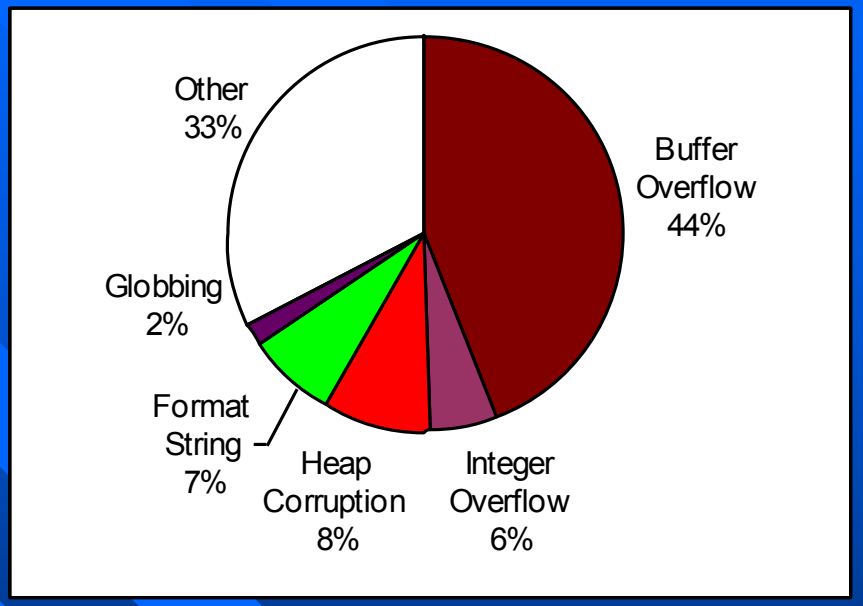
Formal Reasoning of Security Vulnerabilities by Pointer Taintedness Semantics

S. Chen, K. Pattabiraman, Z. Kalbarczyk and R. K. Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign

Motivation

- Our analysis on CERT advisories shows
  - Many vulnerabilities (≥ 66%) due to incorrect pointer dereferences
  - A significant portion of vulnerabilities (≥ 33.6%) due to errors in library functions or incorrect invocations of library functions



- Motivating questions
  - What is the common characteristic among most security vulnerabilities?
  - How to develop a generic reasoning approach to find a wide spectrum of security vulnerabilities?

Formal Analysis of Pointer Taintedness

- **Pointer Taintedness**: a pointer value, including a return address, is derived directly or indirectly from user input. (formally defined using equational logic)
- It provides a unifying perspective for reasoning about a significant number of security vulnerabilities.
- The notion of pointer taintedness enables:
  - Static analysis: reasoning about the possibility of pointer taintedness by source code analysis;
  - Runtime checking: inserting assertions in object code to check pointer taintedness at runtime;
  - Hardware architecture-based support to detect pointer taintedness.
- Current focus: extraction of security specifications of library functions based on pointer taintedness semantics.

