CS6843-Program Analysis	Jan-May 2017
Scribe Week #11	
Lecturer: Prof. Rupesh Nasre	Indrani Roy & Sayantan Ray

.

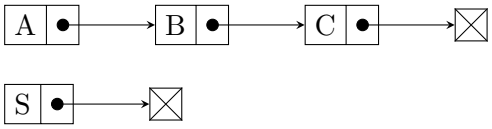
1 Parallelization (contd.)

1.1 Compare And Swap

1.1.1 Classwork - Insertion in the middle of a Single Linked List using CAS

In the previous scribe we saw how to write parallel single linked-list "insertion at the end" routines using atomics. Let us now come up with a routine for inserting new nodes in the middle of the list.

Given a pointer S to a new node, and pointers A and B to nodes in the list such that $A \rightarrow next = B$,



we initialize $S \rightarrow next = B$, and then swing the next field of A to S.

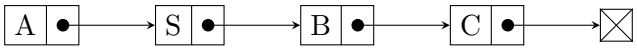
This can be done in the following manner using Compare and Swap (CAS) :

Single Linked List Insertion

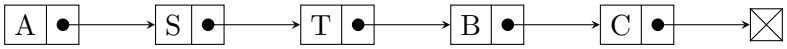
```
while(1)
{
    ...
    // iterate the list and figure out prev_node
    ...
    new_node->next = prev_node->next;
    old_next_node = prev_node->next;
    if(CAS(&prev_node->next, old_next_node, new_node)==old_next_node)
        return;
}
```

Let us say two threads $T1$ and $T2$ are both trying to insert new nodes (S and T) in between A and B simultaneously. T1 sets $S \rightarrow next = B$ and T2 sets $T \rightarrow next = B$. Now since CAS is an atomic instruction, any one of the threads will be able to execute it at a time. Say T1 executes CAS ahead of T2. T1 sees $prev\_node \rightarrow next = B$ and changes it to S. Thus when T2 executes CAS, $prev\_node \rightarrow next != B$ (as T1 changed it to S). Therefore T2 will loop back and retry the insertion with updated links from T1's insertion.

So after T1 executes CAS, we will have :



Then after T2 executes CAS :

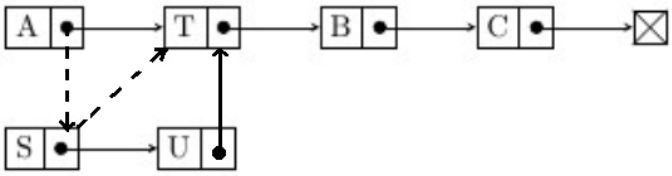


1.1.2 Deletion Routine for Single Linked List using CAS

Suppose two threads (T1 and T2) are trying to delete and insert nodes respectively at the same time in the above obtained linked list :

- T1 wants to delete S.
- T2 wants to insert a node U between S and T.

We may end up with a scenario where T1 makes A point to T, and T2 makes U point to T and S point to U :



Also, if both threads are trying to delete items concurrently, one of the deletions maybe undone by the other. These problems make deletion using atomics alone a bit more complicated than we would like. To satisfy one's curiosity, here is a paper that goes into greater detail regarding this topic.

1.2 Inserting Locks

Code from Slide

```
if(i == 1 || i == 2 || i == 4 || ...)
{
        lock(f(i));
        item = items.remove();
        moreitems = process(item);
        items.add(moreitems);
        unlock(f(i));
}
```

In the above code, let us assume that "$moreitems = process(item)$" takes longer than the rest of the statements. So once a thread acquires a lock, all other threads will have to wait for some significant amount of time. To improve things, we can introduce the following two unlock/lock statements :

Code from Slide modified

```
if(i == 1 || i == 2 || i == 4 || ...)
{
        lock(f(i));
        item = items.remove();
        unlock(f(i));                //This
        moreitems = process(item);
        lock(f(i));                  //This
        items.add(moreitems);
        unlock(f(i));
}
```

Here, *moreitems* is a local variable for each thread, so we don't need to synchronize it between the threads. On the other hand, *items* is a global list which is read and written by all threads. Thus any operation on *items* is enclosed in a lock.

Enclosing these instructions within locks ensures that no second thread can read or modify the shared data resource "*items*" while another thread is modifying it.

Let us observe the dependencies that can arise between two instances of this code, when run on two different threads.

