Static Defenses against Memory Corruption

Vitaly Shmatikov
Reading Assignment

◆ Dor, Rodeh, Sagiv. “CSSV: Towards a realistic tool for statically detecting all buffer overflows in C” (PLDI 2003).
Static Analysis

- Goal: catch buffer overflow bugs by analyzing the source code of the program
  - Typically at compile-time, but also binary analysis
- Static analysis is necessarily imprecise
  - Soundness: finds all instances of buffer overflow
    - Problem: false positives (good code erroneously flagged)
  - Completeness: every reported problem is indeed an instance of buffer overflow
    - Problem: false negatives (misses some buffer overflows)
  - No technique is both sound and complete (why?)
  - Maybe don’t need either...
Static vs. Dynamic

◆ Both static and dynamic approaches have their advantages and disadvantages (what are they?)
◆ Hybrid approaches (example: CCured)
  • Try to verify absence of memory errors statically, then insert runtime checks where static verification failed
◆ Performance and usability are always important
  • Does source code need to be modified?
  • Does source code need to be recompiled?
  • How is backward compatibility (if any) achieved?
    – Rewriting binaries vs. special runtime environment
BOON

- Treat C strings as abstract data types
  - Assume that C strings are accessed only through library functions: strcpy, strcat, etc.
  - Pointer arithmetic is greatly simplified (what does this imply for soundness?)
- Characterize each buffer by its allocated size and current length (number of bytes in use)
- For each of these values, statically determine acceptable range at each point of the program
  - Done at compile-time, thus necessarily conservative (what does this imply for completeness?)
Safety Condition

- Let $s$ be some string variable used in the program.
- $\text{len}(s)$ is the set of possible lengths.
  - Why is $\text{len}(s)$ not a single integer, but a set?
- $\text{alloc}(s)$ is the set of possible values for the number of bytes allocated for $s$.
  - Is it possible to compute $\text{len}(s)$ and $\text{alloc}(s)$ precisely at compile-time?
- At each point in program execution, want $\text{len}(s) \leq \text{alloc}(s)$. 
Integer Constraints

◆ Every string operation is associated with a constraint describing its effects

\begin{align*}
\text{strcpy(dst,src)} & \quad \text{len(src)} \subseteq \text{len(dst)} \\
\text{strncpy(dst,src,n)} & \quad \text{min(len(src),n)} \subseteq \text{len(dst)} \\
\text{gets(s)} & \quad [1,\infty] \subseteq \text{len(s)} \\
\text{s=“Hello!”} & \quad 7 \subseteq \text{len(s)}, 7 \subseteq \text{alloc(s)} \\
\text{s[n]=‘\0’} & \quad \text{min(len(s),n+1))} \subseteq \text{len(s)} \\
\end{align*}

and so on

Does this fully capture what strncpy does?

Range of possible values
Constraint Generation Example

char buf[128];
while (fgets(buf, 128, stdin)) {
    if (!strchr(buf, '\n')) {
        char error[128];
        sprintf(error, "Line too long: %s\n", buf);
        die(error);
    }
}

128 ⊆ alloc(buf)
[1,128] ⊆ len(buf)
128 ⊆ alloc(error)
len(buf)+16 ⊆ len(error)
Imprecision

- Simplifies pointer arithmetic and pointer aliasing
  - For example, $q = p + j$ is associated with this constraint: $\text{alloc}(p) - j \subseteq \text{alloc}(q)$, $\text{len}(p) - j \subseteq \text{len}(q)$
  - This is unsound (why?)
- Ignores function pointers
- Ignores control flow and order of statements
  - Consequence: every non-trivial `strcat()` must be flagged as a potential buffer overflow (why?)
- Merges information from all call sites of a function into one variable
Constraint Solving

• “Bounding-box” algorithm (see paper)
  • Imprecise, but scalable: sendmail (32K LoC) yields a system with 9,000 variables and 29,000 constraints

• Suppose analysis discovers len(s) is in [a,b] range, and alloc(s) is in [c,d] range at some point
  • If b ≤ c, then code is “safe”
    – Does not completely rule out buffer overflow (why?)
  • If a > d, then buffer overflow always occurs here
  • If ranges overlap, overflow is possible

