Week 7

Defending NAND-Flash

References

Maecenas hendrerit tortor velit[1], nec hendrerit[2], mauris accumsan feugiat[3].

Purpose of the week

We need to define implementation logistics more clearly, including:

- ~~Defining how functions are to be implemented in a Flash Block~~
- Determining how feasible is using 3D NAND Flash as logic unit, by analysing lifetime versus number of inputs processed
- ~~Defining how basic key structure could be related to either V_{PASS} and V_{READ} application on a particular Flash cell in a NAND string~~
- Understanding how a NAND flash is optimised on system level, that ensures uniform usage of cells throughout a block
- Understanding the capability of the above "control" circuit, and how we can bend it according to usage
- Defining how varying inputs would lead to usage of different cells for homogenous usage of the entire string
- Defining how locking can include inducing a lock in the sequential part of Flash implementation
- Analysing the key space of the 2 types of implementation
- Analysing with an example, the entire pipeline, from function implementation, getting output, to cell allotment. Say, create a 1-bit adder!

Brief

Having realised a core idea on how to lock functions implemented using 3D-NAND Flash Logic-in-Memory, this week tries to uncover any loopholes during the entire realisation of a logic implementation, and locking the same.

SideNotes

Revisiting what was explained before, the adversary needs to feed the initial state of the "control" circuit FSM that decides the subsequent cells to be used. Note that there are again 2 seperate processes, cell allotment, and V_G value allotment. They should ideally work in sync, to always give correct results, independent of input values or the function itself. The initial state of FSM of cell allotment is decided by the "control" circuit, and this needs to be hidden from the adversary, because that itself is the secret key. The adversary inputs the key which would now run the FSM of V_G value allotment. To hide this, the designer can instead take another input secret key init state, that has some fixed transitions only known by the designer, which would lead to eventual synchronisation of both FSMs.

Another thing to note. Input cells are uniquely identified by cell number and string number (we're just talking about a NAND Flash Block).

The FSMs described above, already implemented by the "control" circuit (not exactly as needed, unknown how, but existence can be proved because of optimisation algorithms implemented inherently in the memory). I assume they work as follows:

The Programming FSM has a state that is equivalent to which cell in a particular string should be used, and this state changes on the basis of the previous state. So if the user wants to encode a function: $\mathbb{F}(A, B, C, D, E, F) = DAB + EBC + FCA$, she would be essentially requesting for 3 cells in 3 different strings.

> Need to understand how input embedded cells, which may be any in number, would have a corresponding Vg vector, that can be deterministically decided by the Allotment FSM's init state. Difference lies in Programming State having init from designer, Allotment having init from user. To be safe, the user shouldn't be able to understand how to map transitions of programming FSM back to init state which led to here, else that would reveal the secret key. Else think otherway as you described above, to create mapping of secret key to actual key of Vg vector.

Topic 2:

Now is the time to think like an adversary.

You know the architecture, and the constraints. Think how you would go for getting the secret key, cause this time you can't bypass it at all. The only way the circuit works if you get the key.

Note that the locking is agnostic of the logic implemented. Once the proper data bits have been loaded, activating correct gates would give the correct output. Therefore it boils down to the adversary knowing the Vg vector. In order to verify that she has the correct key, she would need to apply that key, and check outputs of multiple inputs, with each input needing a new loading process altogether.

We are not protecting IP in this scenario, because the logic is implemented by the adversary itself. We're just protecting the right access of the device itself.

Week 7