[EE392] Undergraduate Project

Protection of Intellectual Property for NAND Flash Logic-in-Memory Architecture

---

Supervised by

Professor Shubham Sahay

Done by

B. Anshuman

EE-UG Y20 [200259]

Abstract

Hardware pervades us, from IoT devices to embedded systems, all controlled by ICs manufactured in concentrated foundries holding most of the designing companies as clients. This leads to asymmetry in resource sharing, foundries have access to the IP of designing clients, but designers don't have control over how the foundry (or later testing phases) takes care of their design. Copyright infrigement against softwares have received huge attention, compared to that hardware protection doesn't rank the same, which brings up the importance of the field of IP protection.

Every microcontroller used is accompanied by a flash storage units, and Sahay et al.[1] proposed multiple multi-variable logic-in-memory architecture, that utilises NAND flash cells to perform computation, dwelling away from Von Neumann architecture. It was proposed that the architecture is intrinsically resilient against known IP attacks, because of it's camouflaged nature.

This work tries to enhance the defense by moving from the domain of camouflaging to one of logic locking, that can be shown to be protective throughout the supply chain of IC manufacturing, which refers to NAND flash storage modules in this context. It includes a novel implementation of logic locking, that can be extended to memory cell protection.

Introduction

In today's digital era, flash memory has become ubiquitous, powering various devices such as smartphones, tablets, laptops, and data centers. The rapid growth of data-driven applications and the need for efficient storage solutions have made flash memory a crucial technology in modern computing. Alongside this, the concept of logic-in-memory (LIM) architecture, where computation is performed within the flash memory itself, has been pitched by Professor Shubham Sahay to overcome the memory bottleneck in traditional von Neumann architectures.

As with any other IP, protecting design becomes critical to invite attention. Unauthorized access, reverse engineering, and IP theft pose significant risks, potentially leading to financial losses, competitive disadvantages, and legal disputes. Therefore, implementing robust defense strategies to secure the IP for NAND flash LIM architecture is of paramount importance.

This work therefore aims to address the challenges of protecting the IP for NAND flash LIM architecture through comprehensive research, analysis, and development of defense strategies. By understanding the significance of hardware protection and implementing effective measures, we can ensure the integrity, and confidentiality of IP, thereby fostering innovation and advancing the field of flash memory-based computing.

Outline

We'll first dive into evolution of Logic Locking, followed by camouflaging and metering. This would be followed by discussing SAT solving and the SAT attack on logic locking. Then we'll dive into discussing the LIM architecture, and explain it's inherent camouflaging capability. This would follow a novel idea about implementing a logic locking mechanism that would require no hardware overhead. Then we would look at why Satisfiability attack can't even be performed as it is on this defense strategy. Finally we will look into simulation results of applying brute force technique to obtain the key, and analysing it's results.

Discussion

Evolution of Hardware Protection

Watermarking

A watermark must remain functionally correct, and prove the ownership of the IP.
For non-intrusive watermarking, it should be on the lines of[2]:

- Optimization problem that's difficult to solve, with it's solution space capable to handle a digital watermark
- Well-defined interpretation of solutions of the problem as an IP
- Existing algorithms that solve the problem without watermark involved
- Protection requirements include: forgining as hard as recreating, tampering is provable

During the creation of the IP in several stages, watermark each stage with a set of "constraints", then use pre- and post-processing of i/o to disproportionately satisfy large number of such constraints. These constraints aren't revealed.

Active Metering

Active metering as described here[3] is based on inherent unclonable invariability of ICs.

This is performed via Boosted Finite State Machine implementation, which are powered by unique IDs. Consists of states that leads to "power-up" (back to normal IC behavior) upon provision of some fixed inputs, and because only the designer knows the transition table, it's not possible with finite resources to crack that input with high dimensional input space.

The above implementation has the capability to lock each IC individually, and the lock designed during designing is realised inherently upon manufacturing. The authors have also proposed the option to remotely disable the IC. The authors emphasise on the use of:

i. uniqueness of ICs due to manufacturing variations
ii. structural manipulation of ICs can still lead to same behaviour

Logic Locking

Assumptions: No-one except the designer is trusted. Adversary has access to netlist AND a functional IC. The only unknown is the key, which is/can be represented as a bit-vector.

