Static Analysis for Memory Safety

Salvatore Guarnieri
sammyg@cs.washington.edu

Papers

- A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities
  - **Using static analysis and integer range analysis to find buffer overflows**
- A Practical Flow-Sensitive and Context Sensitive C and C++ Memory Leak Detector
  - **Identifying memory ownership with static analysis**
  - **Detecting double frees**

A FIRST STEP TOWARDS AUTOMATED DETECTION OF BUFFER OVERRUN VULNERABILITIES

Problem

```
char s[10];
strcpy(s, “Hello world!”);
```

- “Hello world!” is 12 + 1 characters
- s only holds 10 characters
- How do we **detect** or prevent this buffer overflow?

“Modern” String Functions Don’t Fix the Problem

- The strn*() calls behave dissimilarly
- Inconsistency makes it harder for the programmer to remember how to use the “safe” primitives safely.
- strncpy() may leave the target buffer unterminated.
- strncat() and snprintf() always append a terminating ’\0’ byte
- strncpy() has performance implications: it zero-fills the target buffer
- strncpy() and strncat() encourage off-by- one bugs (Null character)

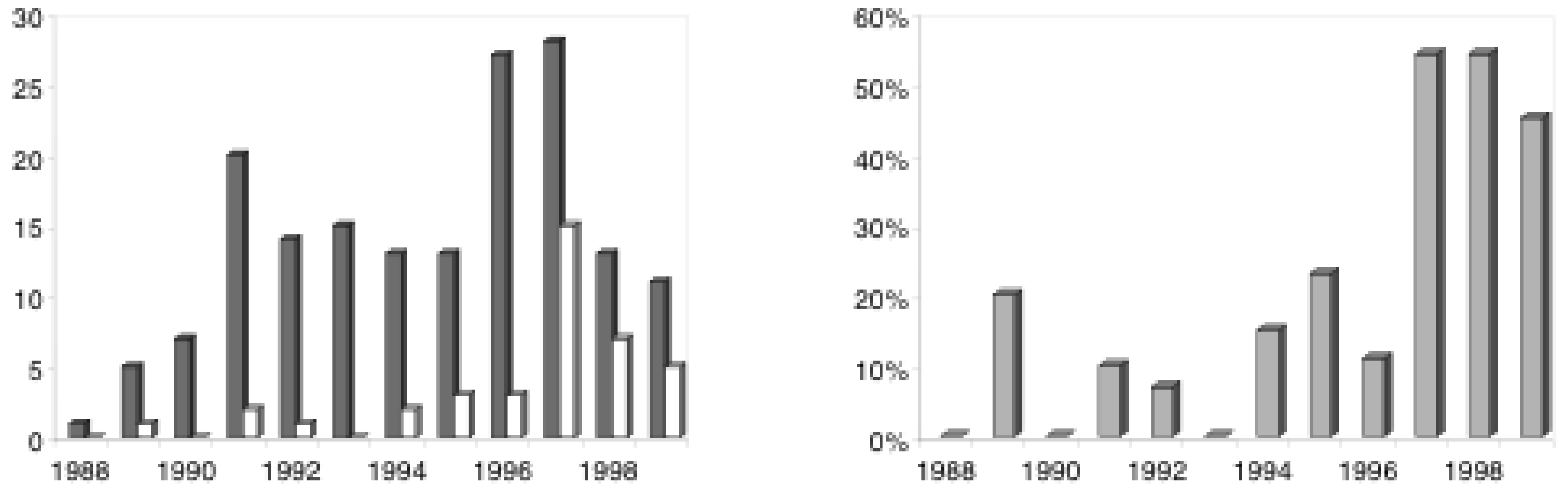


Figure 1. Frequency of buffer overrun vulnerabilities, derived from a classification of CERT advisories. The left-hand chart shows, for each year, the total number of CERT-reported vulnerabilities and the number that can be blamed primarily on buffer overruns. The right-hand chart graphs the percentage of CERT-reported vulnerabilities that were due to buffer overruns for each year.

Insight

- We care about when we write past the end of an array

```
a[i] = ...
```

Should be

```
if (i < sizeof(a)) {
  a[i] = ...
} else {error}
```

Basic Approach

- Treat C strings as an abstract data type
  - Ignore everything but str* library functions

- Model buffers as a pair integer ranges
  - `len(a)` is **how far into the array** the program accesses
  - `alloc(a)` is **how large** the array is

- If **`len(a) > alloc(a)`**, there is a buffer overrun

```
char *array = malloc(10);
```

```
array[1] = 'h';
```

```
array[9] = '\0';
```

```
strcpy(array, "0123456789012");
```

len(array) =

alloc(array) =

```
char *array = malloc(10);
```

```
array[1] = 'h';
```

```
array[9] = '\0';
```

```
strcpy(array, "0123456789012");
```

len(array) = 0 alloc(array) = 10

```
char *array = malloc(10);
```

```
array[1] = 'h';
```

```
array[9] = '\0';
```

```
strcpy(array, "0123456789012");
```

len(array) = 2

alloc(array) = 10

```
char *array = malloc(10);
```

```
array[1] = 'h';
```

```
array[9] = '\0';
```

```
strcpy(array, "0123456789012");
```

len(array) = 10 | alloc(array) = 10

```
char *array = malloc(10);
array[1] = 'h';
array[9] = '\0';
strcpy(array, "0123456789012");
```

len(dest) = len(src)

len(array) = 14 | alloc(array) = 10

```
char *array = malloc(10);
array[1] = 'h';
array[9] = '\0';
strcpy(array, "0123456789012");
```

len(dest) = len(src)