• Ganapathy et al.: model and solve the constraints as a linear program (see paper)
BOON: Practical Results

- Found new vulnerabilities in real systems code
  - Exploitable buffer overflows in nettools and sendmail
- Lots of false positives, but still a dramatic improvement over hand search
  - sendmail: over 700 calls to unsafe string functions, of them 44 flagged as dangerous, 4 are real errors
  - Example of a false alarm:
    
    ```c
    if (sizeof from < strlen(e->e_from.q_paddr)+1) break;
    strcpy(from, e->e_from.q_paddr);
    ```
Context-Insensitivity is Imprecise

foo () {
    int x;
    x = foobar(5);
}

bar () {
    int y;
    y = foobar(30);
}

int foobar (int z) {
    int i;
    i = z + 1;
    return i;
}

False path
Result: \( x = y = [6..31] \)
Adding Context Sensitivity

- Make user functions context-sensitive
  - For example, wrappers around library calls
- Inefficient method: constraint inlining
  - ![Smiley] Can separate calling contexts
  - ![Sad] Large number of constraint variables
  - ![Sad] Cannot support recursion
- Efficient method: procedure summaries
  - Summarize the called procedure
  - Insert the summary at the callsite in the caller
  - Remove false paths
Context-Sensitive Analysis

foo () {
   int x;
   x = foobar(5);
}

bar () {
   int y;
   y = foobar(30);
}

int foobar (int z) {
   int i;
   i = z + 1;
   return i;
}

x = 5 + 1
y = 30 + 1

Summary: i = z + 1
foo () {
  int x;
  x = foobar(5);
}

bar () {
  int y;
  y = foobar(30);
}

int foobar (int z) {
  int i;
  i = z + 1;
  return i;
}

No False Paths

[Ganapathy et al.]

Jump functions

Constraints
x = [6..6]
y = [31..31]
i = [6..31]
Computing Procedure Summaries

- If function produces only difference constraints, reduces to an all-pairs shortest-path problem
- Otherwise, Fourier-Motzkin variable elimination
- Tradeoff between precision and efficiency
  - Constraint inlining: rename local variables of the called function at each callsite
    - Precise, but a huge number of variables and constraints
  - Procedure summaries: merge variables across callsites
    - For example, constraint for i in the foobar example
Off-by-one Bug in sendmail-8.9.3

- `orderq()` reads a file from the queue directory, copies its name into `d->d_name` and `w->w_name`  
  - As long as 21 bytes, including the ‘\0’ terminator
- `runqueue()` calls `dowork(w->w_name+2,...)`  
  `dowork()` stores its first argument into `e->e_id`
- `queueuname()` concatenates "qf" and `e->e_id`, copies the result into 20-byte `dfname` buffer

- Wagner et al.: a pointer to a structure of type T can point to all structures of type T  
  - Finds the bug, but do you see any issues?
- Ganapathy et al.: precise points-to analysis
Goal: **sound** static detection of buffer overflows

- What does this mean?

Separate analysis for each procedure

“Contracts” specify procedure’s pre- and post-conditions, potential side effects

- Analysis only meaningful if contracts are correct

Flow-insensitive “points-to” pointer analysis

Transform C into a procedure over integers, apply integer analysis to find variable constraints

- Any potential buffer overflow in the original program violates an “assert” statement in this integer program
Example: strcpy Contract

char* strcpy(char* dst, char *src)

requires string(src) ∧
alloc(dst) > len(src)

modifies dst.strlen, dst.is_nullt

ensures len(dst) = = pre@len(src) ∧
return = = pre@dst
```c
#define BUFSIZ 1024
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
    char temp[BUFSIZ];
    int i;
    for (i=0; &buf[i] < cp; ++i){
        temp[i] = buf[i];
    }
    strcpy (&temp[i], "(long)");
    strcpy (&temp[i + 6], cp);
    strcpy (buf, temp);
    return cp + 6;
}
```