Cryptographic notion of a **lock**: an adversary that does not have infinite computational power should not be able to unlock the IC without the knowledge of a key.

A locked circuit has *key* inputs that are driven by on-chip tamper-proof memory. The additional logic may consist of XOR gates or LUTs. The IC needs to be activated by loading the secret key onto the tamper-free chip memory.

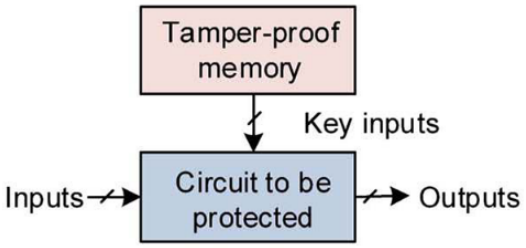

Figure 1: Diagram representation of a locked circuit

Let's define the literature now[4]:

The original netlist can be represented as a boolean function $F : \vec{X} \rightarrow \vec{Y}$ where $\vec{X} \in \{0,1\}^M$ and $\vec{Y} \in \{0,1\}^N$. The locked boolean function can be represented by some structural modification and mathematically modelled as $\mathcal{C} : \vec{X} \times \vec{K} \rightarrow \vec{Y}$ with $\vec{K} \in \{0,1\}^L$. Upon correct activation,

$$\mathcal{C}(i, k_s) = F(i)$$

Earlier, Logic Locking focused on **best locations for inserting key gates:** random, fault analysis-based, strong-interference-based.

SAT attacks emerged and changed the paradigm to anti-SAT logic locking (including SARLock, Anti-SAT, TTLock, SFLL), but they became vulnerable back to the previous attacks such as removal attacks.

The following are some widely known attacks and their combinations[5]:

1. Algorithmic: Exploit weakness of logic locking algorithm, brute forced, exact attacks. Includes sensitization attack, SAT attack, circuit partitioning attack
2. Approximate: Extract an approximately same netlist as original. Includes Double-DIP
3. Structural: Bypass protection logic. Includes signal probability skew attack, AppSAT guided removal attack, Bypass attack
4. Side-Channel: Exploit covert physical channels like power, timing, test-data. Includes differential power analysis attack, desynthesis attack.

An encrypted **combinational** circuit can be also be represented by a Conjunctive Normal form using Tseytin's transformation[6] or otherwise:

$$\mathcal{C}(\vec{X}, \vec{K}, \vec{Y}) \subseteq \{0,1\}^{M+L+N}$$

such that $\vec{X}$ represents *M* primary inputs to the circuit, $\vec{K}$ represents *L* length key, $\vec{Y}$ represents *N* primary outputs of the circuit. Also,

$$\mathcal{C}_O(\vec{X}, \vec{Y}) \subseteq \{0,1\}^{M+N}$$

represents the oracle, which is a decoded and correctly behaving circuit. Note that if $\vec{K}$ is the correct key provided to $\mathcal{C}$, then $\mathcal{C} \equiv \mathcal{C}_O$. Note that the attacker doesn't have the relation $\mathcal{C}_O$, but can apply input patterns and observe outputs of the same. This is modelled by $eval : \{0,1\}^M \rightarrow \{0,1\}^N$.

The attacker goal now becomes obtaining K_C such that

$$\forall \vec{X} : \mathcal{C}(\vec{X}, \vec{K_C}, \vec{Y}) \iff eval(\vec{X}) = \vec{Y}$$

This is equivalent to solving the quantified boolean formula (a boolean formula with the formula is constraint of course, but also constraints in input/variables)

$$\exists K \;\; \forall X \;\; \mathcal{C}(\vec{X}, \vec{K}, \vec{Y}) \wedge \mathcal{C}_O(\vec{X}, \vec{Y})$$

In the above, the attacker can't construct a formula for $\mathcal{C}_O$, and has to use *eval* for a small number of inputs possibly.

For a given set of inputs $\{\vec{X}_1, \vec{X}_2, \cdots, \vec{X}_p\}$ and outputs $\{\vec{Y}_1, \vec{Y}_2, \cdots, \vec{Y}_p\}$, the adversary can find out a $\vec{K}$ that satisfies $\wedge_{i=1}^{p} \mathcal{C}(\vec{X}_i, \vec{K}, \vec{Y}_i)$. But upon a new ($i$+1)$^{\text{th}}$ observation, there's no guarantee that the $\vec{K}$ found before would satisfy $\wedge_{i=1}^{p+1} \mathcal{C}(\vec{X}_i, \vec{K}, \vec{Y}_i)$. The insight lies in considering keys as a part of *equivalent classes*, instead of individually.