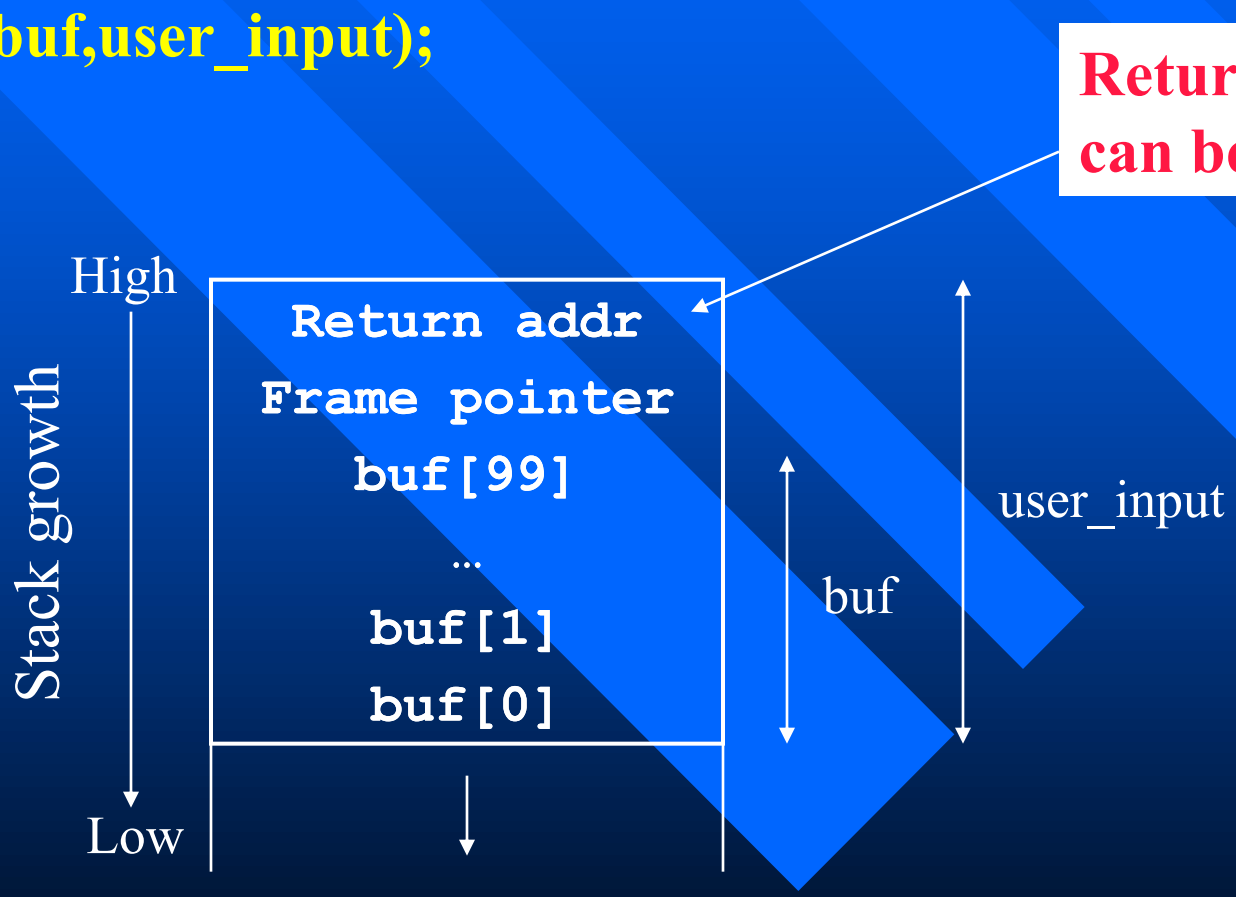
Examples Vulnerabilities Caused by Pointer Taintedness

- Format string vulnerability
  - Taint an argument pointer of functions such as *printf, fprintf, sprintf* and *syslog*.
- Stack buffer overflow (stack smashing)
  - Taint a return address.
- Heap corruption
  - Taint the free-chunk doubly-linked list of the heap.
- Glibc *globbing* vulnerabilities
  - User input resides in a location that is used as a pointer by the parent function of *glob()*.

