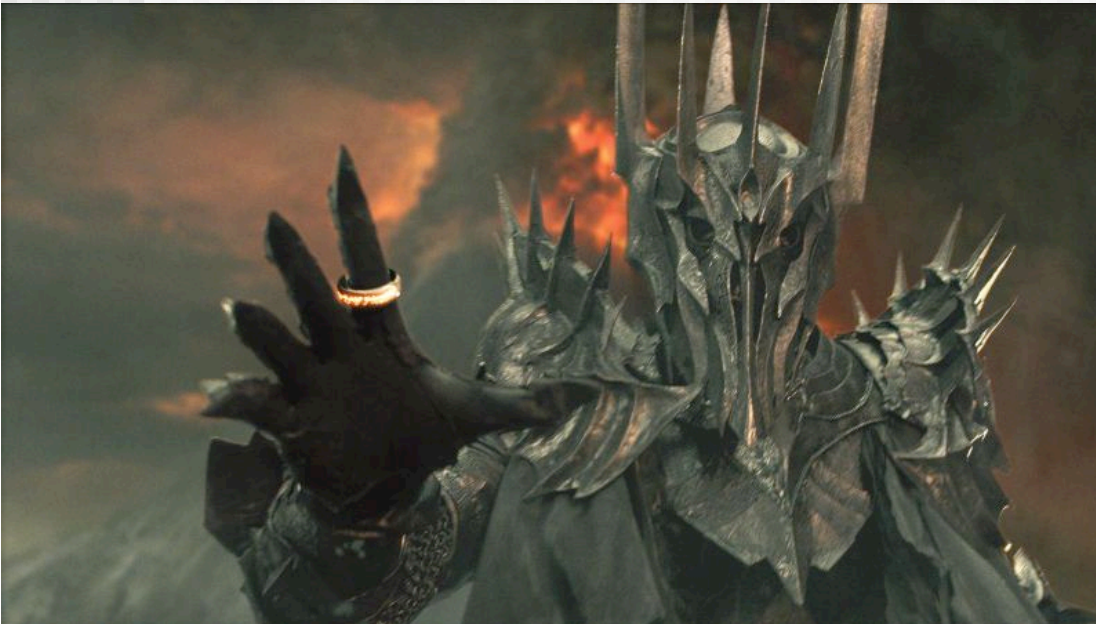
CCured

CMSC 838F – 11 Feb 2008

Michael Lam

Lord of the Pointers

"One Pointer to dereference them all and in a buffer overflow segfault them."



Some slides from
"CCured: Taming C Pointers"
by Wes Wiemer (2002)