Example: `insert_long()`

[Dor, Rodeh, Sagiv]
```c
#define BUFSIZ 1024
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
    char temp[BUFSIZ];
    int i;
    for (i=0; &buf[i] < cp; ++i){
        temp[i] = buf[i];
    }
    strcpy (&temp[i],"(long)");
    strcpy (&temp[i + 6], cp);
    strcpy (buf, temp);
    return cp + 6;
}
```

**insert_long() Contract**

```c
char * insert_long(char *cp)
    requires  string(cp) ∧
                   buf ≤ cp < buf + BUFSIZ
    modifies  cp.strlen
    ensures
           cp.strlen = = pre[cp.strlen] + 6
                   ∧
           return_value = = cp + 6 ;
```
Pointer Analysis

Goal: compute points-to relation
- This is highly nontrivial for C programs (see paper)
- Pointer arithmetic, typeless memory locations, etc.

Abstract interpretation of memory accesses
- For each allocation, keep base and size in bytes
- Map each variable to their abstract locations
- We’ll see something similar in CCured

Sound approximation of may-point-to
- For each pointer, set of abstract locations it can point to
- More conservative than actual points-to relation

[Dor, Rodeh, Sagiv]
C2IP: C to Integer Program

- Integer variables only
- No function calls
- Non-deterministic
- Constraint variables
- Update statements
- Assert statements

Based on points-to information

- Any string manipulation error in the original C program is guaranteed to violate an assertion in integer program
Transformations for C Statements

For abstract location \( l \),
- \( l.\text{val} \) - potential values stored in the locations represented by \( l \)
- \( l.\text{offset} \) - potential values of the pointers represented by \( l \)
- \( l.\text{aSize} \) - allocation size
- \( l.\text{isNullt} \) - null-terminated?
- \( l.\text{len} \) - length of the string

For pointer \( p \),
- \( l_p \) - its location
- \( r_p \) - location it points to

<table>
<thead>
<tr>
<th>C Construct</th>
<th>IP Statements</th>
</tr>
</thead>
</table>
| \( p = \text{Alloc}(i); \) | \( l_p.\text{offset} := 0; \)
|               | \( r_p.\text{aSize} := l_i.\text{val}; \)
|               | \( r_p.\text{isNullt} := \text{false}; \)                                                                                                    |
| \( p = q + i; \) | \( l_p.\text{offset} := l_q.\text{offset} + l_i.\text{val}; \)                                                                                             |
| \( *p = c; \)  | if \( c = 0 \) then \{ \)
|               | \( r_p.\text{len} := l_p.\text{offset}; \)
|               | \( r_p.\text{isNullt} := \text{true}; \} else \)
|               | if \( r_p.\text{isNullt} \land l_p.\text{offset} = r_p.\text{len} \) then \( l_p.\text{isNullt} := \text{unknown}; \) |
| \( c = *p; \)  | if \( r_p.\text{isNullt} \land l_p.\text{offset} = r_p.\text{len} \) then \( l_c.\text{val} := 0; \)
|               | else \( l_c.\text{val} := \text{unknown}; \)                                                                                                 |
| \( g(a_1, a_2, \ldots, a_m); \) | \( \text{mod}[g](a_1, a_2, \ldots, a_m); \)                                                                                                    |
| \( *p == 0 \)   | \( r_p.\text{isNullt} \land r_p.\text{len} = l_p.\text{offset} \)                                                                                |
| \( p > q \)     | \( l_p.\text{offset} > l_q.\text{offset} \)                                                                                                    |
| \( p.\text{alloc} \) | \( r_p.\text{aSize} = l_p.\text{offset} \)                                                                                                    |
| \( p.\text{offset} \) | \( l_p.\text{offset} \)                                                                                                                         |
| \( p.\text{isNullt} \) | \( r_p.\text{isNullt} \)                                                                                                                        |
| \( p.\text{strlen} \) | \( r_p.\text{len} = l_p.\text{offset} \)                                                                                                      |
Correctness Assertions

[Dor, Rodeh, Sagiv]