Stack Buffer Overflow

Vulnerable code:

```
char buf[100];
strcpy(buf,user_input);
```

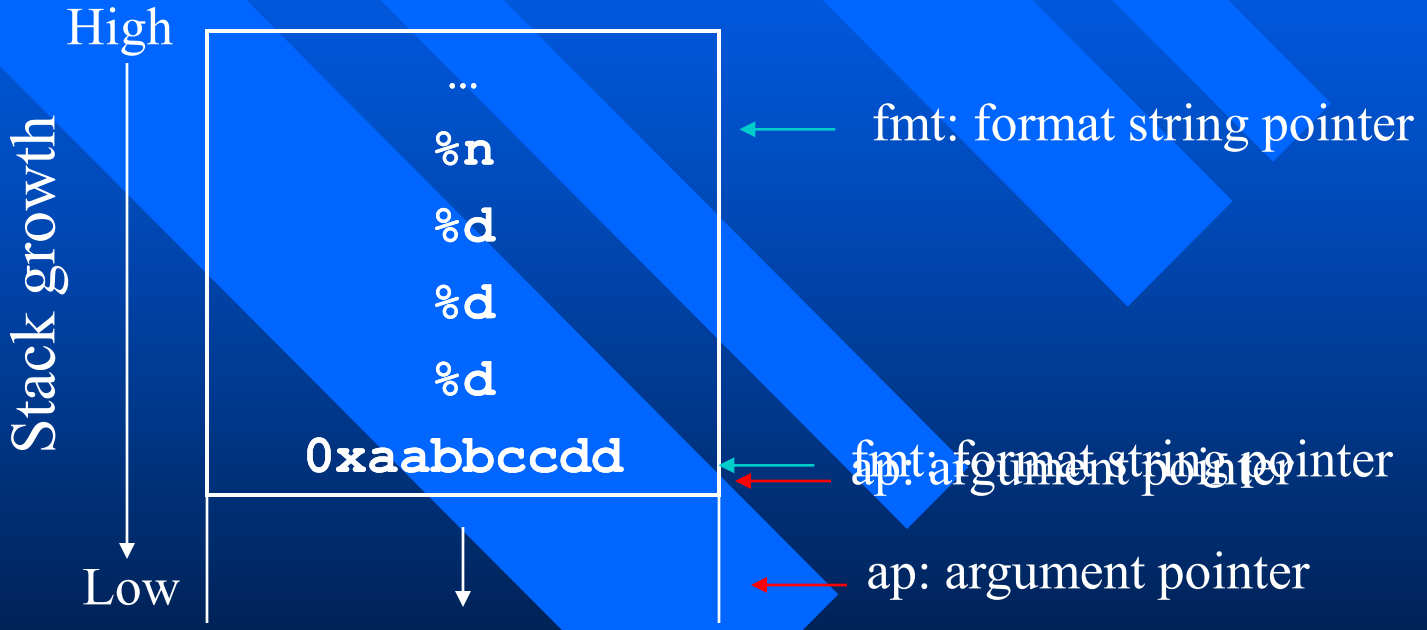


Format String Vulnerability

Vulnerable code:

```
recv(buf);
printf(buf);   /* should be printf("%s",buf) */
```

