What is the paper about?

- This paper presents a new approach to provide security to the data used in the security-sensitive operations during run time.
- This is done by introducing some extra code into the source code and monitoring it during the run time.

Implementation

- They introduce two arrays, one for storing ‘0’ or ‘1’ for the entire memory available for the program and the other for storing ‘0’ or ‘1’ for the stack values of the program stack, namely,

  \[ \text{char tagmap}[2^{30}] \text{ and char tagstack}[2^{15}] \]

  0 means that there is untainted data at that particular memory location or stack location and 1 means there is tainted data.

- So, let us consider two source code examples which are instrumented with this approach. This means whenever there is a new variable declaration (new memory to be allocated) or a variable is being passed as an argument to a function, this approach introduces some piece of code to update the stack array or the map array accordingly.

```c
void foo(void)
{
    int X,Y;
    read(STDIN,&X,sizeof(X));
    tagmap[&x>>2]=1;
    Y=X;
    tagmap[&Y>>2]=tagmap[&X>>2];
    tagstack[const++]=tagmap[&Y>>2];
    bar[Y];
}
void bar(int Z)
{
    tagmap[&Z>>2]=tagstack[...]; // it the argument gets stored in the stack.
    if(tagmap[&Z>>2])
        alert();
-----------------------→ check_policy(&Z);
    dangerous_operation();
}
The above piece of instrumented code updates tagmap and tagstack accordingly and before the dangerous_operation() is executed, it should be forced to have a check_policy on the arguments on which the operation will be performed. Now, it’s the programmer’s discretion to choose the way he wants to implement the check_policy.

- Few things to check with this piece of code is that, if foo and bar are both in the same .c file, compiler can see them at the same time and the code can be instrumented.
- If foo and bar are in different .c files, even then the compiler can access the folder in which they exist and the code can be instrumented.
- If foo program is instrumented and bar is not, even then the code works fine as the tagmap and tagstack will be updated as in foo and bar will be executed as usual (with dangerous_operation executed probably).
- But if the foo program is not instrumented and the bar program is, the bar then tries to dereference the tapstack array for some value that does not exist (not updated in the foo as it is not instrumented) and so the program crashes.

Why tainting at byte level and not object level?

- There are many cases in which tainting at object level does not solve the problem of finding out which part of the object is exactly tainted.
- In SQL, tainting at object level is definitely not helpful as the common string part in the SQL queries can never be tainted and it is only the arguments (or values) that can be tainted. And hence, this approach solves the problem by providing byte level tainting.

Evaluation and Issues

- Due to the additional code and checks, the execution time of the program increases.
- There is around 25% memory overhead due the huge array for the entire program memory. For this, they allocate memory when required instead of allocating an array for the entire memory at once.
- Effectiveness of the code is compromised with insertion of random statements.
- There can be additional programmer effort per library.
- As mentioned above, not being able to instrument pieces of code if suppose they are in a library is an issue with this approach. It does not talk about the libraries.
- Also, compiler optimizations are not so good at array dereferencing and hence they try to eliminate memory references whenever possible. For example, the foo program above can be changes this way, by using local variables wherever possible.

```c
void foo(void)
{
    int X,Y;
    char taint_Y;
    read(STDIN,&X,sizeof(X));
    tagmap[&x>>2]=1;
}```
Y = X;
taint_Y = tagmap[&X >> 2];
tagstack[const++] = taint_Y;
bar[Y];
}

- Now, when we say
  
  Tagmap[&p >> 2] = 0;
  *p = 0;

  and if p maps to tagmap? We have to prevent the program from writing to this chunk of memory as it means to make some index in the tagmap as untainted. As we know, a new index in the array (tagmap) is allocated when required (dynamically) and so when something like the above happens, it can be checked it is trying to write something that is not already allocated in the tagmap and if it is writing a zero. If such is the case, an alert can be raised.