- *items.remove*() is a Read+Write operation.
- *items.add*(*moreitems*) is a Write operation.

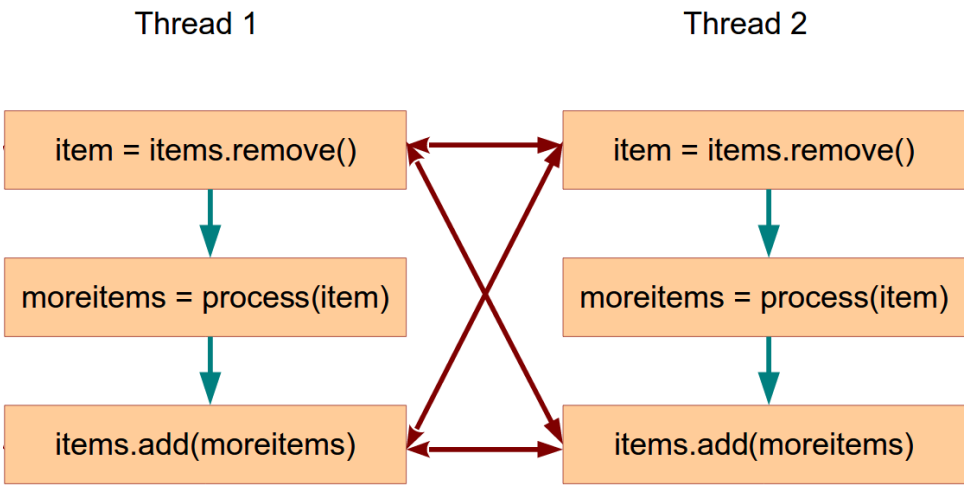


Figure 1: Dependencies

Note : The green arrows represent flow of execution within a thread. As we have a control flow (sequential execution within a thread), we don't need to include dependencies between instructions of the same thread as they are already accounted for.

The above dependency diagram has Bidirectional arrows as the code will mostly be inside a loop. This leads to cyclic dependencies. In the worst case, cyclic dependencies can be dealt with by putting the entire loop code into a single thread. The cycle will be taken care of sequentially. We can parallelize the rest of the non loop code. However, if we are given a trace of a program, then there will be no cyclic dependencies as the loops will be laid out as sequential code.

Data Race vs Data Dependence

Data Race occurs when two or more threads in a single process access the same memory location, with at least one of the accesses being a write operation, and the threads are not using any exclusive locks to control their accesses to that memory.
Dependency on the other hand is when the effect of one instruction depends on the outcome of another, with one instruction being a write operation. Dependency can exist within instructions in the same thread, as well as between instructions belonging to different threads.

So does the above code exhibit both data race and dependency?
Two instances of "*item* = *items.remove*()" in two different threads have a data race. This also means both are dependent on each other's order of execution. In the same thread, "*item* = *items.remove*()" and "*items.add*(*moreitems*)" have dependency, due to which we must execute the instructions of a thread sequentially (maintaining the control flow designated by the green arrows)

Data races can be handled by enclosing the instructions in between lock/unlock instructions, as shown in the above given code. However this doesn't rid us of the dependencies. They may still exist. Parallelization cannot guarantee dependency preservation, and hence provides incorrect outputs. If **order** of the program doesn't matter, then we can parallelize without losing precision. However the significance of maintaining order cannot be determined by most compilers. Such information is generally contributed by the programmer.

Sometimes, even if the dependencies are dissatisfied, we can still end up with a different but semantically correct output.

E.g.– Malloc() statement returns a random memory location. If two threads try to allocate a memory location to some global pointer concurrently, it should lead to a WAW dependence. However we don't really need to satisfy this dependence as *malloc*() assignment isn't affected by order of execution. The threads can execute the *malloc*() statements in any order, as the main aim is to get back any memory location.

1.3 Barriers

A **barrier** is a point in the code, which must be reached by all threads executing the same code, before any one of them can proceed beyond that point. This gives a guarantee that all instructions before the barrier will be executed ahead of the ones written after it. Our goal is to preserve dependencies while trying to parallelize it across threads.

