pass	pass
hypermail-2.1.5	mail to HTML converter	27.6K	yes	yes	fail	pass
monkey-0.7.1	web server	2.5K	yes	no	pass	pass
OpenSSH-3.2.2p1	SSH1 protocol implementation	43.4K	no	no	fail	pass
OpenSSL-0.9.7b	SSL & TLS toolkit	162.7K	no	yes	fail	pass
pgp4pine-1.76	mail encryption tool	3.3K	yes	no	fail	pass
polymorph-0.40	filesystem unixier	0.4K	yes	no	pass	pass
tar-1.13	archiving utility	18.2K	no	yes	pass	pass
WsMp3-0.0.10	web server	3.4K	yes	no	pass	pass
wu-ftpd-2.6.1	FTP server	18.3K	no	no	pass	pass
zlib-1.13	data compression library	8.3K	no	yes	pass	pass

Imperial College London

Jones & Kelly compiler

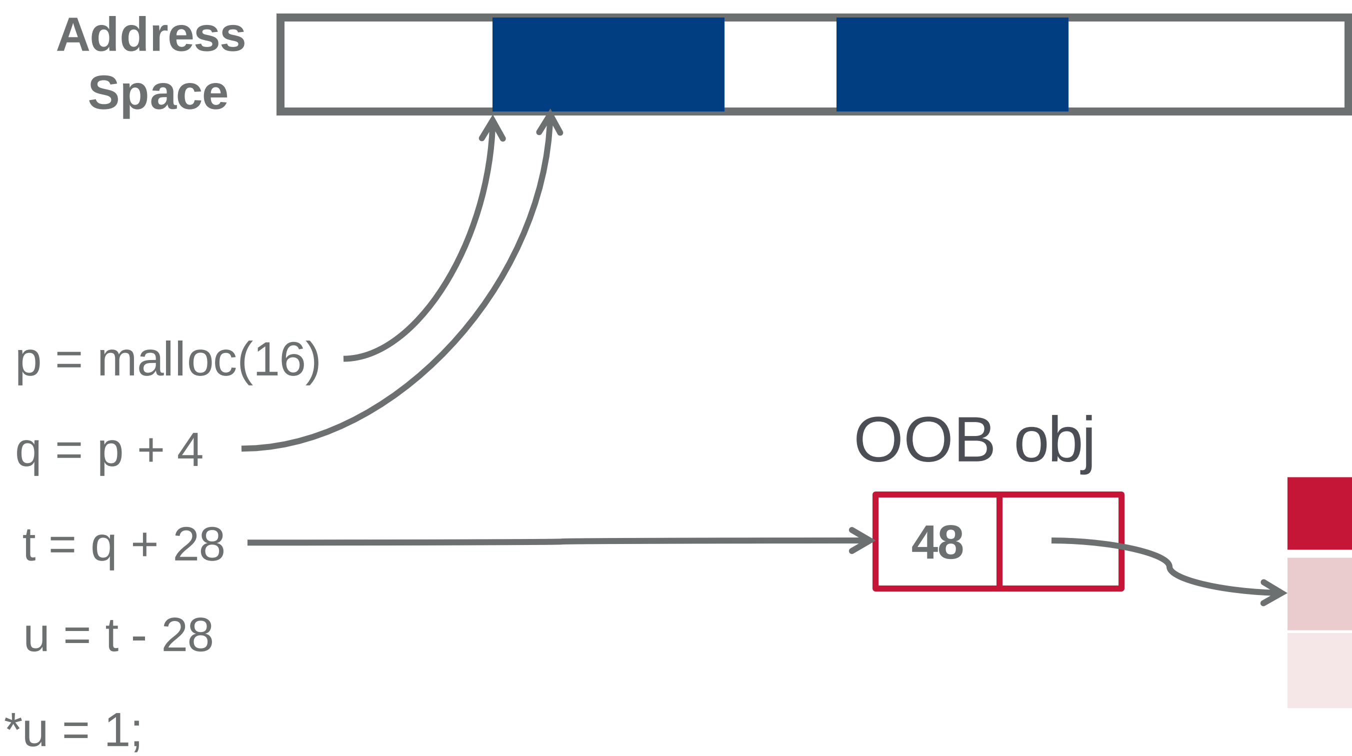


Object table

Start	End
16	32
40	56

Imperial College London

CRED compiler

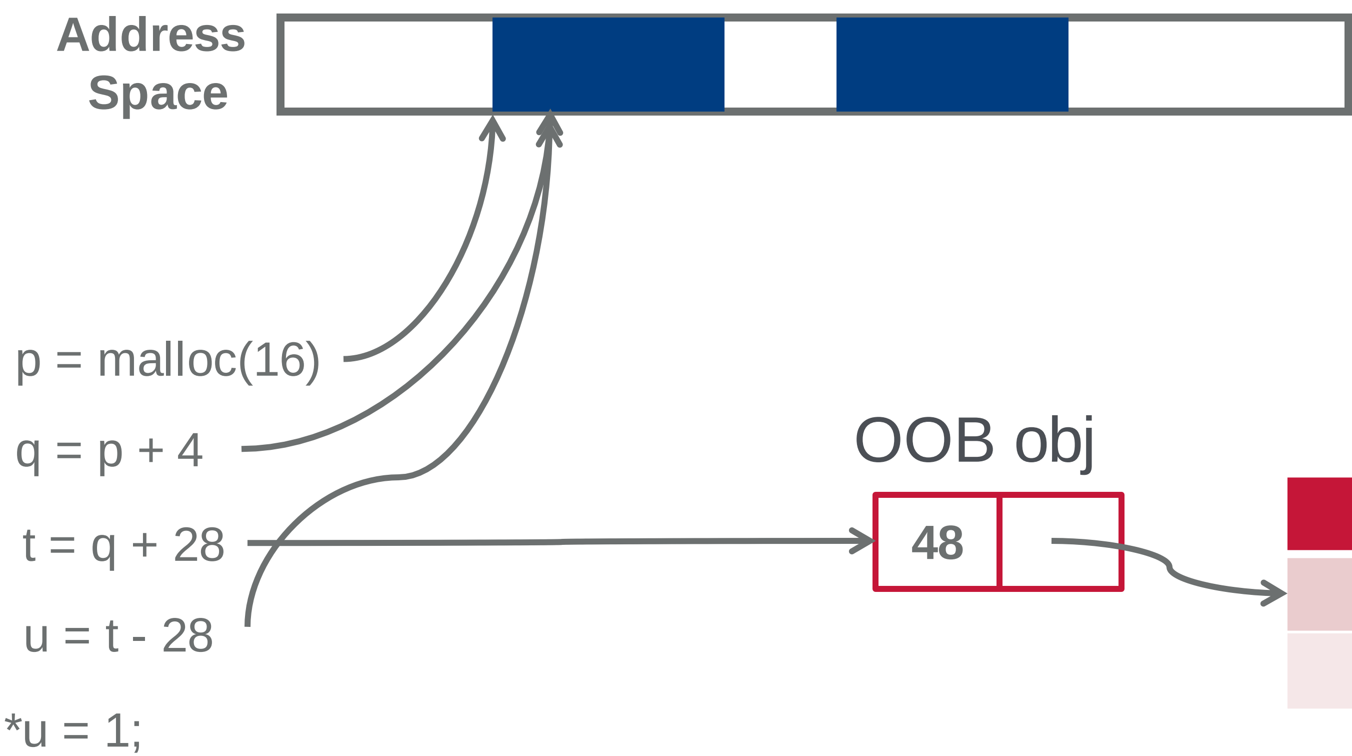


Object table

Start	End
16	32
40	56

During pointer arithmetic, check if pointer points to OOB obj. If so, use its stored value.

CRED compiler



Object table

Start	End
16	32
40	56

If pointer arithmetic brings the pointer in-bounds, switch back to standard pointer representation

CRED vs J&K

1) Compatibility implications: passing out-of-bounds to uninstrumented code would incorrectly pass the OOB address
   - Not seen in any of the benchmarks
2) Extra memory consumption due to OOB objects
   - OOB objects stored in a hash table
   - When a memory object is deallocation, all OOB objects referring to it are also deallocated
3) Extra runtime overhead
   - Shown to introduce negligible additional overhead in most cases, except for 15% slowdown compared to J&K in tar

