Week 5

Logic-in-Memory

References:

Logic-in-memory exploiting 3D-NAND Flash Array[1], SAT Attack-Resistant Camouflaged Two-Dimensional Heterostructure Devices[2], Oracle-Guided Incremental SAT Solving to Reverse Engineer Camouflaged Logic Circuits[3]

Purpose of the week:

To understand how the Flash Memory works, why is it owning up the market, and how to use it as a memory unit as well as a logic unit.

NAND Flash

NAND Flash memory is made up of memory cells strings, which are arranged in a grid-like structure on the memory chip. Each cell consists of a transistor and a storage node, which is a floating gate that can hold an electrical charge. The transistor acts as a switch that controls the flow of current to and from the storage node.

To write data to a NAND Flash cell, a voltage is applied to the control gate of the transistor, which opens the switch and allows a charge to be transferred to or from the storage node. The charge on the storage node represents the data that is being stored. Because the charge can be either positive or negative, each cell can store multiple bits of data. Reading data from a NAND Flash cell involves applying a voltage to the control gate and measuring the current that flows through the transistor. The amount of current that flows is determined by the charge on the storage node, which can be used to determine the data that is stored in the cell. NAND Flash memory is organized into pages and blocks. A page is the smallest unit of data that can be read or written, and typically consists of multiple cells. A block is a group of pages, and is the smallest unit of memory that can be erased. To erase a block of NAND Flash memory, the charge on the storage nodes in all of the cells in the block is set to the same level, effectively erasing all of the data in the block at once.

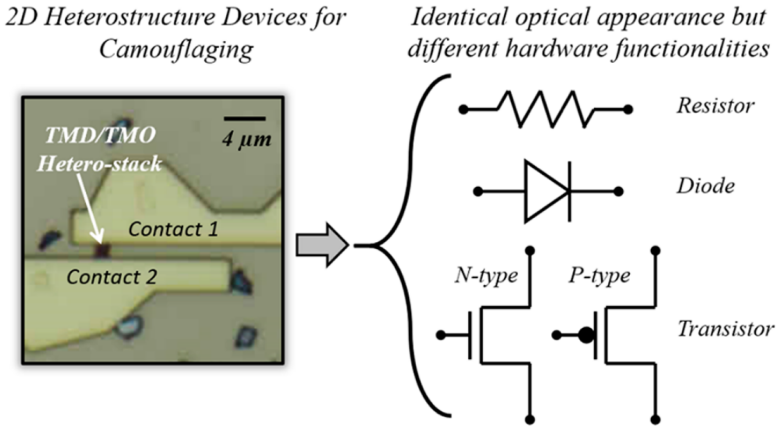
Because NAND Flash memory has a limited number of erase cycles, wear-leveling algorithms are used to evenly distribute the erase cycles across all of the blocks in the memory. This helps to extend the lifespan of the memory and prevent any individual blocks from wearing out too quickly.

2-D Heterostructures Devices

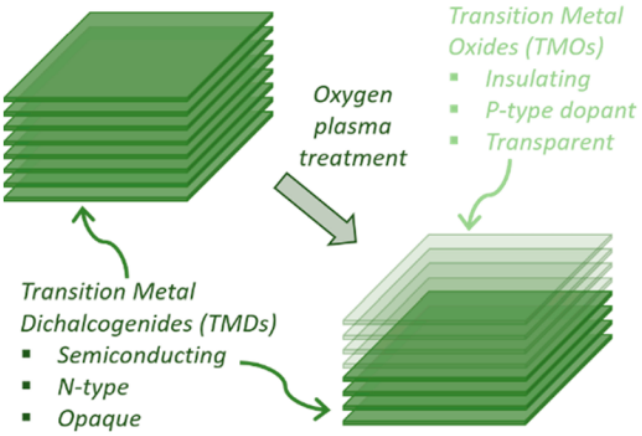
The camoflaging technique discussed by the authors describes how they made individual device components optically indistinguisable from each other. Existing IC camoflaging involved transformable interconnects, covert gates where varying doping and dummy contacts chnage the apparent circuit structure. Another approach involves building cells that look alike but have different functionalities. All these lead to large area overhead and are defeated by SAT-based RE.