The author explains to look for a member of the *equivalence class* of keys which produce the correct output for all input patterns seen. For this, iteratively rule out *equivalence classes* which produce the wrong output values for at least one Differetiating Input Pattern (DIP).
Note that $\vec{X}_d$ behaves as a DIP iff

$$\mathcal{C}(\vec{X}^d, \vec{K}_1, \vec{Y}_1^d) \wedge \mathcal{C}(\vec{X}^d, \vec{K}_2, \vec{Y}_2^d) \wedge (\vec{Y}_1^d \neq \vec{Y}_2^d)$$

The second insight is if $\vec{X}_d$ is found, then the oracle can be used to reject either of $\vec{K}_1$ or $\vec{K}_2$ or both as not being of the equivalence class of the correct key.

The following was orignally from Subramanyan et al[7].

```
1. function SAT_Attack(Circuit Cl, Circuit eval)
2.       i <= 0;
3.       F[0] <= Cl(x, k1, y1) ^ Cl(x, k2, y1);
4.       while SAT(F[i] ^ (y1 != y2)) do
5.             xd[i] <= sat_assign_x (Fi ^ (y1 != y2));
6.             yd[i] <= eval(xd[i]);
7.             F[i+1] <= F[i] ^ Cl(xd[i], k1, yd[i]) ^ Cl(xd[i], k2, yd[i]);
8.             i <= i + 1;
9.       kc <= sat_assign_k1 (F[i]);
```

To apply this, we've seen how to generate CNF format of the given logic netlist. Once obtained, it can be directly fed to a SAT solver, iteratively as above. Complexity analysis of the same yields the hardness of the defense in context.

Logic-in-Memory Architecture

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.

Logic-in-Memory Exploiting 3D-NAND Flash

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.

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

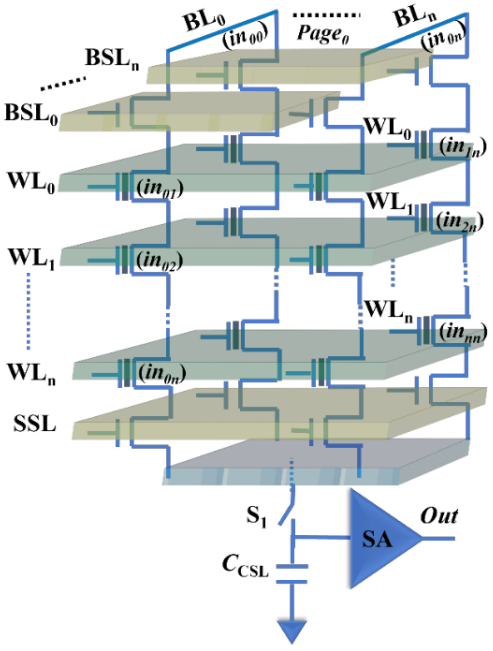

Figure 2: Architecture of NAND Flash, has a direct analogy to an SoP form

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 $A \oplus B$.

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.

Inherent problem in Logic-in-Memory

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.

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.

This has been countered by the fact that this is not a processor, it's still a memory, used for maybe secondary method to perform computation if a protocol is created that can encode inputs as this needs. Also availability of huge number of blocks per module is in favour of using this LIM approach.

We'll see that this breakdown **problem** is **helpful** for the defense strategy discussed later.

Protecting LIM (WFSM Strategy):

As noted above, the entire implementation of Logic-in-Memory is **agnostic of the actual logic being implemented**. Once the user decides what logic is needed, he 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.

> Use WordLine voltages to corrupt the output.

