BENCHMARKING SAT SOLVERS IN HARDWARE SECURITY AND THEIR GPU ACCELERATION

by

Harshith Kumar Thirumala
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

_____ Dr. Avesta Sasan, Thesis Director

_____ Dr. Houman Homayoun, Committee Member

_____ Dr. Setareh Rafatirad, Committee Member

_____ Dr. Monson Hayes, Department Chair

_____ Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Summer Semester 2018
George Mason University
Fairfax, VA

Benchmarking SAT Solvers in Hardware Security and Their GPU Acceleration

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

by

Harshith Kumar Thirumala
Bachelor of Engineering
Jawaharlal Nehru Technological University, Hyderabad – 2016.

Director: Avesta Sasan, Associate Professor
Electrical and Computer Engineering

Summer Semester 2018
George Mason University
Fairfax, VA

Copyright 2018: Harshith Kumar Thirumala
All Rights Reserved

DEDICATION

This thesis is dedicated to my uncle, late Thirumala Swami Krishna, and his family.
Thank you, *Pedha daddy* for all the memories, we miss you.

ACKNOWLEDGEMENTS

I would like to express my very great appreciation to my advisor Dr. Avesta Sasan for his patience, motivation and guidance through the research and thesis documentation. I would also like to take this opportunity to thank Shervin Roshanisefat and Hadi Mardani Kamili for their immense help.

Finally, I would like to thank my parents, grandparents and friends for their moral support and encouragement without which the thesis would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

Satisfiable ..... SAT
Un-Satisfiable ..... UNSAT
Set of Completion Keys ..... SCK
Distinguishing Input Pattern ..... DIP/DI
Key-Programmable Gate ..... KPG
Key-Programmable Circuit ..... KPC
Functional Circuit ..... FC
DI Validation Circuit ..... DIVC
SCK Validation Circuit ..... SCKVC
Learnt-Clause Avoidance Circuit ..... LCAC
Boolean Constraint Propagation ..... BCP
Unit Implication Rule ..... UIR

ABSTRACT

BENCHMARKING SAT SOLVERS IN HARDWARE SECURITY AND THEIR GPU ACCELERATION

Harshith Kumar Thirumala, M.S.

George Mason University, 2018

Thesis Director: Dr. Avesta Sasan

The recent trend in the Integrated Circuit (IC) supply chain process involves, in-house development and out-sourcing the designs to major fabrication labs located globally. This process introduces several concerns and threats like Intellectual Property (IP) piracy, reverse engineering, IC counterfeiting and introduction of hardware Trojans. An awareness of these attacks and threats have led to the introduction of several hardware-level security techniques. This thesis comprises a review of the Hardware-level security techniques on Logic Locking, their strength against the Boolean Satisfiability solvers and an attempt to parallelize the SAT-Solver on a GPU, which is at the core of the de-obfuscation tool used for the security analysis.

We investigate the strength of six different SAT solvers in attacking various obfuscation schemes. Our investigation revealed that Glucose and Lingeling SAT solvers are generally suited for attacking small-to-midsize obfuscated circuits, while

MapleGlucose, if the system is not memory bound, is best suited for attacking mid-to-difficult obfuscation methods.

The GPU version of the SAT-Solver was implemented using the NVIDIA's CUDA toolkit, as to take advantage of the hundreds/thousands of CUDA cores present on the GPUs. However, the results of GPU parallelization are not as promising as one would think in the general sense, and the drawbacks of the approach are also reported clearly. The results of the GPU based SAT Solver are derived using the benchmarks available in the SATLIB website.

CHAPTER 1: INTRODUCTION

1.1 IC Supply chain model:

In the pursuit of designing energy-efficient, high-performance and low power dissipating electronics that has an insignificant footprint, the semiconductor businesses are rapidly shrinking the size of the semiconductor device, this phenomenon is known as “Technology Scaling”, which leads the device integration density to double every two years, in accordance with “Moore’s Law”. Therefore, this has led to the dramatic increase in the cost of IC fabrication process, which forced the semiconductor business to a contract foundry model. The traditional supply chain model of designing, fabricating, testing and package the ICs has been stressed to its limit by the rapidly changing technology node.

Therefore, the leading-edge design houses outsource the fabrication and packaging to offshore facilities, where the operational costs, resources and employee wages are less. For example, AMD contracts some of its IC production to the foundries throughout the world, and Texas Instruments chose not to develop sub-45-nm fabrication in-house partnering with major foundries worldwide to outsource manufacturing, other companies include, Nvidia, Qualcomm, Broadcom etc. to name a few. This idea is a boon for reducing costs at the design houses but has many threats and concerns regarding the

security of the design. Figure 1 shows the IC supply chain and the location of vulnerabilities at different levels.