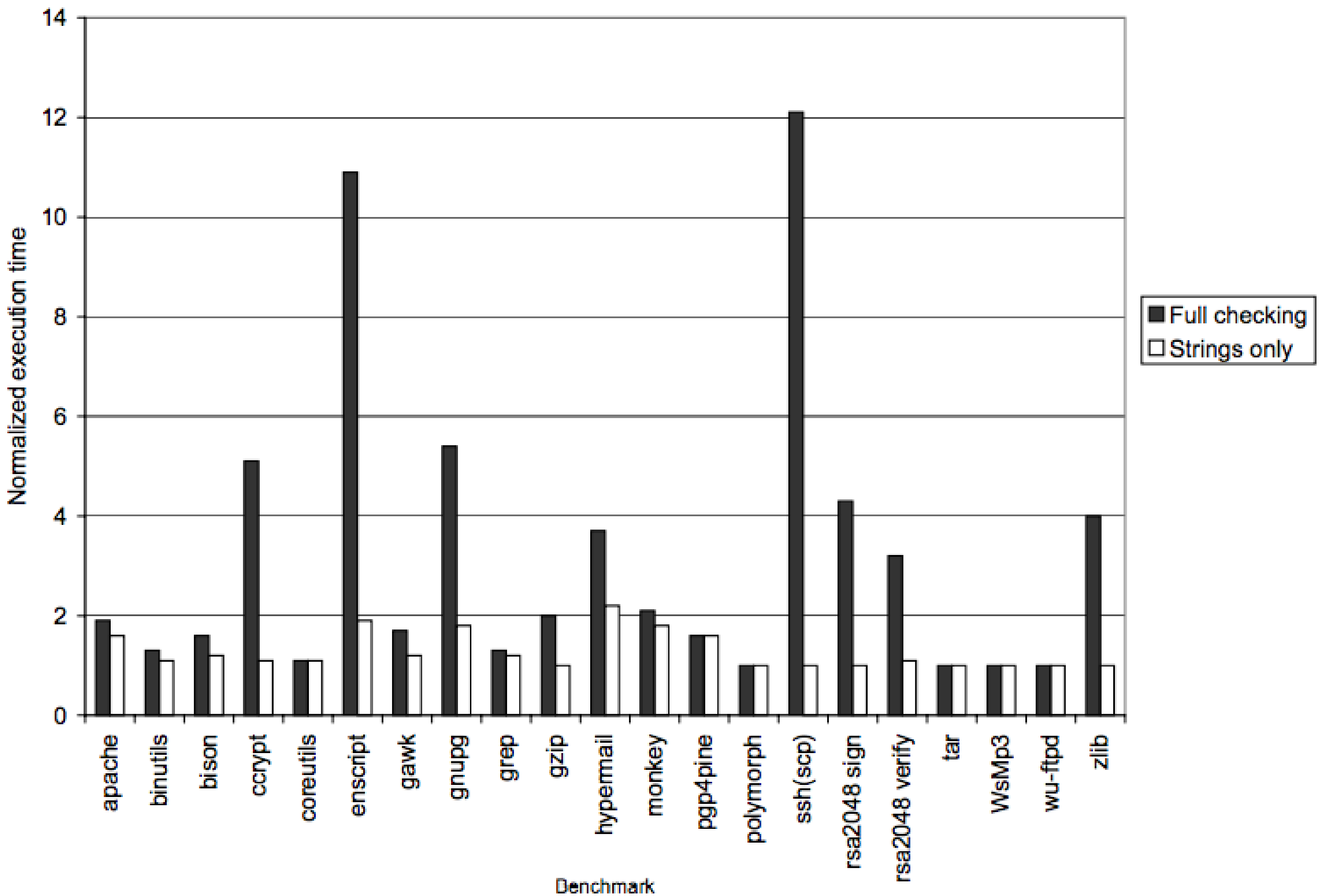
Performance

- Despite optimizations, J&K/CRED compiler introduces a large slowdown
  - 5-6x on the original benchmarks (Tcl/Tk, Ghostscript, GCC, MicroEmacs)
  - But acceptable (<26%) on other benchmarks (coreutils, tar, wu-ftpd, etc.)

- CRED authors argues for a version which protects only char* pointers
  - ssh overhead goes down from 12x to <26%

Imperial College London

CRED performance



Boundless-Memory Blocks

- Detection critical, but not the whole story
  - Terminating the program can be extremely disruptive
  - In some cases, early benign overflows can completely disable execution under Safe C compilers
  - Doesn’t avoid denial of service attacks