Note that the function being implemented is a seperate procedure than obtaining it's output. Therefore once the user encodes her inputs according to her required function, he would expect the LIM structure to still be applicable. Also 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 (or otherwise[8]), and the memory controller 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 memory controller decides this deterministically on the basis of the designer who created it (and it's initial state k_s that becomes the correct key), according to a deterministic **allotment FSM**, we would know which cells in sequence would be used for the input in hand. Have another **wordline FSM**, that would have the **exactly same transition table** as allotment FSM. Have the initial state of wordline FSM be decided by the user acting as the input key. If initial states of both of them are the same, we would have both FSMs' transitions following each other, giving correct read voltage on pages that are being used currently. Otherwise "non-significant" cells would have read voltage applied to them, that can non-deterministically pass the signal or not, according to their state.

The adversary needs to feed the initial state of the "control" circuit FSM that decides the subsequent cells to be used.

Reiterating, there are again two 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 wordline 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: $\mathbb{F}(A, B, C, D, E, F) = DAB + EBC + FCA$, he would be essentially requesting for 3 cells in 3 different strings.

> Difference lies in allotment State having initialization from the designer, wordline state having init from user.

The allotment FSM already exists somewhat in microcontrollers, in the form of wear levelling algorithms. But wear levellers currently consider block level wearing, and we would need the same for page level instead. Similarly wordline FSM is required to be implemented that follows every input handling and activates wordlines sequetially.

Why SAT Attack is inapplicable?

Note the following block diagram of the entire programming of the block to infering it's output correctness.

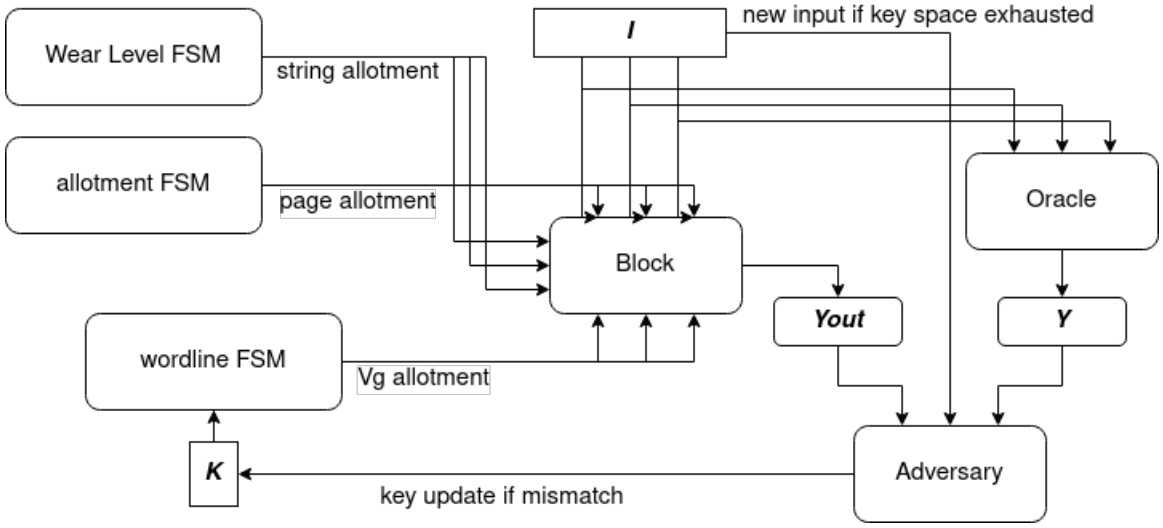

Figure 3: Block Diagram showing data path of changing keys K for every function input I

A block would require for every input cell I, a (page, string) pair to place that bit. This is yielded by the allotment FSM, and wear level FSM (already present in memory controllers, no actual need for defense perspective). Then output is provided upon Vg allotment from wordline FSM.

If outputs of oracle and the block mismatch, Adversary moves on to his next best guess of key and repeats the process.

If key space is exhausted, there is no guarantee to have found the correct key, just because it has worked so far, because theoritically the key space can be $\mathbb{Z}$, limited by the key length, which again can be theoritically of any size (note that this input corresponds to a state, and not electric lines to

any gate, so no overhead is incurred). Probabilistically and as we see from the experiments it's likely that upon finding the correct approximate range of key space, one can pin-point the correct key by using at least two different functions. Later on in the experiments section we'll see that even finding the correct key in such a manner would render the flash module unusable, just because of the problem of low life span of this architecture.