\xdd \xcc \xbb \xaa %d %d %d %n



In vfprintf(),

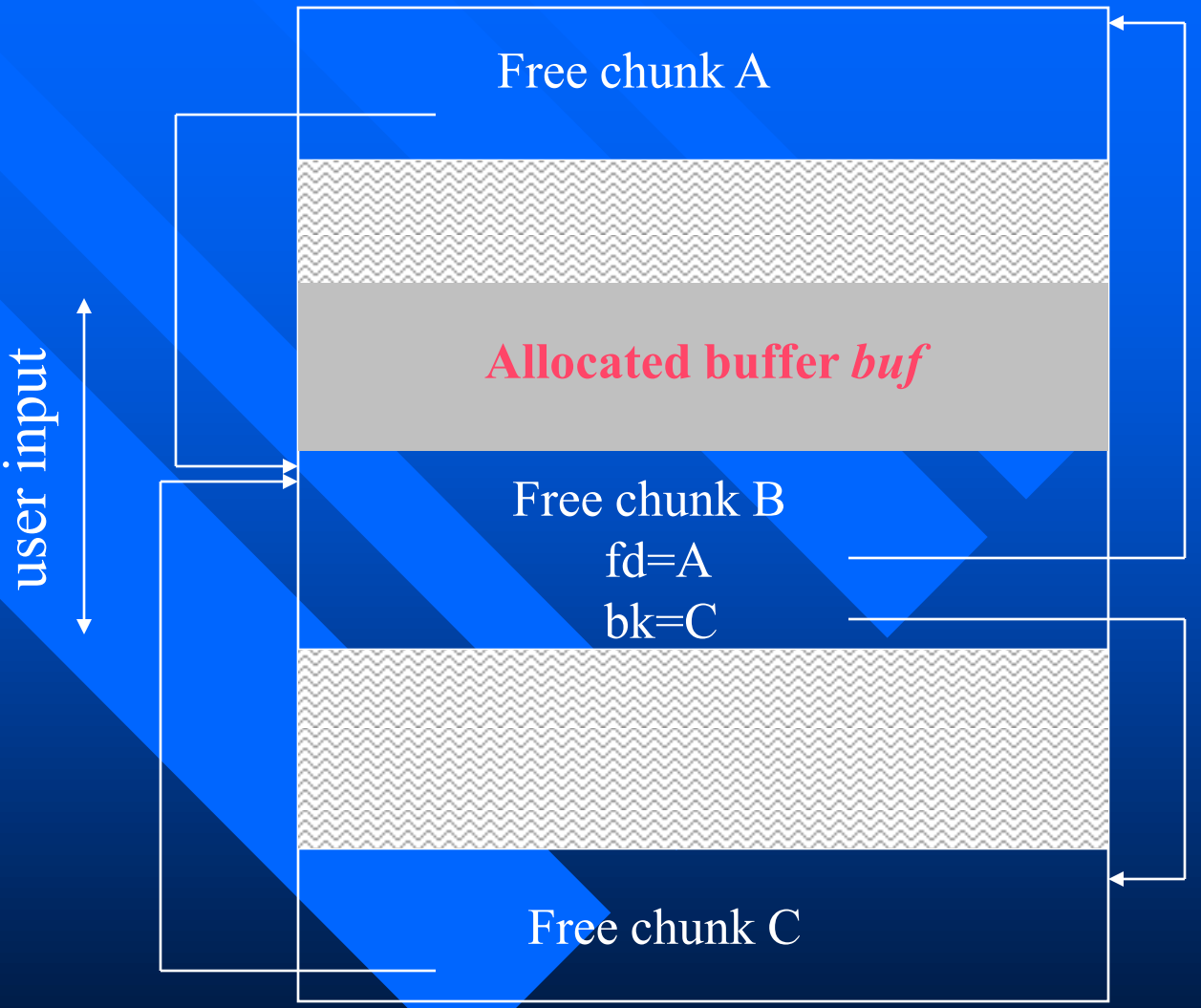
```
if (fmt points to "%n")
then **ap = (character count)
```

***ap is a tainted value.**

Heap Corruption Vulnerability

Vulnerable code:

```
buf = malloc(1000);
recv(sock,buf,1024);
free(buf);
```



In free():

```
B->fd->bk=B->bk;
B->bk->fd=B->fd;
```

When ***B->fd*** **and B->*bk*** **are tainted**, the effect of *free()* is to write a user specified value to a user specified address.

Semantic Definition of Pointer Taintedness

One-Slide Intro to Equational Logic

- Use term rewriting to establish proofs of theorems.
- Natural number addition expressed in the Maude system.

```
0 :  Natural .
s_  :  Natural -> Natural .
_+_  : Natural Natural -> Natural .

vars N M : Natural
Axiom:  N + 0 = N .
Axiom:  N + s M = s (N + M) .
```

(s s s 0) + (s s 0) = s ((s s s 0) + (s 0)) = s(s((s s s 0) + 0))
= s(s((s s s 0)) = s s s s s 0

Intuitively, this is a proof of "3 + 2 = 5" in natural number algebra.

Semantics of a Memory Model

- A ***store*** represents a snapshot of the memory state at a point in the program execution.
- For each memory location, we can evaluate two properties: content and taintedness (true/false).
- Operations on memory locations:
  - The *fetch* operation Ftch(S,A) gives the **content** of the memory address A in store S
  - The *location-taintedness* operation LocT(S,A) gives the **taintedness** of the location A in store S
- Operations on expressions:
  - The *evaluation* operation Eval(S,E) evaluates expression E in store S
  - The *expression-taintedness* operation ExpT(S,E) computes the taintedness of expression E in store S.

