

CCured: Type-Safe Retrofitting of Legacy Code

Language Based Security
CS 6v81 – 002

February 20, 2008

Problem: Adding Type Safety to Unsafe Legacy C Code

- Significant legacy C code exists
- C offers flexibility and performance advantages (critical to systems programmers)
- Drawback is a weak type system
- Hardware and software performance have improved
- Desirability of type safety increasing

Problem Continued

- Easy mistakes
 - Array out of bounds
 - Inadvertent memory access
 - Difficult to debug
- Malicious attacks
 - Buffer overflows (50% of CERT advisories)

What do we gain from type safety?

- Avoid need for separate address spaces
- Improved interfacing with type safe languages
- Improved performance over other security techniques
- Identification of subtle errors

Observations leading to CCured

- Much code is statically verifiable as safe
- Run time checks can ensure safety
- Reduced performance is an acceptable trade for increased safety
- High cost of reprogramming
 - Emphasis on ease of porting existing code

The Authors Solution

- Extension of the C type system with explicit pointer types
- Type inference algorithm
 - Identify pointers that can be statically verified
 - Identify pointers that require run-time checks
- Combines typed and untyped concepts

CCured Pointers

- CCured pointers track additional attributes
 - Memory area size
 - Types of values
- Computed at compile or run-time
- Why might this be a problem?

Basic Cured Pointer Types

- SAFE – Basic pointer; statically verified
- SEQUENCE – Allows pointer arithmetic; statically verified
- DYNAMIC – Can have multiple types; requires run-time checks

Pointer “home”

- Home – the starting address of a pointer, used for bounds checks and checking for pointers converted to integers
- Safe pointers are only integers n
- Sequence and dynamic pointers store home information plus the address $\langle h, n \rangle$
- Home of a pointer converted to an integer is 0
- Also used to track pointer conversions (i.e. sequence to int to sequence yields $\langle h, n \rangle$ to $\langle 0 \rangle$ to $\langle 0, h+n \rangle$) to determine if a resulting pointer should be allowed to be used as a memory address

Pointer Types in CCured

Kind	Invariants maintained	Capabilities	Access checks required
Safe pointer to τ	<ul style="list-style-type: none"> • Either 0 or a valid address containing a value of type τ. • Aliases are either safe or sequence pointers of base type τ. 	<ul style="list-style-type: none"> • Cast from sequence pointer of base type τ. • Set to 0. • Cast to integer. 	<ul style="list-style-type: none"> • Null-pointer check when dereferenced.
Sequence pointer to τ	<ul style="list-style-type: none"> • Knows at run-time if it is an integer, and if not, knows the memory area (containing a number of values of type τ) to which it points. • Aliases are safe and sequence pointers of base type τ. 	<ul style="list-style-type: none"> • Cast to safe pointer of base type τ. • Cast from integer. • Cast to integer. • Perform pointer arithmetic. 	<ul style="list-style-type: none"> • Non-pointer check (subsumes null-pointer check). • Bounds check when dereferenced or cast to SAFE.
Dynamic pointer	<ul style="list-style-type: none"> • Knows at run-time if it is an integer, and if not, knows the memory area (containing a number of integer or dynamic pointer values) to which it points. • The memory area pointed to maintains tags distinguishing integers from pointers. • Aliases are dynamic pointers. 	<ul style="list-style-type: none"> • Cast to and from any dynamic pointer type. • Cast from integer. • Cast to integer. • Perform pointer arithmetic. 	<ul style="list-style-type: none"> • Non-pointer check. • Bounds check when dereferenced. • Maintain the tags in the pointed area when reading and writing.

Figure 2: Summary of the properties and capabilities of various kinds of pointers.

