CCured

Type-Safe Retrofitting of C Programs

[Necula, McPeak, Weimer, Condit, Harren]

One-Slide Summary

- **CCured** enforces **memory safety** and **type safety** in legacy C programs. CCured analyzes how you use pointers and either proves the usage safe statically or inserts **run-time checks**.
- Along the way we'll see cameo appearances by just about every CS 415 topic.

Lecture Outline

- Type and Memory Safety
- CCured Motivation
- SAFE Pointers
- SEQuence Pointers
- WILD Pointers
- Experimental Results
- Analysis

Why Now, Brown Cow?

• Type Systems	[Type Safety]
• Language Security	[Memory Safety]
• Static and Dynamic Types	[Run-Time Type Info]
• Runtime Organization	[Pointer Layout]
• Subtyping	[Convertibility]
• Garbage Collection	[Tag Bits]
• Dataflow Analysis	[Pointer-Kind Inference]
• Object-Oriented Programming	[OOP in C]
• Aspect-Oriented Programming	[Checks Everywhere]
• Libraries	[Library Compatibility]
• Debuggers and Profilers	[Purify and Valgrind]

Two Kinds of Safety

- **Type safety** is a property of a programming language that prevents certain errors (**type errors**) that result from attempts to perform an operation on a value of the wrong type.
  - Type safety prevents: "hello" + 3
  - Not Type safe: 3 + (int)"hello"
    - Some languages allow **unsafe casts** between types.
- **Memory safety**: if a value of type T_1 is read from address A, then the most recent store to A had type T_2 with $T_2 \leq T_1$
  - Store a Dog in memory, read an Animal later

We Need Them Both

- You *cannot* have true Type Safety without Memory Safety
  - *Why?* Hint: an unsafe cast defeats type safety

Explore your creative side with photos and videos

Windows® XP enhances your ability to share photos and videos with yo friends and family. You can easily download pictures from your digital ca into the My Pictures folder.

The improved My Pictures folder shows thumbnail views of your picture in the My Pictures fo or send them throug can even use the Ph d quantity.

lsass.exe - Unable To Locate Component

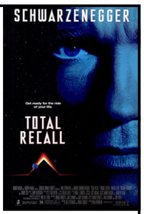
This application has failed to start because

We Need Them Both

- You *cannot* have true Type Safety without Memory Safety
  - Why?
- Example:
  - **table * t = new table(); char buf[10];**
  - **buf[10] = buf[11] = buf[12] = buf[13] = 55;**
  - **t->countLegs();**

Buffer overrun!

Memory Safety



- Essential component of a security infrastructure
  - Prevents interference
  - ≥50% of reported attacks are due to buffer overruns
- Software engineering advantages
  - Memory bugs are hard to find (*why?*)
  - Memory safety ensures component isolation
  - Required for soundness of many program analyses (*why?* hint: aliasing)
  - Does not even need an explicit specification

C and Memory Safety

- C was designed for flexibility and efficiency
  - Many operators can be used unsafely
  - Memory safety is *sacrificed!*

- In practice, many C programs use those operators safely
  - Only a small portion of the pointers and operators are responsible for the unsafe behavior

CCured Idea

1. Devise a sound **type system** and a type inference algorithm that handles most C programs
   - Combination of *static* and *dynamic* types

2. Insert **run-time checks** (e.g. array-bounds checks and dynamic type checking) in those places where safety cannot be verified statically

This way we sacrifice performance instead of safety

- Makes sense for more and more applications every day
- Hardware progress improves performance but not safety

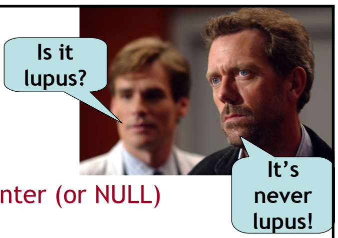
CCured Goals

- Compatibility: support **existing C code**
  - Source-to-source transformation
  - Handle GCC/MSVC source, Makefiles
  - All that is needed is a recompilation: `make CC=ccured`

- Efficiency: 0-50% overhead rather than 1000%
  - Other research: 10x, Purify: 20x, BoundsChecker: 150x

- More effective and more efficient than Purify
  - Because it leverages existing type information in source
  - Use for production code not just during testing

Diseases We Want to Cure



- Focus on pointer usage

- Dereferencing a non-pointer (or NULL)
  - Invoking a non-function
  - Complicated by casts and union types

- Dereferencing outside of object bounds
  - Buffer overruns
  - Complicated by pointer arithmetic
  - Not always caught by Purify