<table>
<thead>
<tr>
<th>C Exp.</th>
<th>Generated IP Condition</th>
</tr>
</thead>
</table>
| *p     | \( l_p.offset \geq 0 \land \)  
          \( ((r_p.is\_nullt \land l_p.offset \leq r_p.len) \lor \)  
          \( (-r_p.is\_nullt \land l_p.offset < r_p.aSize)) \) |
| p + i  | \( l_p.offset + l_i.val \geq 0 \land \)  
          \( l_p.offset + l_i.val \leq r_p.aSize \) |

All dereferenced pointers point to valid locations

Results of pointer arithmetic are valid
Example

assert (5 <= q.alloc && (!q.is_nullt || 5 <= q.len))

p = q + 5;

p.offset = q.offset + 5;
Nondeterminism

\[ *p = 0; \]

\[
\begin{align*}
\text{if (...)} & \{ \\
& \text{alloc}_1.\text{len} = p.\text{offset}; \\
& \text{alloc}_1.\text{is}_\text{nullt} = \text{true}; \} \\
\text{else } & \{ \\
& \text{alloc}_5.\text{len} = p.\text{offset}; \\
& \text{alloc}_5.\text{is}_\text{nullt} = \text{true}; \}
\end{align*}
\]
Integer Analysis

- Interval analysis not enough
  - Loses relationships between variables
- Infer variable constraints using abstract domain of polyhedra [Cousot and Halbwachs, 1978]
  - $a_1 \cdot \text{var}_1 + a_2 \cdot \text{var}_2 + \ldots + a_n \cdot \text{var}_n \leq b$

\[
\begin{align*}
V &= \langle (1,2) \ (2,1) \rangle \\
R &= \langle (1,0) \ (1,1) \rangle
\end{align*}
\]
```c
#define BUFSIZ 1024
#include "insert_long.h"
char buf[BUFSIZ];
char * insert_long (char *cp) {
  char temp[BUFSIZ];
  int i;
  for (i=0; &buf[i] < cp; ++i){
    temp[i] = buf[i];
  }
  strcpy (&temp[i], "(long)");
  strcpy (&temp[i + 6], cp);
  strcpy (buf, temp);
  return cp + 6;
}
```

[Dor, Rodeh, Sagiv]
Integer Analysis of insert_long()

```
buf.offset = 0
temp.offset = 0
0 ≤ cp.offset = i
i ≤ sbuf.len < sbuf.msize
sbuf.msize = 1024
stemp.msize= 1024
```

```
assert(0 ≤ i < stemp.msize - 6);  // strcpy(&temp[i],"(long)");
```

Potential violation when cp.offset ≥ 1018
Goal: make legacy C code type-safe

- Treat C as a mixture of a strongly typed, statically checked language and an “unsafe” language checked at runtime
  - All values belong either to “safe,” or “unsafe” world

Combination of static and dynamic checking
- Check type safety at compile-time whenever possible
- When compile-time checking fails, compiler inserts run-time checks in the code
- Fewer run-time checks ⇒ better performance
Safe Pointers

- Either NULL, or a valid address of type T
- Aliases are either safe pointers, or sequence pointers of base type T

What is legal to do with a safe pointer?
- Set to NULL
- Cast from a sequence pointer of base type T
- Cast to an integer

What runtime checks are required?
- Not equal to NULL when dereferenced
Sequence Pointers

◆ At runtime, either an integer, or points to a known memory area containing values of type T
◆ Aliases are safe, or sequence ptrs of base type T
◆ What is legal to do with a sequence pointer?
  • Perform pointer arithmetic
  • Cast to a safe pointer of base type T
  • Cast to or from an integer
◆ What runtime checks are required?
  • Points to a valid address when dereferenced
    – Subsumes NULL checking
  • Bounds check when dereferenced or cast to safe ptr
Dynamic Pointers

- At runtime, either an integer, or points to a known memory area containing values of type T
- The memory area to which it points has tags that distinguish integers from pointers
- Aliases are dynamic pointers
- What is legal to do with a dynamic pointer?
  - Perform pointer arithmetic
  - Cast to or from an integer or any dynamic pointer type
- Runtime checks of address validity and bounds
  - Maintain tags when reading & writing to base area
Example

```c
int **a;
int i;
int acc;
int **p;
int *e;
acc = 0;
for (i = 0; i < 100; i++) {
    p = a + i;
    e = *p;
    while (((int) e % 2 == 0) {
        e = *(int **) e;
    }
    acc += ((int) e >> 1);
}
```