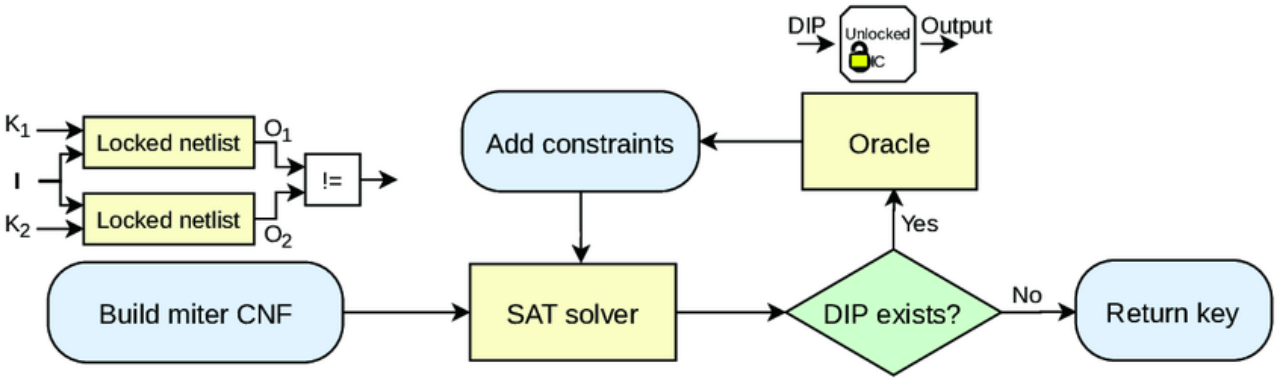

Figure 4: SAT Attack Block Diagram[1], [5]

Looking at how SAT solver takes in input, it assumes, as stated above, a combinational circuit's CNF format. But this circuit above in all includes 3 sequential components. Therefore it's not straightforward in converting a sequential circuit to its CNF format, having SAT attack as it is, inapplicable. We can do that by treating each state of the circuit independent of the previous state, and the current state to just be a combinational circuit, but this remains a prospective future work.

Method

Experimentation

It's shown that SAT Attack on the defense on LIM architecture doesn't seem feasible as it is. Therefore to attain a benchmark, I have simulated the architecture[9] in Python as below:

> Architecture is structured the same as proposed, except for one assumption. Suppose we have a functional input in form of Sum of Products as:
>
> $$A\bar{B}C + D\bar{C}$$
> with $A = 0, B = 1, C = 1, D = 0$.
>
> This is the computation we want to calculate, which can be equivalently written in functional agnostic manner as $0 \cdot 0 \cdot 1 + 1 \cdot 1$.
> Note that each product term enters a particular NAND string, with each literal going into different strings. Each term would correspond to a cell on that string, which in turn corresponds to a page. But we have the constraint of each page having a single Vg input, also we need to have "non-significant" cells just pass the signal, aka. be in ON state.
> Therefore I propose to consider inputing the function with the largest number of cells used as the first string, which would decide which pages are going to be used according to the allotment FSM output. Now insert input values for other strings, and for the non-needed cells of required pages, put them in ON state to pass the signal.
>
> I have structured the entities in context as classes, and with the highest class as follows:

```
class FlashModule:
    def __init__(self, key, correct_key, num_strings) -> None:
        self.block = Block(0, num_strings, correct_key)
        self.keyHandler = KeyApply(key)
        self.oracle = Oracle()
        self.function = Function(self.block)

    def write(self, func: list):
        func.sort(key=lambda x: x[0] + x[1], reverse=True)
        self.block.disable_strings()
        pages_used = self.function.write_to_block(func)
        Y = self.oracle.true_value(func)
        Vg = self.keyHandler.apply(func)
        Yhat = self.block.output(Vg)
        return Y, Yhat

    def reinit(self, key):
        self.block.reset()
        self.keyHandler.reinit(key)
```

> Here this provides an API that uses `write` to enter a new function in the flash module, with each step explained below:

`func.sort(key=lambda x: x[0] + x[1], reverse=True)`
This sorts the function as proposed above.

`self.block.disable_strings()`
`This disables all enable lines of that block`

`pages_used = self.function.write_to_block(func)`
For writing that function to the block. Each change in state of any cell is kept tracked of, and allotment of strings is done according to static wear algorithm of lowest-used-first. This returns which pages are used in that block.