Week 5

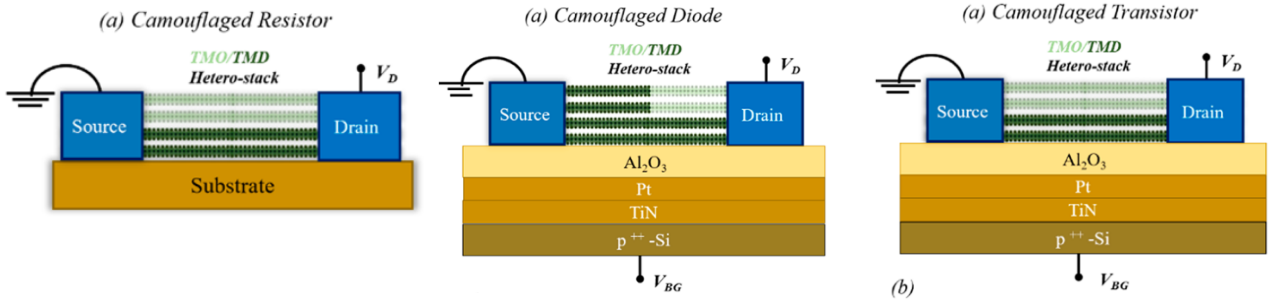
The authors now depict this picture:



This shows how a single structure can act as one of the basic 4 components of an electronic circuit. This is done by using stacked Transition Metal Dichalcogenides (TMDs) and using Oxygen plasma treatment to convert some layers of that to Transition Metal Oxides (TMOs) in a controlled manner.



Because of their different properties, and TMOs being transparent, they together can behave as the above devices if generated in manners depicted below. The optical behavior of all of them were shown to be similar, while the duration of oxygen plasma applied (which led to how deep TMOs were formed) changed their electrical properties drastically.



This was shown by pre-plasma Resistor formation having it's resistance changed by 8 orders of magnitude by 75s of exposure, diode having categories of thin (late diode) and thick (early diode) possibilities, and changing characterstics with changing V_{BG}, and FETs having it's type and characterstics changed by amount of exposure time.

All these leads to the possibility of using the same structures throughout the IC, making individual components indistinguisable. This is also represented by the picture below:

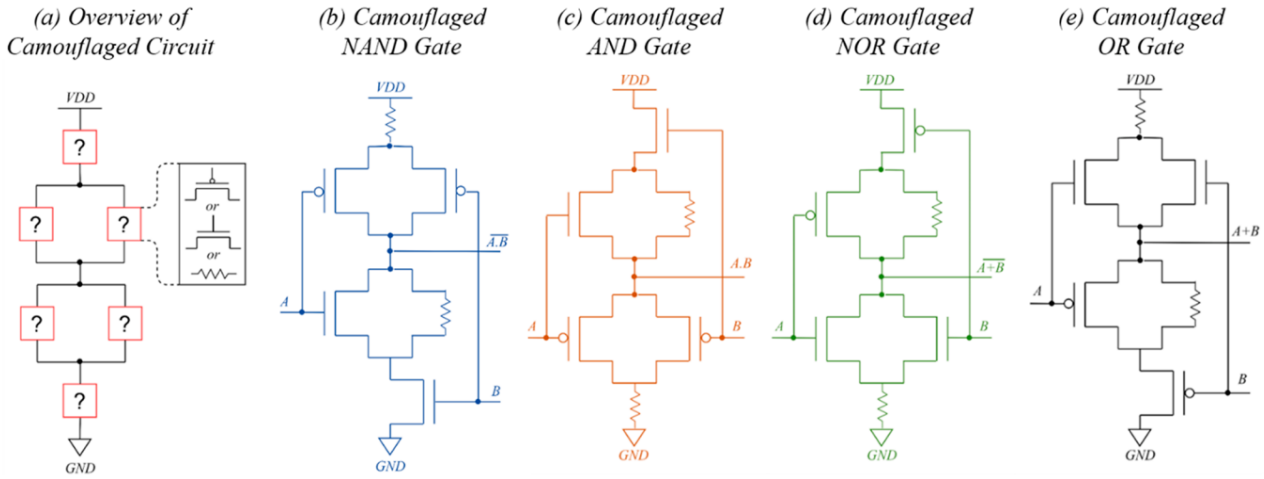


Figure 1: Camouflaged logic gate, that can function as NAND, AND, NOR, OR, all with the same optical properties.

Due to the same optical properties, the authors claim that the search space becomes astronomical. If there are *M* hardware components in the IC, and each with *P* possibilities, number of possible functionalities of the IC becomes $F = P^M$. So even for *M* = 1000 and *P* = 4, possibilies become

