Type Qualifiers and Security

- This presentation will discuss two papers that use qualifiers for security purposes
- Qualifiers are used to extend the normal C type system to provide more rigorous (and clever) type checking, both statically and dynamically
- First paper: qualifiers for intelligent instrumentation of runtime checks
- Second paper: qualifiers for tracking tainted data flow

CCured: Type-Safe Retrofitting of Legacy Code

George C. Necula, Scott McPeak & Westley Weimer

Presented by Jeff Johnson

The Problem Space

- As we all know...
- C is extremely flexible with types and data representation
- Great for low level nitty gritty, but often causes subtle bugs when manipulating pointers
  - Array out of bounds access
  - NULL dereferencing
  - Accidental aliasing
  - Bad casting
  - Etc...

What Can We Do?

- Naïve approach: during runtime, hold extra information with each pointer and perform checks on all memory reads and writes
- For example, Purify
- But slow
  - Usually lots of reads and writes to check
  - Ignoring *context* of read or write

Runtime Checks Needed?

```
int *cat;
...
int dog = *cat;
```

cat is non-NULL

```
int fish[5];
...
int *shark = fish + 10;
...
int squid = *shark;
```

shark is in bounds

shark is non-NULL

Runtime checks can be done selectively based on usage

CCured Approach

- Key insight: Type safety can be verified statically for a large portion of a C program
- The rest can be checked at runtime
- In other words, CCured will separate type checking into two parts
  - Static checks when possible
  - Instrumentation for runtime checks only when needed
- CCured will use extensions to the C type-system to do so

Presentation Overview

- We will discuss the following
  - CCured dialect and type system
  - Runtime checks/operational semantics
  - Dealing with legacy code – type inference
  - Results and discussion
  - Post-paper developments (it was published in 2002)

CCured Dialect (Simplified)

Types:	τ	$::=$	$\texttt{int} \mid \tau\ \texttt{ref SAFE} \mid \tau\ \texttt{ref SEQ}$
		$\mid$	$\texttt{DYNAMIC}$
Expressions:	e	$::=$	$x \mid n \mid e_1\ \texttt{op}\ e_2 \mid (\tau)e$
		$\mid$	$e_1 \oplus e_2 \mid\ !\,e$
Commands:	c	$::=$	$\texttt{skip} \mid c_1; c_2 \mid e_1 := e_2$

- Important to note:
  - p ⊕ i → p + i (pointer arithmetic)
  - !p → *p
  - Pointer types: ref SAFE, ref SEQ, DYNAMIC

T ref SAFE

```
int *cat;
...
int dog = *cat;
```

→

```
int ref SAFE cat;
...
int dog = !cat;
```

- Pointers used in a statically checkable safe way
- At runtime, either `NULL` or valid address containing type *T*
- Aliases are either *T ref SAFE* or *T ref SEQ*

T ref SEQ

```
int *fish;   // array
int *shark;
shark = fish + 10;
```

```
int ref SEQ fish;
int ref SAFE shark;
shark =
 (int ref SAFE)fish ⊕ 10;
```

- Pointers involved in pointer arithmetic
- At runtime, holds information about the memory area (a sequence of type *T*) it points to
- Aliases are either *T ref SAFE* or *T ref SEQ*

DYNAMIC

```
int **wild;
int *crazy = (int*) wild;
int nuts = *crazy;
```

```
DYNAMIC wild;
DYNAMIC crazy = wild;
int nuts = !crazy;
```

- Pointers involved in unsafe operations that are not checkable at compile time
- At runtime, holds information about the memory area it points to (or if it is actually an integer)
- Aliases are always *DYNAMIC*

Type System

Expressions:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \;\texttt{op}\; e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash (\tau) e : \tau} \qquad \frac{}{\Gamma \vdash (\tau\;\texttt{ref SAFE})0 : \tau\;\texttt{ref SAFE}}$$

$$\frac{\Gamma \vdash e_1 : \tau\;\texttt{ref SEQ} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \tau\;\texttt{ref SEQ}} \qquad \frac{\Gamma \vdash e_1 : \texttt{DYNAMIC} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{DYNAMIC}} \qquad \frac{\Gamma \vdash e : \tau\;\texttt{ref SAFE}}{\Gamma \vdash\, !e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{DYNAMIC}}{\Gamma \vdash\, !e : \texttt{DYNAMIC}}$$