`Y = self.oracle.true_value(func)`
Calculates the true value of the function

`Vg = self.keyHandler.apply(func)`
This uses the key initiated by the enduser, to calculate which the Vg vector, aka which pages should V_{READ} be applied to.

`Yhat = self.block.output(Vg)`
Computes output of LIM architecture according to applied Vg and initially alloted inputs, both uses the same FSM, but with possibly different initial states.

Note that in this experiment, we want to show the following:

1. For every input function, only the correct key initialisation will **always** give the correct output as the oracle
2. Any key initialisation that's incorrect should give **unpredictable** results, i.e. it may match with the oracle or won't, as there's only 2 outputs possible for this architecture as of now.
3. We want incorrect key values to give reasonable winning streaks (as defined later). Too low winning streak would lead to less computational need and lower time complexity, leading to quicker search through the key space. Too high winning streaks would lead to every key being able to act approximately as the correct key value. Therefore implementation of such an FSM is desirable that would keep a balance as required by the specification of security needed.
4. Trying to brute-force the key will lead to usage of the flash block, leading to ultimate unusability of that module. Note that each correct_key (the initial state of allotment FSM) made by the designer can be unique to that particular Flash Module! This leads to watermarking capability, as well as possiblities of black hole states as described here[3]
5. Therefore upon exhaustion of 10,000 cycles of flips, a cell is no longer trusted to store bits properly, and if the total experiment exceeds the same, we would obtain the key but at the cost of losing the Flash's usability itself!

Experiment Results

For the initial setup of the following:

The FSM unit is defined as a pseudo random number generator, I took up the most basic one, aka Linear Congruential Generator[10].

The "key" for either of the FSM becomes the seed of the generator. Same seed would generator identical sequences, while incorrect ones won't. Note that the enduser can NOT observe the generator's output. He can only observe whether the output calculated by the flash module is the same as the oracle's.

```
correct_key = 400
cell_per_string = 177
num_strings = 10
functions = [[(3, 2), (0, 1)],
             [(2, 4), (5, 6)]]
```

```
● → Simulation python test.py -correct_key 400
  For function 0:
  [81, 3, 0, 8, 0, 3, 0, 27, 1, 7, 19, 8, 8, 5, 0, 6, 1, 3, 2, 0, 1, 4, 5, 2, 3, 13, 0, 0, 0, 0, 0, 8, 1, 7, 6, 17, 3, 4, 19, 1, 1, 12, 0, 1, 8, 0, 13,
   8, 0, 5, 5, 18, 0, 5, 0, 0, 9, 0, 3, 0, 4, 4, 1, 3, 3, 1, 14, 1, 1, 0, 6, 0, 8, 2, 3, 6, 30, 0, 1, 0, 2, 0, 2, 0, 7, 8, 8, 1, 4, 5, 0, 12, 0, 5, 0,
  0, 1, 10, 1, 0, 178, 2, 2, 0, 1, 0, 0, 1, 2, 2, 0, 1, 0, 1, 0, 4, 5, 2, 1, 0, 0, 4, 0, 12, 6, 3, 2, 0, 0, 0, 0, 0, 1, 6, 0, 1, 0, 0, 0, 3, 1, 2, 1, 0
  , 0, 0, 0, 0, 0, 0, 3, 0, 0, 1, 2, 1, 0, 0, 5, 4, 0, 1, 3, 1, 0, 1, 0, 2, 0, 4, 1, 0, 2, 0, 5, 0, 0, 0, 2, 9, 2, 2, 4, 0, 0, 3, 4, 1, 0, 1, 0, 0, 2,
  0, 0, 0, 4, 4, 6, 2]
  For function 1:
  [3, 1, 178, 59, 1, 37, 122, 65, 102, 75, 112, 178, 106, 61, 66, 84, 46, 178, 71, 78, 0, 178, 29, 178, 5, 178, 95, 100, 75, 32, 69, 40, 47, 83, 74, 28
  , 178, 178, 8, 19, 8, 11, 0, 53, 58, 63, 84, 41, 62, 84, 8, 178, 178, 40, 29, 18, 7, 78, 81, 178, 75, 109, 53, 58, 15, 36, 74, 32, 54, 10, 30, 19, 8,
   63, 116, 55, 28, 81, 24, 13, 75, 53, 178, 178, 55, 178, 33, 22, 9, 64, 35, 153, 77, 68, 120, 111, 67, 24, 56, 0, 178, 45, 178, 23, 10, 127, 70, 154,
   62, 37, 58, 15, 4, 39, 77, 35, 178, 46, 35, 22, 11, 0, 53, 26, 15, 20, 178, 102, 35, 137, 78, 178, 58, 63, 34, 23, 12, 1, 167, 59, 64, 55, 12, 1, 38
  , 59, 32, 5, 10, 15, 178, 24, 77, 66, 7, 12, 178, 135, 124, 64, 53, 58, 15, 4, 25, 49, 70, 3, 48, 149, 120, 178, 178, 87, 14, 83, 105, 83, 0, 178, 28
  , 66, 6, 4, 15, 6, 170, 126, 150, 137, 128, 84, 25, 178, 98, 178, 178, 178, 178, 29]
  Usage count per string: [5440. 5440. 5441. 5439. 5441. 5441. 5440. 5440. 5438. 5438.]
  Each flip requires 3ms to complete, therefore total time taken: 163194.0 ms
```