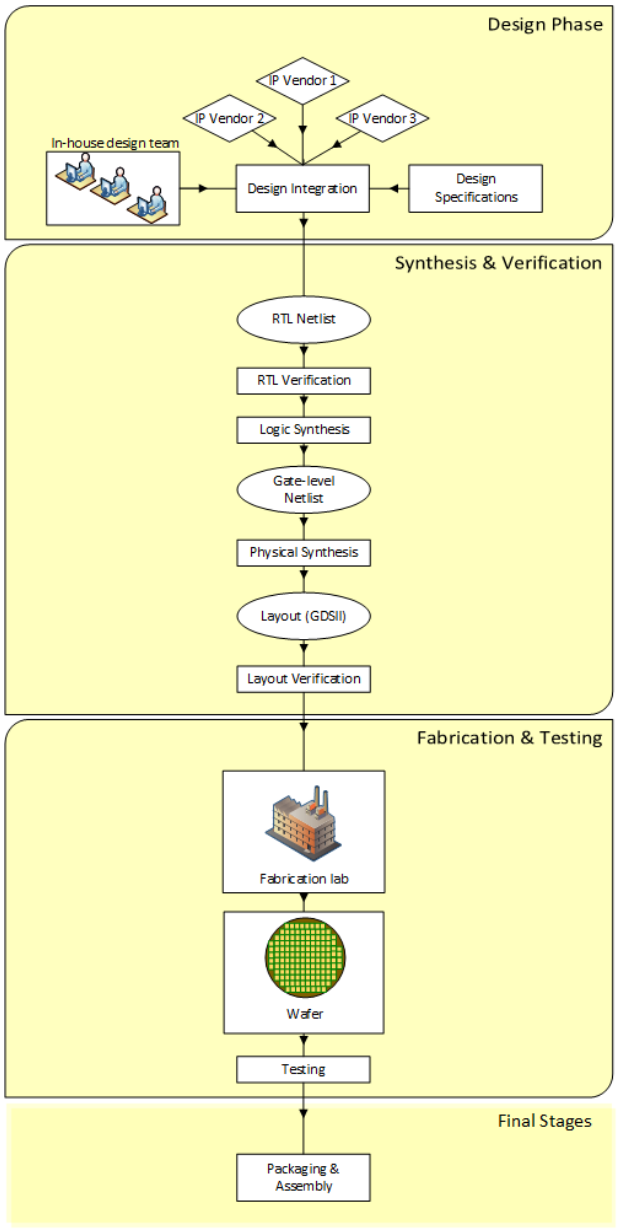


Figure 1: Vulnerable locations in IC supply chain.

The threats and concerns that arises due to the above process change include, Reverse engineering, counterfeit ICs, Hardware Trojan insertion, side channel attacks, fault attacks. A detailed description of each of the attacks is explained later in this chapter.

The effects of these attacks have an adverse effect on the consumers using these faulty products and the semiconductor businesses. The effects on the consumers would be: Consumers purchase products relying on the vendor's reputation, who in-turn rely on the Original Equipment Manufacturers (OEMs) and Contract Manufacturers (CMs) for the production of the final product, if the OEMs or the CMs are a victim to these hardware threats, their products would be termed unreliable and cause harm to the Health, Safety and, Security depending on the nature of the end product purchased.

On the other hand, the effects on the semiconductor businesses include economic harm to the businesses that are victim to these attacks. These companies spend tens of billions of US dollars per ear developing, manufacturing, and supporting the products that will operate reliably for many years in the consumers applications. In contrast, many attackers implement their plan of attack according to the weakness of the OEM or OCM products and take advantage of them. For example, if an IC is counterfeited, the attackers spend minimal amount of money developing and "manufacturing" the products and they

provide no post-sales customer support. Therefore, the low quality and poor reliability of the products, damage the reputation of the OEMs/OCMs, especially if the users do not realize that these products are counterfeited.

Therefore, to curb the above-mentioned attacks, a number of hardware design-for-trust (DFTr) techniques such as Logic Locking, IC metering, IC camouflaging, watermarking etc. have been proposed. Logic locking or Logic encryption, in particular, has significant interest in the research community because of its versatility and it can protect against attacks located anywhere on the IC supply chain. Other techniques like camouflaging, split manufacturing also provide ample protection but only for a limited set of malicious attacks, such as trying to Reverse Engineer the logic by washing off the top layer of the IC to reveal the internal logic.

1.2 Threats and Attacks on the IC supply chain:

Each of the attacks and threats have their own attack goals and are aimed at different locations of the IC supply chain.

1.2.1 Hardware Trojans:

An attacker may place a malicious hardware on the original circuitry or modify the existing circuitry. This aids the attacker to control, modify or monitor the

contents of the underlying computing device. The attacker can be located in the design house or maybe in a malicious foundry.

- Goal: Leak sensitive information, deny service to the user.
- Defenses: Design- Obfuscation, IC metering, Split manufacturing.

1.2.2 IP Piracy and IC overbuilding:

An attacker, in most cases a rogue foundry, may illegally pirate or copy the design without the consent of the designer. He can either sell the stolen designs to other designers or over build the same design and sell them to the grey market.

- Goal: To pirate a design, identify the trade secrets.
- Defenses: IP Watermarking, Fingerprinting, Obfuscation, Metering, Split manufacturing.

1.2.3 Reverse Engineering:

An attacker can reverse engineer the IC/IP design to identify the internal gate-level netlist and/or inferring its functionality. He can then re-implement the same design or improve the design for his advantage. Reverse engineering could be done either in the supply-chain or after the product is in the marketed.

The steps involved in the process of Reverse Engineering are hierarchical destruction of the physical IC, by de-lidding the IC using chemicals, annotating the images of IC internals and finally extracting the gate-level netlist.

- Goal: Steal design, modify functionality, leak sensitive information, Reduce reliability.
- Defenses: Design obfuscation, IC metering, Split manufacturing, IC camouflaging.

1.2.4 Side-Channel Attacks:

These attacks exploit the leakage of information from the physical IC during normal operation of the IC. The information includes power consumption, timing, or electromagnetic emission from the IC. These types of attacks were successful in breaking most of the existing important cryptographic algorithms. For example, the secret key of the RSA algorithm could be extracted by observing the time taken to compute the "square and multiply" step of the algorithm [3]. This type of attack is called the Timing side-channel attack.

- Goal: Leak sensitive information, compromise security features.
- Defenses: Regular changing the secret key, reducing the leakage information, deliberate noise injection.

1.2.5 Counterfeiting:

A counterfeit semiconductor component is an illegal forgery or imitation of the original component. Counterfeited ICs are different from pirated/overbuilt ICs as the latter involves in overbuilding the ICs without the consent from the designers, while the former is produced by trying to imitate the functionality and appearance

of the original ICs in the market. Due to the advances in the 3-D packaging technology, it is difficult to differentiate the fake ICs from the original ICs.

As the fake ICs are of lower quality and due to their poor performance than the original ICs, the overall systems performance/reliability are affected. These components if found a way into the supply chain of critical systems such as, airplanes, cars, or defense electronics etc. can adversely affect the performance/reliability of those systems. These components due to their drawbacks, hurt the reputation and cause financial loss to the original manufactures in the market. Apart from the performance of such components in the system, counterfeiting leads to financial advantage to the adversaries, the potential threat to these components from Trojan insertion, or spyware is very high.

- Goal: Financial advantage, deny services, reduce reliability.
- Defenses: Physical un-cloneable functions (PUFs), IP watermarking, IP fingerprinting, Design obfuscation.