OVERRUN

len(array) = 14 | alloc(array) = 10

It’s not that simple

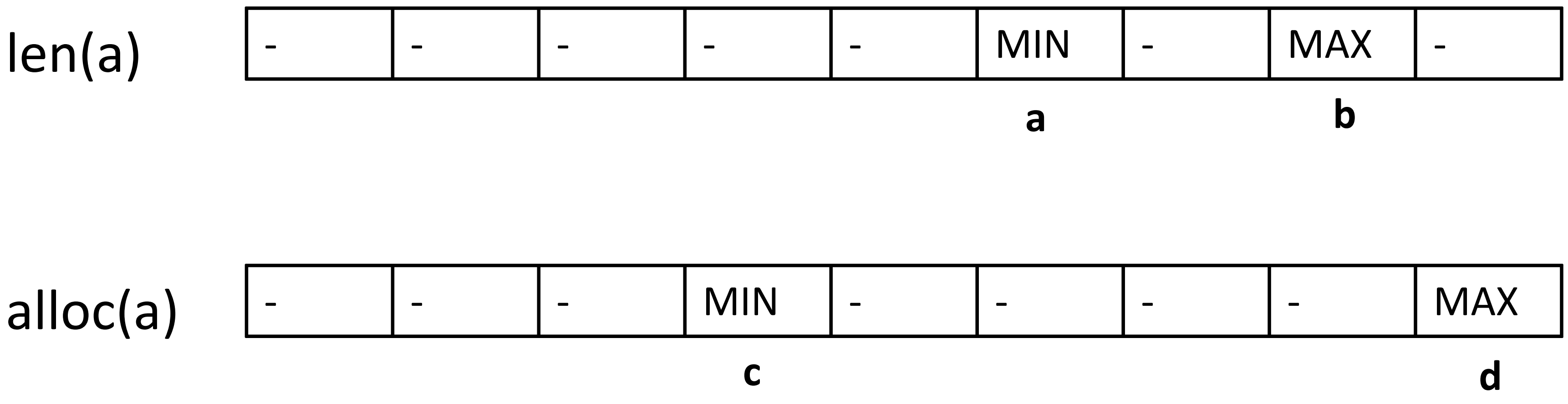
```
char *array = malloc(10);
if (k == 7) {
  strcpy(array, “hello”);
} else {
  free(array); array = malloc(3);
  strcpy(array, “world!”);
}
```

- What is len(array)? What is alloc(array)?

Use Ranges

```
char *array = malloc(10);
if (k == 7) {
  strcpy(array, “hello”);
} else {
  free(array); array = malloc(3);
  strcpy(array, “world!”);
}
```

- len(array) = [5, 6], alloc(array) = [3,10]
- 5>3 so we have a possible overrun



- If b <= c, **no overrun**
- If a > d, **definite overrun**
- Otherwise the ranges overlap and there **may be an overrun**

Implementation Overview

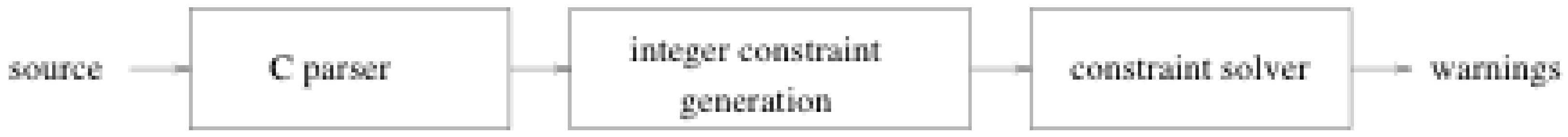
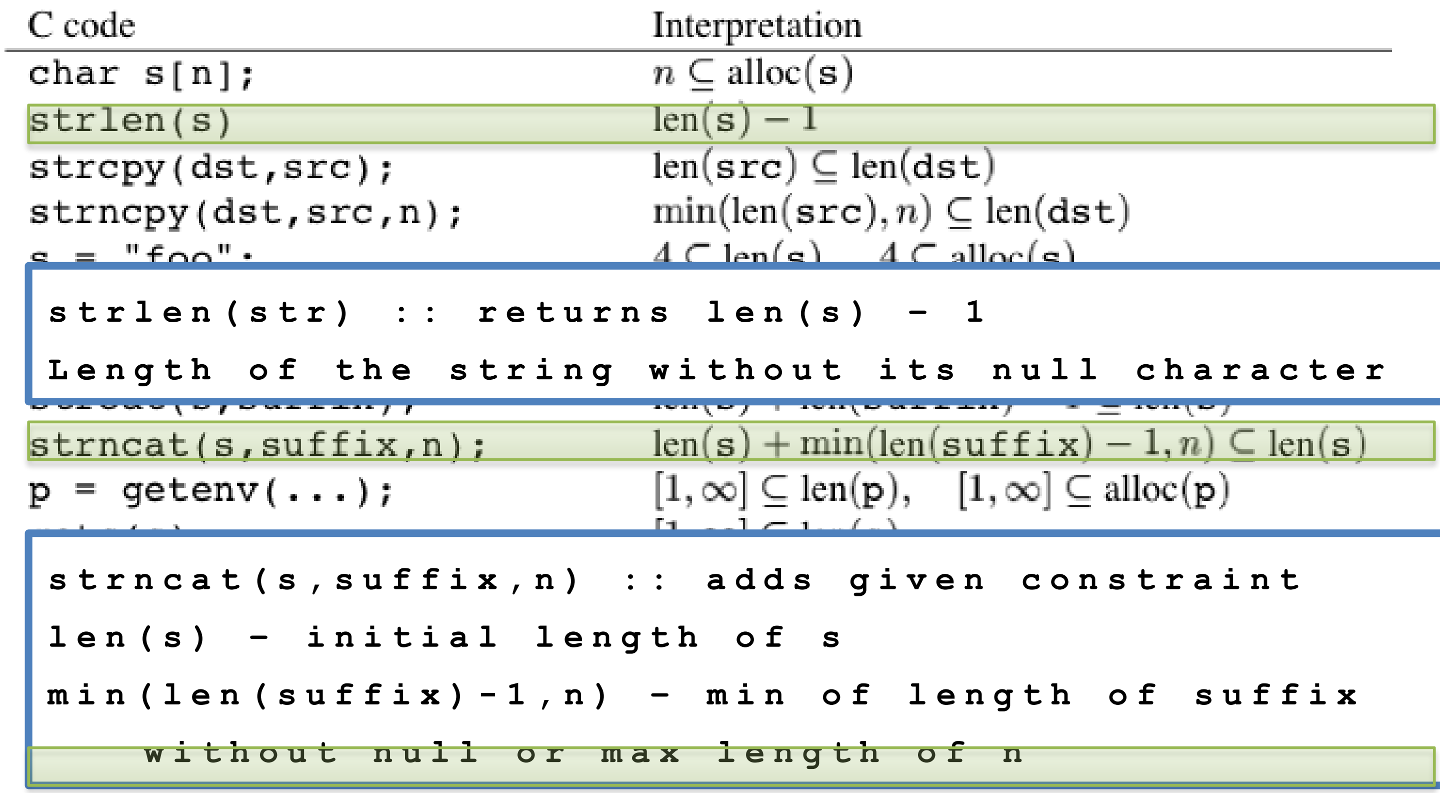


Figure 2. The architecture of the buffer overflow detection prototype.