Let us consider the following example :

Barrier

```
                T1                                      T2

S1:          ....                                    ....
S2:    item = items.remove();                  item = items.remove();
    -------------------------- BARRIER --------------------------
S3:    items.add(moreitems);                    items.add(moreitems);
S4:          ....                                    ....
```

In the above code excerpt, following dependencies need to be satisfied :

- S2 of T1 $\longleftrightarrow$ S2 of T2
- S2 of T1 $\longleftrightarrow$ S3 of T2
- S3 of T1 $\longleftrightarrow$ S2 of T2
- S3 of T1 $\longleftrightarrow$ S3 of T2

On introducing a barrier between S2 and S3, we guarantee that both threads have to finish executing S1 and S2 before either thread can start executing S3 and S4. This maintains an order of execution between S2 and S3, thus preserving the dependency between them. What this means is things on either side of the barrier cannot be parallelized.

However, we can still parallelize the phases in between barriers. In the above example, S1 and S2 of T1 and T2 can be parallelized (assuming S1 and S2 have no dependencies). Similarly, S3 and S4 can be parallelized in between the two threads.

Note : In the above example, we actually ended up with more dependencies than we had begun with. (Assuming S1 and S4 had no dependencies, their order of execution is now restricted by the inclusion of barriers. This is equivalent to having a dependency $S1 \longleftrightarrow S4$)

Let us now apply barriers to the code we saw in the previous section and see how many dependencies we can satisfy :

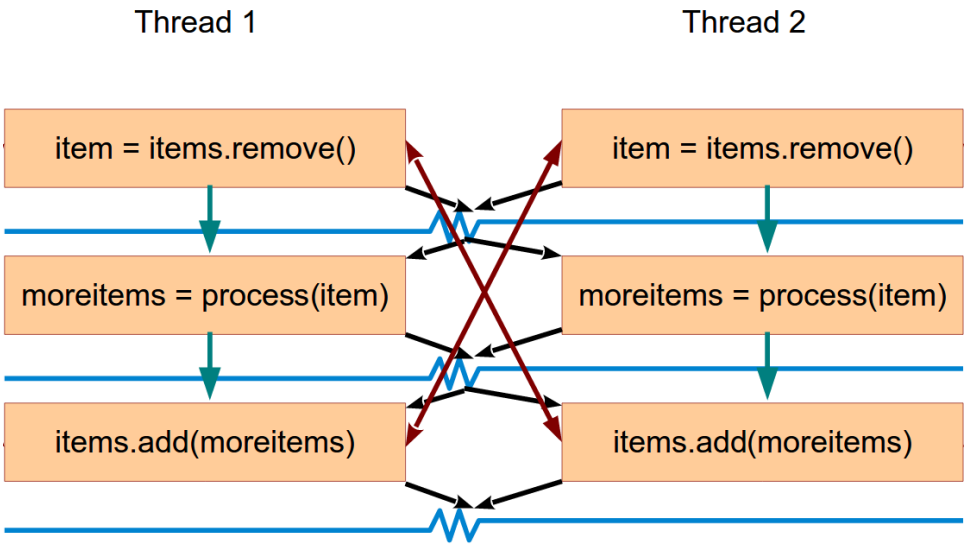


Figure 2: Inserting barriers into the previous dependency diagram

- The Blue lines represent the barriers
- The Black arrows above the barriers represent the meeting of all global variables at that point. This means that all threads finished up to this point shall wait for others who haven't. Once they do, all global variables will be synchronized at this point, and only then can all threads once again diverge and carry on execution beyond the barrier point. This is indicated by the black arrows below the barriers.
- The Maroon arrows indicate the dependencies that have been satisfied due to inclusion of the barriers.
- The Green arrow indicates sequential execution within a thread.

Note : The 1st and 2nd Barriers are both not simulataneously required. They indicate suitable positions to includce a barrier. Either one will be enough to satisfy the criss-cross dependencies. Also, the dependencies between the two instances of "$item = items.remove()$" is not satisfied with the help of barriers. Same for the two instances of "$items.add(moreitems)$". As instructions of

different threads inside a phase can execute in any order, dependency is not satisfied. However, dependency between "$item = items.remove()$" of T1 and "$items.add(moreitems)$" of T2 is satisfied as barrier guarantees the former getting executed first.