- **sequence pointer**
- **safe pointer**
- **dynamic pointer**
Modified Pointer Representation

- Each allocated memory area is called a home (H), with a starting address h and a size.
- Valid runtime values for a given type:
  - Integers: $||\text{int}|| = N$
  - Safe pointers: $||\tau\text{ ref SAFE}|| = \{ h+i \mid h \in H \text{ and } 0 \leq i < \text{size}(h) \text{ and } (h=0 \text{ or } \text{kind}(h)=\text{Typed}(\tau)) \}$
  - Sequence pointers: $||\tau\text{ ref SEQ}|| = \{ <h,n> \mid h \in H \text{ and } (h=0 \text{ or } \text{kind}(h)=\text{Typed}(\tau)) \}$
  - Dynamic pointers: $||\text{DYNAMIC}|| = \{ <h,n> \mid h \in H \text{ and } (h=0 \text{ or } \text{kind}(h)=\text{Untyped}) \}$

For sequence and dynamic pointers, must keep track of the address and size of the pointed area for runtime bounds checking.

Safe pointers are integers, same as standard C.
Runtime Memory Safety

- Each memory home (i.e., allocated memory area) has typing constraints
  - Either contains values of type \( \tau \), or is untyped

- If a memory address belong to a home, its contents at runtime must satisfy the home’s typing constraints
  - \( \forall h \in H \setminus \{0\} \ \forall i \in \mathbb{N} \)
    - if \( 0 \leq i < \text{size}(h) \) then
      - \((\text{kind}(h) = \text{Untyped} \Rightarrow \text{Memory}[h+i] \in \|\text{DYNAMIC}\|) \text{ and } \text{kind}(h) = \text{Typed}(\tau) \Rightarrow \text{Memory}[h+i] \in \|\tau\|)\)
Runtime Checks

◆ Memory accesses
  • If via safe pointer, only check for non-NULL
  • If via sequence or dynamic pointer, also bounds check

◆ Typecasts
  • From sequence pointers to safe pointers
    – This requires a bounds check!
  • From pointers to integers
  • From integers to sequence or dynamic pointers
    – But the home of the resulting pointer is NULL and it cannot be dereferenced; this breaks C programs that cast pointers into integers and back into pointers
Inferring Pointer Types

- Manual: programmer annotates code
- Better: type inference
  - Analyze the source code to find as many safe and sequence pointers as possible
- This is done by resolving a set of constraints
  - If \( p \) is used in pointer arithmetic, \( p \) is not safe
  - If \( p_1 \) is cast to \( p_2 \)
    - Either they are of the same kind, or \( p_1 \) is a sequence pointer and \( p_2 \) is a safe pointer
    - Pointed areas must be of same type, unless both are dynamic
  - If \( p_1 \) points to \( p_2 \) and \( p_1 \) is dynamic, then \( p_2 \) dynamic
- See the CCured paper for more details
Various CCured Issues

- Converting a pointer to an integer and back to a pointer no longer works
  - Sometimes fixed by forcing the pointer to be dynamic
- Modified pointer representation
  - Not interoperable with libraries that are not recompiled using CCured (use wrappers)
  - Breaks sizeof() on pointer types
- If program stores addresses of stack variables in memory, these variables must be moved to heap
- Garbage collection instead of explicit deallocation
Performance

- Most pointers in benchmark programs were inferred safe, performance penalty under 90%
  - Less than 20% in half the cases
  - Minimal slowdown on I/O-bound applications
    - Linux kernel modules, Apache
  - If all pointers were made dynamic, then 6 to 20 times slower (similar to a pure runtime-checks approach)
  - On the other hand, pure runtime-checks approach does not require access to source code and recompilation

- Various bugs found in test programs
  - Array bounds violations, uninitialized array indices
Other Static Analysis Tools

- Coverity
- PREfix and PREfast (from Microsoft)
- PolySpace
- Cyclone dialect of C
- Many, many others
  - For example, see http://spinroot.com/static/