Commands:

$$\frac{}{\Gamma \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \qquad \frac{\Gamma \vdash e : \tau\;\texttt{ref SAFE} \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e'} \qquad \frac{\Gamma \vdash e : \texttt{DYNAMIC} \quad \Gamma \vdash e' : \texttt{DYNAMIC}}{\Gamma \vdash e := e'}$$

Convertibility:

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\tau \leq \texttt{int}} \qquad \frac{}{\texttt{int} \leq \tau\;\texttt{ref SEQ}} \qquad \frac{}{\texttt{int} \leq \texttt{DYNAMIC}} \qquad \frac{}{\tau\;\texttt{ref SEQ} \leq \tau\;\texttt{ref SAFE}}$$

Note that it seems that we could do
`DYNAMIC <: int <: SEQ <: SAFE`
But we cannot, because of operational semantics we'll see later

Runtime Model

- Need to do the following checks dynamically
  - SAFE: not-NULL on reads/writes
  - SEQ: not-NULL on reads/writes, within bounds on reads/writes and casts to SAFE
  - DYNAMIC: not-NULL and within bounds on reads/writes
- To do this, we will use the following representation
  - SAFE, int: as normal integers
  - SEQ, DYNAMIC: as `<home, value>`
    - `home` holds information about the memory area the pointer refers to and `value` refers to the pointer's value (usually an offset from `home`)

Expressions:

$$\frac{}{\Sigma, M \vdash n \Downarrow n}\ \text{INT} \qquad \frac{\Sigma(x) = v}{\Sigma, M \vdash x \Downarrow v}\ \text{VAR} \qquad \frac{\Sigma, M \vdash e_1 \Downarrow n_1 \quad \Sigma, M \vdash e_2 \Downarrow n_2}{\Sigma, M \vdash e_1 \texttt{ op } e_2 \Downarrow n_1 \texttt{ op } n_2}\ \text{OP}$$

Casts:

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\texttt{int})e \Downarrow n}\ \text{C1} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n\rangle}{\Sigma, M \vdash (\texttt{int})e \Downarrow h + n}\ \text{C2}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\tau \texttt{ ref SEQ})e \Downarrow \langle 0, n\rangle}\ \text{C3} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n\rangle}{\Sigma, M \vdash (\tau \texttt{ ref SEQ})e \Downarrow \langle h, n\rangle}\ \text{C4}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\texttt{DYNAMIC})e \Downarrow \langle 0, n\rangle}\ \text{C5} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n\rangle}{\Sigma, M \vdash (\texttt{DYNAMIC})e \Downarrow \langle h, n\rangle}\ \text{C6}$$

$$\frac{\Sigma, M \vdash e \Downarrow n}{\Sigma, M \vdash (\tau \texttt{ ref SAFE})e \Downarrow n}\ \text{C7} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n\rangle \quad \boxed{0 \leq n < \texttt{size}(h)}}{\Sigma, M \vdash (\tau \texttt{ ref SAFE})e \Downarrow h + n}\ \text{C8}$$

Pointer arithmetic:

$$\frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n_1\rangle \quad \Sigma, M \vdash e_2 \Downarrow n_2}{\Sigma, M \vdash e_1 \oplus e_2 \Downarrow \langle h, n_1 + n_2\rangle}\ \text{ARITH}$$

Memory reads:

$$\frac{\Sigma, M \vdash e \Downarrow n \quad \boxed{n \neq 0}}{\Sigma, M \vdash {!}\,e \Downarrow M(n)}\ \text{SAFERD} \qquad \frac{\Sigma, M \vdash e \Downarrow \langle h, n\rangle \quad \boxed{h \neq 0} \quad \boxed{0 \leq n < \text{size}(h)}}{\Sigma, M \vdash {!}\,e \Downarrow M(h + n)}\ \text{DYNRD}$$