Axioms of *Eval* and *ExpT* operations

Eval(S, I) = I // I is an integer constant
Eval(S, ^ E1) = Ftch(S, Eval(S,E1))
Eval(S, E1 + E2) = Eval(S, E1) + Eval(S, E2)
Eval(S, E1 - E2) = Eval(S, E1) - Eval(S, E2)
… …
ExpT (S, I) = false
ExpT(S, ^ E1) = LocT(S,Eval(S,E1))
ExpT(S,E1 + E2) = ExpT(S,E1) or ExpT(S,E2)
ExpT(S,E1 - E2) = ExpT(S,E1) or ExpT(S,E2)
… …

E.g., is the expression `(^100)-2` tainted in store S?

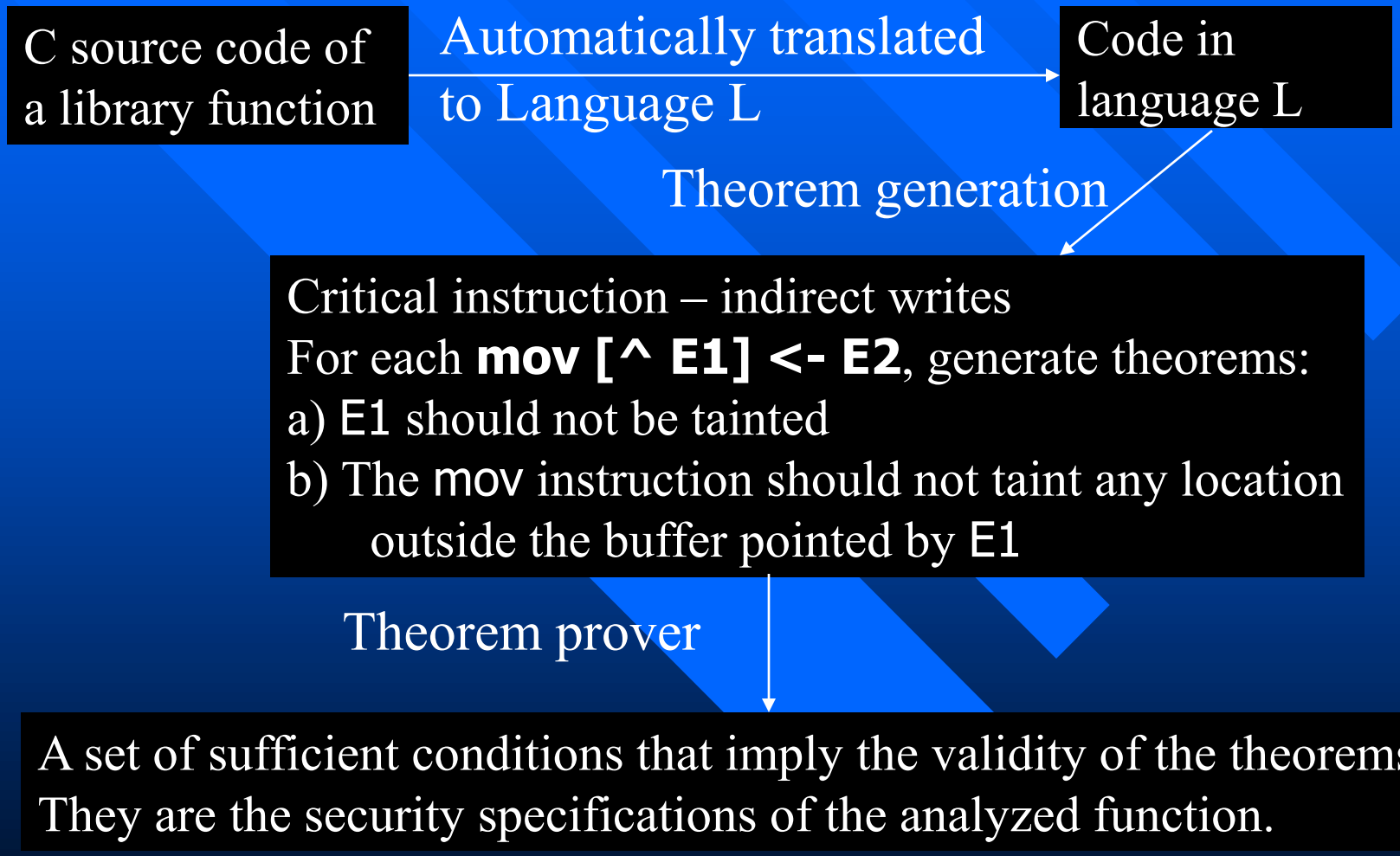
ExpT(S, (^100)–2) = ExpT(S, (^100)) or ExpT(S, 2)
= LocT(S,100) or false =
LocT(S,100)