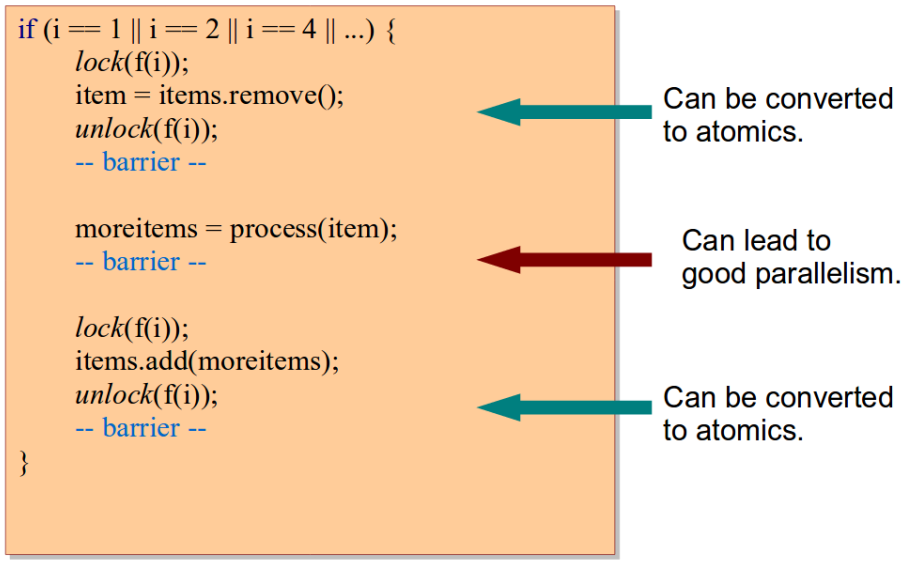


Figure 3: Inserting barriers into the given code

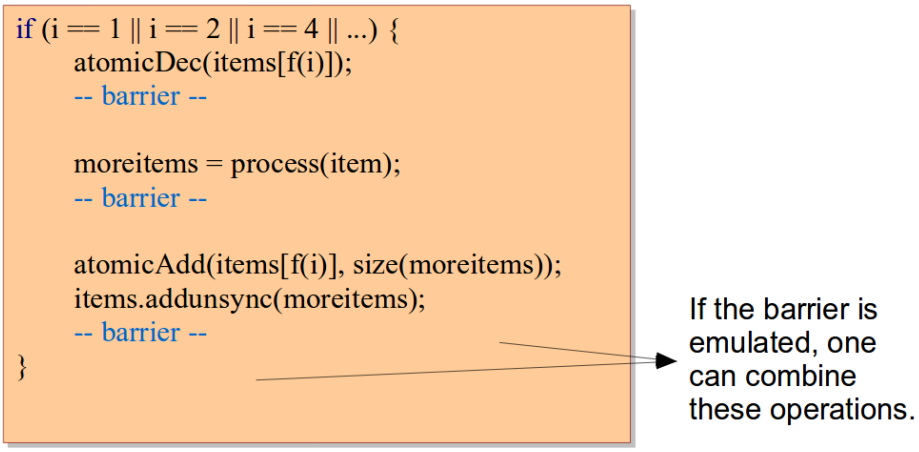


Figure 4: Implementing Atomics

Barriers and Dependences

- A barrier may be considered in effect similar to loop distribution.
- If dependences are sparse, use atomics/locks; otherwise barriers work well.
- A barrier may add more dependences than required.

- But it must preserve all the existing dependences.

1.4 Limitations of Static Parallelization

- Some programs cannot be effectively parallelized using static techniques. – e.g. *graph algorithms, pointer-savvy programs*

  One such instance of a graph problem is *Delaunay Mesh Refinement*. One can find more on this here.

- Existing static optimization techniques (analysis) are also very conservative for such programs.
- Ineffectiveness of static techniques forces us to use dynamic approaches.

1.5 Irregularity

Data Access or Control flow patterns are often unpredictable at compile time. Simply put, the compiler finds it difficult to carry out the analyses if the data access is dynamically determined. Let us consider the example :

```
for(j=0; j<B.length();j++)
        scanf("%d", &B[j]);
for(j=0; j<A.length();j++)
        scanf("%d", &A[j]);
for(i=0; i<N; i++)
{
        ....
        C[i+3] = ... B[A[2*i]];
        ....
}
```

It is impossible for the compiler to know the location of B, as we need to know the value of A[2*i]. We can find an access pattern for C[i+3], as we have a bound for i, but we do not have the same for A[2*i], as the value of A[2*i] can be any user defined value. Thus B[A[2*i]] cannot be analyzed statically. We need dynamic approaches.