Figure 5: Simulation terminal output for 2 different functions, showing flip count per string for 10 strings per block.

For simulation's sake, I kept correct_key as a small value. Note that any bit sequence can be transformed to such a number with any transform $\mathcal{F}_K : \{0,1\}^L \to \mathbb{Z}$.

The experiment has the following attack algorithm:

```
for function in functions:
      key_streak = List()
      for key in key_space:
            flash_module.insert_key(key)
            winning_streak <= 0
            while winning_streak < WINNING_THRESHOLD:
                  Y, Yhat <= flash_module.write(function)
                  if Y != Yhat:
                        break
                  winning_streak += 1
            key_streak.append(winning_streak)
```

Therefore finally we have winning_streak of each key for every function, which can be plotted as:

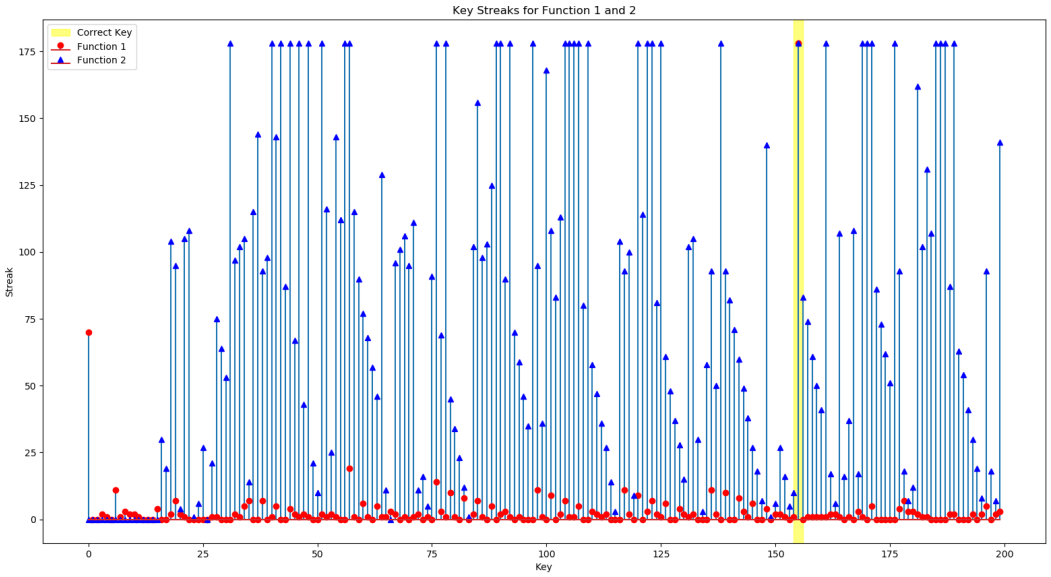

Figure 6: Case: correct_key = 155, key_space is {0, 200}. Yellow highlight is for the correct key.

Note that for only the correct key, both functions reached a winning streak will winning_threshold.

Other key values that were able to reach the same threshold were not able to do so for the other function.

Figure 7: Case: correct_key = 400, key_space = {300, 499}. Tried for 3 different functions

Note that this apparently localises the correct key if the module is tried for two different functions. That's the defense's strength as well as it's apparent weakness.