Commands:

$$\frac{}{\Sigma, M \vdash \texttt{skip} \Longrightarrow M}\ \text{SKIP} \qquad \frac{\Sigma, M \vdash c_1 \Longrightarrow M' \quad \Sigma, M' \vdash c_2 \Longrightarrow M''}{\Sigma, M \vdash c_1; c_2 \Longrightarrow M''}\ \text{CHAIN}$$

$$\frac{\Sigma, M \vdash e_1 \Downarrow n \quad \boxed{n \neq 0} \quad \Sigma, M \vdash e_2 \Downarrow v_2}{\Sigma, M \vdash e_1 := e_2 \Longrightarrow M[{}^{v_2}/_{n}]}\ \text{SAFEWR} \qquad \frac{\Sigma, M \vdash e_1 \Downarrow \langle h, n\rangle \quad \boxed{h \neq 0} \quad \boxed{0 \leq n < \text{size}(h)} \quad \Sigma, M \vdash e_2 \Downarrow v_2}{\Sigma, M \vdash e_1 := e_2 \Longrightarrow M[{}^{v_2}/_{h+n}]}\ \text{DYNWR}$$

Instrumenting Code (SAFE Reads)

```
int ref SAFE cat;
/* allocate space for cat */
int dog = !cat;                         // read
```

Instrumentation

```
int ref SAFE cat;                       // cat = 0
/* allocate space for cat */            // cat = n
int dog;
if (cat != 0)                           // check null
   dog = !cat;                          //  dog = *n
else
   // error – halt
```

Runtime Casting Rules

- `int n <: SEQ,DYNAMIC` → n becomes <0, n> (i.e. a `NULL` pointer)
- `SEQ <: SAFE` → <h, v> becomes h + v (plus a bounds check)
- `SEQ, DYNAMIC <: int` → <h, v> becomes h + v
- `SAFE <: int` → no change in memory
- Note that casting from a pointer to int and back creates a `NULL` pointer, disallowing

`DYNAMIC <: int <: SEQ <: SAFE`

Instrumenting Code (Casting)

```
int ref SEQ fish; // array
/* ...allocate space for fish */
int ref SAFE shark;
shark = (int ref SAFE)fish ⊕ 10;
```

Instrumentation

```
int ref SEQ fish;                   // fish = <0,0>
/* ...allocate space for fish */    // fish = <h,n>
int ref SAFE shark;                 // shark = 0
if (0 <= n+10 < size(h))            // check bounds
   shark = (int ref SAFE)fish ⊕ 10; // shark= h+n+10
else
   // error – halt
```

Type Inference

- No one wants to annotate legacy code to use CCured pointer-types
- Instead, use a type inference algorithm to maximize the number of SAFE, SEQ pointers used and minimize the number of DYNAMICS
- Follows same inference work-flow we've been seeing
  - Constraint Generation
  - Constraint Normalization
  - Constraint Solving

Constraint Generation

- Generate variables for pointers in program
- Generate constraints based on pointer use
- Possible values: {SAFE, SEQ, DYNQ}

Example constraints (for qualifier variable **q**):

$T \text{ ref } q \oplus n \rightarrow q \;!= \text{SAFE}$

$T_1 \text{ ref } q_1 <: T_2 \text{ ref } q_2 \rightarrow$
$(q_1 = q_2 \vee (q_1 = \text{SEQ} \wedge q_2 = \text{SAFE})) \wedge$
$(q_1 = q_2 = \text{DYNQ} \vee T_1 \approx T_2)$

$T \text{ ref } q' \text{ ref } q \wedge q = \text{DYNQ} \rightarrow q' = \text{DYNQ}$

Constraint Normalization/Solving

- Simplify constraints
- Solve using the following steps
  - Propagate (q = DYNQ) to all qualifiers that are references or aliases of q
  - Set all unsolved qualifiers with (q != SAFE) to SEQ and propagate to references and aliases of q
  - Set all other qualifiers to SAFE
  - Lastly, do: q = DYNQ $\rightarrow$ T ref q = DYNAMIC

Inference Example: SAFE and SEQ

```
int *foo;
int *baz;
...
foo = baz + 10;
```

→

```
int ref Q1 foo;
int ref Q2 baz;
...
foo = (int ref Q1) baz ⊕ 10;
```

↓

Q2 != SAFE
Q2 = Q1 OR (Q2 = SEQ AND Q1 = SAFE)
Q2 = Q1 = DYNQ OR int = int

↓

Q2 != SAFE
Q2 = Q1 OR (Q2 = SEQ AND Q1 = SAFE)