Simply put :

Static Access
$C[5] = B[4];$

Dynamic Access
$C[5] = B[A[4]];$

Graph based programs are good examples of irregularity in the real world. At compile time, information regarding adjacency among nodes is not known. On the other hand, a program like Matrix Multiplication is a good example of a regularity, as we strictly know which rows and columns need to be accessed in order to obtain elements of the product.

2 Security Analysis

Most of the times security is one of the main aspect of a program. Followings are some ways in which security attacks can take place.

- **Memory leak** - This happens when memory is allocated but there is no reference to it. Sometimes it might happen that a programmer allocated some memory and it is not freed after usage, so in this case any other person might be able to access that data.

```
func()
{
    int *p= malloc(sizeof(int));
    return 0;
}
```

- **Accessing unallocated memory** - Accessing any out of bounds array element, or null pointers.
- **Accessing unauthorized memory** - Trying to access some other process' variables or some other process' memory location. It might create problems for some privileged processes.

Among the above mentioned types last one is mainly programming language specific. Some other effects of security threats are: crash, non-termination, wrong output, unintended actions etc which might be caused by dangling pointers, buffer overruns, null pointer dereference, wrong opcode[1], arbitrary data-change etc.

C language is more susceptible to buffer overflow attack. As space and performance is primary concern of C it ignores some of the security aspects which causes vulnerability. For example we can say C allows direct pointer manipulation.

Some standard library functions are also not very safe to use if they are not handled carefully.
E.g. *gets, strcpy, strcat*, etc. do not give any bound on string length. So it might lead to buffer overflow attack as any arbitrary length of string (greater than the size of array) can be manipulated using these functions.
– To solve this problem we can use *str**n**cpy* instead of *strcpy* which will allow only n bytes to copy thus gives bound and combats buffer-overflow attack.

[1] If attacker change any bit value in the opcode, the newly generated bit pattern may not represent a valid opcode.

2.1 Design of an analysis of NULL pointer

Here we are checking if a pointer p is going to be $NULL$ or not. If p points to $NULL$ in any one path we include it in our set. Hence it is "May Analysis"

1. **Direction:** Forward
2. **Analysis Dimensions:** Flow Sensitive
3. **Data Flow Facts:** Nullity of pointers.
4. **Transfer Function:** We can write transfer function as following:
   $Out(B) = f(In(B))$
   $In(B) = meet(Out(P))$ [*meet* operation is dependent on the type of analysis we are performing].
5. **Meet Operation:** Union (As we are doing 'may analysis' here)
6. **Statements of Interest:**
   - Copy statement: $p = q$
     If q is pointing to $NULL$ p will also points to $NULL$
   - $p = f()$
     Nullity of p will depend on the return value of the function. We need to perform inter procedural analysis in this case.
   - $p = malloc()$;
     This will set nullity of pointer p to $NOTNULL$
   - Store: $*p = q$;
     Nullity of points to set of p will be set according to nullity of q
   - Load: $q = *p$;
     Nullity of q will depend on nullity of points to set of p
   - $p = NULL$;
     Set nullity of p to $NULL$
   - "Address of " operator : $p = \&q$;
     p will point to address of q. This will set nullity of p to $NOTNULL$

2.2 Stack Smashing

Stack smashing is a stack buffer overflow attack deliberately caused by some attacker. Here attacker fills buffer with some arbitrary length of executable code and takes control of the program. If the affected program is running with special privileges, or accepting data from untrusted network hosts then this bug is a potential security vulnerability.

2.2.1 How Stack Smashing works

In the code given below initially all the elements of array is pushed inside the stack. Before the control goes to the function f the return address of the function main is pushed on to the stack, here it is $L1$. From the function f other elements are pushed inside the stack after $L1$. IR is also given below.

Code from Slide

```
C-code                                          Assembly-Code

void f(char *b)                          f:     pop b
{                                               push L2
        gets(b);                                push b
        L2:                                     jump gets
}                                               ...
void main()                                     pop PC
{                                        main:
        char a[4];                              mov a, SP
        f(a);                                   add SP,4
        L1: ...                                 push L1
}                                               push a
                                                jump f
```

If the return address to main in the stack (here $L1$) is modified by some malicious programmer, he can shift the control to some other code. As *gets* has no bound on input string it can read more bytes than the number of bytes allocated to the array. So if any malicious programmer gives input greater than the total size of array it will override the other location of the stack. Having proper information of array size attacker can change the return address $L1$ to some other address where malicious code is present.