This can be countered by the following:

1. Better FSM initialisation OR FSM implementation would lead to higher streak values for even incorrect keys, leading to high complexity even in a single function's search.
2. Some functions lead to outputs biased to 1, others lead to outputs biased towards 0. This leads to partition of the functional space, leading to the first category reaching high win streaks for incorrect values (blue plot), and second category having extremely low win streaks for the same (red plot).
3. Keeping a reasonable winning_threshold is necessary, and can be arbitrary
4. The key space can be anything, $\mathcal{F}_{\mathcal{K}}$ being unknown to the user.

Possible Problems faced

FSM implementation in the Memory controller of Flash Memory is an unknown area. As the experiment concluded, there exists some functions that have keep winning streaks too low for incorrect keys. Having more complex FSMs would require better processing power.

Such a linear congruence generator is possible to perform. Flash Memory controllers are microcontrollers like the *89C51Rx2*, an 8bit 40MHz processor or *Atmel AT89C51RD2*, which are capable of handling such a defense strategy. Note that the overhead lies just in FSM calculation, connecting Vg decision to the FSM output. There is no hardware overhead. But this would incur computation to be required by the memory controller, making task scheduling necessary, needing possibly more complicated microprocessors.

Conclusion

This project concludes with the implementation of WFSM defense strategy for the LIM architecture and showing it's strength against bruteforce attacks, and incompatibility with explored and well known logic locking attacks including sensitisation attacks, side channel attacks, and the SAT attack. We've seen that this defense strength lies on an FSM implementation on-chip of the memory controller, which needs some features stated before. We've also seen that even if the FSM implementation itself is simple, key space can be increased as well, by including larger number of states and transitions. Future works can lie in determining which automaton implementations would lead to a sweet spot between complexity and key space size. Also because this is function agnostic architecture, we can reduce the concept of accessing bits (aka using flash as memory) to such a function in CNF form, making this defense applicable to memory modules as well, but right now it is still unexplored.

Acknowledgements

I received help from Bhogi Satya Swaroop in understanding his architecture, and going through the line of thoughts about things can work as suggested, and are the assumptions correct. He also helped me at every step of providing relevant papers and books to go through, and clearing any lingering doubts.

I have been advised at every turning point by my UGP advisor, Professor Shubham Sahay, who had to begin with suggested me to work on this project and accepted being a supervisor of the same.

I finally want to thank Aditya Jain for having a look at this architecture, and providing his criticism keeping me on track with assumptions made and limits to what is available at such low level computing without hampering performance.

References

[1]…S. Sahay, B. S. Swaroop, and A. Saxena, “Satisfiability Attack-resilient Camouflaged Multiple Multivariable Logic-in-Memory Exploiting 3D NAND Flash Array.” TechRxiv, Nov. 11, 2022. doi: 10.36227/techrxiv.21532455.v1.
[2].....A. B. Kahng *et al.*, “Watermarking techniques for intellectual property protection,” in *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, Jun. 1998, pp. 776–781. doi: 10.1145/277044.277240.
[3].....Y. M. Alkabani and F. Koushanfar, “Active Hardware Metering for Intellectual Property Protection and Security”.
[4].....M. Yasin and O. Sinanoglu, “Evolution of Logic Locking,” Oct. 2017. doi: 10.1109/VLSI-SoC.2017.8203496.
[5].....H. M. Kamali, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, “Advances in Logic Locking: Past, Present, and Prospects”.
[6].....“Tseytin transformation - Wikipedia.” https://en.wikipedia.org/wiki/Tseytin_transformation (accessed Feb. 13, 2023).
[7]…D. Sirone and P. Subramanyan, “Functional Analysis Attacks on Logic Locking,” *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 2514–2527, 2020, doi: 10.1109/TIFS.2020.2968183.
[8].....“Wear leveling,” *Wikipedia*. Feb. 24, 2023. Accessed: Mar. 11, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Wear_leveling&oldid=1141408307
[9].....B.Anshuman, “Simulation of EE392.” Mar. 22, 2023. Accessed: Apr. 16, 2023. [Online]. Available: https://github.com/ba-13/Courses/EE392/Simulation
[10]. .“Linear congruential generator,” *Wikipedia*. Feb. 19, 2023. Accessed: Apr. 16, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linear_congruential_generator&oldid=1140372972