1.3 The subject of Hardware Security:

Considering the above-mentioned vulnerabilities in the supply chain, and to neutralize those threats and attacks, the concept of “Hardware Security” has emerged to the surface. Hardware security can be defined as:

"A property of the system at the fundamental level so as to thwart efforts that lead to resources of value from being copied, damaged, or made unavailable to genuine users." [8]

A good example of hardware security progress is pay-tv. Around 1994 providers started using smartcards for conditional access control. At the same time a large community of hackers was trying to reverse engineer and clone these cards to get free access to the contents of cable and satellite channels. The very first cards were quite simple and had lots of vulnerabilities, like sensitivity to UV light, power fluctuation and reduced clock rates. Service providers learned lessons from this and significantly improved not only the smartcards they used, but the encryption protocols too. Yet after several months determined hackers were again able to find a hole in the security. That again forced the providers into a cycle of hardware security improvements.

Attackers very often are ahead of the defenders for some obvious reasons. Firstly, a single vulnerability could let an attacker succeed, while a defender must protect against all known attacks. Secondly, even if the device can withstand all known attacks, no one can guarantee that a new attack will not be discovered. Initially, security-related bugs are always present in software. The question is who finds one first, and either fixes or exploits it. All these problems force the developer of a secure system to look for better ways of designing and exhaustive testing of his system. If the key element of his system is a standard smartcard, or off-the-shelf secure chip, he should test it properly, to be sure it will not be broken, and reverse engineered in a few weeks' time. One way might be to hire a specialized security evaluation company, but the better way, especially for large

manufacturers, would be to build his own research laboratory. In this case he will not leak any confidential information, and also do it faster.

1.4 Recent Work:

Hardware security is a vast field and there is active research taking place to address these issues in most of the research centers, this also includes GATE "Green, Accelerated & Trustworthy Engineering" Laboratory located in the Electrical and Computer Engineering department at George Mason University. Here, active research is being conducted in developing security measures to address the above-mentioned threats. This is done under the guidance of Dr. Avesta Sasan.

During the initial days of my thesis, I have been working on tool contributed to the Hardware Security community by the authors of the paper "Evaluating the Security of Logic Encryption Algorithms". The core functionalities of it includes, obfuscation of a given netlist for any specified encryption algorithm, and de-obfuscation of the encrypted netlist using any of the 2 in-built decryption algorithms. One of this algorithm was implemented with a SAT Solver at its core and the other was implemented using the CPLEX ILP Solver.

My work included in interfacing the different kinds of off-the-shelf SAT Solvers that could be interfaced with the tool, to evaluate their capabilities and limitations in de-obfuscating the encryption algorithms. The SAT Solvers were chosen based on their performance in the international SAT Competition, which takes place every year.

The SAT Competition is an evaluation platform where several SAT Solvers under development, in the SAT community, are submitted based on their respective tracks,

Main, Parallel, Random and No-limits tracks, each of these tracks have their own set of regulations. The organizers of the competition then, run these SAT Solvers on several benchmark problems that are found in the hardware and software application development.

The above work resulted in publishing of paper “Benchmarking the Capabilities and Limitations of SAT Solvers in Defeating Obfuscation Schemes”, to the On-Line Testing and Robust System Design (IOLTS) 2018 conference.

Later, I was involved in the development of a logic encryption schemes, which would try to defeat the current logic encryption de-obfuscation techniques. The implemented idea had to be to experimentally be tested, hence, I was responsible for generating the RTL files with the implementation idea embedded in it. Then, evaluate area, timing, power-consumption, of the obfuscated circuits by synthesizing them.

In the context of de-obfuscation of the encrypted circuits, the idea proposed was to accelerate the SAT Solver algorithm, present at the core of the tool, on a GPU to achieve speed-up in obfuscating the circuits. This thesis is a report of an attempt to accelerate the SAT Solvers on the GPU.

1.5 Motivation and Overview:

This thesis concentrates on the concept of Logic Encryption, as it has a potential to defend over wide range of attacks in the supply chain. In this technique, additional logic is added to the functional original circuits in an attempt to hide/obfuscate the original circuit. This additional logic added to the design, however, requires a specific

key to unlock the circuit, otherwise, it output would be invalid. The encryption logic that is being included could be implemented using either XOR *key gates* or look up tables (LUTs). The key inputs are driven by an on-chip tamper proof memory. Figure 2 depicts an abstract representation of the logic-locked design.

The key inputs are not revealed to the foundry, they manufacture the integrated circuits and send them back to the design house. In context, the design loads the correct key values into the ICs, hence, "activating" the IC and these activated ICs are then marketed to the users.

Logic encryption evolved from development of algorithms for deciding the best locations for inserting the *key gates*, such as:

- Random insertion-based logic encryption – Logic gates (mainly XOR gates) are inserted randomly into the original design, (known as salting), using TRNGs and PUFs [3].
- Strong-interference-based logic encryption – Logic gates inserted into the original circuit using an "interference graph", i.e. insert non-mutable gates that interfere with output of next gate [3].
- Fault analysis-based logic encryption – Involves in analysis of the faults in obfuscated circuits that would result in correct output and compensating over those faults by inserting XOR/X-NOR gates at logically selected locations [6].

Over the years, many key-recovery attacks have been mounted that exploit the vulnerabilities of these techniques, however, a powerful attack that broke all the logic

encryption techniques existing then is *Boolean satisfiability (SAT)* - based key-pruning attack, referred to as SAT attack. To overcome the SAT attack, techniques such as,

- SAT-Resistant Cyclic Logic Locking (SRCLock) – presents an idea of inserting logical cycles into the circuit, cyclic obfuscation, when properly implemented, poses exponential complexity on SAT or CycSAT attack [10].
- Anti-SAT – presents a circuit block (referred to as Anti-SAT block) to enhance the security of existing logic locking techniques against the SAT attack [16].
- SARLock – this technique exponentially increases the execution time of the SAT attack by maximizing the required number of distinguishing number of input patters to recover the secret key [15].
- Tenacious and traceless logic locking (TTLock) – Both SARLock and Anti-SAT are vulnerable to structural attacks since they implement the original function as is. In TTLock, the original logic cone F(I) is modified for exactly one input pattern is to hide the true implementation from an attacker and thwart structural attacks [1].
- Stripped functionality logic locking (SFLL) – SFLL is based on the notion of "strip and restore", where some functionality from the original circuit is stripped

and is stored in form secret keys in an on-chip tamper-proof memory. SFLL has two variants: SFLL-HD and SFLL-flex [1].