Note: ^ is the dereference operator, ^100 gives the content in the location 100

Semantics of Language L

- Extend the semantics proposed by Goguen and Malcolm
- The following operations (arithmetic/logic) are defined:
  - +, -, *, /, %, !, &&, ||, !=, ==, ……
- The following instructions are defined:
  - **`mov [Exp1] <- Exp2`**
  - **`branch (Condition) Label`**
  - **`call FuncName(Exp1,Exp2,…)`**
- Axioms defining `mov` instruction semantics
  - Specify the effects of applying `mov` instruction on a store
  - Allow taintedness to propagate from Exp2 to [Exp1].
- Axioms defining the semantics of *recv* (similarly, *scanf*, *recvfrom:* user input functions)
  - Specify the memory locations tainted by the *recv* call.

Extracting Function Specifications by Theorem Prover



Example: strcpy()

```
char * strcpy (char * dst,
               char * src) {
char * res;
0: res =dst;
   while (*src!=0) {
1:   *dst=*src;
     dst++;
     src++;
    }
2:  *dst=0;
   return res;
}
```

Translate to Language L

```
0: mov  [res] <- ^ dst
lbl(#while#6)
   branch (^ ^ src is 0) #ex#while#6
1:  mov  [^ dst] <- ^ ^ src
    mov  [dst] <- (^ dst) + 1
    mov  [src] <- (^ src) + 1
    branch true #while#6
lbl(#ex#while#6)
2:  mov  [^ dst] <- 0
    mov  [ret] <- ^ res
```

Theorem generation

a) Suppose S1 is the store before Line L1, then **LocT(S1,dst) = false**

b) If S0 is the store before Line L0, and S2 is the store after Line L1, then

I < Eval(S0, ^dst) or Eval(S0, ^dst+dstsize) ≤ I =>

LocT(S2,I) = LocT(S0, I)

c) Suppose S3 is the store before Line L2, then **LocT(S3,dst) = false**

Theorem prover

Specifications Suggested by Theorem Prover

- Suppose when function strcpy() is called, the **size** of destination buffer (dst) is **dstsize**, the **length** of user input string (src) is **srclen**
- Specifications that are extracted by the theorem proving approach
  - `srclen <= dstsize`
  - The buffers `src` and `dst` do not overlap in such a way that the buffer `dst` covers the string terminator of the `src` string.
  - The buffers `dst` and `src` do not cover the function frame of strcpy.
  - Initially, `dst` is not tainted

Documented in Linux man page

Not documented

Other Examples

- A simplied version of *printf()*
  - 55 lines of C code
  - Four security specifications are extracted, including one indicating **format string vulnerability**
- Function *free()* of a heap management system
  - 36 lines of C code
  - Seven security specifications are extracted, including several specifications indicating **heap corruption** vulnerabilities.
- Socket read functions of Apache HTTPD and NULL HTTPD
  - The Apache function is proved to be free of pointer taintedness.
  - Two (known) vulnerabilities are exposed in the theorem proving process of NULL HTTPD function.

Conclusions

- A common characteristic of many categories of widely exploited security vulnerabilities: pointer taintedness
- A memory model and a language can be formally defined using equational logic to allow reasoning of pointer taintedness.
- A theorem proving approach is developed to extract security specifications from library function code, based pointer taintedness analysis.

Future Directions

- Provide higher degree of automation on the theorem generation and theorem proving process.
- Apply the pointer taintedness analysis on a substantial number of commonly used library functions to extract their security specifications.
- Compiler techniques for inserting “guarding code” to check unproved properties at runtime.
- Architecture supports for pointer taintedness detection. A module working with RSE (Reliability and Security Engine).