$$F = 4^{1000} = 1.148 \times 10^{602}$$

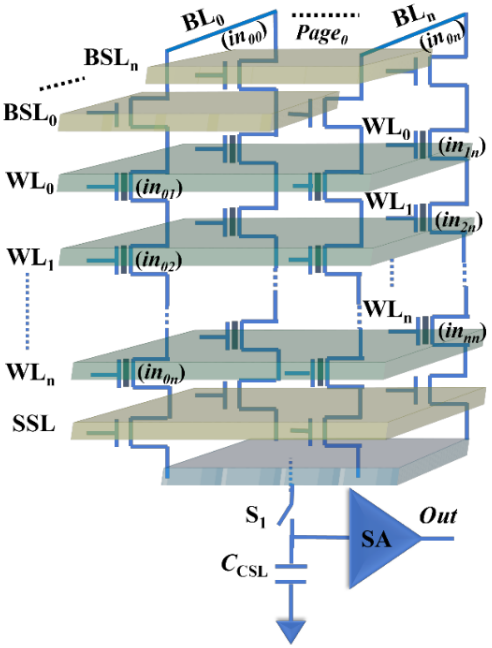
The authors benchmarked their procedure on **ISCAS'85** suite, comprising of 11 circuits with multi-input gates from multiple logical families. I think they didn't actually manufacture anything, but gave the SAT solver just the layout of the netlist like in Fig1 (a), and an oracle.

Logic-in-Memory Exploiting 3D-NAND Flash

The paper describes how to implement logic using 3D-NAND Flash blocks. A usual 3D-NAND Flash block comprises of strings of 177 cells, each cell being in a page of 4-16KB worth cells, i.e. $2^{12} - 2^{14}$ cells in a page.

The first approach

Use the fact that all logic can be refactored in SoP form. Due to this fact, the authors claim to be capable of realising logic functions with ≤ 177 literals and $\leq 2^{14}$ minterms parallely.



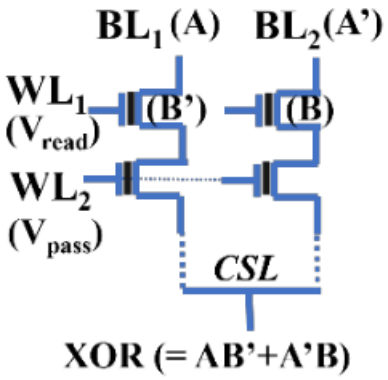
This is done by using BL as the first input value, flash cells' threshold voltages as the other literal values, and applying V_{READ} = 2.5V on those input encoded cells, while applying V_{PASS} = 8V on other redundant series cells (to decrease series resistance).

Let's consider an example.

Week 5

We want to implement $A \oplus B$ operation, given A and B values. Note the later part of the line, given the values.

Suppose $A = 0$, $B = 1$. So apply 0 at BL_1, apply 1 at BL_2, encode $B' = 0$ as $V_{TH} = 3.5V$ on cell marked as B', and encode $B = 1$ as $V_{TH} = 0V$ on cell marked as B. Apply $V_G = 2.5V$ on WL_1 as that encodes your inputs, and apply $V_G = 8V$ on other WLs. This will lead to CSL effectively accumulating the value of $A \oplus B$.



Problems

The problem arises when one cell is encoded with an input, and another cell is not, but both lie in the same page, so are activated by the same WL. This may lead to the second input having an unknown V_{TH}, having it's BL activating or not depending on that, corrupting the output (this concept can be used furthur).

Another problem arises that the whole inputs have to be structured according to the need as above, with the need to change threshold voltage of cells everytime inputs need to be changed (can be made easier by analysis of most frequent changing input being the BL input), but still changing V_{TH} leads to destruction of oxide layers of the cells, and considering the possible frequency with which data may need to be changed, i.e. 10^9 per second, life cycle of NAND flash of 10^5 cycles stand no chance. But again remembering that this isn't a processor, but a memory, easies things, because the NAND flash has been tasked on storing the values as well, so we won't expect such high frequency changes during storage. This only leads to the fact that we would need to arrange data in the manner as above, instead of in the manner received, and this might lead to redundancy in storage, with both parts of memory storing the same data just in different formats.
Also once a particular SoP is calculated, it would be much more efficient to just store it's value in a single cell instead of applying all V_G mentioned above to calculate the value again, making such calculation relevant only once, leading to the incentive of changing inputs again for a new calculation, getting us back to the point of life cycles. Considering the problem of life cycles aside for now, going back to

The second approach

Preprogram different logic gates with different input combinations in different strings across 3D-NAND flash array in form of logic gate planes, using a Content Addressable Memory, to search the strings encoding the logic function needed to be performed. This works as follows:

1. Inputs and logic function to be performed are encoded and given as search query vector to the CAM.

2. CAM stores the information regarding the location of the logic gate planes performing a particular logic operation on a set of inputs in the array.
3. The flash cells within different strings encoding a logic funciton are programmed in the array such that output obtained as voltage on C_{CSL}, when the input literals are applied on BLs and BSLs during reading.

Problems

Limitation of such an approach is the constraint of having only 2 literals per minterm in the logic function needed to be implemented. Example, if I want to implement $A \wedge B \wedge C$ using the second approach of preprogrammed strings, I don't have any place to encode the value of C if A takes up the BL and B takes up BSL of that string.