These techniques have in most cases rendered the SAT attack obsolete, by pushing the execution time of the embedded SAT Solver in the de-obfuscation tool to exponential.

Therefore, this led to the idea of accelerating the embedded SAT Solver, so that it can overcome this limitation of timing out before it completes execution and finds the secret key. For accelerating the SAT Solver, the GPU approach was chosen as to take advantage of their highly parallel architecture and the multiple CUDA cores on them.

Chapter 2 of this thesis has some background in the GPU Hardware architecture, Memory levels and programming environment on the NVidias CUDA toolkit known as CUDA. It also speaks about the internal methods and algorithms that are implemented inside the famous Minisat SAT Solver, which is at the core of the de-obfuscation tool. Also, many other solvers like glucose-4.1, maple-glucose, COMiniSatPS etc.… are built upon Minisat.

Chapter 3 speaks briefly about the SAT attack model and the techniques that are proposed to curb the SAT attack, its implemented algorithms for de-obfuscation, definitions and theorems for developing the attack model. The SAT solvers role in this attack.

Chapter 4 speaks about the implementation of the attack algorithm of the SAT attack and the stages that were involved in generating the SAT circuit for finding the keys. Also, the GPU SAT Solver implementation has been discussed in this chapter.

The results of benchmarking the logic obfuscation techniques and their strength is tested against the SAT attack implementation are discussed in chapter 5. The results are then discussed and analyzed to survey the capabilities and limitations of the solvers. The results of the GPU implementation of the SAT solvers are also discussed.

In chapter 6, the conclusions of benchmarking the different logic obfuscation techniques and the properties of the SAT Solvers are discussed in detail and the survey results are also finalized in this chapter. Also, results of the GPU implementation of SAT Solvers are justified and finally, a direction for future work is proposed for parallelizing the SAT Solvers.

CHAPTER 2: BACKGROUND

2.1 Overview of GPGPU:

Modern graphic cards (Graphics Processing Units) are true multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy to support graphical processing (e.g., DirectX and OpenGL APIs). Efforts like NVIDIA's CUDA – Compute Unified Device Architecture [26] aimed at enabling the use of the multicores of a graphic card to accelerate general (non-graphical) applications by providing programming models and APIs that enable the full programmability of the GPU. This movement allowed programmers to gain access to the parallel processing capabilities of a GPU without the restrictions of graphical APIs. We consider the CUDA programming model proposed by NVIDIA. The underlying conceptual model of parallelism supported by CUDA is Single-Instruction Multiple-Thread (SIMT), a variant of the popular SIMD model; in SIMT, the same instruction is executed by different threads that run on identical cores, while data and operands may differ from thread to thread. CUDA's logical architectural model is represented in Figure 2**.** A disadvantage of this platform is that SMT provides mechanisms of concurrency that are not present in GPUs such as locks and semaphores.

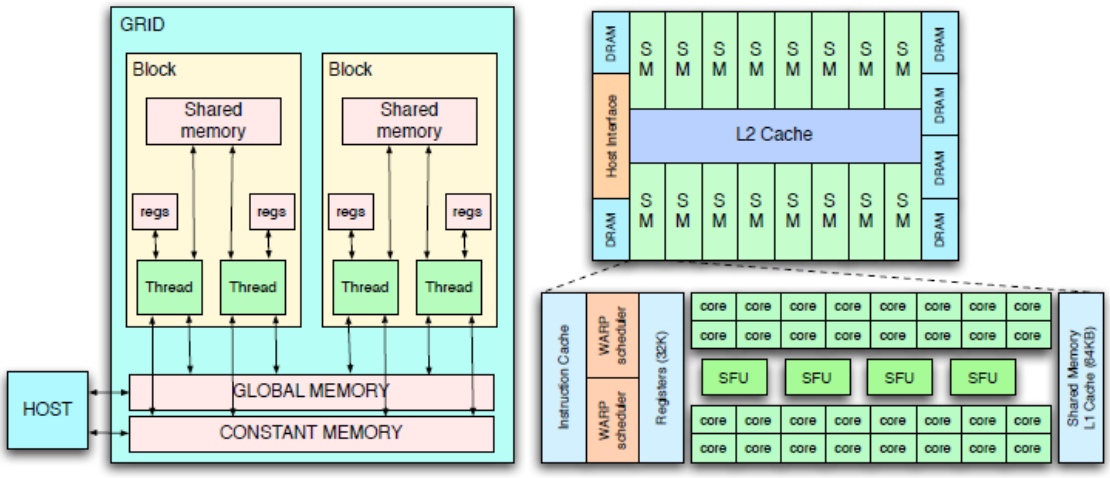


Figure 2: Logical CUDA architecture.

2.1.1 Hardware Architecture:

A General-Purpose GPU (GPGPU) is a parallel machine and different NVidia GPUs are distinguished by the number of cores, their organization, and the amount of memory available. The GPU is composed of a series of Stream Multi-Processors (SMs); the number of SMs depends on the specific characteristics of each class of GPU e.g., the Fermi architecture, shown in Figure 2, provides 16 SMs and the Kepler architecture shown in Figure 3**,** and provides 192 cores.

In turn, each SM contains a collection of computing cores, typically containing a fully pipelined ALU and floating-point processing unit. The number of cores per SM may range from 8 (in the older G80 platforms) to 192 (e.g., in the Kepler platforms). Each GPU provides access to both on-chip memory (thread registers and shared memory) and off-chip memory (L2 cache, global memory and constant memory).

The GPGPUs used to run the experiments are Tesla K80 GPU accelerators. They are based on 2 individual GK210 GPUs connected using a PLX switch for communication [30]. Each GK210 has 13 SMs, which contain 192 CUDA cores. Therefore, in total, each GK210 consists of 2496 cores. Finally, the Tesla K80 has 2 x 2496 = 4992 CUDA Cores. Figure 4 (above) shows the Kepler architecture of the GK210 GPU and SM is shown in the Figure 4 (below).