Constraint Generation



C code	Interpretation
`char s[n];`	$n \subseteq \text{alloc}(s)$
`strlen(s)`	$\text{len}(s) - 1$
`strcpy(dst,src);`	$\text{len}(src) \subseteq \text{len}(dst)$
`strncpy(dst,src,n);`	$\min(\text{len}(src), n) \subseteq \text{len}(dst)$
`s = "foo";`	$4 \subseteq \text{len}(s),\quad 4 \subseteq \text{alloc}(s)$
`strncat(s,suffix,n);`	$\text{len}(s) + \min(\text{len}(suffix) - 1, n) \subseteq \text{len}(s)$
`p = getenv(...);`	$[1,\infty] \subseteq \text{len}(p),\quad [1,\infty] \subseteq \text{alloc}(p)$

strlen(str) :: returns len(s) - 1

Length of the string without its null character

strncat(s,suffix,n) :: adds given constraint

len(s) - initial length of s

min(len(suffix)-1,n) - min of length of suffix without null or max length of n

p[n] = NULL :: Sets the new effective length of p

The min doesn't really make sense here

Constraints

```
char *array = malloc(10);                 10 ⊆ alloc(array)
if (k == 7) {
  strcpy(array, “hello”);      len("hello") ⊆ len(array)
} else {
  free(array); array = malloc(3);    3 ⊆ alloc(array)
  strcpy(array, “world!”);    len("world!") ⊆ len(array)
}
```

len = [5,6]

alloc = [3,10]

Limitations

- Double pointer
  - Doesn’t fit in with their method
- Function pointers and union types
  - Ignored
- Structs
  - All structs of same “type” are aliased
  - Struct members are treated as unique memory addresses
- Flow Insensitive

Pointer Alias Limitations

```
char s[20], *p, t[10];
strcpy(s, “Hello”);
p = s + 5;
strcpy(p, “ world!”);
strcpy(t, s);
```

- What is len(s)?

Evaluation

- Run tool on programs from ~3kloc to ~35kloc
- Does it find new bugs?
- Does it find old bugs?
- What is the false positive rate?
- Are there any false negatives in practice?
- How long does it take to execute on CPU?
- How long does it take the user to use the tool?

Linux nettools

- Total 3.5kloc with another 3.5kloc in a support library
- Recently hand audited
- Found several serious new buffer overruns

- **They don’t talk about the bugs that they find**

Sendmail

- ~35 kloc
- Found **several minor bugs** in latest revision
- Found **many already discovered** buffer overruns in an old version

- 15 min to run for sendmail
  - A few minutes to parse
  - The rest for constraint generation
  - A few seconds to solve constraint system

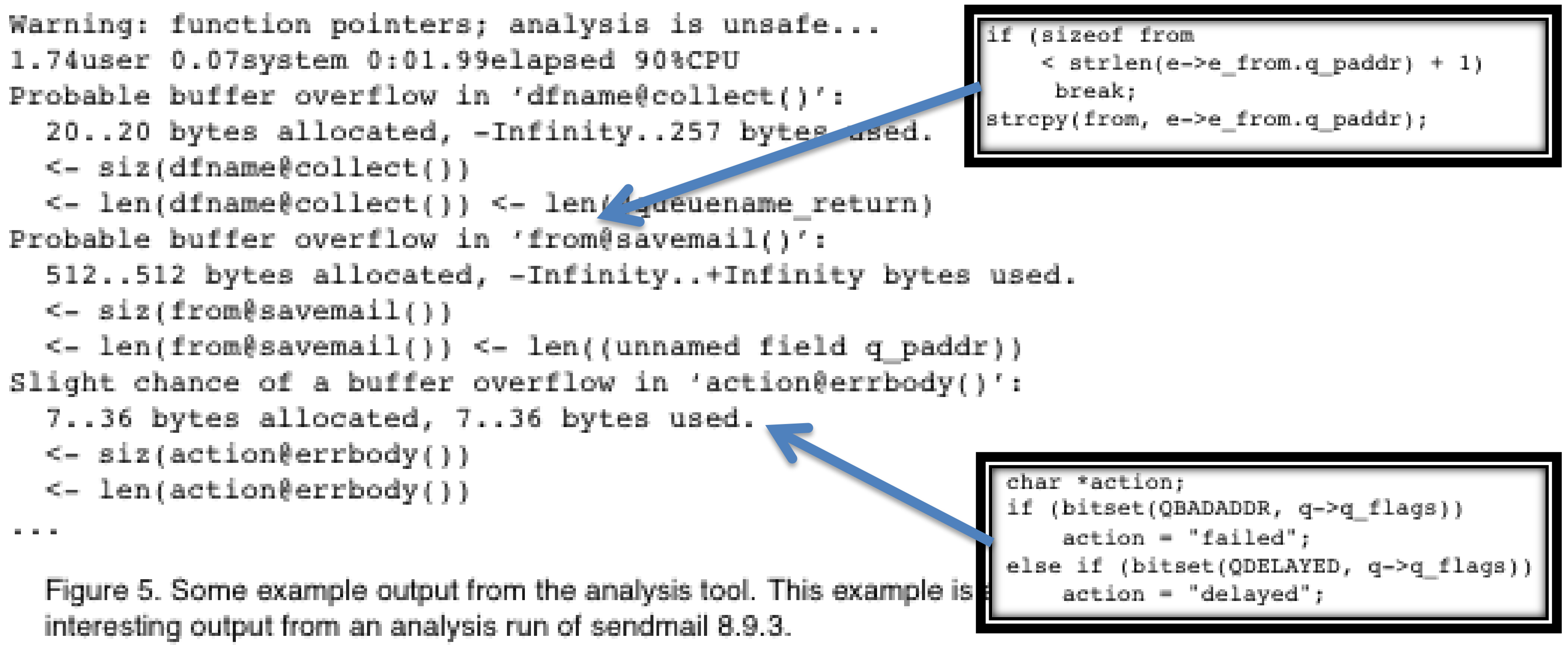
Sendmail findings

- An unchecked sprintf() from the results of a DNS lookup to a 200-byte stack-resident buffer; exploitable from remote hosts with long DNS records. (Fixed in sendmail 8.7.6.)
- An unchecked strcpy() to a 64-byte buffer when parsing stdin; locally exploitable by "echo /canon aaaaa... | sendmail -bt". (Fixed in 8.7.6)
- An unchecked copy into a 512-byte buffer from stdin; try "echo /parse aaaaa... | sendmail -bt". (Fixed in 8.8.6.)
- An unchecked strcpy() to a (static) 514-byte buffer from a DNS lookup; possibly remotely exploitable with long DNS records, but the buffer doesn't live on the stack, so the simplest attacks probably wouldn't work.
- Several places where the results of a NIS network query is blindly copied into a fixed-size buffer on the stack; probably remotely exploitable with long NIS records. (Fixed in 8.7.6 and 8.8.6.)