Problem

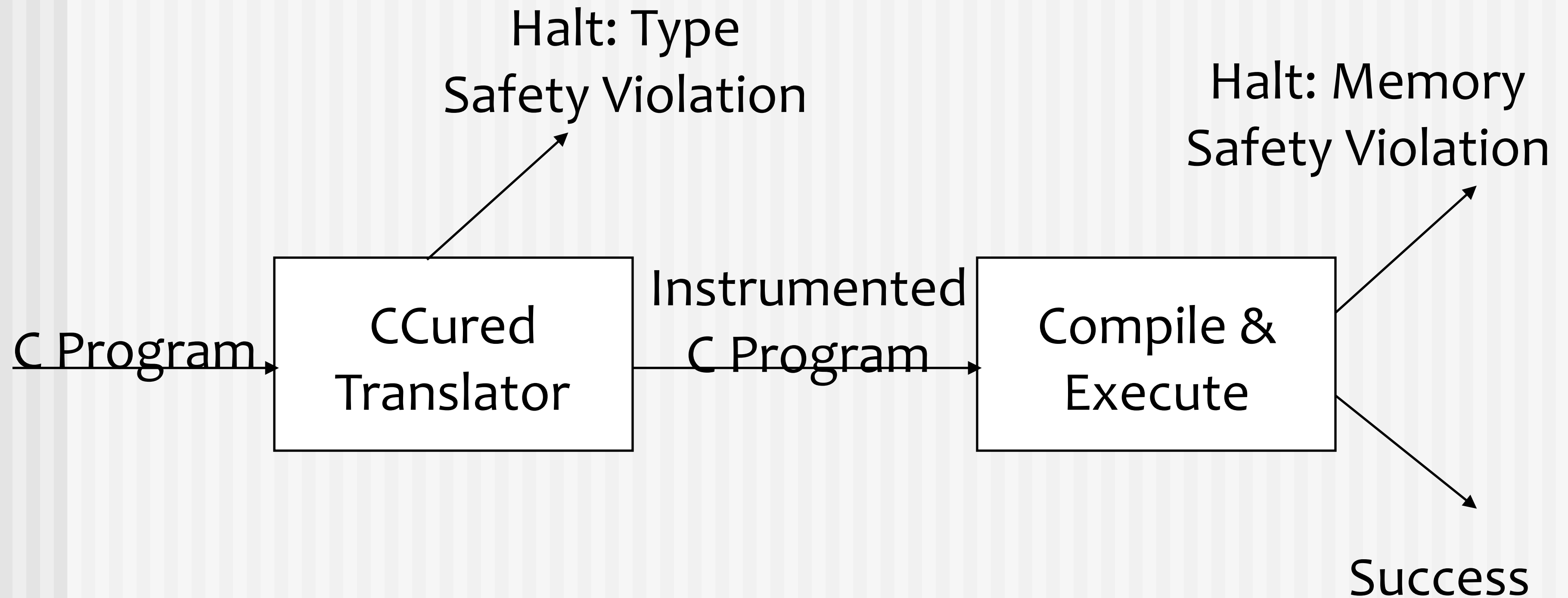
- Pointers -- ultimate power!
  - Speed of direct memory access
  - Terrible price: array overflows
    - Errors and instability
    - Memory corruption
    - Security holes

- *But we'd really like to use all our legacy C code!*
- Can we tame the pointer problems and "cure" the legacy code?

Proposed Solution

- CCured
  - Type system for array pointers
  - Static type inference
  - Run-time checks
  - Ensure no unsafe pointer use
  - Minimal user effort
  - Make C “feel” safe!

CCured Overview



CCured Overview

- Added bookkeeping for pointers
  - Base and size of allocated memory (“home”)
  - Range of pointer types
    - Safe (immutable)
    - Sequence (iterator)
    - Dynamic (polymorphic)

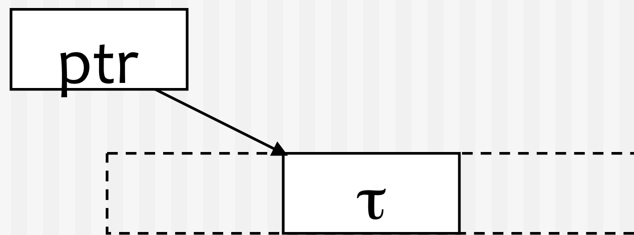
faster

more capable

- Infer the fastest safe representation for each pointer by examining constraints