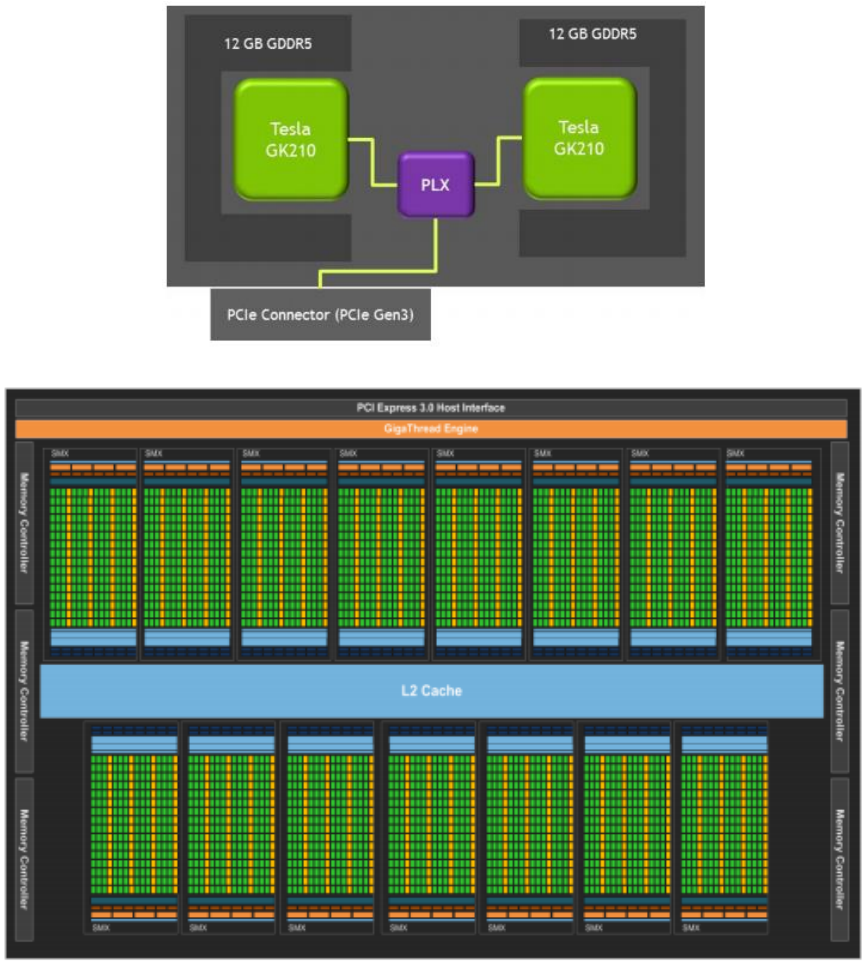


Figure 3: Tesla K80 GPU Architecture (above), Kepler Architecture (below)

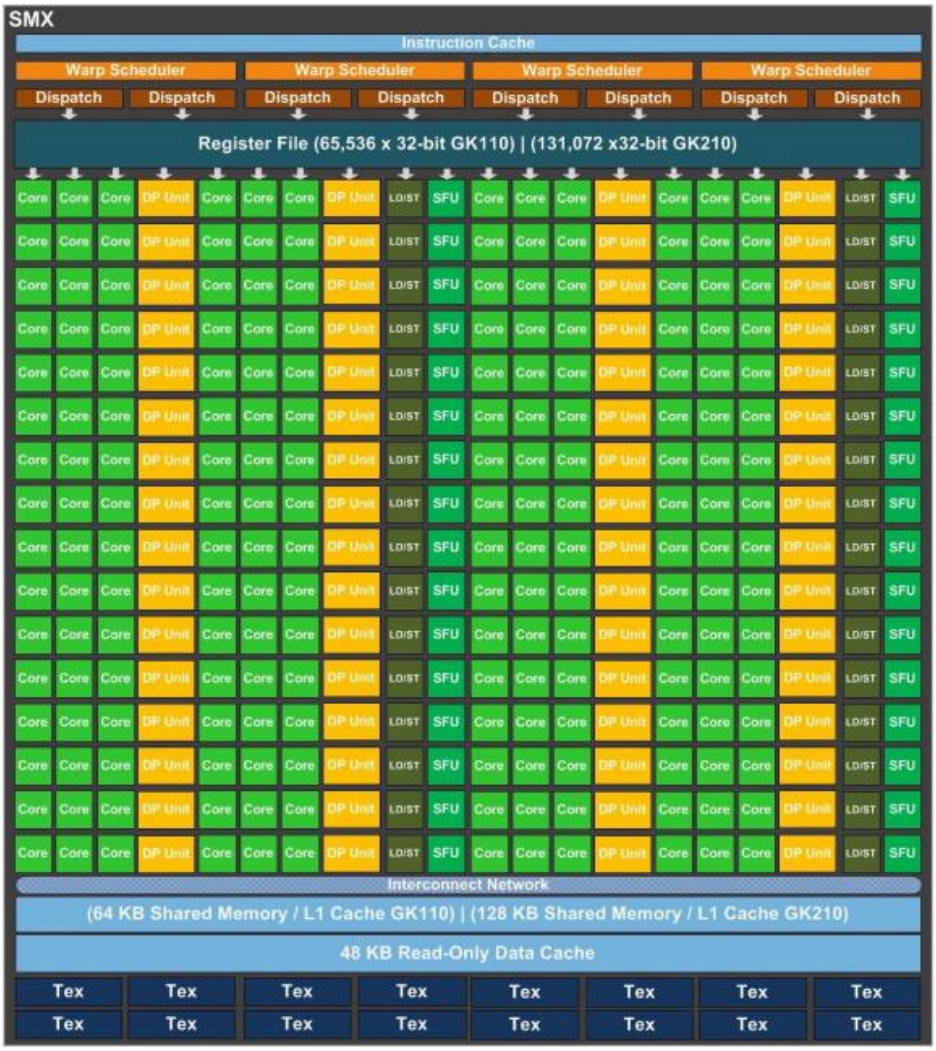


Figure 4: Inside the *Stream Multiprocessor* of the GPU.

2.1.2 CUDA Programming Model:

The CUDA architecture recognizes two entities capable of performing computational – traditional CPU (referred as Host) and cores of GPU (referred as Device). As a result, a CUDA program is a high-level program that is decomposed in parts that can be run on the host and parts that can be executed on the device. Thus, the

CUDA program oversees managing both sets of computing resources. For the C based implementation every function in a CUDA program is labeled as host, global, or device. A host function runs in the host and it is a standard C function. A global function is called by a host function, but it runs on the device.