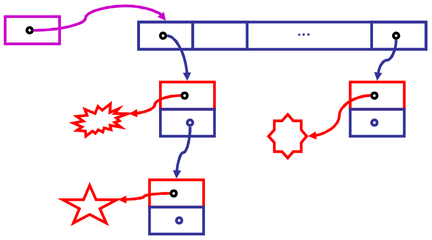
- Freeing non-pointers, using freed memory

Example Pointer Usage in C

- Consider an implementation of a hash table

`struct list {void * data; struct list * next} *  * hash;`

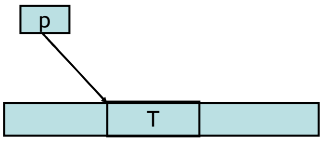
Cast allowed	Yes	No	No
Arith allowed	Yes	No	Yes



SAFE Pointer Invariants and Representation

T * safe

- Can be 0 or a pointer to storage containing a T
- All aliases agree on the type of the referenced storage
- Must do null-check before dereference
- Inexpensive to store and to use
- Prototypical example: `FILE *`



statically typed home area

Quiz

- How many pointers in an average C program are SAFE according to the previous definition?

- Answer: between 40-95% of the declared pointers

Typing Rules for SAFE Pointers

$$\frac{}{0 : T * \text{safe}} \qquad \frac{e : T * \text{safe}}{* e : T}$$

$$\frac{e_1 : T * \text{safe} \qquad e_2 : T}{* e_1 = e_2 : T}$$

$$\frac{e : T * \text{safe} \qquad T * \text{safe} \lesssim T_1 * \text{safe}}{(T_1 * \text{safe})\ e : T_1 * \text{safe}}$$

$T * \text{safe} \lesssim T_1 * \text{safe}$ means that a pointer of the first kind is **convertible** to a pointer of second kind

Convertibility of Pointers

- The **convertibility** relation is based on the physical layout of types (flatten structures and arrays)
- Examples:

`struct { int x, y; int *p;} *` $\lesssim$ `int *`

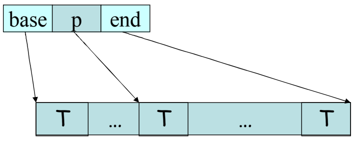
It's like an upcast in OOP

```
struct { int x;
         struct { int y; int *p;} s;
     } *
```
$\lesssim$ `struct { int x, y; } *`

SEQ Pointer Invariants and Representation

T * seq

- Can be 0
- Can be involved in pointer arithmetic
- Null check and bounds check before use
- Carries the bounds of a home area consisting of a sequence of T's
- All SAFE or SEQ aliases agree on the type of the referenced area



statically typed home area

Typing Rules for SEQUENCE Pointers

$$\frac{\text{e : T * seq} \qquad \text{e' : int}}{\text{e + e' : T * seq}}$$

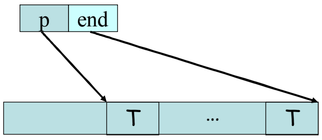
$$\frac{\text{e : T * seq} \qquad \text{T * seq} \lesssim \text{T}_1 \text{ * seq}}{(\text{T}_1 \text{ * seq) e : T}_1 \text{ * seq}}$$

- Before dereferencing a SEQ pointer it must be converted to SAFE (with a bounds check and with dropping the base and the end fields)

$$\frac{\text{e : T * seq}}{\text{(T * safe) e : T * safe}}$$

Forward SEQUENCE Pointers

- Often a sequence pointer is only advanced
  - We call it a Forward Sequence (**FSEQ**)
  - Needs to carry only the end and needs only an upper bound check



- Pointer arithmetic must be checked to advance
- A SEQ is converted to FSEQ via a lower-bound check and dropping the lower-bound field

Quiz

- How many forward-thinking Cylon characters are in a typical Battlestar episode?
- Answer: “Six” or “Eight”

Quiz

- How many forward sequence and sequence pointers are in a typical C program?

- Answer: about 25% FSEQ, 1% SEQ

The WILD West

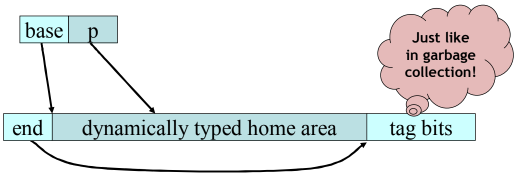
- So far we have not faced the really ugly pointers
  - Those that are cast to incompatible types
- We call them WILD pointers
- For these we cannot count on the static type!
  - We must keep run-time type tags (cf. Python, Ruby)

- Operations allowed:
  - read/write
  - assign an integer
  - pointer arithmetic
  - cast to other ***WILD*** pointers

WILD Pointer Invariants and Representation

T * wild

- Can be a non-pointer (any integer)
- Carries the bounds of a dynamic home area (**containing only integers and dynamic pointers**)
- Has only WILD pointer aliases
- Must do a non-pointer and a bounds check
- Must maintain the tags when writing (1 bit per word)

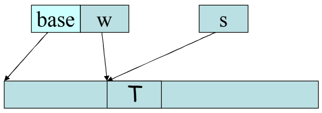


Just like in garbage collection!

