Week 6

Defending NAND-Flash Logic

References

Active Hardware Metering for Intellectual Property Protection and Security[1]

Purpose of the week

Defining the function implementation through Flash Logic-in-Memory, describing it's fallacies. Furthur describing the assumptions taken before describing the defense strategy itself, I'll call it WordLineExchange.

Brief

Having realised a core idea on how to lock functions implemented using 3D-NAND Flash Logic-in-Memory, this week I try to cover an example, and derive inspiration from Active Metering paper for implementing a one-time, unique to device, secret key that persists throughout changing inputs and input embedded flash cells.

Logical Functions in 3D-NAND Flash

As explained in the last report, the idea behind this is to use the structure of 3D-NAND Flash, which consists of 177 flash cells in series as a string, with 16K strings in parallel, all strings having a BitLine (BL) at "top", selected through a BitSelectLine (BSL), and each cell being "activated" by WordLine (WL) connected to it's gate.

For example, select a function needed to be implemented, say

$$F(A, B, C, D, E, F) = DAB + EBC + FCA$$

We choose D, E, F as the inputs applied at BL (this choice matters). For simplicity, we choose the first two cells in first 3 strings for encoding F. For this, A, B, C values are already stored somewhere in the memory, and through some mechanism, A, B are transported to the first cell and second cell in the first string, with the cells storing these bits by changing it's V_{TH} accordingly. Similarly done for B, C and C, A. This finishes the function implementation itself. Now to obtain the output for this particular combination of bits for this function, BSL of these strings is put to 1, apply D, E, F at BL_1, BL_2, and BL_3 respectively. Apply V_{READ} on cells embedded with input, and V_{PASS} on the rest of cells in the activated strings. The output would be obtained at CSL, "bottom" of all strings in that block, of course output being just 1 bit.

Problems

3D NAND Flash Logic-in-Memory is a novel idea indeed, but has a few hurdles nonetheless.

We're facing the following issues:

1. Lifetime of NAND Flash is low compared to computing speed needed these days. So we need to utilise the massive number of Flash string to compensate for low lifetime, thus making the entire block feasible to use practically.
2. A function being implemented is agnostic to what the inputs are. The strings just perform AND operations in series and OR operation in parallel strings. It now lies on the user to encode the input appropriately in order to obtain the results accordingly. Encoding input would involve using data stored somewhere else in the Flash block, and piping it to the correct locations (particular cell) in correct form (the bit or it's complement) determined by the "control" circuit as above. This needs to be performed by some other mechanism (let's call it "allotment mechanism") that is yet to be described.
3. Implementing functions usually involve smaller SoPs, that keeps them modular. Because of massive 177 series (and thus literals) per minterm possible, it would be a waste of storage cells (or they may be still in use, but ignored due to V_{PASS} but again, logistics undefined on how to arrange cells involved in logic, and cells used as storage) and massive time delay to implement smaller functions. Large SoPs with only one output seems less used than smaller functions with multiple outputs, which this architecture can handle by involving another block, but that's a big investment.

These are some to name a few. Some more thinking needs to be done in order to uncover more problems in this architecture. Exposing problems helps in avoiding making wrong assumptions later on about this.

Protecting a function-agnostic circuit

As noted above, the entire implementation is agnostic of the actual logic being implemented. Once the user decides what logic is needed, she needs to encode the bits appropriately through allotment mechanism in the layout as explained above.

Note that such encoding is not something special needed by this architecture. Von Neumann architecture needs a set of registers and a seperate processing unit on entirety, with inputs being encoded conveniently because of years of refinement of the layout organisation. If this idea is developed upon, one can expect to find solutions of such mechanism being efficiently implemented.

So we need to ensure that irrespective of any kind of encoding, the output gets corrupted if the user is not autheticated, i.e. doesn't have a secret key.

Here comes the first idea:

Use WordLine voltages to corrupt the output. Note that the function being implemented is seperate procedure than obtaining it's output. So once the user encodes her inputs according to her required function, she would expect the rule to still be applicable. Note that the user herself doesn't have a say on which cells should be used to encode particular inputs, because in a string, all cells can be symmetrically used, and the "control" circuit chooses the set of cells for homogenous usage.

Therefore the user doesn't know which cells should have V_{READ} (consider as key bit 0) applied on it or V_{PASS} (consider as key bit 1). As the "control" circuit decides this deterministically on the basis of the designer who created it, we would know which cells in sequence would be used, so the secret key would be having those cells alloted key bit 0, and other cells alloted key bit 1. This may be provided to the user to enter manually in some format, maybe the key even being written by the user in another string, and read by the "control" circuit to apply those as gate voltages.

Note that "control" circuit plays 2 roles here, one of alloting input bits to particular cells for maintaining homogenous usage throughout the string, and the second role of application of Gate voltages, in my case, the key being stored as 177 bit key in a particular string (say $\mathbf{S_K}$), and being read by the "control" circuit and applied as gate voltage to every enabled string (assuming every string has the same cell usage sequences).

To remove the above assumption, here's the second idea:

The control circuit would be using a automaton to determine the next cell to be used in a particular string for storing the next bit (in our case, embedding the next input). Use the secret key (stored in $\mathbf{S_K}$) as the initial state of this state machine. If inserted with an incorrect key, all the following transitions would yield incorrect mapping of the cell having V_{READ} applied to, and the actual input-encoded cell, in all subsequent input changes! This covers up the case of having a persistent secret key for all inputs throughout usage of the Flash strings.

Note that this key string can be initialised as a PUF value of the Flash Memory itself, thus uniquely creating a key for every memory created. This would allow the key string to act as a watermark as well.

Attempt to remove allotment mechanism

Note, regarding the "transferring" mechanism, which is different than allotment mechanism, that deals with transferring correct bit values to chosen cells, this is simpler than it would seem.

If there exists another mechanism that can invert a bit stored, in place, easily, then our computation technique is well applicable to that string itself! You just need to make sure all required bits are present, and make them in correct format via the inverting mechanism. Your function is ready rn, as other cells are redundant, even if they keep on storing actual storage-meant bits, we're still applying V_{PASS} on those cells. What remains to be solved is, because now we are trying to remove the transferring mechanism out of the picture, input cells are no longer controlled by allotment mechanism, but according to a pattern of storage which is decided by the data itself. So the secret key would be fixed according to the storage type.

Piecing all together

I discussed about a "control" circuit, and 3 possible mechanisms namely, "allotment", "transferring", and "inverting". They are for input cells allotment, data transfer between actual storage and input cells, and inverting stored bit in-place. No more implicit assumptions had been made.