A global function is also referred to as a CUDA kernel function. A device function can be called only by a function running on the device (global or device) and it runs on the device as well. Functions running on the device must adhere to some restrictions, depending on GPU's capabilities. In particular, limitations about the number of registers per thread and shared memories should be carefully handled.

The programming model is based on the logical architecture shown in Figure 2. When a CUDA global kernel is called from the host program, it generates the indicated number of parallel instances of the kernel to run on the GPU. These kernels are then logically arranged into *blocks*, which are in recursion grouped into g*rids*.

When mapping a kernel to a specific GPU, each block is scheduled on each of the SMs in the GPU. The schedulers in these blocks then schedule a group of 32 threads from their respective blocks. These 32 threads from the blocks are called as *warps*, which is the smallest work unit of execution on the device.

The number of blocks, grids and threads to be launched on the device are specified in the kernel call, which is invoked in the host device. The kernel call can be showed as,

Kernel_name<<< gridDim, blockDim, 1 >>> (args);

where,

Kernel_name	= global kernel function call from the host.
gridDim (a, b)	= tuple specifies number of blocks in a & b dimensions.
blockDim (x, y, z)	= triplets specify number of threads in x, y & z dimensions.
"1"	= default (instantiating threads in 3-D, no hardware support yet.)
args	= arguments to be passed to the global kernel function.

The dimensions of the grid and blocks are defined by the programmer, depending on his application. To identify the threads on the GPU the threads are indexed based on the location on the GPU, which are derived from gridDim(a, b) and blockDim(x, y, z). variables blockIdx.x and blockIdx.y contain the unique 2D block identifiers, also, threadIdx.x, threadIdx.y and threadIdx.z, contain the unique thread identifiers.

2.1.3 CUDA Memory Model:

In CUDA there are memory levels that have different latencies and characteristics depending on their location on chip, caching, scope/lifetime. These memories are connected by the system bus and the DMA transfers are used to transfer the data to the GPU. The data is copied to the GPU either through implicitly (memory mapping) and/or explicitly (through the cudaMemcpy function). There are 6 different memory levels of

memory, *(1) Registers, (2) Local memory, (3) Shared memory, (4) Global memory, (5) Constant memory and (6) texture memory.*

- **Registers** – They are allocated automatically by the compiler. However, the number of registers can be reduced by hand at compile time, through a compiler flag. Their scope and lifespan are dependent on the threads. The number of registers is defined per block in the GPU, hence, the number of registers required per thread is variable.
- **Local Memory** – This memory is used to store the overflow data from the registers. However, this memory is in the global memory space, hence has the problems of the global memory, like high latency effect the performance.
- **Shared Memory** – This memory has block scope, hence, this memory is used for thread co-operation. It uses the same memory bank as of the L1 cache, and their sizes range from 16KB to 48KB for each block. This memory is divided in 32 banks, which cannot be accessed at the same time. If a concurrent bank access exists, they are serialized resulting in a repetition of the instruction, causing performance degradation. This memory should be used when the access pattern to the data is random resulting in performance degradation if global memory is used. This memory has the lifetime of a block.
- **Global Memory** – This memory is generally in the order of Gigabytes, it is accessible to all the threads on the GPU and has a lifespan of the

application. However, its access speed id slow, usually takes about 400 to 800 cycles per access. The preferred way of accessing global memory is by doing coalesced accesses, i.e., a half warp (16 threads) from a block access the memory. In context, the maximum memory throughput is possible when these threads access a contiguous memory location.

- **Constant Memory** – This memory is used for read-only access and/or caching purposes. It is allocated by the host on the GPU, has faster access times but has limited space. This memory is visible to both the device and host.
- **Texture Memory** – Texture memory is different from Constant memory, because, it is possible to have un-coalesced accesses to it. This memory is also used for read-only accesses and/or caching purposes. However, its latency is like that of the global memory.

In a complex scenario like this, design of an efficient parallel protocol depends on the choice of the type of memory to use and the access patterns generated by the execution.

2.2 Boolean Satisfiability Problem:

2.2.1 Overview:

The Boolean Satisfiability Problem consists of, given a Boolean formula, determine if there is an assignment, true or false, to all, or a subset, of the Boolean variables, such that the formula evaluates to true.

In a formula, there might be more than an instance of each variable. To each instance of a variable or its negation we call a literal. So, both "x" and "~ x" are literals of the same variable x. A variable can have one of three values, true, false or not a value. In a satisfied formula, that evaluates to true, not all variables may have an assigned value. An example formula is shown below,

$$((x \vee \neg z) \wedge y) \vee (\neg x \wedge y)$$

Equation 1: Boolean Equation.

However, the input to the SAT Solver is given in the Conjunctive Normal Form (CNF). In this form, the formula is composed as a conjunction of clauses that in-turn are disjunction of literals that compose the Boolean formula. For example, Equation 2 displays the CNF form of Boolean formula in Equation 1.

$$(x \vee \neg z) \wedge (\neg x \vee y) \wedge \neg z$$

Equation 2: Boolean equation in CNF form.

For the formula to be *Satisfiable*, the following criterions should be met,

1. At least, one of the literals should be assigned in every clause, and
2. All the clauses in the formula should be true.

The formula is said to be *Un-Satisfiable*, if any of the above two criterions is not met.

2.2.2 DPLL Algorithm:

The DPLL algorithm also known as, Davis-Putnam-Longemann-Loveland algorithm is used to verify the satisfiability of the Boolean equation in the CNF form. This algorithm was developed in 1962 and is the basis of most of the modern off-the-shelf SAT Solvers.

The DPLL algorithm implements several routines, such as back tracking, variable selection procedure, Conflict analysis etc.… The modern version of the DPLL algorithm is shown below,

Algorithm 1: DPLL Algorithm pseudo-code:

```
1.  procedure DPLL_loop
2.      status ← preprocess()
3.      if ( status = / = UNKNOWN ) then
4.          return status
5.          while (true) do
6.              decideNextBranch()
7.              while (true) do
8.                  status ← propagate()
9.                  if (status == CONFLICT) then
10.                     BTstatus ← backtrack()
11.                     if (BTstatus == UNSATISFIABLE) then
12.                         return UNSATISFIABLE
13.                     else if (status == SATISFIABLE) then
14.                         return SATISFIABLE
15.                     else break
```

Table 1: The DPLL algorithm of the SAT Solvers.