Human Experience

- 15 minutes to run…
- 44 warnings to investigate
- 4 real bugs

- Without tool you would have to investigate **695 potentially unsafe call sites**



```
Warning: function pointers; analysis is unsafe...
1.74user 0.07system 0:01.99elapsed 90%CPU
Probable buffer overflow in 'dfname@collect()':
  20..20 bytes allocated, -Infinity..257 bytes used.
  <- siz(dfname@collect())
  <- len(dfname@collect()) <- len(queuename_return)
Probable buffer overflow in 'from@savemail()':
  512..512 bytes allocated, -Infinity..+Infinity bytes used.
  <- siz(from@savemail())
  <- len(from@savemail()) <- len((unnamed field q_paddr))
Slight chance of a buffer overflow in 'action@errbody()':
  7..36 bytes allocated, 7..36 bytes used.
  <- siz(action@errbody())
  <- len(action@errbody())
...
```

```
if (sizeof from
    < strlen(e->e_from.q_paddr) + 1)
     break;
strcpy(from, e->e_from.q_paddr);
```

```
char *action;
if (bitset(QBADADDR, q->q_flags))
     action = "failed";
else if (bitset(QDELAYED, q->q_flags))
     action = "delayed";
```

Figure 5. Some example output from the analysis tool. This example is interesting output from an analysis run of sendmail 8.9.3.

Improvements

Improved Analysis	**False alarms that would be removed**
Flow-sensitive	19/40 (**47%**)
Flow-sensitive with pointer analysis	25/40 (**62%**)
Flow and context sensitive with linear invariants	28/40 (**70%**)
Flow and context sensitive with linear invariants and pointer analysis	38/40 (**95%**)

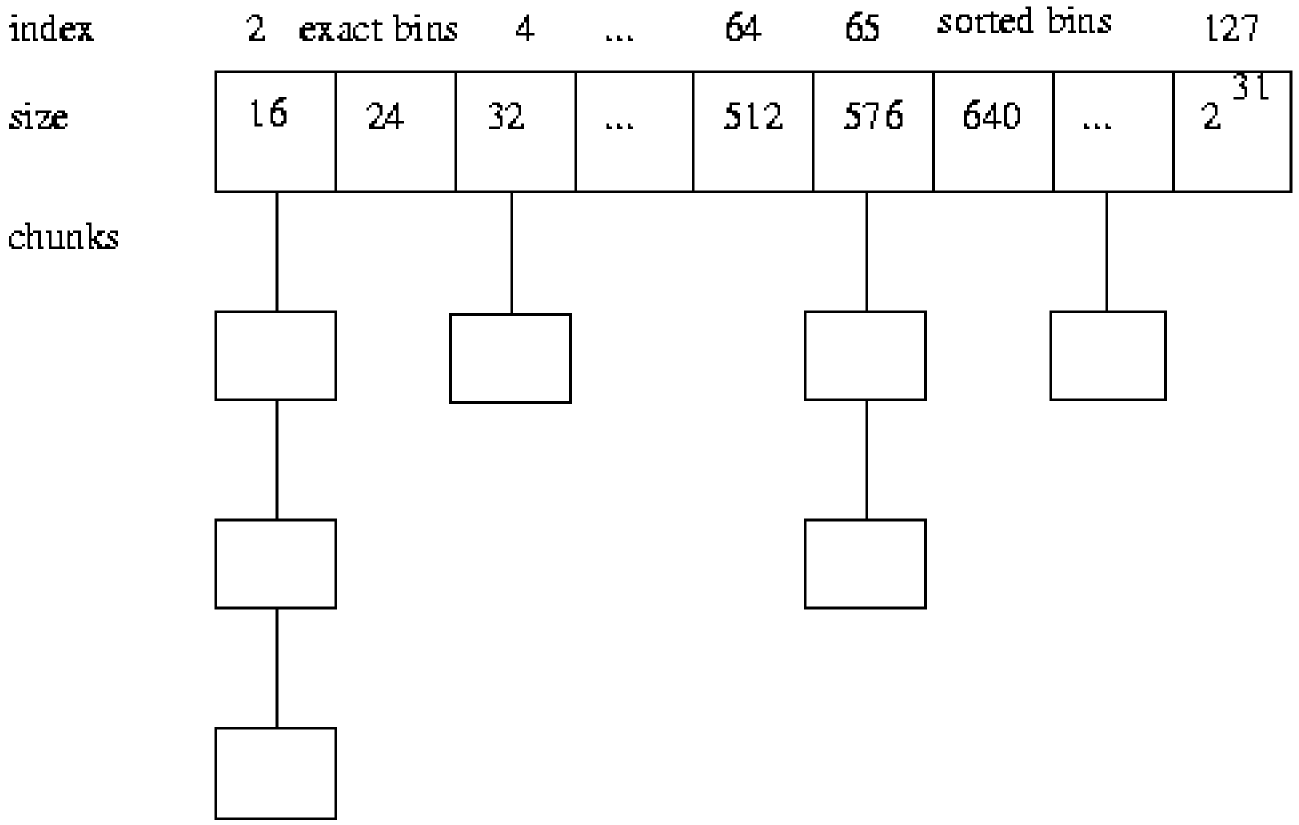
IDENTIFYING MEMORY OWNERSHIP
-- CLOUSEAU

From overruns to memory errors

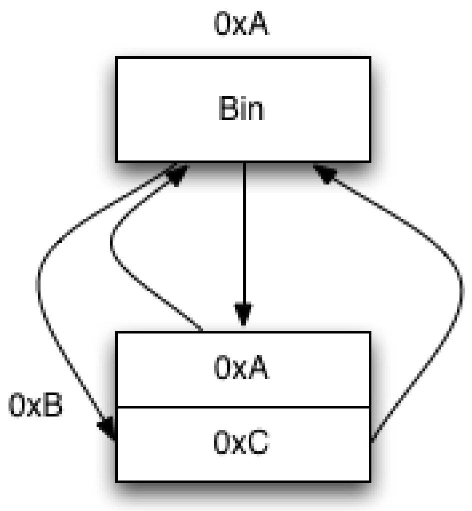
- Memory Leaks
  - Bloat
  - Slow performance
  - Crashes

- Dangling pointers/Double free
  - Crashes
  - Unexpected behavior
  - **Exploits**