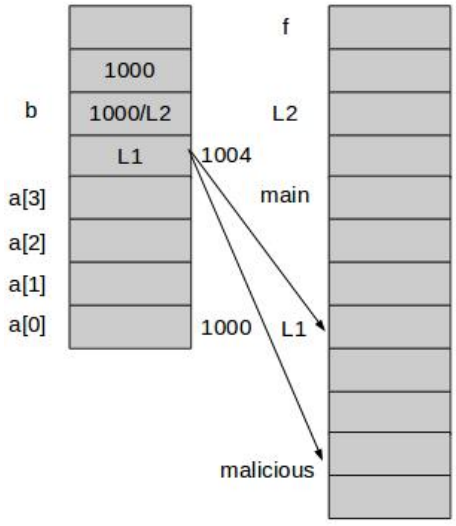


Figure 5: Stack Smashing

2.2.2 How to solve Stack Smashing problem:

The problem of stack smashing can be handled in two ways: Prevention and Detection

- **Prevention:** We can instrument the code in such a way that we pass the length of the array to the function which is using *gets*. So, while filling the array the length also can be kept in mind.
- **Detection:** To detect stack smashing we can use a special bit pattern called *sentinel*/*canary*. *canary* is a special bit pattern which is inserted between the data of array and the return address(As given in fig 6).

```
C-code                                          Assembly-Code

void f(char *b)                          f:     pop b
{                                               push senti
        gets(b);                                push L2
        L2:                                     push b
}                                               jump gets
void main()                                     ....
{                                               intact senti?
        char a[4];                              pop PC
        f(a);                            main:
        L1: ...                                 mov a, SP
}                                               add SP,4
                                                push senti
                                                push L1
                                                push a
                                                intact senti?
                                                jump f
```

This *senti* value is checked before function f is called and after f is called and compared. If stack smashing occurs then the *senti* value is likely to be changed before the return address $L1$ is overridden. So, while comparing before and after *senti* value if we find that the values are different then we can say that stack smashing happened.
Sometimes it might happen that the attacker is aware of the fact that *senti* is used for security check. In this case the attacker might tactfully override *senti* with exactly previous *senti* value, so that the stack smashing cannot be detected. To avoid this, generally a random bit pattern is used as *senti*, hence it will be very difficult for the attacker to guess the *senti* value.

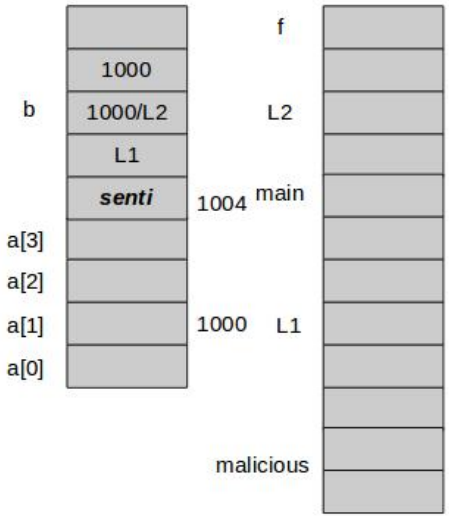


Figure 6: Use of Sentinel/Canary

Is heap smashing possible? In heap we allocate data dynamically. Mainly program data is added to the heap in runtime. If data is written in heap without any bound then it might cause heap overflow. Say, two buffers are allocated in heap next to each other. Writing on one buffer without any bound might cause other's data to be overwritten. Sometimes this might lead to overwriting some critical data structures in the heap such as the heap headers, or any heap-based data such as dynamic object pointers, which in turn can lead to overwriting the virtual function table.

What is the output of the given program?