↓

Q2 = SEQ
Q1 = SAFE

$T \text{ ref } q \oplus n \rightarrow q \text{ != SAFE}$

$T_1 \text{ ref } q_1 <: T_2 \text{ ref } q_2 \rightarrow$
$(q_1=q_2 \lor (q_1=\text{SEQ} \land q_2=\text{SAFE})) \land$
$(q_1=q_2=\text{DYNQ} \lor T_1 \approx T_2)$

Inference Example: DYNQ

```
int **wild;
int *crazy = (int*)wild;
```

```
int ref Q1 ref Q2 wild;
int ref Q3 crazy = (int ref Q3)wild;
```

Q2 = Q3 OR (Q2 = SEQ AND Q3 = SAFE)
Q2 = Q3 = DYNQ OR (int ref Q1) = int

Q2 = Q3 = DYNQ

```
int ref Q1 ref DYNQ wild;
int ref DYNQ crazy = (int ref DYNQ)wild;

DYNAMIC wild;
DYNAMIC crazy = wild;
```

T_1 ref q_1 <: T_2 ref q_2 →
$(q_1=q_2 \lor (q_1=SEQ \land q_2=SAFE)) \land$
$(q_1=q_2=DYNQ \lor T_1 \approx T_2)$

Experimentation

Program	LOC	Description
compress	1,590	LZW data compression
go	29,315	Plays the board game Go
ijpeg	31,371	Compresses image files
li	7,761	Lisp interpreter
bh	2,053	n-body simulator
bisort	707	Sorting algorithm
em3d	557	Solves electromagnetism problem
health	725	Simulates Colombia's health care system
mst	617	Computes minimum spanning tree
perimeter	395	Computes perimeters of regions in images
power	763	Simulates power market prices
treeadd	385	Builds a binary tree
tsp	561	Approximates Traveling Salesman Problem

Source Changes

- To make using CCured possible, had to change the source of some test programs slightly
  - `sizeof` gives incorrect size when passed a type, because of "fat" pointers. Fixed by passing an expression (i.e. `sizeof(int*)` → `sizeof(p)`)
  - Moving locals to the heap (because of issues involving saving stack references using address-of)
- Other changes that might be needed
  - pointer cast to `int` then back to pointer: don't do it
  - incompatibility with library functions: use wrapper functions to convert "fat" pointers to normal representations and back

Results

Name	Lines of code	Orig. time	CCured sf/sq/d	CCured **ratio**	Purify ratio
SPECINT95					
compress	1590	9.586s	87/12/0	**1.25**	28
go	29315	1.191s	96/4/0	**2.01**	51
ijpeg	31371	0.963s	36/1/62	**2.15**	30
li	7761	0.176s	93/6/0	**1.86**	50
Olden					
bh	2053	2.992s	80/18/0	**1.53**	94
bisort	707	1.696s	90/10/0	**1.03**	42
em3d	557	0.371s	85/15/0	**2.44**	7
health	725	2.769s	93/7/0	**0.94**	25
mst	617	0.720s	87/10/0	**2.05**	5
perimeter	395	4.711s	96/4/0	**1.07**	544
power	763	1.647s	95/6/0	**1.31**	53
treeadd	385	0.613s	85/15/0	**1.47**	500
tsp	561	3.093s	97/4/0	**1.15**	66

Bugs Found

- `compress` and `ijpeg` each have one array bounds violation
- `go` has eight bounds violations, and one use of an uninitialized integer used for array indexing
- The paper lacks further discussion...

Conclusion

- CCured uses type qualifiers to track pointer usage and optimize runtime checks for safe memory access
- What else can we do with qualifiers and type inference?

Detecting Format String Vulnerabilities with Type Qualifiers

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster and David Wagner

Presented By Jeff Johnson

Problem Space and Approach

- Addressing the problem of format vulnerabilities
  - e.g. `printf(buf)`
- Use type qualifiers to detect vulnerabilities *statically*
  - Annotate small set of typed elements as tainted or untainted
  - Infer taintedness for other elements throught the program
  - Complain if tainted element can reach a format string function
  - Similar to Perl, but Perl tracks taintedness during runtime

Example

Declare

```
tainted char *get_string_from_user();
void printf(untainted *char format, … );
```

Vulnerable Code

```
char *response =
   get_string_from_user(); // infer tainted
...
printf(response);
```

Raise error at compile time!