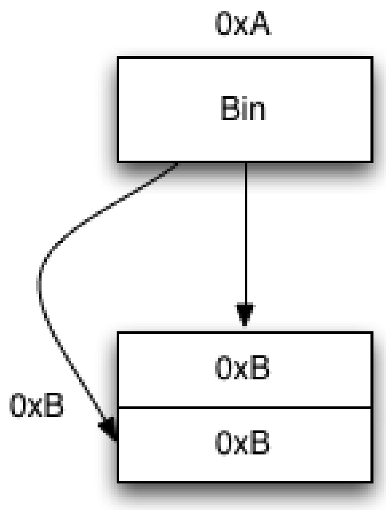
Double Free


index
2
exact bins
4
...
64
65
sorted bins
127
size
16
24
32
...
512
576
640
...
2^{31}
chunks

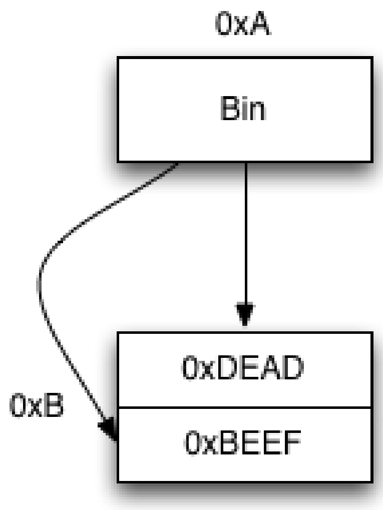
After Normal Free



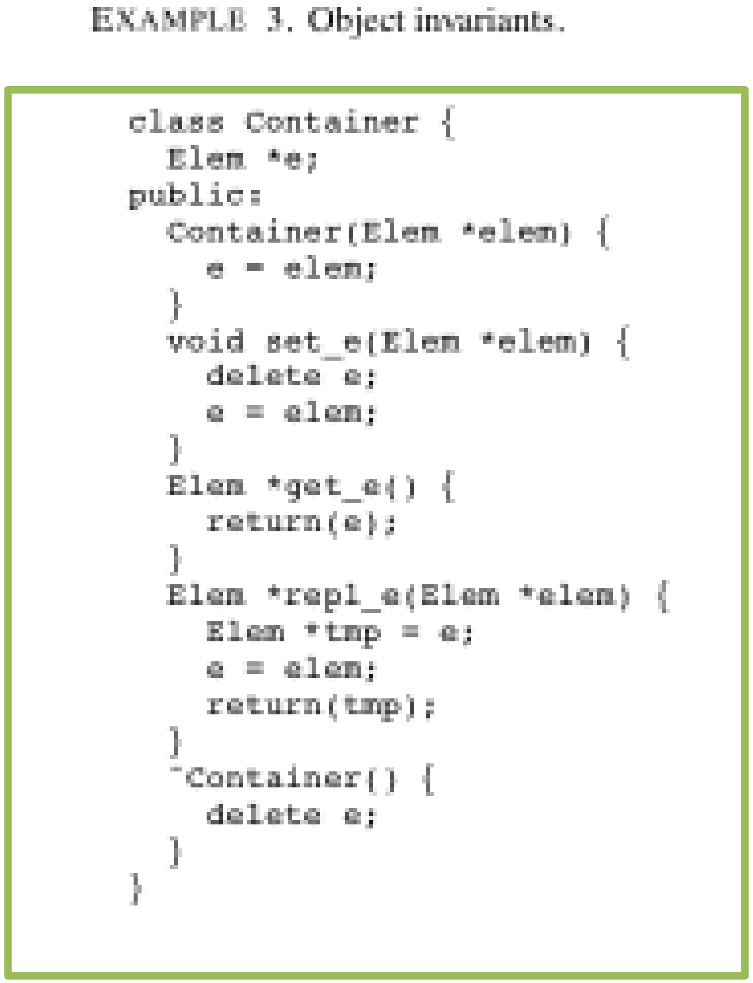
After Double Free



Alloc same size chunk again and get same memory. Write 8 bytes

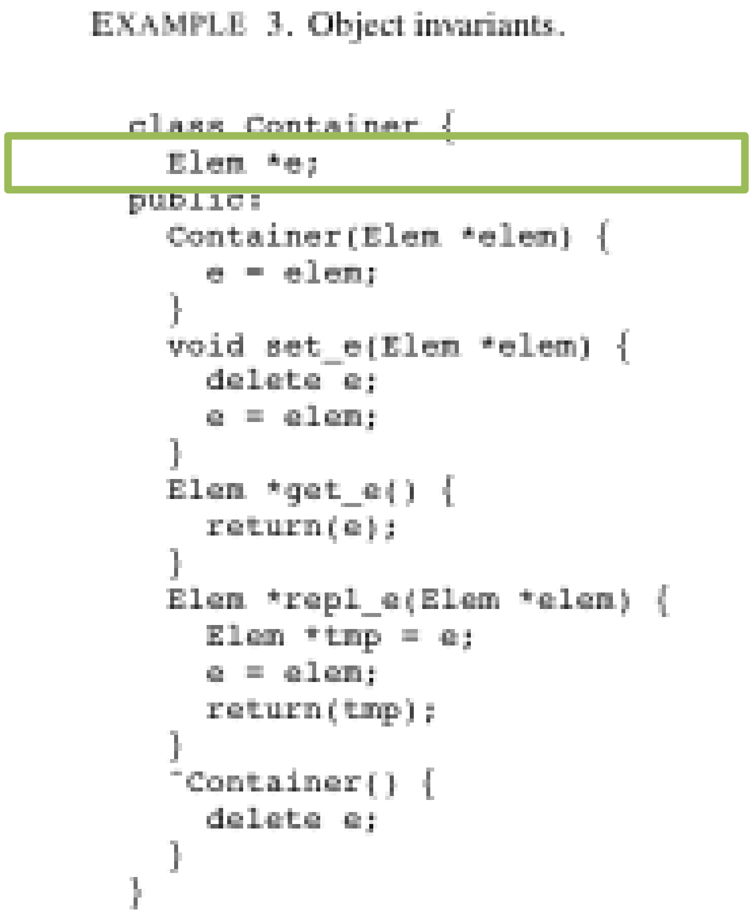


Motivating Example


EXAMPLE 3. Object invariants.

```
class Container {
  Elem *e;
public:
  Container(Elem *elem) {
    e = elem;
  }
  void set_e(Elem *elem) {
    delete e;
    e = elem;
  }
  Elem *get_e() {
    return(e);
  }
  Elem *repl_e(Elem *elem) {
    Elem *tmp = e;
    e = elem;
    return(tmp);
  }
  ~Container() {
    delete e;
  }
}
```

