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
- Language Security
- Static and Dynamic Types
- Runtime Organization
- Subtyping
- Garbage Collection
- Dataflow Analysis
- Object-Oriented Programming
- Aspect-Oriented Programming
- Libraries
- Debuggers and Profilers

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
We Need Them Both

- You cannot have true Type Safety without Memory Safety
  - Why?
- Example:
  - `table * t = new table(); char buf[10];`
  - `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;
  ```

<table>
<thead>
<tr>
<th>Cast allowed</th>
<th>Yes</th>
<th>No</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith allowed</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

SAFE Pointer Invariants and Representation

- 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 *`

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

\[
\begin{align*}
0 & : T \* \text{safe} & e & : T \\
T_1 & : T \* \text{safe} & e_1 & : T \\
& e_1 = e_2 & : T \\
T & : T \* \text{safe} & T_1 & \preceq T_1 \* \text{safe} \\
& (T_1 \* \text{safe}) e & : T_1 \* \text{safe}
\end{align*}
\]

\(T \* \text{safe} \preceq T_1 \* \text{safe}\) 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:
  \[
  \text{struct \{} \text{int} x, y; \text{int *p}\text{;} \text{\}} \* \preceq \text{int *}
  \]
  \[
  \text{struct \{} \text{int} x; \\
  \quad \text{struct \{} \text{int} y; \text{int *p}\text{;} \text{\}} s;
  \text{\}} \* \preceq \text{struct \{} \text{int} x, y; \text{\}} *
  \]

SEQ Pointer Invariants and Representation

- 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

Base | p | End

T ... T ... T

Statically typed home area
Typing Rules for SEQUENCE Pointers

\[
\frac{e : T^* \text{seq} \quad e' : \text{int}}{e + e' : T^* \text{seq}}
\]

\[
\frac{e : T^* \text{seq} \quad T^* \text{seq} \preceq T_1^* \text{seq}}{(T_1^* \text{seq}) e : T_1^* \text{seq}}
\]

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

\[
\frac{e : T^* \text{seq}}{(T^* \text{safe}) e : T^* \text{safe}}
\]

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)
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 *(T *(T *(T
  ```

  *(T *(T *(T *(T
  ```

  // w is an alias for s // w is an alias for s // w is an alias for s // w is an alias for s

  *(T *(T *(T *(T
  ```

  random_stuff_of_type_T random_stuff_of_type_T random_stuff_of_type_T 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?

  ```
  T * safe p;
  T * safe q;
  ```

  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)
  
\[
T_1 \rightarrow \text{RTTI} \rightarrow T \\
\]

- 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):

```c
struct Figure { double (*area)(struct Figure *); };
struct Circle { double (*area)(struct Figure *); double radius; };

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

- 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**

```c
int *a = ... ;  // so a is FSEQ too
int *b = a ;    // bounds check here
int *c = b ;    // but c can be SAFE
print(*c) ;     // arithmetic: b must be FSEQ
b = b + 1 ;     //
```

Example: **WILD**

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

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)

<table>
<thead>
<tr>
<th></th>
<th>jpeg</th>
<th>compress</th>
<th>go</th>
<th>li</th>
<th>bh</th>
<th>tsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCured</td>
<td>1.6</td>
<td>1.2</td>
<td>1.1</td>
<td>1.8</td>
<td>1.5</td>
<td>1.1</td>
</tr>
<tr>
<td>Purify</td>
<td>30</td>
<td>28</td>
<td>51</td>
<td>50</td>
<td>94</td>
<td>42</td>
</tr>
</tbody>
</table>

- 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
  - ...

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/

Homework

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