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;

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

cat is non-NULL
shark is non-NULL
shark is in bounds

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: \( \tau ::= \text{int} \mid \tau \text{ ref SAFE} \mid \tau \text{ ref SEQ} \mid \text{DYNAMIC} \)

Expressions: \( e ::= x \mid n \mid e_1 \text{ op } e_2 \mid (\tau)e \mid e_1 \oplus e_2 \mid !e \)

Commands: \( c ::= \text{skip} \mid c_1;c_2 \mid e_1 := e_2 \)

- Important to note:
  - \( p \oplus i \to p + i \) (pointer arithmetic)
  - \( !p \to *p \)
  - Pointer types: ref SAFE, ref SEQ, DYNAMIC
• 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
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
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 \textit{DYNAMIC}
Note that it seems that we could do
\texttt{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:
\[
\begin{align*}
\frac{\Sigma, M \vdash n \downarrow n}{\text{INT}}, & \quad \frac{\Sigma(x) = v}{\text{VAR}}, & \quad \frac{\Sigma, M \vdash c_1 \downarrow n_1}{\text{OP}} \quad \frac{\Sigma, M \vdash c_2 \downarrow n_2}{},
\end{align*}
\]

Casts:
\[
\begin{align*}
\frac{\Sigma, M \vdash e \downarrow n}{\Sigma, M \vdash \mathtt{(int)} e \downarrow n} \quad & \quad \text{C1} \\
\frac{\Sigma, M \vdash (\tau \mathtt{ref} \mathtt{SEQ}) e \downarrow \langle 0, n \rangle}{\Sigma, M \vdash e \downarrow n} \quad & \quad \text{C3} \\
\frac{\Sigma, M \vdash e \downarrow n}{\Sigma, M \vdash \mathtt{(DYNAMIC)} e \downarrow \langle 0, n \rangle} \quad & \quad \text{C5} \\
\frac{\Sigma, M \vdash e \downarrow \langle h, n \rangle}{\Sigma, M \vdash (\tau \mathtt{ref} \mathtt{SAFE}) e \downarrow \langle h \rangle} \quad & \quad \text{C7} \\
\frac{\Sigma, M \vdash e \downarrow \langle h, n \rangle}{\Sigma, M \vdash (\tau \mathtt{ref} \mathtt{SAFE}) e \downarrow h + n} \quad & \quad \text{C8}
\end{align*}
\]

Pointer arithmetic:
\[
\frac{\Sigma, M \vdash e_1 \downarrow \langle h, n_1 \rangle \quad \Sigma, M \vdash e_2 \downarrow n_2}{\Sigma, M \vdash e_1 \oplus e_2 \downarrow \langle h, n_1 + n_2 \rangle} \quad \text{ARITH}
\]

Memory reads:
\[
\begin{align*}
\frac{\Sigma, M \vdash e \downarrow n \quad n \neq 0}{\Sigma, M \vdash ! e \downarrow M(n)} \quad & \quad \text{SAFERD} \\
\frac{\Sigma, M \vdash e \downarrow \langle h, n \rangle \quad h \neq 0 \quad 0 \leq n < \text{size}(h)}{\Sigma, M \vdash ! e \downarrow M(h + n)} \quad & \quad \text{DYNRD}
\end{align*}
\]

Commands:
\[
\begin{align*}
\frac{\Sigma, M \vdash \text{skip} \Rightarrow M}{\text{SKIP}} \\
\frac{\Sigma, M \vdash e_1 \downarrow n \quad n \neq 0 \quad \Sigma, M \vdash e_2 \downarrow v_2}{\Sigma, M \vdash e_1 := e_2 \Rightarrow M[v_2/n]} \\
\frac{\Sigma, M \vdash e_1 \downarrow \langle h, n \rangle \quad h \neq 0 \quad 0 \leq n < \text{size}(h)}{\Sigma, M \vdash e_2 \downarrow v_2} \quad & \quad \text{SAFEWR} \\
\frac{\Sigma, M \vdash c_1 \Rightarrow M' \quad \Sigma, M' \vdash c_2 \Rightarrow M''}{\Sigma, M \vdash c_1 ; c_2 \Rightarrow M''} \quad & \quad \text{CHAIN}
\end{align*}
\]

\[
\frac{\Sigma, M \vdash e_1 \downarrow \langle h, n \rangle \quad h \neq 0 \quad 0 \leq n < \text{size}(h)}{\Sigma, M \vdash e_2 \downarrow v_2} \quad & \quad \text{DYNWR}
\]