The preprocess step of the SAT Solver is used to instantiate the required data structures, and read/parse the input CNF file, in which the input is in the DIMACS format. Then the solver then translates the read variables from the CNF file into its own format (data structures), so to keep track of all the variables and clauses.

Then the variable selection procedure selects a variable from the formula using any of the defined selection algorithm, and randomly assigns the variable to either true/false. Therefore, by the end of this procedure, the SAT Solver has picked a branch in the search.

In the next procedure, the Solver has to verify the assigned variable with all the clauses and see if they are satisfied. This procedure is done by the iterative procedure known as *propagate*(). The propagate procedures details are explained in the next section. This procedure either returns “NULL” or returns “conflict”, if the current assignment causes one of the clauses to be falsified, which in-turn renders the whole formula Un-Satisfiable.

If propagate returns conflict, then the Solver backtracks to implicit assignment (assignment made by the solver) that caused the formula to be falsified. In context, if the solver has returned to the first level of assignment and tries to backtrack, then the formula is *Un-Satisfiable*.

While searching and iterating over this loop, if we find an assignment that satisfies all the clauses in the formula, then the problem is classified as *Satisfiable*.

The next sections speak about the different procedures and algorithms that are implemented in most of the SAT Solvers, to optimize their search.

2.3 Conflict-Driven Clause-Learning:

Modern solvers, based on the simple DPLL, employ the Conflict-Driven Clause-Learning (CDCL) scheme. This name comes from the fact that unlike DPLL, modern solvers learn every time a conflict happens. This allows the solver to prevent the same mistake from happening again and it also enables the solver to better lead the search based on what it already knows [5].

2.3.1 Clause learning and Implication graph:

The central part of the conflict analysis is a concept known as implication graph. It represents the reasoning leading to a certain conclusion, generally a conflict. At any given time, the implication graph represents the current assignments and the implications generated. This graph is a concept, as it does not exist, it is instead built implicitly in the solver's state.

Every time a conflict happens, this graph can be analyzed and through this analysis we can create new clauses that prune the search by allowing us not to repeat the same mistakes. Additionally, by analyzing the conflict clause, we can also conclude to what level we should backtrack. After we are done with the analysis of the implication graph, we store the reason for the conflict in the form of a new clause. This clause is also known as the learnt clause, it helps the solver to avoid the same branch to be taken again.

2.3.2 Non-Chronological Backtracking:

Non-Chronological Backtracking helps the solver to return to the state where the conflict assignment had been made by the solver. Therefore, the solver must revert all the assignments, and other data structures to that particular state. This can be done by traversing the above-mentioned implication graph.

When a conflict happens in the propagation stage, the solver then looks at the data structure which stores all the variable assignments that took place in that level and then, assigns them back to their previous state. Therefore, when a conflict happens, the solver directly goes to the state where the "conflict assignment" was made and then picks some different branch and proceeds with execution.

2.3.3 Two-Watched Literals:

Another addition to the modern SAT Solvers is a data structure known as two watched literals. This is a lazy data structure, in the sense that it is only evaluated when it needs to, that aids in the process of unit propagation and backtrack.

With watched literals, instead of having to process the entire clause to check for any implications, we just check two literals. Mostly, the watched literals are the first two literals in a clause. These literals should be unassigned. Every time one of those two literals evaluates to false or is satisfied, we move our watch to another literal in the clause that that is not falsified. If we have only one place left, and it is unassigned, we have found an implication. If, on the other hand, we have no places left to put any of the watches, we have a conflict. The two watched literals structure has another feature.

2.3.4 Variable Selection Heuristics:

The Variable State Independent Decaying Sum (VSIDS) heuristic was introduced with CHAFF. This heuristic gives priority to variables that were involved in recent conflicts. It does this by watching the activity of a variable by an amount, during conflict analysis. Then, periodically, it divides all the activities effectively making old conflicts worth less than recent conflicts. This procedure then helps the solver to assign the variables that are active and leave the in-active variables to assign them at last.

2.3.5 Unit Propagation:

The propagate stage is also known as Boolean Constraint Propagation (BCP) stage, which has the following computational pattern,

1. For each variable that is assigned, fetch a list of all clauses that contain this variable. This was improved to a smaller sized *watched literals* list.
2. For each of these clauses, fetch the variable assignments. If all variables except for one are assigned such that the clause is not satisfied, we know we must assign the unassigned variable to satisfy the clause.
3. As BCP implies new variables, repeat step 1.

Notice that what is fetched in step 1 indicates what to fetch in step 2. Once we have the variable assignments, we apply the BCP rules and may imply a new variable. Eventually that new variable will later tell us what to fetch in its step 1. This access pattern is analogous to traversing a link list, as what we fetch now tells us where to look next [29].

CHAPTER 3: THE SAT ATTACK

3.1 Introduction:

The main idea of the SAT attack is to reveal the correct key of an obfuscated circuit, by selectively applying the *Distinguishing Input Patterns* (DIPs) [12], to a functional IC. The attack rules out incorrect key values by using DIPs iteratively. A DIP is an input value for which at least two unique key values, K_1 and K_2, produce differing outputs, O_1 and O_2, respectively. Since O_1 and O_2 are different, at least one of the key values is incorrect. Let the set of complete inputs be defined by *Set Correct keys* (SCK). It is possible for a single DIP to rule out multiple incorrect key values, as shown in Figure 5.

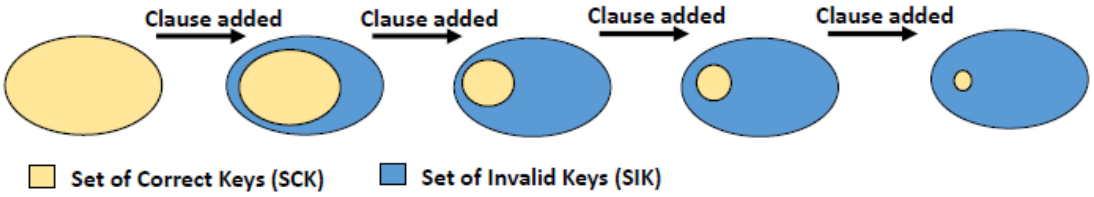


Figure 5: SCK reducing after each iteration.