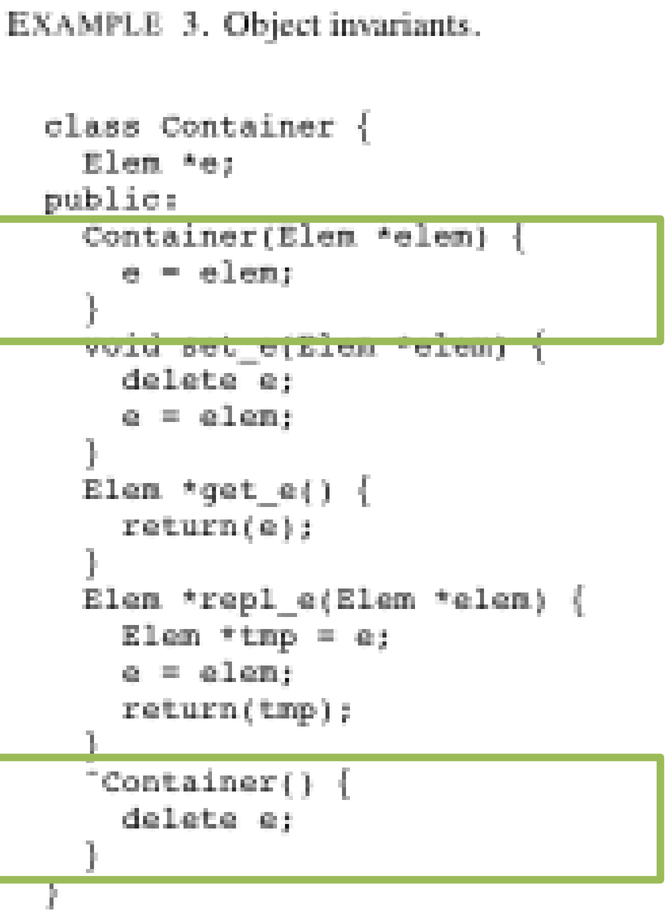
Motivating Example



EXAMPLE 3. Object invariants.

```
class Container {
  Elem *e;
public:
  Container(Elem *elem) {
    e = elem;
  }
  void set_e(Elem *elem) {
    delete e;
    e = elem;
  }
  Elem *get_e() {
    return(e);
  }
  Elem *repl_e(Elem *elem) {
    Elem *tmp = e;
    e = elem;
    return(tmp);
  }
  ~Container() {
    delete e;
  }
}
```

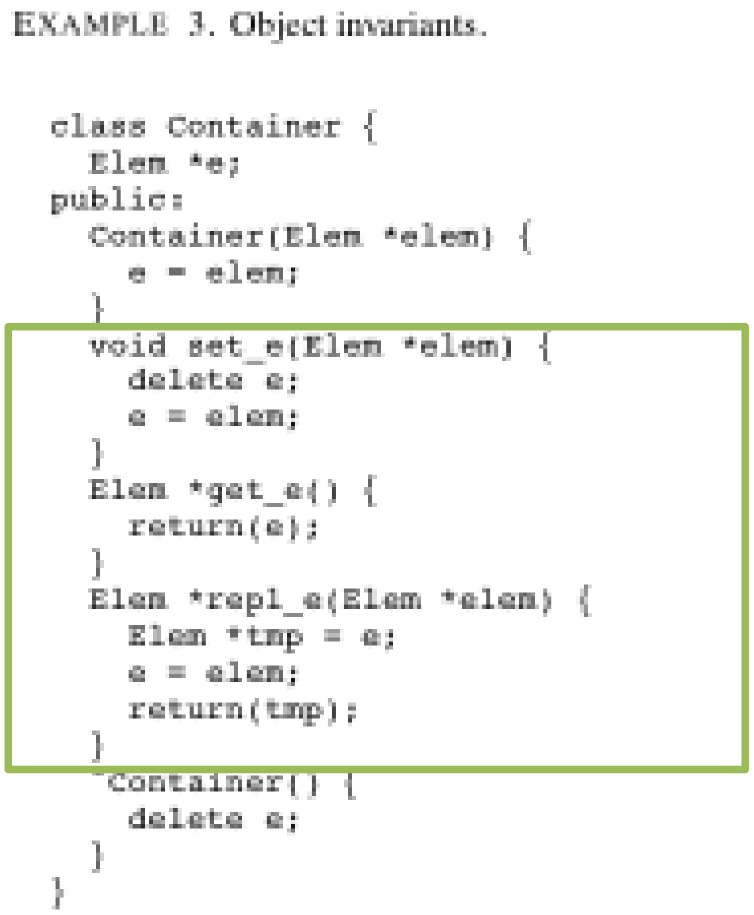
Motivating Example



EXAMPLE 3. Object invariants.

```
class Container {
  Elem *e;
public:
  Container(Elem *elem) {
    e = elem;
  }
  void set_e(Elem *elem) {
    delete e;
    e = elem;
  }
  Elem *get_e() {
    return(e);
  }
  Elem *repl_e(Elem *elem) {
    Elem *tmp = e;
    e = elem;
    return(tmp);
  }
  ~Container() {
    delete e;
  }
}
```

Motivating Example



EXAMPLE 3. Object invariants.

```
class Container {
  Elem *e;
public:
  Container(Elem *elem) {
    e = elem;
  }
  void set_e(Elem *elem) {
    delete e;
    e = elem;
  }
  Elem *get_e() {
    return(e);
  }
  Elem *repl_e(Elem *elem) {
    Elem *tmp = e;
    e = elem;
    return(tmp);
  }
  ~Container() {
    delete e;
  }
}
```

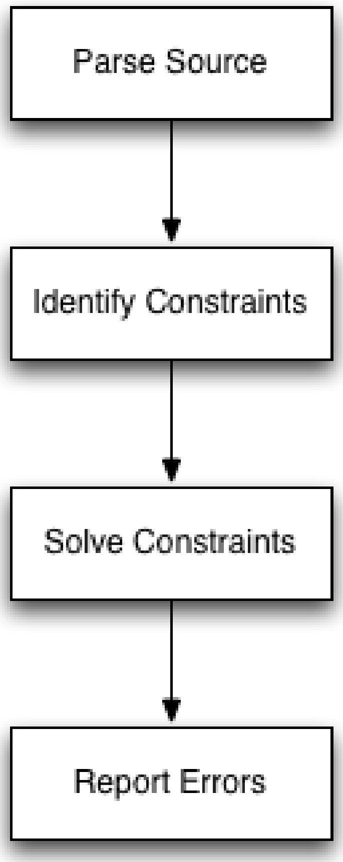
Ownership

- Introduce **ownership** to identify who is **allowed and responsible to free memory**
- PROPERTY 1. *There exists one and only one owning pointer to every object allocated but not deleted.*
- PROPERTY 2. *A delete operation can only be applied to an owning pointer.*