Safe Pointers

SAFE pointer to type τ

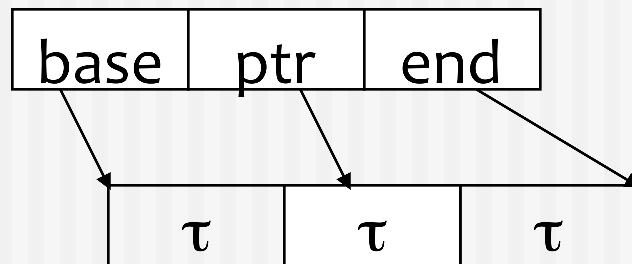


On use:
- null check

Capabilities:
- dereference

Sequence Pointers

SEQ pointer to type τ

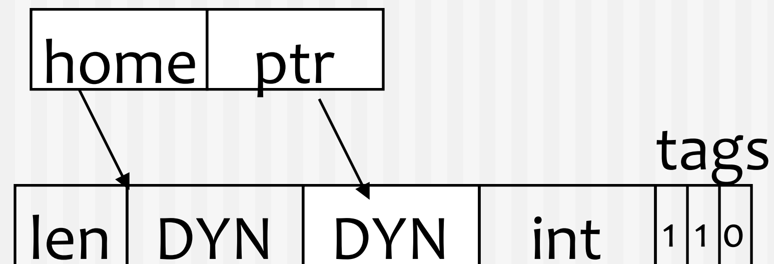


On use:
- null check
- bounds check

Capabilities:
- dereference
- pointer arithmetic

Dynamic Pointers

DYN pointer



On use:
- null check
- bounds check
- tag check/update

Capabilities:
- dereference
- pointer arithmetic
- arbitrary typecasts

Simple Language

$$
\begin{array}{llll}
\text{Types:} & \tau & ::= & \texttt{int} \mid \tau\ \texttt{ref SAFE} \mid \tau\ \texttt{ref SEQ} \\
 & & \mid & \texttt{DYNAMIC} \\
\text{Expressions:} & e & ::= & x \mid n \mid e_1\ \textbf{op}\ e_2 \mid (\tau)e \\
 & & \mid & e_1 \oplus e_2 \mid\ !e \\
\text{Commands:} & c & ::= & \texttt{skip} \mid c_1; c_2 \mid e_1 := e_2
\end{array}
$$

Type System

- Three judgments:
  - Expression types: $\Gamma \vdash e : \tau$
  - Command types: $\Gamma \vdash c$
  - Convertibility: $\tau \leq \tau'$
- NOT transitive:
  - DYNAMIC ≤ int ≤ DYNAMIC not identity

Type Rules

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:

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

Figure 5

Operational Semantics

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:

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

Figure 6

Operational Semantics

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:

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

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

Figure 6

Progress and Preservation

Theorem 1 (Progress and type preservation) *If* $\Gamma \vdash e : \tau$ *and* $\Sigma \in \|\Gamma\|_H$ *and* $WF(M_H)$ *then either* $\Sigma, M_H \vdash e \Downarrow \mathit{CheckFailed}$ *or else* $\Sigma, M_H \vdash e \Downarrow v$ *and* $v \in \|\tau\|_H$.

Theorem 2 (Progress for commands) *If* $\Gamma \vdash c$ *and* $\Sigma \in \|\Gamma\|_H$ *and* $WF(M_H)$ *then either* $\Sigma, M_H \vdash c \Longrightarrow \mathit{CheckFailed}$ *or else* $\Sigma, M_H \vdash c \Longrightarrow M'_H$ *and* $WF(M'_H)$.

Type Inference

- Pointer type: τ ref q
  - Where $q \in$ {SAFE, SEQ, DYNQ}
- Inference process
  - Introduce qualifier variable for each occurrence of a pointer type constructor
  - Scan the program and collect constraints
    - Constraint judgments: $\Gamma \vdash e : \tau \hookrightarrow C$
  - Solve the system of constraints
  - Apply substitutions to the original program

Constraint Generation Rules

Expressions and commands:

$$\frac{\Gamma \vdash e_1 : \tau \ \mathtt{ref}\ q \mapsto C_1 \quad \Gamma \vdash e_2 : \mathtt{int} \mapsto C_2}{\Gamma \vdash e_1 \oplus e_2 : \tau \ \mathtt{ref}\ q \mapsto C_1 \cup C_2 \cup \{q \neq \mathtt{SAFE}\}} \qquad \frac{\Gamma \vdash e : \tau' \mapsto C \quad \tau' \leq \tau \mapsto C'}{\Gamma \vdash (\tau)e : \tau \mapsto C \cup C'} \qquad \frac{}{\Gamma \vdash (\tau \ \mathtt{ref}\ q)0 : \tau \ \mathtt{ref}\ q \mapsto \emptyset}$$

$$\frac{\Gamma \vdash e : \tau \ \mathtt{ref}\ q \mapsto C}{\Gamma \vdash\, !e : \tau \mapsto C} \qquad \frac{\Gamma \vdash e_1 : \tau \ \mathtt{ref}\ q \mapsto C_1 \quad \Gamma \vdash e_2 : \tau_2 \mapsto C_2 \quad \tau_2 \leq \tau \mapsto C_3}{\Gamma \vdash e_1 := e_2 \mapsto C_1 \cup C_2 \cup C_3}$$