\[
\frac{\Sigma, M \vdash e_1 \downarrow \langle h, n \rangle}{\Sigma, M \vdash e_2 \downarrow v_2} \quad & \quad \text{DYNWR}
\]
Instrumenting Code (SAFE Reads)

```c
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

- \( \text{int } n <: \text{SEQ,DYNAMIC} \rightarrow n \text{ becomes } <0, n> \) (i.e. a NULL pointer)
- \( \text{SEQ} <: \text{SAFE} \rightarrow <h, v> \text{ becomes } h + v \) (plus a bounds check)
- \( \text{SEQ, DYNAMIC} <: \text{int} \rightarrow <h, v> \text{ becomes } h + v \)
- \( \text{SAFE} <: \text{int} \rightarrow \) no change in memory
- Note that casting from a pointer to \text{int} and back creates a NULL pointer, disallowing
\( \text{DYNAMIC} <: \text{int} <: \text{SEQ} <: \text{SAFE} \)
Instrumenting Code (Casting)

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

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 \(q\)):

\[ T \text{ ref } q \oplus n \rightarrow q \neq \text{SAFE} \]

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

\[ T \text{ ref } q' \text{ ref } q \land q = \text{DYNQ} \rightarrow q' = \text{DYNQ} \]
Constraint Normalization/Solving

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

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

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

T ref q ⊕ n → q != SAFE

T₁ ref q₁ <: T₂ ref q₂ →
(q₁=q₂ v (q₁=SEQ ∧ q₂=SAFE)) ∧
(q₁=q₂=DYNQ v T₁≈T₂)

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
Inference Example: DYNQ

```c
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

T ref q1 <: T ref q2 →
(q1=q2 v (q1=SEQ ∧ q2=SAFE)) ∧
(q1=q2=DYNQ v T1≈T2)

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

DYNAMIC wild;
DYNAMIC crazy = wild;
```
## Experimentation

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of code</th>
<th>Orig. time</th>
<th>CCured sf/sq/d</th>
<th>Purify ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECINT95</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>1590</td>
<td>9.586s</td>
<td>87/12/0</td>
<td>1.25</td>
</tr>
<tr>
<td>go</td>
<td>29315</td>
<td>1.191s</td>
<td>96/4/0</td>
<td>2.01</td>
</tr>
<tr>
<td>jpeg</td>
<td>31371</td>
<td>0.963s</td>
<td>36/1/62</td>
<td>2.15</td>
</tr>
<tr>
<td>li</td>
<td>7761</td>
<td>0.176s</td>
<td>93/6/0</td>
<td>1.86</td>
</tr>
<tr>
<td>Olden</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bh</td>
<td>2053</td>
<td>2.992s</td>
<td>80/18/0</td>
<td>1.53</td>
</tr>
<tr>
<td>bisort</td>
<td>707</td>
<td>1.696s</td>
<td>90/10/0</td>
<td>1.03</td>
</tr>
<tr>
<td>em3d</td>
<td>557</td>
<td>0.371s</td>
<td>85/15/0</td>
<td>2.44</td>
</tr>
<tr>
<td>health</td>
<td>725</td>
<td>2.769s</td>
<td>93/7/0</td>
<td>0.94</td>
</tr>
<tr>
<td>mst</td>
<td>617</td>
<td>0.720s</td>
<td>87/10/0</td>
<td>2.05</td>
</tr>
<tr>
<td>perimeter</td>
<td>395</td>
<td>4.711s</td>
<td>96/4/0</td>
<td>1.07</td>
</tr>
<tr>
<td>power</td>
<td>763</td>
<td>1.647s</td>
<td>95/6/0</td>
<td>1.31</td>
</tr>
<tr>
<td>treeadd</td>
<td>385</td>
<td>0.613s</td>
<td>85/15/0</td>
<td>1.47</td>
</tr>
<tr>
<td>tsp</td>
<td>561</td>
<td>3.093s</td>
<td>97/4/0</td>
<td>1.15</td>
</tr>
</tbody>
</table>
Bugs Found

• compress and jpeg 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 \textit{statically}
  - Annotate small set of typed elements as tainted or untainted
  - Infer taintedness for other elements throughout 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

cchar *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:

\[
\begin{align*}
Q1 <: Q2 & \quad T1 <: T2 \\
\hline
Q1 T1 <: Q2 T2 \\
Q1 <: Q2 & \quad T1 = T2 \\
\hline
Q1 \text{ ptr}(T1) <: Q1 \text{ ptr}(T2)
\end{align*}
\]
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, ...);

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