Code from Slide

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";
main()
{
        printf(f,34,f,34,10);
}
```

The program will print the program itself. We call these type of programs as: *Quine*
The format of printf given above is prone to buffer overflow attack as it's not taking any format parameter. As Quine prints its own code attacker can get access to any source code using Quine.

2.2.3 Static Buffer Overrun Detection

Sentinel can protect against naive malicious code or memory inadvertent code. But there will be more intelligent attackers who can bypass this. Due to this sometimes human involvement is needed. Human involvement guarantees the safety of the code to some extent at the cost of time and sometimes inaccurate results (False positive and False Negative).

False positive: If our analysis says that the access is "unsafe" and during runtime it turns out to be "safe" then we call it as false positive.
False negative: If our analysis says that the access is "safe" and during runtime it turns out to be "unsafe" then we call it as false negative.

Compared to analysis which only gives false negative, false positive analysis is more conservative.

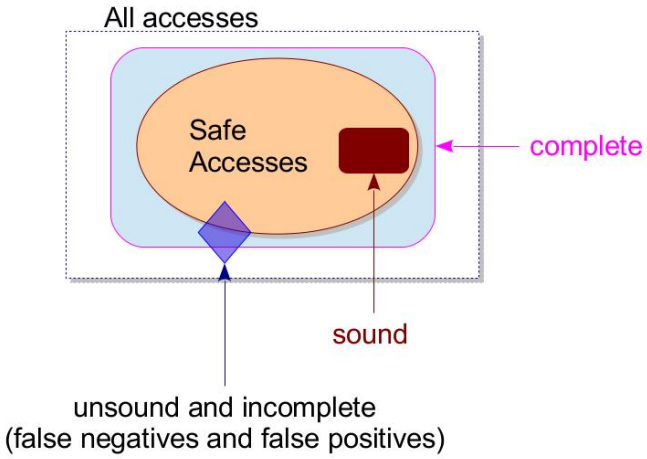


Figure 7: Memory Accesses

In the above diagram ($Fig7$) *Orange* part shows the accesses which are safe at runtime and the whole block (including $White$) are all the accesses done during runtime. $White$ region contains those accesses which are unsafe.
If the result of our analysis is within the safe access region (*Red* box in figure) then we call it as "*Sound*" (if our analysis tells us the access is safe, then it **must** be safe).
If the result of our analysis includes superset of all the safe accesses of runtime, we call our analysis as "*Complete*" (*Blue* region in Figure).
In general, as human involvement is there in the analysis, our analysis will be the *Purple* region of Figure. Which is both unsound and incomplete as it includes both "False Positives and False Negatives.

2.2.4 Pre-conditions and Post-conditions

To avoid buffer overrun sometimes users can provide some additional information regarding the sizes of arrays or other data structures. Using these information, the compiler can define a bound on arrays and might be able to prevent attack in some cases.

Using Annotations: Annotations are provided by the programmers which will define the properties of the arrays like size of the array etc.
Some terms are defined like $minDef, maxDef, minUse, maxUse$ to write annotations.
@$requires$ and @$ensures$ are used to write annotations. @$requires$ is *precondition* which defines the conditions which need to be checked before the operation on the array. @$ensures$ is

postcondition which needs to be verified after accessing the array. Annotations can be written as following:

```
char *strcpy(char *s1, char *s2)
/* @requires maxDef(s1) >= maxDef(s2) */
/* @ensures maxUse(s1) == maxUse(s2) and result == s1*/;
```

In the above piece of code we wrote pre-condition and post-condition for *strcpy* operation.
Precondition says that $maxDef$ (Maximum defined value) for $s1$(destination) should be greater than $s2$(source).
Postcondition says that $maxUse$ (Maximum index value till where we can read) for $s1$ should be equal to $s2$. Also we need to verify that the result after *strcpy* should be equal to the source.

```
 void *malloc(size_t size)
/* @ensures maxDef(result) == size or result == null */;
```

For malloc we don't need any precondition. For postcondition we need to verify that the allocated value for the result ($maxDef(result)$) should be equal to the size defined or it should be $NULL$.
Sometimes we use $notNull(ptr)$, $restrict(ptr)$ etc to write annotations.
$restrict(ptr)$- It Gives compiler a hint that ptr is the only pointer which is pointing to that memory location and the location will be modified by ptr only.

2.2.5 Inferring Constraints

If compiler has to give the constraints on the bound of array without any help from programmer then compiler can use following information to add constraints.

- From for loop: As in most of the cases array access pattern of the for loop is sequential, compiler can find the access pattern and get some idea regarding the bound of arrays. This bound check is identified at compile time as we want the system to be guaranteed to be secure at compile time.
  while loop cannot be used in this case as the array access pattern of *while* is random in general.
- From the array declarations and malloc statements compiler can determine the array size.
- Using conditions of code we can figure out which domain and locations of array we are accessing.
- Small number of heuristics often cover large part of the program.

Security vs Optimization

Security	Optimization
Security is must for any system. This is safety property. We cannot compromise security of a system.	Optimization is optional. This is may property.
To achieve a secure system we need to perform some expensive operations e.g. flow sensitive analysis etc	For optimization we can settle with less expensive analysis like flow insensitive etc.

Some points regarding constraints:

- We check user annotations using the compiler defined constraints and verify the constraints.
- Effectiveness of user annotations are checked using these constraints.
- Sometimes to reduce the number of constraints we perform flow insensitive analysis.
- While checking the bounds in loop it's better to give condition in such a way that it does not involve accessing the array elements. E.g.

  `while (a[i] != '\0') vs while (i < n)`

while (a[i] != '\0')	while (i <n)
In this case we need to look inside the array and check the content of array, so this will be more complex	This is an affine expression so it is very easy to perform as no array access is required.

Precision vs. Efficiency: Let us discuss precision and efficiency based on the given program:

```
void main() {
        int *a;
        a = malloc(N);
        ii = N / 2 + f(N);
        a[ii] = 0;
}
```

```
...
int f(int N) {
        return N % 5;
}
```

If we perform intra procedural analysis then we cannot say anything about the bound of ii as the value is determined by the function $f(N)$ and we are unaware of the internal structure of the function. This conservative analysis will be efficient but less precise as we need to assume the return value of function which might lead to false $+ve$.

If we perform inter procedural analysis then we get to know the bound which will be helpful in analysis. But inter procedural analysis is expensive compared to intra procedural analysis. So, this will be less efficient but we will get more precise information in return.

2.2.6 Stack Smashing in gcc

Code from Slide

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    char buff[15];
    int pass=0;
    printf("\n Enter the Password \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
        printf ("\n Wrong Password \n");
     else
        printf ("\n Correct Password \n"),    pass = 1;
        if(pass)
        /* Now Give root or admin rights to user*/
                printf ("\n Root privileges given to the user \n");
        return 0
}
```

Older gcc

```
Enter the password :
hhhhhhhhhhhhhhhhhhhh
Wrong Password
Root privileges given to the user
```

This happens in older version of gcc mainly because *gets* have no bound on string length. So when we give string greater than 15 it will override the content of *pass*, as *pass* is stored just after the $buff$. In most cases it will override 0 with a non zero value. Hence the *pass* variable value will become non zero and user will get access to root as per given condition.

New gcc

```
Enter the password :
hhhhhhhhhhhhhhhhhhhh
Wrong Password
*** stack smashing detected ***: ./a.out terminated
```

In newer version of gcc, if string length is greater than the size of buffer then we get message as "stack smashing detected" and the program gets terminated.

New gcc with -fno-stack-protector

```
Enter the password :
hhhhhhhhhhhhhhhhhhhh
Wrong Password
Root privileges given to the user
```

-fno-stack-protector is a flag which emits extra code to check buffer overflow attack, which is by default enabled to detect stack smashing in newer version of gcc. So when we compile our code with this flag, stack smashing is not checked and we get root access.

2.3 Vulnerability Analysis as a DFA

Here we need to check if the array accesses are within the bound defined in the program.

1. **Direction:** Forward
2. **Analysis Dimensions:** Flow Sensitive/ Path Sensitive
3. **Data Flow Facts:** We need to track the array length, use/def of variables, array index and other variables. Mainly the data flow facts will be values of index variables.
4. **Transfer Function:** We can write transfer function as following:
   $Out(B) = f(In(B)) \equiv Out(B) = In(B) + (Gen(B) - Kill(B))$
   $In(B) = meet(Out(P))$ here *meet* operation is dependent on the type of analysis we are performing.
5. **Meet Operation:** If we are performing safety analysis then the meet operation will be "$Intersection$" else we can use "$Union$". In case of security generally we need to perform safety analysis, hence we take "$Intersection$" as meet operator.
   In some cases we need to take "Max" as meet operator. E.g. say from two basic block same index values are coming to a basic block (from the predecessor of a basic block). Let us assume the index values from two predecessors are 5 and 7 respectively. So, in this case we need to take the maximum value of index coming from predecessor to determine the bound of the array.
6. **Statements of Interest**: Array access, malloc, assignments, pointer arithmetic etc.

References

- Professor Rupesh Nasre's slides and lectures.
- *Lock-free Linked Lists using Compare and Swap* - by John D. Valois.
- http://www.thegeekstuff.com/2013/02/stack-smashing-attacks-gcc/
- The Almighty Internet.