- **Focus on availability / continued execution**

Imperial College London

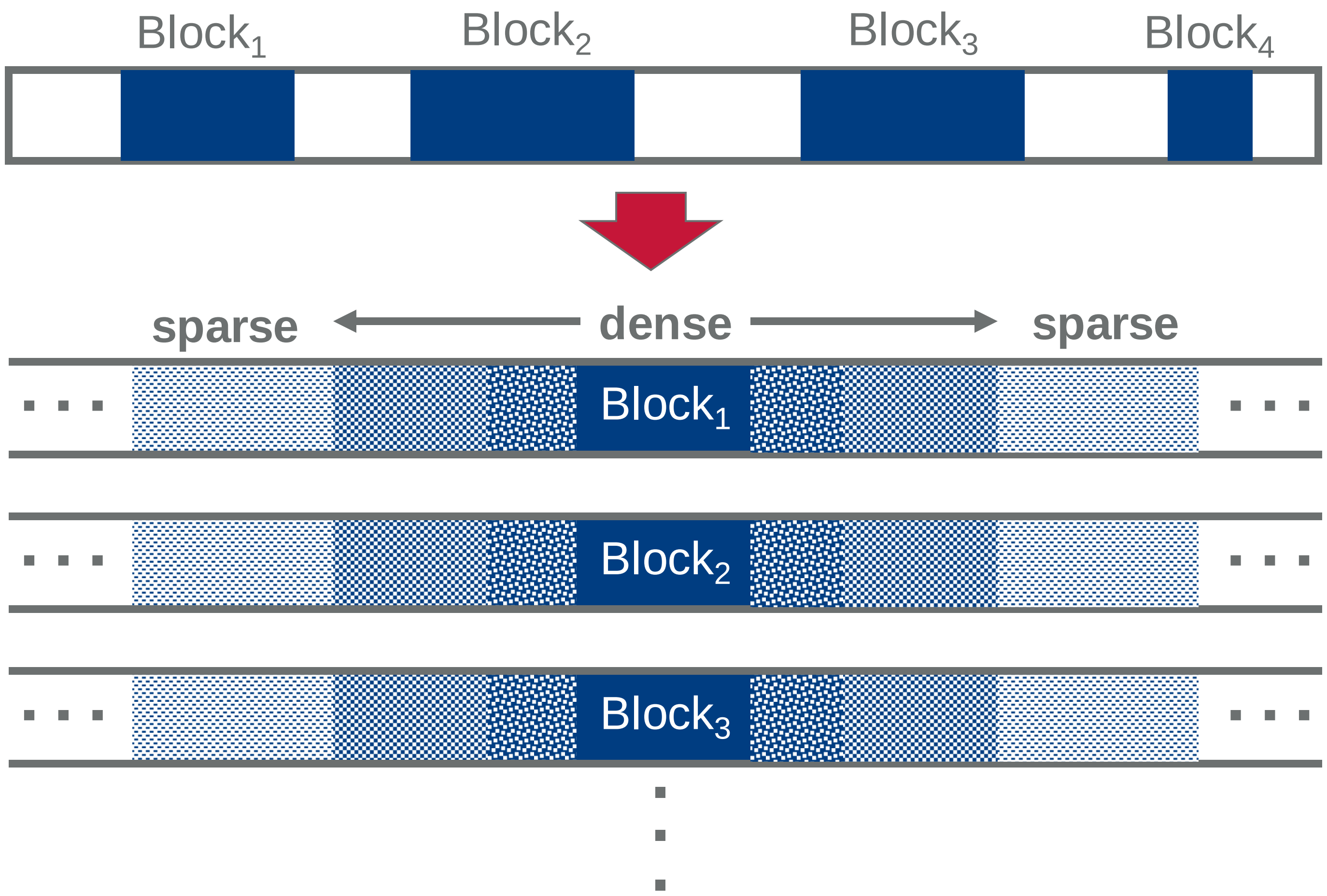
Boundless Memory Blocks Compiler

- Use a Safe C compiler to detect all out of bounds accesses
- Store out of bounds writes in a hash table
- Retrieve out of bounds reads from the hash table
- Conceptually give each allocated memory block its own address space and unbounded size

A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, Tudor Leu. Annual Computer Security Applications Conference (ACSAC 2004)