Defense strategies

Because of the above limitation of the second approach, I will try to explain how to lock the first logic-in-memory approach 3D-NAND Flash array.

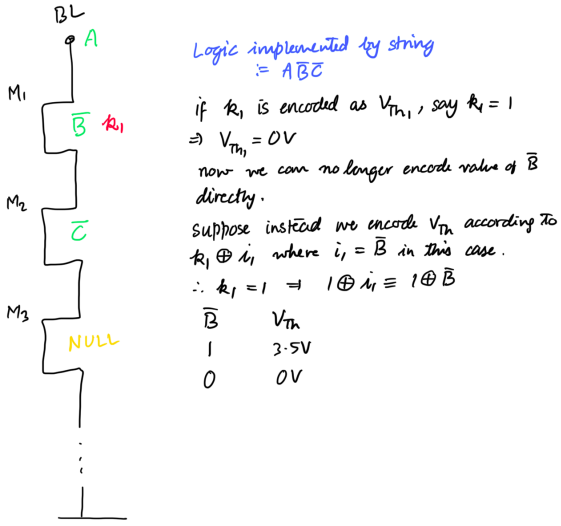
Consider the input space *I* and key space *K*. These spaces must be independent of each other, with the key remaining the same inspite of any input and vice versa.

As explained above, the inputs are divided into 2 types, one of the input is applied directly to the BitLine of it's string, and the other inputs are encoded as threshold voltage of flash cells, with 0 being V_{TH} = 3.5V and 1 being V_{TH} = 0V. The other cells are not involved because of High 8V being applied to them rendering V_{TH} value irrelevant. Suppose I want to calculate the minterm $A \wedge \bar{B} \wedge \bar{C}$ for values of A = 1, B = 0, C = 1. This can be done by applying V_{BL} = 0.5V representing bit 1 of A, applying V_{BSL} = 2V for selecting that string to the output, converting the first cell's V_{TH} = 0V representing bit 1 of $\bar{B}$, and converting the second cell's V_{TH} = 3.5V, representing bit 0 of $\bar{C}$. Note that you need to apply the actual values already calculated before, so need the availability of inputs and their complements beforehand. This entire setup would give the correct output of 0 for the given input values.

Suppose now we want to lock a 3D-NAND flash string.

Locking can be viewed as embedding a change in the circuit that doesn't change with inputs. The first approach can be to tamper with threshold voltages themselves.

But if the redundant cell's V_{TH} are changed, that won't cause any corruption in output. So we're left with changing V_{TH} of input cells. But that already encodes the input. This means we



need to somehow encode both a permanent key and a possibly changing input as the same value in the circuit.

One possibility is XORing, but that would lead to including XOR operation. But this can be solved by using the second approach which only provides 2 literals per minterm, but we need only 2 this time, a key and an input, so the keys would be applied to a set of BLs of an array, that would then unlock (apply correct threshold voltages during changing inputs) the logic.

EDIT: Thinking again, XORing always the key and input, would need the key to always be 0, so we need to have *some* function $F(k, i)$ that decides the V_{TH} of that input *i* cell instead.

Another possiblity is tampering with the the redundant cells. This would work by controlling the application of V_G on strings. If under control, V_G can either be 8V (key bit 0) or 2.5V (key bit 1). This approach would proliferate the key space to 2^{177} per string! Note that this works because the logic function to be applied on inputs itself is **fixed**. Therefore which cells should have inputs embedded and which cells should not is fixed. Assuming by some means, the inputs are encoded without the adversary knowing which cells were used, such type of locking becomes viable. This is also viable when only one input per minterm needs to change, so allot that to the BL, needing no change in the threshold voltages.

The third possibility includes using Multi-Level Controlled cells.

Because each cell now can store 2 bits (4 Vth ranges kept for simplicity), the cells can encode $\bar{B}$ and $\bar{C}$ in one cell itself. Or we can encode one input **and** a key in the same cell, with the key being a biased shift in V_{TH} inherent to the FET design.

Launching SAT Attack itself

This is described here with Swaroop acknowledging that he used the same for the above paper's benchmarking. It would involve encoding the netlist in VHDL format and running the solver on it directly. Another manner is to use Tseitin Transformation, using the above key pattern described included with the actual logic circuit, and feeding the whole CNF form in one of the leading SAT solvers described in previous report.

Piecing all together

Going through the paper uncovered some logistic issues with actual implementation, but also showed how the defenses in the 3D-NAND flash in-memory logic can be included which would generate huge key spaces. As the memory is already camouflaged, this would only enhance the security of this method. But the method's problems need to be clarified before looking into more details about possible defense, because if this isn't going to be used as an regular processing, how are we meant to think of it in terms of a general attack-defense circuit? One can instead opt for encrypting the data instead, encrypted data leading to correct but encrypted output.