Convertibility:

$$\frac{}{\tau \leq \mathtt{int} \mapsto \emptyset} \qquad \frac{}{\mathtt{int} \leq \tau \ \mathtt{ref}\ q \mapsto \{q \neq \mathtt{SAFE}\}} \qquad \frac{}{\tau_1 \ \mathtt{ref}\ q_1 \leq \tau_2 \ \mathtt{ref}\ q_2 \mapsto \{q_1 \preceq q_2\} \cup \{q_1 = q_2 = \mathtt{DYNQ} \vee \tau_1 \approx \tau_2\}}$$

Figure 7

Constraint Solver

- Constraint types
  - INDYN: q = DYNQ
  - EQ: $q = q'$
  - CONV: $q \leq q'$
  - POINTSTO: q = DYNQ $\Rightarrow$ q' = DYNQ
  - ARITH: $q \neq$ SAFE
- Algorithm:
  - Start with INDYN information
  - Propagate DYNQ via EQ, CONV, & POINTSTO
  - Everything involved in ARITH to SEQ
  - Everything else is SAFE

Special Handling

- Specially-handled functions
  - Function polymorphism
  - Argument constraints
  - Variable argument functions
  - Examples:
    - `malloc/free`
    - `memcpy`
    - string functions
    - `printf`

Points of Interest

- Function pointers are (usually) easy!
- DYNAMIC pointers are contagious
- Boehm-Demers-Weiser garbage collector

Source Changes

- `sizeof(type)` doesn’t work
  - Use `sizeof(expression)` instead
- Deallocation of stack pointers
  - Annotation to enforce heap allocation
- Interoperability with third-party libraries
  - Wrapper functions (ex. `__ptrof`)
  - 120 functions from standard C library

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

Figure 8

Issues

- Analysis is conservative
  - Some safe operations are not allowed
  - Examples:
    - Cast from SAFE to int to SAFE
    - Tagged union types (ie. OOP polymorphism)
- Performance hit
  - 30-150% slowdown
- Requires source code

TOPLAS Changes

- DYNAMIC → WILD
- Physical subtyping for upcasts
- New RTTI pointer type for downcasts
- New FSEQ pointer type for forward iterators
- Faster (only 3-85% slowdown)
- More extensive testing
  - More benchmarks and applications
  - Linux kernel modules
  - Apache modules

TOPLAS Rules

Expression	Typing Premises	Run-time checks and translation
Memory Reads		
$*x$	$x : \tau * \texttt{SAFE}$	$\mathsf{assert}(x.p \neq \texttt{null});\ *(x.p)$
$*x$	$x : \tau * \texttt{SEQ}$	$\mathsf{assert}(x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau));\ *(x.p)$
$*x$	$x : \texttt{int} * \texttt{WILD}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 1);\ *(x.p)$
$*x$	$x : \tau * \texttt{WILD} * \texttt{WILD}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 2);$ $\mathsf{assert}(\mathsf{tag}(x.b, x.p + 1) == 1);\ *(x.p)$
Memory Writes		For all $*x = y$, check $*x$ as above, except omit the tag check. The type of y must match that of $*x$.
$*x = y$	$x : \texttt{int} * \texttt{SAFE}$	$*(x.p) = y$
$*x = y$	$x : \texttt{int} * \texttt{SEQ}$	$*(x.p) = y$
$*x = y$	$x : \tau * q * \texttt{SAFE}$	$\mathsf{assert}(\mathsf{NotStackPointer}(y));\ *(x.p) = y$
$*x = y$	$x : \tau * q * \texttt{SEQ}$	$\mathsf{assert}(\mathsf{NotStackPointer}(y));\ *(x.p) = y$
$*x = y$	$x : \texttt{int} * \texttt{WILD}$	$\mathsf{tag}(x.b, x.p) = 0;\ *(x.p) = y$
$*x = y$	$x : \tau * \texttt{WILD} * \texttt{WILD}$	$\mathsf{assert}(\mathsf{NotStackPointer}(y));\ \mathsf{tag}(x.b, x.p) = 0;$ $\mathsf{tag}(x.b, x.p + 1) = 1;\ *(x.p) = y$