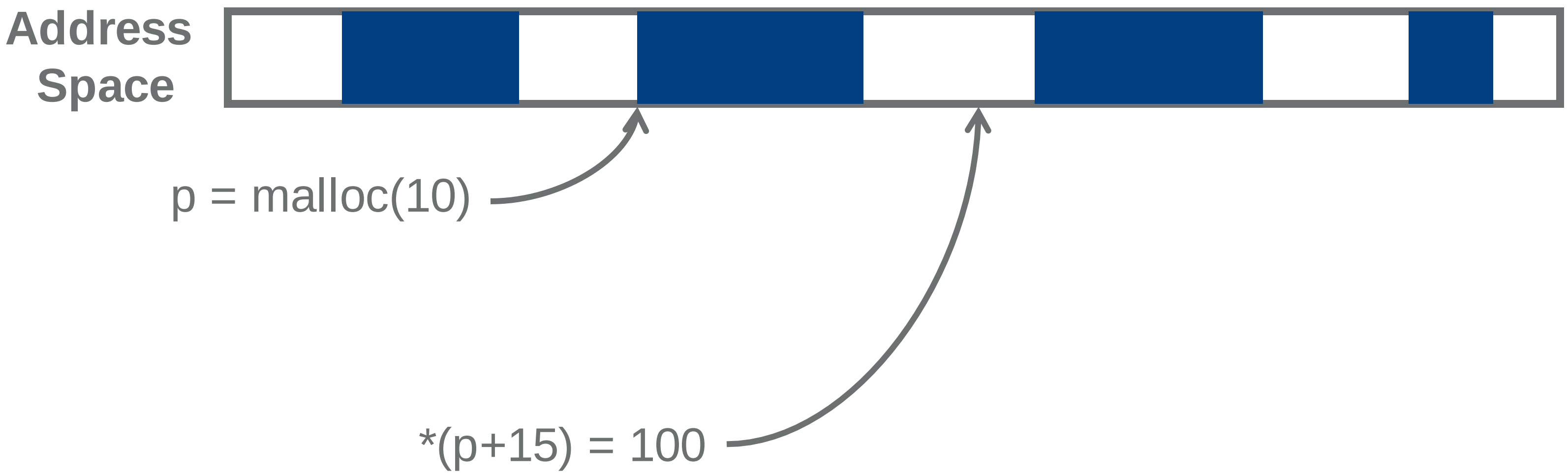
Imperial College London

Our Philosophy

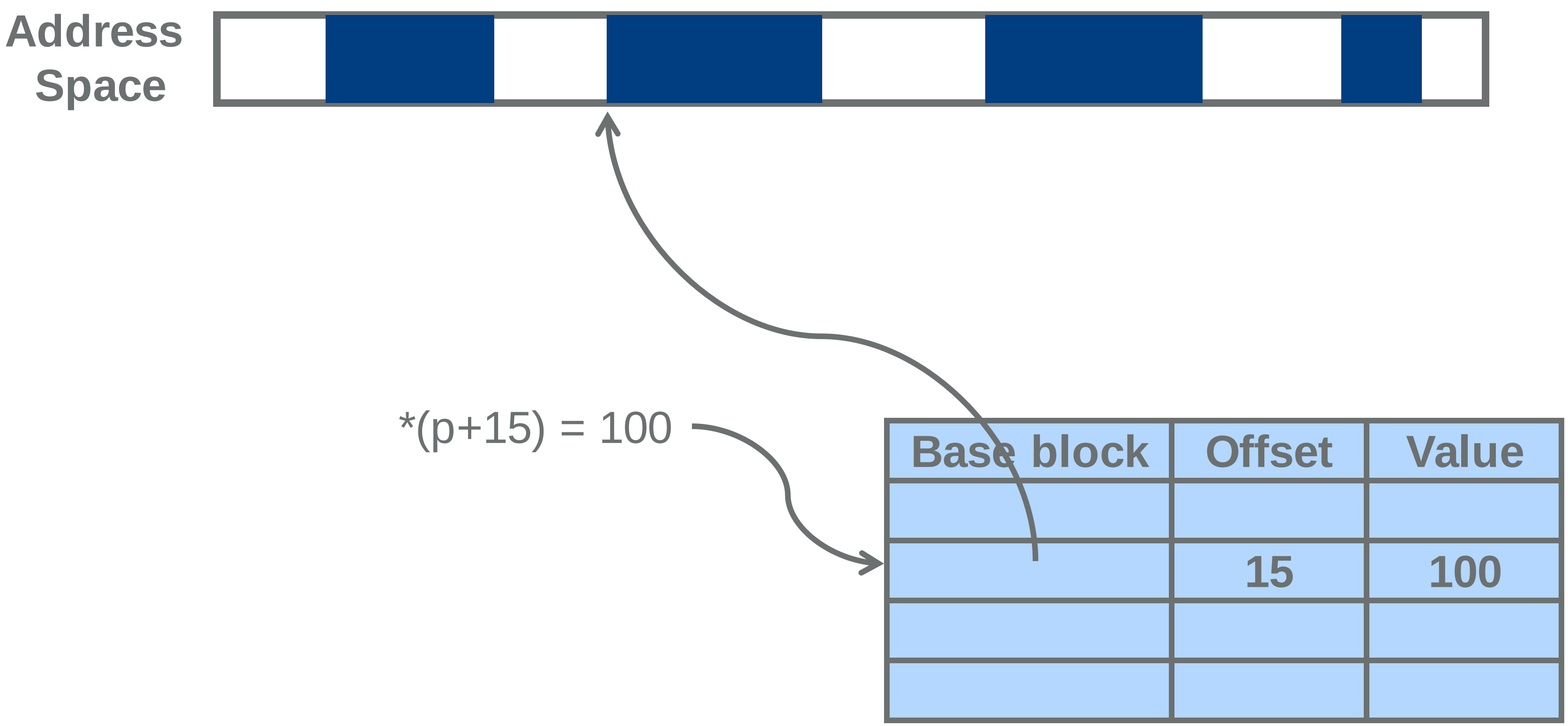