Key Design Choices

- Ownership is connected with the pointer variable, not the object
- Ownership is tracked as 0 (non-owning) or 1 (owning)
  - Partially to make solving the linear inequality constraints easier
- Rank warnings with heuristics to minimize impact of false positives

System Overview



Flow Sensitive Analysis

```
u = new int; //u is the owner
z = u;
delete z; //right before this line z is the owner
```

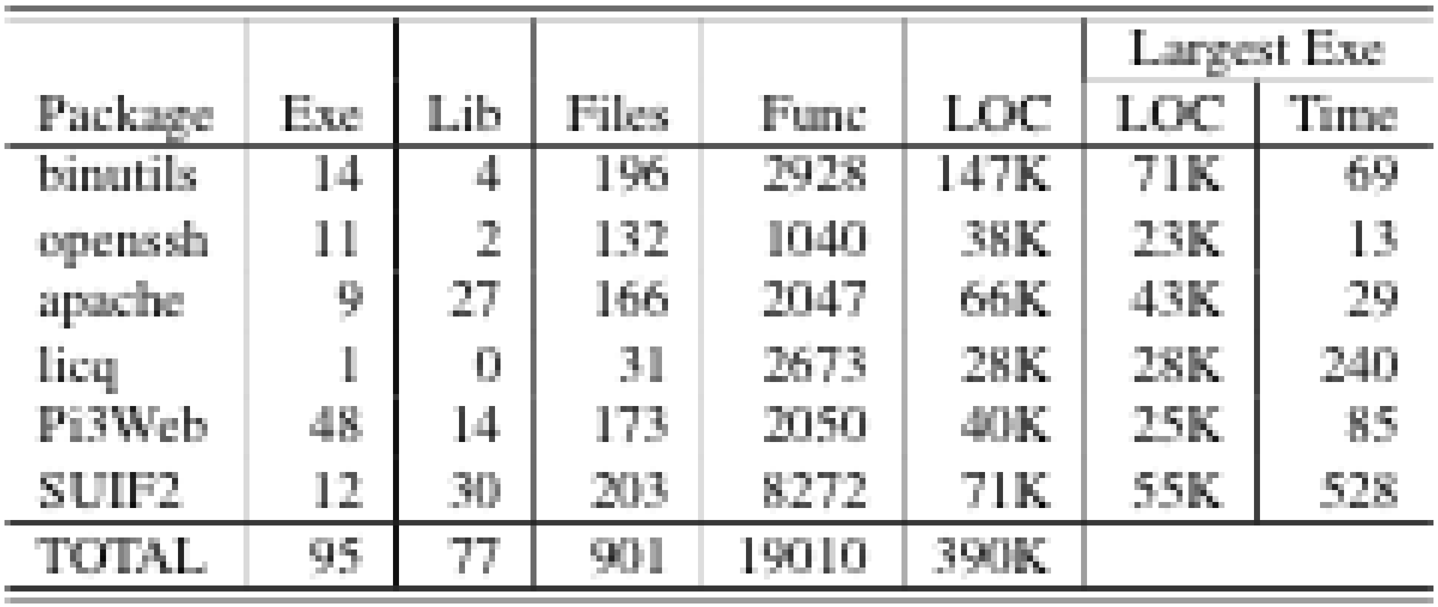
- Order of instructions matters
- Analysis identifies line 2 as a possible ownership transfer point

Constraint Solving Problem

```
u = new int; //u is the owner
z = u;
delete z; //right before this line z is the owner
```

- Constructors indicate ownership
- Deletion indicates desired/intended ownership
- Generate all other constraints from assignments
- Solve to identify owners

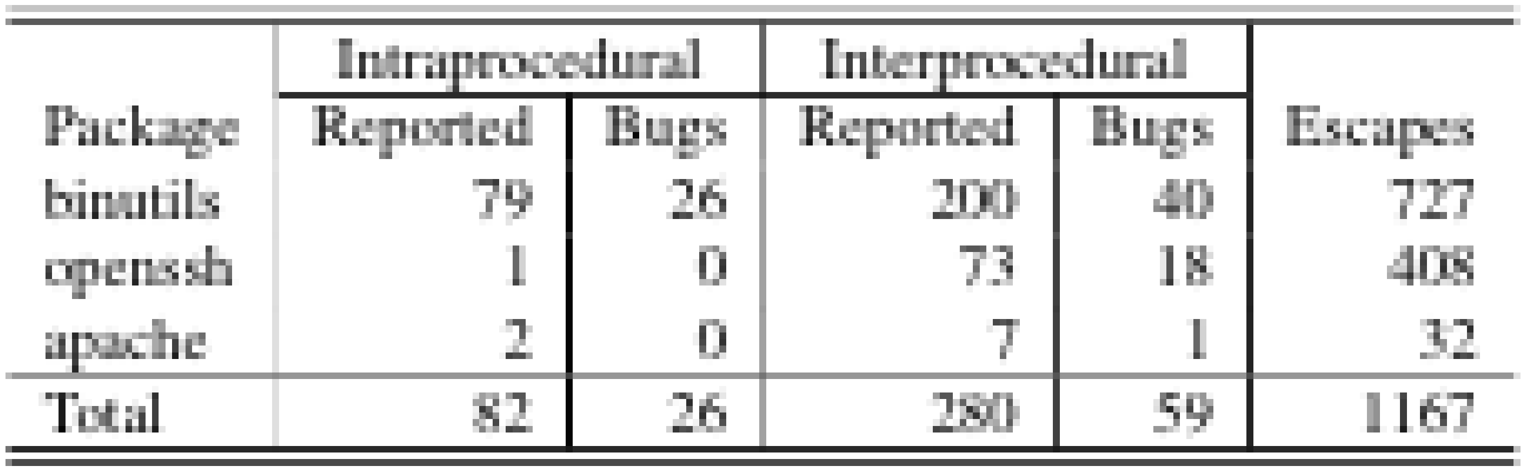
Evaluation



						Largest Exe	
Package	Exe	Lib	Files	Func	LOC	LOC	Time
binutils	14	4	196	2928	147K	71K	69
openssh	11	2	132	1040	38K	23K	13
apache	9	27	166	2047	66K	43K	29
licq	1	0	31	2673	28K	28K	240
Pi3Web	48	14	173	2050	40K	25K	85
SUIF2	12	30	203	8272	71K	55K	528
TOTAL	95	77	901	19010	390K		

Figure 4: Application characteristics: number of executables, libraries, files, functions, lines of code, lines of code in the largest executable and its ownership analysis time in seconds.