Expression	Typing Premises	Run-time checks and translation
Allocation		
$(\tau * \mathtt{SAFE})\mathtt{malloc}(n)$		$\mathsf{assert}(n \geq \mathtt{sizeof}(\tau));\ \{\ p = \mathtt{zeroed_malloc}(n)\ \}$
$(\tau * \mathtt{SEQ})\mathtt{malloc}(n)$		$\{\ p = \mathtt{zeroed_malloc}(n),\ b = p,\ e = p + n\ \}$
$(\tau * \mathtt{WILD})\mathtt{malloc}(n)$		$\mathtt{let}\ m = \mathtt{zeroed_malloc}(1 + n + \lceil n/\mathtt{bitsperword}\rceil);$ $m[0] = n;\ \{\ p = m + 1,\ b = m + 1\ \}$
Type Casts		
$(\mathtt{int})x$	$x : \tau * \mathtt{SAFE}$	$x.p$
$(\mathtt{int})x$	$x : \tau * \mathtt{SEQ}$	$x.p$
$(\mathtt{int})x$	$x : \tau * \mathtt{WILD}$	$x.p$
$(\tau' * \mathtt{SAFE})x$	x is the literal 0	$\{\ p = \mathtt{null}\ \}$
$(\tau' * \mathtt{SEQ})x$	$x : \mathtt{int}$	$\{\ p = x,\ b = \mathtt{null},\ e = \mathtt{null}\ \}$
$(\tau' * \mathtt{WILD})x$	$x : \mathtt{int}$	$\{\ p = x,\ b = \mathtt{null}\ \}$
$(\tau' * \mathtt{WILD})x$	$x : \tau * \mathtt{WILD}$	x
$(\tau' * \mathtt{SAFE})x$	$x : \tau * \mathtt{SAFE},\ \tau \approx \tau'$	x
$(\tau' * \mathtt{SEQ})x$	$x : \tau * \mathtt{SEQ},\ \tau \approx \tau'$	x
$(\tau' * \mathtt{SAFE})x$	$x : \tau * \mathtt{SEQ},\ \tau \approx \tau'$	$\mathsf{assert}(x.p == \mathtt{null}\ \Vert$ $x.b \leq x.p \leq x.e - \mathtt{sizeof}(\tau'));\ \{\ p = x.p\ \}$
$(\tau' * \mathtt{SEQ})x$	$x : \tau * \mathtt{SAFE},\ \tau \approx \tau'$	$\{\ p = x.p,\ b = x.p,\ e = x.p + \mathtt{sizeof}(\tau)\ \}$
Arithmetic (including aggregate access)		
$x_1 + x_2$	$x_1 : \tau * \mathtt{SEQ},\ x_2 : \mathtt{int}$	$\{\ p = x_1.p + x_2 * \mathtt{sizeof}(\tau),\ b = x_1.b,\ e = x_1.e\ \}$
$x_1 + x_2$	$x_1 : \tau * \mathtt{WILD},\ x_2 : \mathtt{int}$	$\{\ p = x_1.p + x_2 * \mathtt{sizeof}(\tau),\ b = x_1.b\ \}$
$\&(x \rightarrow f_2) : \tau_2 * \mathtt{SAFE}$	$x : \mathtt{struct}\{\tau_1 f_1;\ \ \tau_2 f_2;\} * \mathtt{SAFE}$	$\mathsf{assert}(x.p \neq \mathtt{null});\ \{p = \&(x.p \rightarrow f_2)\}$
$\&(x \rightarrow f_2) : \tau_2 * \mathtt{SAFE}$	$x : \mathtt{struct}\{\tau_1 f_1;\ \ \tau_2 f_2;\} * \mathtt{SEQ}$	$\mathsf{assert}(x.b \leq x.p \leq x.e - \mathtt{sizeof}(\tau_1) - \mathtt{sizeof}(\tau_2));$ $\{\ p = \&(x.p \rightarrow f_2)\ \}$
$\&(x \rightarrow f_2) : \tau_2 * \mathtt{WILD}$	$x : \mathtt{struct}\{\tau_1 f_1;\ \ \tau_2 f_2;\} * \mathtt{WILD}$	$\{\ p = \&(x.p \rightarrow f_2),\ b = x.b\ \}$