Imperial College London

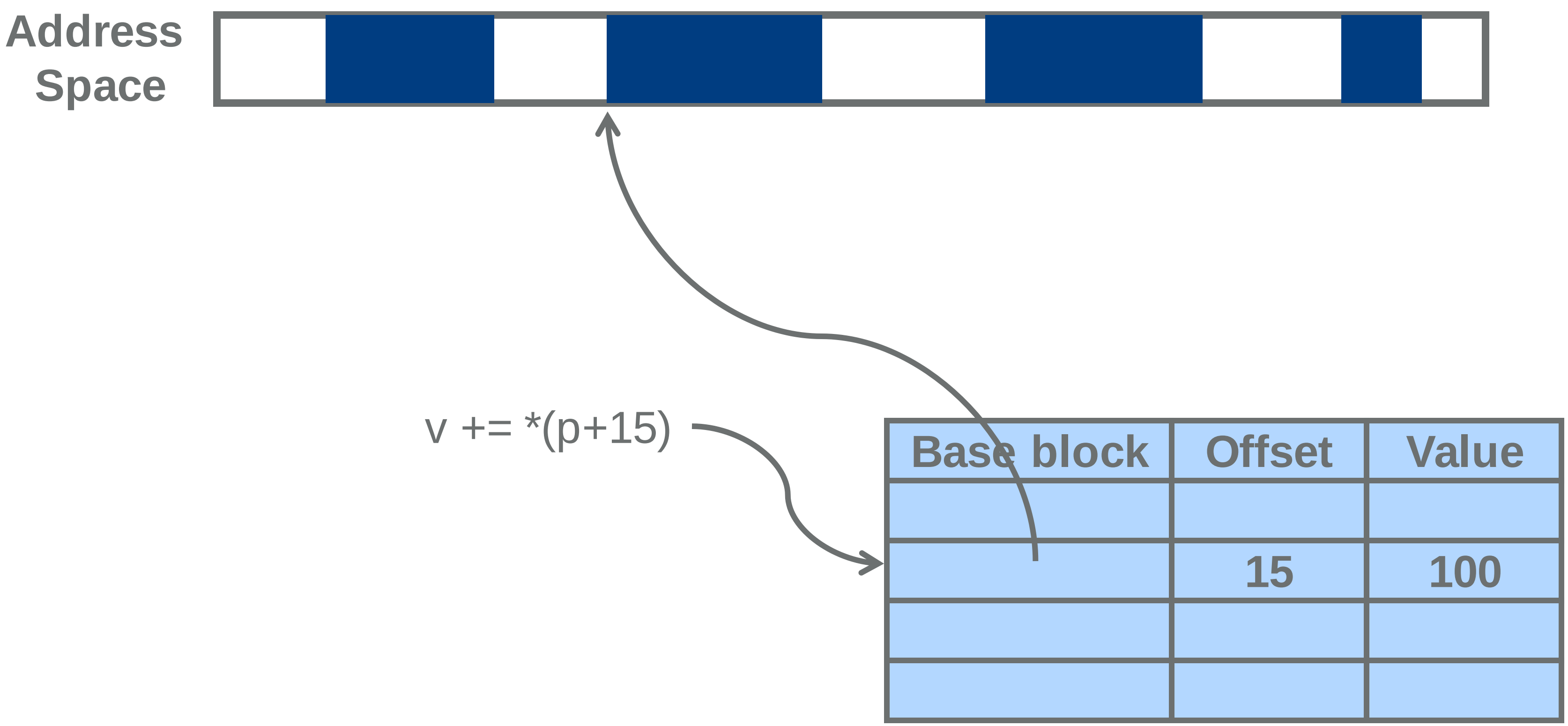
BMB Compiler (cont.)



BMB Compiler (cont.)



BMB Compiler (cont.)