WILD Complications

- What if we have a SAFE alias for a WILD ?

```
T * safe s;
T * wild w = s; // w is an alias for s
```

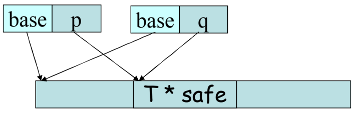


```
*(T' * wild)w = random_stuff_of_type_T';
```

- *For Safety:* WILD pointers can only alias other WILD pointers!

WILD Complications

- What if we have a WILD pointer to a SAFE ptr?



- The type system cannot ensure that all writes through the WILD pointer will write a compatible SAFE pointer
- *For Safety:* WILD pointers must point to areas containing only WILD pointers!

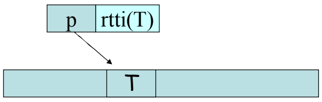
WILD Pointers are Highly Contagious

- A WILD pointer will force other pointers to be WILD as well:
  - All pointers to which it is assigned
  - All pointers from which it is assigned
  - All pointers that it points to
- We need ways to reduce the number of WILD pointers
  - Better understanding of casts

Handling Downcasts

- 10-20% of the pointers are void*
- Cast from void* to T* is *unsafe*!
  - This is a special case of a downcast
  - Downcasts are frequent in C programs (50-90% of bad casts)
- We introduce pointers that carry run-time type info
  - Each downcast is checked at run-time (like in Java)

T_1 * RTTI p;



- Invariant: T is a subtype of T_1

Programming OO-Style in C

- With RTTI pointers we can program safely in a object oriented style (e.g. dynamic dispatch):

```
struct Figure { double (*area)(struct Figure * RTTI   ); }
struct Circle {  double (*area)(struct Figure * RTTI   );
                 double radius; }
double circle_area(struct Figure* RTTI    fig) {
  struct Circle *circ = (struct Circle*)fig;
  return PI * circ->radius * circ->radius;
}
…; struct Figure * RTTI    fig; … fig->area(fig) …
```

- Other places where RTTI helps
  - Heterogeneous data structures
  - Polymorphic code and data structures

Pointer-Kind Inference

- For every pointer in the program
  - Try to infer the fastest sound representation
- We construct a whole-program data flow graph
  - We collect constraints about pointer kinds
  - Then linear-time constraint solving
- Analysis can be modularized if the interfaces are annotated with pointer kind
- Extremely simple, fast and predictable

Example: SAFE/SEQ

```
int *a = ... ;
int *b = a ;
int *c = b ;
print(*c) ;
b = b + 1 ;
```

so **a** is FSEQ too

bounds check here

but **c** can be SAFE

arithmetic: **b** must be FSEQ

Example: WILD

pointer to a pointer

```
int foo(int **p)
{
  int *q = (int *)p;
  return *q;
}
```

cast to incompatible type:
both **p** and **q** are WILD
(so is the caller's argument!)

Experience Using CCured

- CCured handles all of C:
  - vararg, function pointers, union types, GCC extensions
- CCured works on low-level code
  - Apache modules, Linux device drivers
- CCured scales to large programs
  - sendmail, openssl, ssh, bind (>= 100K lines)
  - ACE infrastructure (>= 1M lines)
- CCured often *requires manual intervention*
  - must change between 1/100 to 1/300 lines of code

Experimental Results

Slowdown of CCured and Purify vs. C

(low numbers = good)

	ijpeg	compress	go	li	bh	tsp
CCured	1.6	1.2	1.1	1.8	1.5	1.1
Purify	30	28	51	50	94	42