TOPLAS Results

Module Name	Lines of code	% sf/sq/w/rt	**CCured Ratio**	Lines Changed	Trusted Casts
asis	149	72/28/0/0	**0.96**	2	0
expires	525	77/23/0/0	**1.00**	5	0
gzip	11648	85/15/0/0	**0.94**	136	0
headers	281	90/10/0/0	**1.00**	6	0
info	786	86/14/0/0	**1.00**	62	3
layout	309	82/18/0/0	**1.01**	37	0
random	131	85/15/0/0	**0.94**	47	0
urlcount	702	87/13/0/0	**1.02**	41	0
usertrack	409	81/19/0/0	**1.00**	6	0
WebStone	n/a	n/a	**1.04**		

TOPLAS Results

Name	Lines of code	% sf/sq/w/rt	**CCured ratio**	Valgrind ratio	Memory ratio	Lines Changed	Trusted Casts
pcnet32	1661	92/8/0/0	**0.99**			66	0
ping			**1.00**				
sbull	1013	85/15/0/0	**1.00**			18	0
seeks			**1.03**				
ftpd	6553	79/12/9/0	**1.01**	9.42	1.32	28	0
OpenSSL	177426	67/27/0/6	**1.40**	42.9	1.81	2000	2
cast			**1.87**	48.7	3.56		
bn			**1.01**	72.0	3.47		
OpenSSH	65250	70/28/0/3				365	14
client			**1.22**	22.1	3.88		
server			**1.15**		4.53		
sendmail	105432	65/34/0/1	**1.46**	122	2.32	904	4
bind	336660	79/21/0/0	**1.81**	129	3.84	224	237
tasks			**1.11**	81.4	1.86		
sockaddr			**1.50**	110	1.00		

Name	Lines of code	% sf/sq/w/rt	**CCured ratio**	Purify ratio	Memory ratio	Lines Changed	Trusted Casts
SPECINT95							
compress	1590	90/10/0/0	**1.17**	28	1.01	36	0
go	29315	94/06/0/0	**1.22**	51	1.60	117	0
ijpeg	31371	80/20/0/1	**1.50**	30	1.05	1103	0
li	7761	80/20/0/0	**1.70**	50	2.00	600	0
Olden							
bh	2053	80/20/0/0	**1.44**	94	1.55	271	0
bisort	707	93/07/0/0	**1.09**	42	2.00	469	0
em3d	557	93/06/0/0	**1.45**	7	1.39	22	0
health	725	93/07/0/0	**1.07**	25	1.90	449	0
mst	617	97/03/0/0	**1.87**	5	1.15	44	0
perimeter	395	100/0/0/0	**1.10**	544	1.97	3	0
power	763	94/06/0/0	**1.29**	53	1.58	8	0
treeadd	385	96/04/0/0	**1.15**	500	2.61	14	0
tsp	561	100/0/0/0	**1.06**	66	2.54	7	0
Ptrdist-1.1							
anagram	661	88/12/0/0	**1.43**	34	1.52	37	0
bc	7323	77/23/0/0	**9.91**	100	2.18	58	0
ft	2194	98/02/0/0	**1.03**	12	2.12	59	0
ks	793	88/12/0/0	**1.11**	31	1.65	22	0
yacr2	3999	88/12/0/0	**1.56**	26	1.63	30	0

Case Studies

	LOC	Hours	LOC changed
OpenSSL/SSH	235 K	70	1700
sendmail	105 K	20	200
bind	330 K	5	88
Apache	228 K	2 months (*)	550
monkeyd	5 K	3	30

(*) CCured beginner

Conclusions

- Complementary approach
  - Static analysis
  - Dynamic analysis
- CCured
  - Simple type system
  - Works on “real” C
  - Performance is “good”
- “Surprising” result: most pointer use in C is safe!

Future Work

- Resolve outstanding issues
- Automatic garbage collection
- Improve error reporting
- Extend to C++

Thank you!

- Questions?