Sample Code Fragment

```
1 int *1 *2 a;           // array
2 int i;                 // index
3 int acc;               // accumulator
4 int *3 *4 p;           // elem ptr
5 int *5 e;              // unboxer
6 acc = 0;
7 for (i=0; i<100; i++) {
8     p = a + i;         // ptr arith
9     e = *p;            // read elem
10    while ((int) e % 2 == 0) { // check tag
11        e = * (int *6 *7) e; // unbox
12    }
13    acc += ((int)e >> 1); // strip tag
14 }
```

Figure 1: A short C program fragment demonstrating safe and unsafe use of pointers.

Type Safety

- Each non-null home has one of two “kind”s
- Typed – contains a number of values of a type and has only safe and sequence pointers of the base type pointing to it
- Untyped – contains a number of values of type DYNAMIC and has only pointers of DYNAMIC pointing to it
- At all times the contents of each memory address must conform to the typing constraints of its home
- CCured uses Progress Theorems to state that if all the added run time checks succeed, execution will not halt

Type Inference

- Major innovation of CCured
- Translates existing C programs into CCured with minimum operator intervention
- Establishes all pointers as either SAFE, SEQUENCE or DYNAMIC
- Goal is to mark as many pointers SAFE or SEQUENCE as possible
- Linear run-time

How it works?

- Step 1: Scan the source code and collect a list of constraints for each pointer
- Step 2: Normalize the constraints into a simpler form
- Step 3: Solve the constraints

Constraint Solving

- Step 1: Identify the minimum number of DYNAMIC variables based on the collected constraints and propagate to pointers that alias them
- Step 2: Of the remaining variables mark any that perform pointer arithmetic SEQUENCE
- Step 3: The remaining variables must be SAFE

CCured Performance

- Ran significantly faster than cases where all pointers were marked DYNAMIC
- Significantly faster than run-time techniques such as Purify
- Program size was a factor as larger programs tended to use pointers in more varied ways

Problems with CCured

- Requires source code
- Requires source code changes
 - Generally less than dialects such as Cyclone
- DYNAMIC pointers can spread via aliasing
- Several common C usages require DYNAMIC pointers
- Compatibility issues with non-CCured libraries

Additional CCured Limitations

- CCured takes data representation control from programmers
 - Extra storage required by SEQ and DYN Pointers
- Basic operations may have different costs
 - Dereference, arithmetic, etc based on pointer type
- Only option for memory management is garbage collection

Source Code Change Examples

- Storing a pointer in a int variable, casting back to a pointer and dereferencing
 - change from (ex.) unsigned long to void*
 - replace casts with pointer arithmetic
 - query garbage collection as a last resort
- sizeof(type) must change to sizeof(expression)

More source changes

- `int **p = (int**)malloc(5 * sizeof(int*))`
 - Why does this not work?
- address-of to store stack pointers into memory; instead annotate with a qualifier to allocate them on the heap

CCured in the Real World

- Utilize physical subtyping to reduce the number of dynamic pointers
 - Statically verify upcasts as safe
- Run-time type information combined with physical subtyping to handle downcasts
 - Supports parametric and subtype polymorphism
 - Allows 99% of casts to be verified w/o resorting to WILD (DYNAMIC) pointers
- Split metadata into separate data structures
 - Lessens need for wrappers

How 99% was calculated?

- 63% of casts are between identical types
- Remaining 37% were bad casts in original CCured
 - 93% are safe upcasts
 - 6% are downcasts
 - < 1% fall outside these categories

Future Work

- Further reduce DYNAMIC pointers
- Reduce the need for source code changes
- Combine with PCC or TAL
- Support additional methods of memory management

References

- George C. Necula, Scott McPeak, Westley Weimer. [CCured: Type-Safe Retrofitting of Legacy Code](#). In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*. Portland, OR, 2002.
- Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, Westley Weimer. [CCured in the Real World](#). In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 2003.
- Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. [Cyclone: A Safe Dialect of C](#). In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.

Questions

- Do you think this provides enough benefit to merit revisiting existing C code?
- Why can libraries be a practical problem?
- What are the pros and cons compared to SFI/CFI or PCC?
- Is the loss of control of data representations and memory management significant?
- Why would you choose CCured over Cyclone or vice versa?