"Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels"

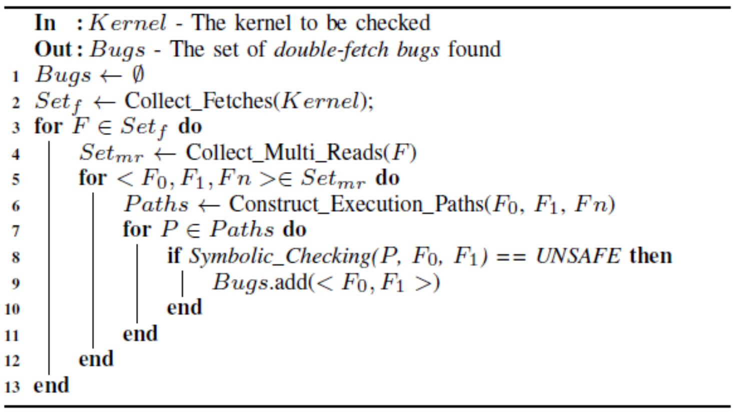
In this paper, the automatic detection of double fetch bugs in the Linux and FreeBSD kernel is discussed using DEADLINE tool designed and developed by the authors of the paper. This article first provides a precise and formal definition of double-fetch bug, distinguishing between this bug and multi-read. In other words, every double fetch bug is multi-read, but the opposite is not always true. The official definition outlined in this article is as follows:

1- There must be two times or more reading of the user space memory (multi-read occurred).
2- There must be an overlap between the two read memories.
3- There must be at least one of the data or control dependencies between the two reads from the user space over the overlapping space.
4- It must not be possible to prove that the dependency is still in place after the second read.

An example of control dependency is sanity check; in this situation if the value is not valid, kernel will stop the system call. An example of a data dependency is first reading a variable such as a "size" from user space, storing it in the kernel, validating it, then receiving all of the data from the user's space. There is a data dependency in this case, if the size will be read in the second read (i.e. it is in the overlapping area).

This bug is important because if the user program is multi-threaded and can alter the data in overlapping space between the two fetches, then it can be malicious depending on the kernel code and its exploitability. The attack is one of the race condition attacks in which the kernel competes for reading and the user's thread to write the memory. The DEADLINE's algorithm is as shown in algorithm 1.

Algorithm 1 abstract view of DEADLINE algorithm



```
In  : $Kernel$ - The kernel to be checked
Out : $Bugs$ - The set of *double-fetch bugs* found
$Bugs \leftarrow \emptyset$
$Set_f \leftarrow$ Collect_Fetches($Kernel$);
for $F \in Set_f$ do
    $Set_{mr} \leftarrow$ Collect_Multi_Reads($F$)
    for $< F_0, F_1, Fn > \in Set_{mr}$ do
        $Paths \leftarrow$ Construct_Execution_Paths($F_0$, $F_1$, $Fn$)
        for $P \in Paths$ do
            if *Symbolic_Checking*($P$, $F_0$, $F_1$) == *UNSAFE* then
                $Bugs$.add($< F_0, F_1 >$)
            end
        end
    end
end
```

This tool first collects all fetches in the kernel code to a set. This is due to the fact that a limited number of functions are available for reading from the user space at the kernel (due to better management of the boundary between these two memory spaces). Thus, the first step can easily be done. Then for each fetch, all the double-reads this fetch is involved in are stored in pairs of fetches in another set (Set_{mr}). How to do this is explained in algorithm 2. Then, the execution paths from the first fetch to the second are placed in the set of paths, and for each path, a static symbolic execution is applied to it to check the existence of the conditions described in the definition of the bug in the previous section.

Algorithm 2 construction of fetch pairs which constitute double-read

```
In  : $F$ - A fetch, i.e., a call to a transfer function
Out : $R$ - A set of triples $< F_0, F_1, Fn >$ representing *multi-reads*
$Fn \leftarrow$ Function that contains $F$;
$R \leftarrow \emptyset$;
$Set_{up} \leftarrow$ Get_Upstream_Instructions($Fn$, $F$);
for $I \in Set_{up}$ do
    if $I$ is a fetch then
        $R$.add($< I, F, Fn >$)
    end
    if $I$ is a call to a function that contains a fetch then
        Inline $I$, redo the algorithm
    end
end
$Set_{dn} \leftarrow$ Get_Downstream_Instructions($Fn$, $F$);
for $I \in Set_{dn}$ do
    if $I$ is a fetch then
        $R$.add($< F, I, Fn >$)
    end
    if $I$ is a call to a function that contains a fetch then
        Inline $I$, redo the algorithm
    end
end
```

To find the fetches which constitute multi-read with fetch F, we need to know the function that F is called in. Then consider F as a reference point in this function and check the higher and lower instructions. If the instruction is a fetch, we add it to the multi-read set and if it is a call for a function containing a fetch, then we open it and repeat the algorithm again.

The control flow graph is also used to find the paths between two fetches. In this step, instructions that do not affect fetching (or are not affected by it) are deleted. An instruction is affected by fetch if derived from or constrains the fetch. An instruction affects the fetch if address or size of the fetch is derived form or constrained by it.

Finally, by forming a symbolic representation using the output of the LLVM code for the path between the fetches, this symbolic representation is handed to an SMT solver to decide on presence or absence of a double-fetch bug.

By applying this static analysis to the Linux kernel and FreeBSD, 23 and 1 new bugs were discovered respectively in their code. These bugs are not recognizable by previous methods for various reasons. For example, one of the previous methods uses lexical analysis (by finding some specific patterns in the code), while all occurrences of the bug cannot be categorized in one or more specific patterns.

Limitations of this method include: 1) Non-convertible codes to LLVM 2) Restricting number of paths to 4096 Different paths between fetches and 3) discarding inline assembly.

References

[1] M. Xu, C. Qian, K. Lu, M. Backes and T. Kim, "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, US, 2018.