Evaluation -- C

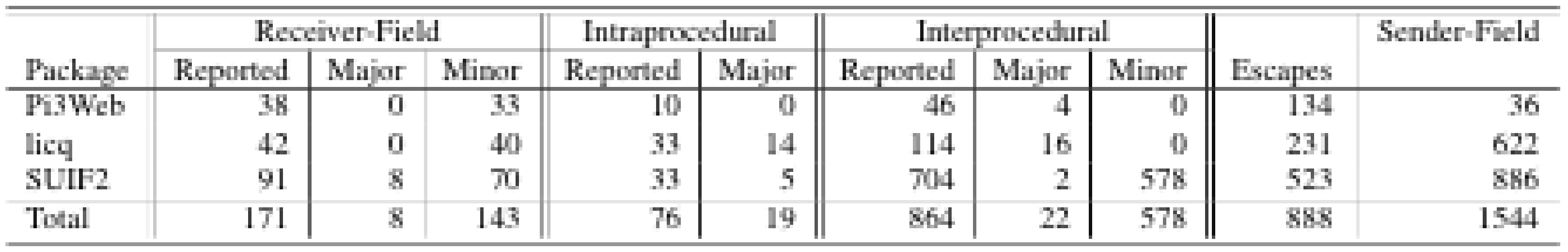


	Intraprocedural		Interprocedural		
Package	Reported	Bugs	Reported	Bugs	Escapes
binutils	79	26	200	40	727
openssh	1	0	73	18	408
apache	2	0	7	1	32
Total	82	26	280	59	1167

Figure 5: Reported warnings and identified errors on C applications

85 bugs / 362 warnings = 23% true positives

Evaluation – C++



	Receiver-Field			Intraprocedural		Interprocedural				Sender-Field
Package	Reported	Major	Minor	Reported	Major	Reported	Major	Minor	Escapes	
Pi3Web	38	0	33	10	0	46	4	0	134	36
licq	42	0	40	33	14	114	16	0	231	622
SUIF2	91	8	70	33	5	704	2	578	523	886
Total	171	8	143	76	19	864	22	578	888	1544

Figure 6: Reported warnings on C++ applications, with identified major and minor errors

False Positives

- For C
- 85 errors for 362 warnings – **23% accuracy**
- Many errors due to abnormal flow paths
  - breaks, error conditions, etc.

- For C++
- 777 errors out of 1111 warnings – minor
- 49 errors out of 390 warnings – **12.5% accuracy**
  - Double deletes, incorrect destructors

END.

Flow Insensitive

- Instruction order doesn’t matter

```
char *a; a=malloc(10); strcpy(a, “hello”); a=malloc(3);
```

Is analyzed the same as

```
char *a; a=malloc(3); strcpy(a, “hello”); a=malloc(10);
```

Not Sound and Not Complete

- Lack of pointer treatment makes this unsound
  - True positives can be missed
- Already an imprecise algorithm, so it is incomplete
  - False positives could be generated
- Evaluation will be very important

Sendmail findings

- An unchecked sprintf() from the results of a DNS lookup to a 200-byte stack-resident buffer; exploitable from remote hosts with long DNS records. (Fixed in sendmail 8.7.6.)
- An unchecked sprintf() to a 5-byte buffer from a command-line argument (indirectly, via several other variables); exploitable by local users with “sendmail -h65534 ...”. (Fixed in 8.7.6.)
- An unchecked strcpy() to a 64-byte buffer when parsing stdin; locally exploitable by “echo /canon aaaaa... | sendmail -bt”. (Fixed in 8.7.6)
- An unchecked copy into a 512-byte buffer from stdin; try “echo /parse aaaaa... | sendmail -bt”. (Fixed in 8.8.6.)
- An unchecked sprintf() to a 257-byte buffer from a filename; probably not easily exploitable. (Fixed in 8.7.6.)
- A call to bcopy() could create an unterminated string, because the programmer forgot to explicitly add a ’\0’; probably not exploitable. (Fixed by 8.8.6.)
- An unchecked strcpy() in a very frequently used utility function. (Fixed in 8.7.6.)
- An unchecked strcpy() to a (static) 514-byte buffer from a DNS lookup; possibly remotely exploitable with long DNS records, but the buffer doesn’t live on the stack, so the simplest attacks probably wouldn’t work.
- Also, there is at least one other place where the result of a DNS lookup is blindly copied into a static fixed-size buffer. (Fixed in 8.7.6.)
- Several places where the results of a NIS network query is blindly copied into a fixed-size buffer on the stack; probably remotely exploitable with long NIS records. (Fixed in 8.7.6 and 8.8.6.)

Double Free Exploit

- Freeing a memory block twice **corrupts allocation structures**
- First free puts block back in list
- Second free mucks with forward and back pointers so they point to the same block (the current block)
- Now future requests for blocks of that size will always return the same block
- Write two memory addresses (8 bytes) to this chunk
- Make another request of the same block size, same block will be used but since we just filled in data for forward and back pointers, we can write to any memory we want

cut

- Enforce ownership with a type system
  - **Infer types from code**
  - No user provided annotations
  - Sound
  - Minimizes false positives by prioritizing constraints

Definition of Escape

- Escaping violations refer to possible transfers of ownership to pointers stored in structures, arrays or indirectly accessed variables. While these warnings tell the users which data structures in the program may hold owning pointers, they leave the user with much of the burden of determining whether any of these pointers leak. Users are not expected to examine the escaping warnings, so we only examined the non-escaping warnings to find program errors.

Minor Errors in C++

- First, many classes with owning member fields do not have their own copy constructors and copy operators; the default implementations are incorrect because copying owning fields will create multiple owners to the same object. Even if copy constructors and copy operators are not used in the current code, they should be properly defined in case they are used in the future.
- Second, 578 of the 864 interprocedural warnings reported for SUIF2 are caused by leaks that occur just before the program finds an assertion violation and aborts. We have implemented a simple interprocedural analysis that can catch these cases and suppress the generation of such errors if desired.