Imperial College London

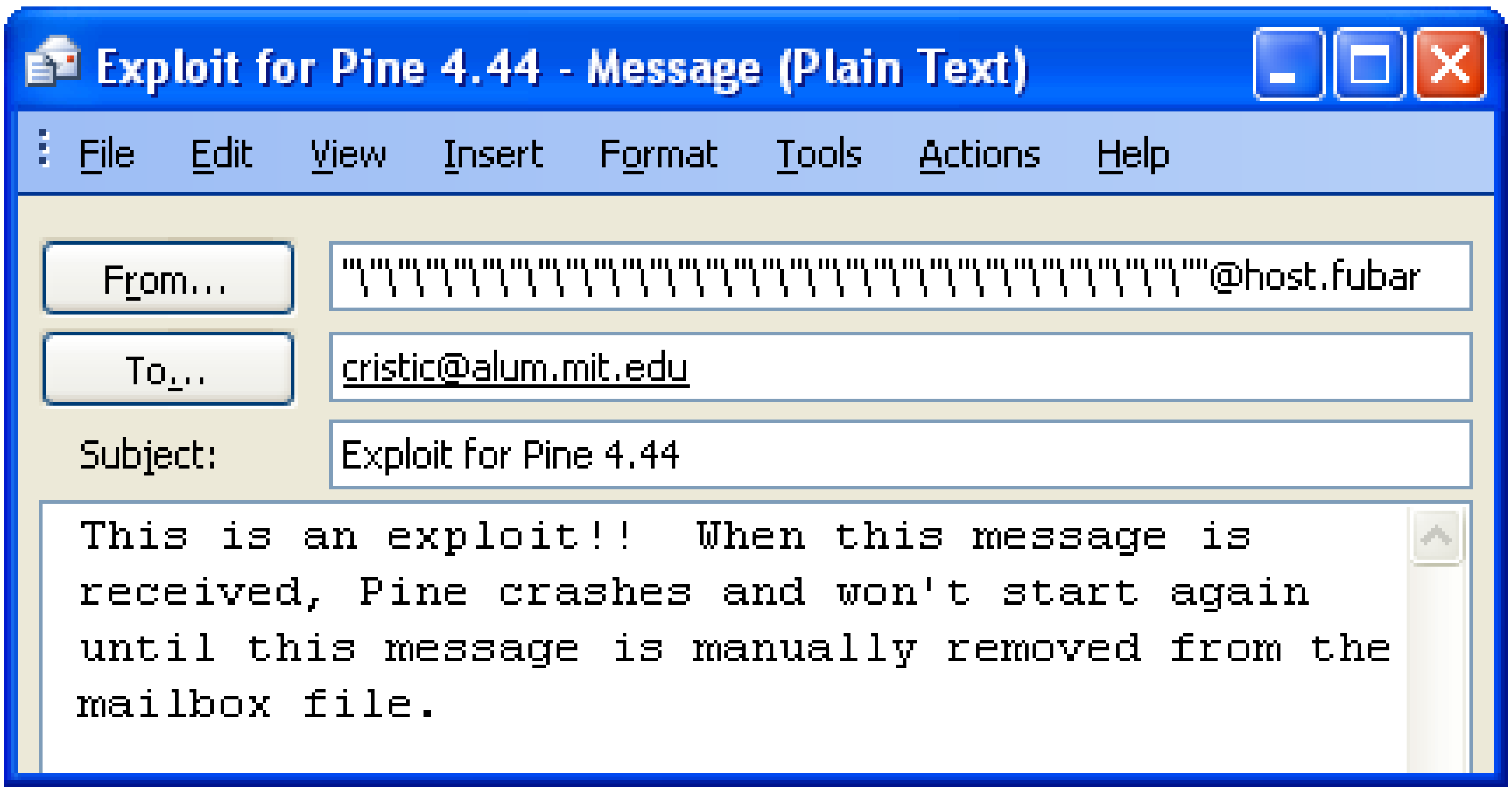
Possible problems

- New DOS attack
  - Craft an input which causes a large number of writes
  - Solution: treat the hash table as a fixed-size cache
    - LRU replacement policy
    - Never observed a case in which the code attempts to access a discarded write entry
- Cache Misses and Uninitialized Reads
  - Returns a default value
  - Absent in most applications

Evaluation

- Acquired several open source programs
  - Servers: Apache, Sendmail
  - Mailers: Pine, Mutt
  - Utilities: Midnight Commander
- Acquired publicized buffer overflow security vulnerabilities
  - SecuriTeam, Security Focus

Vulnerabilities – Pine 4.44

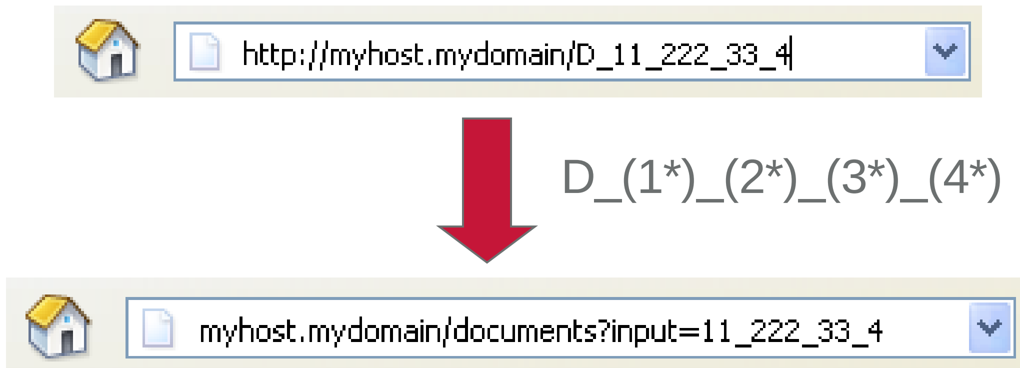


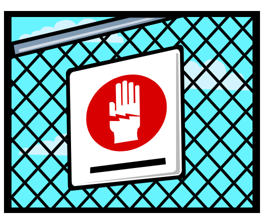
Vulnerabilities – Apache 2.0.47

- Apache can redirect some URLs, which are specified by regular expressions
- Example: redirect URLs of the form *http://myhost.mydomain/D_(1*)_(2*)_(3*)_(4*)* to URLs of the form *http://myhost.mydomain/documents/input=$1_$2_$3_$4*