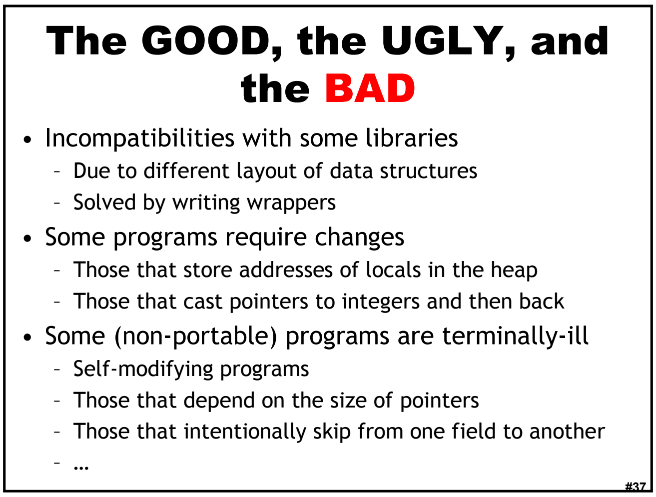
- 0-80% slowdown
- 60-100% of pointers are statically known to be type safe
- Found bugs in SPEC95 benchmarks

The GOOD, the UGLY, and the BAD

- Standard techniques from type theory can be used to understand the “type safety” of existing C programs
- CCured works automatically in most cases
- Most pointers are SAFE and some are SEQUENCE
- The slowdown is minimal in many cases
  - The uglier your program the slower it will be

The GOOD, the UGLY, and the BAD

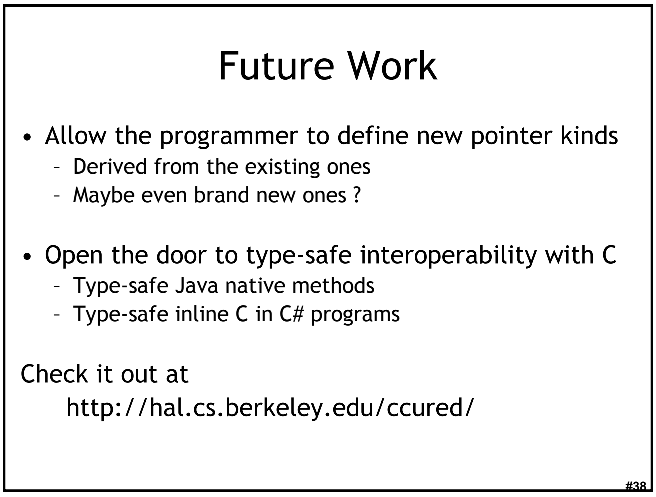
- Occasional significant slowdown
  - Typically due to either large number of WILD or SEQUENCE pointers
- Increased memory footprint
  - Larger code size
  - Some pointers take 64-bits and some even 96-bits
- CCured is confused by custom-memory allocators
  - Forced to treat them WILD
  - Or to trust the allocator (as in the experiments)



The GOOD, the UGLY, and the BAD

- Incompatibilities with some libraries
  - Due to different layout of data structures
  - Solved by writing wrappers
- Some programs require changes
  - Those that store addresses of locals in the heap
  - Those that cast pointers to integers and then back
- Some (non-portable) programs are terminally-ill
  - Self-modifying programs
  - Those that depend on the size of pointers
  - Those that intentionally skip from one field to another
  - …

#37

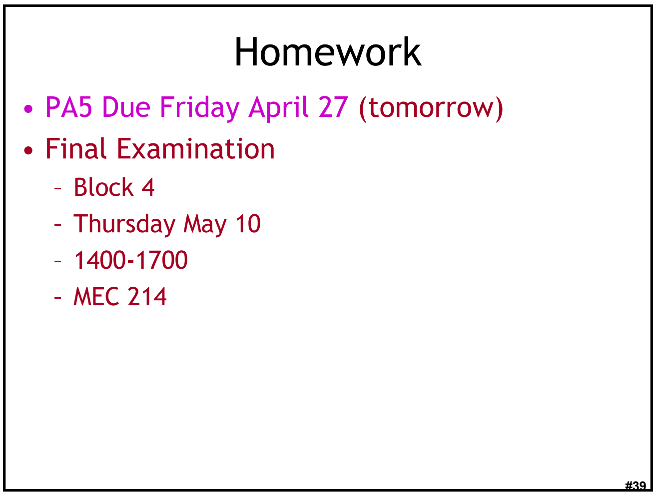


Future Work

- Allow the programmer to define new pointer kinds
  - Derived from the existing ones
  - Maybe even brand new ones ?
- Open the door to type-safe interoperability with C
  - Type-safe Java native methods
  - Type-safe inline C in C# programs

Check it out at
http://hal.cs.berkeley.edu/ccured/

#38



Homework

- PA5 Due Friday April 27 (tomorrow)
- Final Examination
  - Block 4
  - Thursday May 10
  - 1400-1700
  - MEC 214

#39