Why Type Annotations?

- Familiar to programmers
- Easy way to understand error output
- Type theory is well understood
- Provide a sound basis for formal verification

Taintedness Type System

- `tainted` – types of values controllable by user
- `untainted` – types for other values
- Examples:

```
untainted int x;    // integer untouched by user
tainted char *y;    // pointer to a tainted char
char * untainted z;// untainted pointer to char
int a;              // neither tainted nor untainted
```

Taintedness Type System (2)

Sub-typing Relation:

untainted T < tainted T

Allows untainted data to become tainted, but not the reverse

Sub-typing Rules:

$$\frac{Q1 <: Q2 \quad T1 <: T2}{Q1\ T1 <: Q2\ T2}$$

$$\frac{Q1 <: Q2 \quad T1 = T2}{Q1\ ptr(T1) <: Q1\ ptr(T2)}$$

Type Inference

- User introduces a small number of annotations as “constraint seeds”
- Generate qualifier variables for each typed element in the program
- Generate constraints based on variable usage
- Solve using sub-typing rules, find inconsistencies

Example: Solving Constraints

```
tainted char *getenv(char *name);  // seed
...
char * x = getenv("FOO");
```

Generate qualifier variables

```
getenv_ret_p char * getenv_ret
 getenv(getenv_arg0_p char * getenv_arg0 name);
   where (getenv_ret_p = tainted)
...
x_p char * x_v x = getenv("FOO");
```

Generate constraints

```
getenv_ret_p char * getenv_ret <: x_p char * x_v
```

Solve constraints

```
  getenv_ret_p = x_p = tainted, get_ret <: x_v
```

Example: Finding Unsafe Code

```
tainted char *getenv(const char *name);
int printf(untainted const char *fmt, ...);

char *s;
s = getenv(“FOO”);
printf(t);
```

Generates constraints

```
tainted = getenv_ret_p = s_p
   <:  printf_arg0_p = untainted
```

DOES NOT TYPE CHECK

tainted <: untainted is not allowed

Type System Extensions

- Polymorphism
  - For functions, sometimes return value taintedness is dependent on what is passed
  - Solution: hand-write constraints using special qualifier variables to have “conditional” taintedness
- Variable Argument Functions
  - Hand-write special qualifiers to apply to all extra arguments

Other Extensions

- GUI integrated into GNU Emacs
- Taint Flow Graph
  – Trace taintedness using a flow graph tracking where taintedness comes from
  – Present to the user for easy traceback
- Hotspots
  – Present user with hottest quantifiers; those involved in the largest number of taint flow paths

Experimentation

- Metrics
  - How many known vulnerabilities detected and how many undetected?
  - How many false positives?
  - How easy to determine if a warning is a real bug?
  - How long did the automated analysis take
  - How easy was preparing programs for analysis?

Results

Name	Version	Description	Lines	Preproc.	Time	Warnings	Bugs
cfengine	1.5.4	System administration tool	24k	126k	28s	5	1
muh	2.05d	IRC proxy	3k	103k	5s	12	1
bftpd	1.0.11	FTP server	2k	34k	2s	2	1
mars_nwe	0.99	Novell Netware emulator	21k	73k	21s	0	0
mingetty	0.9.4	Remote terminal control utility	0.2k	2k	1s	0	0
apache	1.3.12	HTTP server	33k	136k	43s	0	0
sshd	2.3.0p1	OpenSSH ssh daemon	26k	221k	115s	0	0
imapd	4.7c	Univ. of Wash. IMAP4 server	43k	82k	268s	0	0
ipopd	4.7c	Univ. of Wash. POP3 server	40k	78k	373s	0	0
identd	1.0.0	Network identification service	0.2k	1.2k	3s	0	0

Discussion

- On first run, most programs produced a decent amount of warnings
- Hot spot finder was helpful in finding correct spots for qualifiers
- After inserting several qualifiers, only a few warnings issued
- Timing (per program):
  - 30 – 60 minutes to modify build process
  - usually < 1, no greater than 10 minutes for automated analysis to run
  - tens of minutes for human analysis of results