Imperial College London

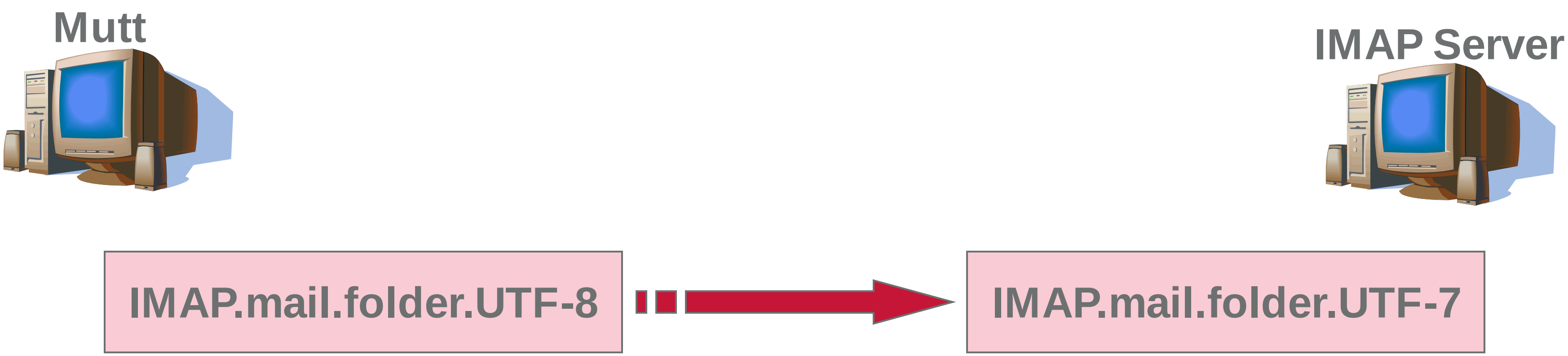
Vulnerabilities – Apache 2.0.47





Static buffer contains space for only 10 parenthesized captures!

Vulnerabilities – Mutt 1.4



- Mutt assumes the UTF-7 string can be at most 2 times longer
- Worst increase ratio is in fact larger

Evaluation (cont.)

- Three versions per benchmark
  - Standard Compilation (GCC)
  - Bounds Check Compilation (CRED)
  - Boundless Memory Blocks Compilation (BMB)
- Tested each versions on the acquired vulnerabilities

Imperial College London

Results

	Secure			Continues Correctly			Initializes			Correct For Attack		
	GCC	CRED	BMB	GCC	CRED	BMB	GCC	CRED	BMB	GCC	CRED	BMB
Pine	✗	✓	✓	✗	✗	✓	✓ (orange)	✓ (orange)	✓	✗	✗	✓
Mutt	✗	✓	✓	✗	✗	✓	✓ (orange)	✓ (orange)	✓	✗	✗	✓
Apache	✗	✓	✓	✓ (orange)	✓ (orange)	✓	✓	✓	✓	✗	✗	✓
Sendmail	✗	✓	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓ (orange)
MC	✗	✓	✓	✗	✗	✓	✓	✓ (orange)	✓	✗	✗	✓ (orange)

Imperial College London

Discussion

- Why it works
  - Developers more likely to incorrectly calculate the size of a buffer or omit a bounds check
  - Cache misses and uninitialized reads are rare
    - Only MC contained some uninitialized reads
- Why it makes sense
  - When allocating memory, hard to reason about the worst case, which is usually exploited by security attacks
  - Although the programmer failed to allocate enough space, the program usually correct when provided with (conceptually) unbounded memory blocks.

Extensible Arrays

- Many languages (e.g. Java) provide extensible arrays
- BMB
  - Preservation of the address space from the original implementation
  - Efficiency – allocates only elements which are actually accessed

Imperial College London

Summary

- Safe C compilers aim to create a memory-safe version of C
  - First generation used fat pointers, which breaks compatibility with unchecked code
  - Second generation does not change pointer representation, ensuring backward-compatibility with unchecked code
- We studied three Safe C compilers:
  - Jones & Kelly: First backward-compatible compiler for standard-compliant programs
  - CRED: allows out-of-bounds pointers for better compatibility with existing code
  - BMB: provides automatically extensible memory blocks, to allow continued execution
- While these compilers generate code that incurs acceptable overhead for some applications, they are overall impractical for use in production (but useful for offline testing)