For the SAT attack to be implemented, the attack should have access to the following resources,

1. An original functional IC.
2. A Reverse Engineered netlist containing the obfuscated circuit.
3. Ability to probe the input/output of the combinational obfuscated circuit.

3.2 Preparing Obfuscated Netlists for SAT Attack:

The attacker must first translate the obfuscated circuit into a format the SAT Solver can understand. This is called the SAT Circuit or *SATC*. For the SATC to be achieved, the circuit must be built into the following intermediate stages. Their functionality is explained in the below sections.

3.2.1 Key-Programmable circuit (KPC):

It is the original obfuscated circuit which contains the *key gates*, to which when only correct *key Input* applied, functions correctly.

3.2.2 Key-Differentiating circuit (KDC):

To build a circuit with functional DIPs, two copies of the KPC are used, their non-key inputs (X in Figure 7) are tied together, and their outputs are XORed. This circuit produces logic 1 when the output of two instantiated KPCs for the same input X but different keys K1 and K2 are different. The resulting circuit is denoted as Key-Differentiating Circuit (KDC).

3.2.3 DI Validation Circuit (DIVC):

The candidate keys in the SCK are capable of producing the correct output for all DIs that have previously been discovered and tested on the KPC circuit. In order to test the keys for one DI, this circuit is instantiated. In this circuit, FC is working copy of the chip, and its output is used for testing the correctness of both KPCs for a given DI and two key values. This circuit is denoted as DI-Validation Circuit (DIVC).

3.2.4 SCK Validation circuit (SCKVC):

To test the keys for all DIs, the DIVC circuit is duplicated D times, with D being the number of current DIs tested, and the output of all DIVC circuits ANDed together. This resulting circuit is a validation circuit for SCK set, which is denoted as SCKVC.

3.2.5 SAT Circuit (SATC):

The SAT circuit is built while finding the DIs. After all the DIs are extracted and added to the SCKVC, there would be only correct keys available in the search space. Therefore, any valid input to this circuit would result in correct output. If two keys K1 and K2 produce the correct output for all previously tested DIs (SCKVC evaluates to true), but produce different results for a new input X_{test}, then X_{test} is a DI that further prunes the SCK. This could be tested by using an AND gate at the output of SCKVC and KDC circuits. The resulting circuit forms a SAT solvable circuit denoted by SATC.

When SATC evaluates to true, the KDC has tested a pair of keys K1 and K2 that produce two different results for an input Xtest, and SCKVC circuit has confirmed that both K1 and K2 belong to SCK set. Hence, the input Xtest is yet another DI. Each time a new DI is found, the SCKVC should be updated by adding yet another DIVC circuit for testing the newly discovered DI. This process is continued until SAT solver no longer finds a solution to the final SAT circuit. In this case, any key remaining in the SCK set is a correct key for the circuit.

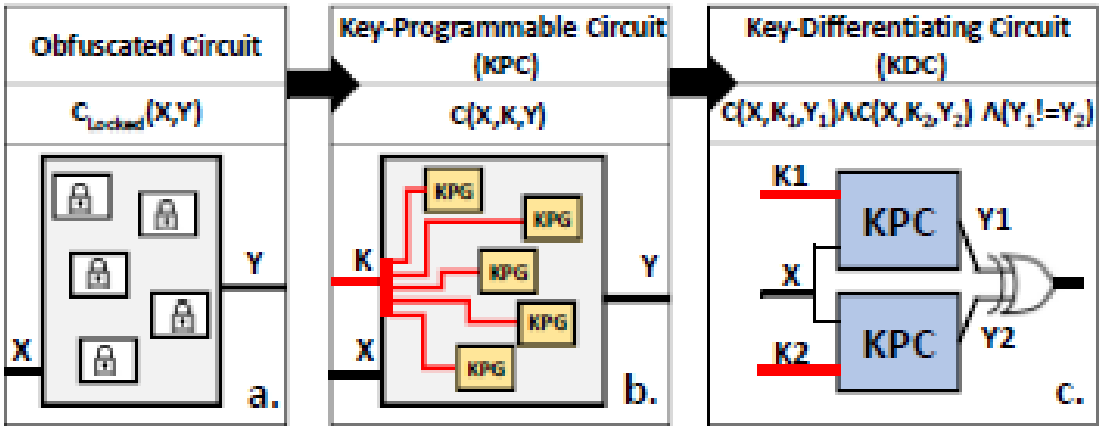


Figure 6: a. Original circuit. **b**. Key-Programmable circuit. **c**. Key-Differentiating Circuit.

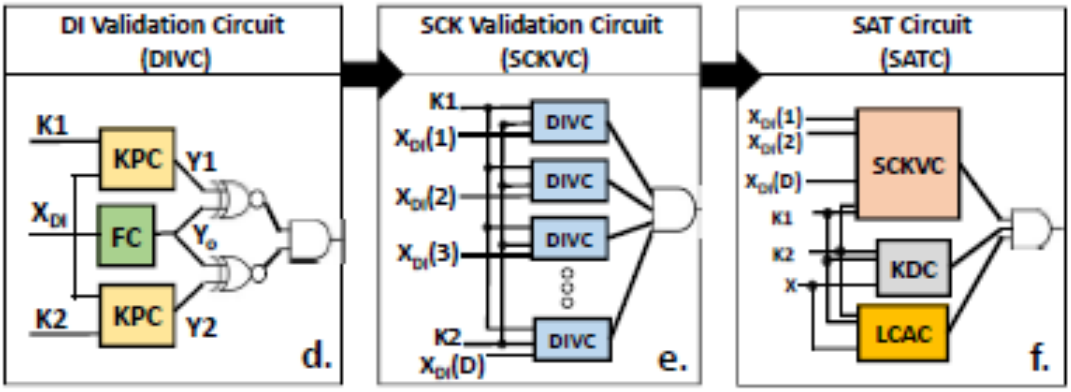


Figure 7: d. DI Validation Circuit. **e**. SCK validation Circuit. **f**. SAT Circuit.

3.3 SAT Attack Algorithm:

The SAT attack algorithm after the construction of the SATC circuit. At first, the SCKVC does not contain any logic in it, as there are no previously rested DIs. Hence, the output of this circuit is set to true. The SAT circuit is a combination of the multiple iterations of the SVCKC, a KDC and a LCAC. Where, LCAC is a circuit which contains the learnt clauses extracted from the SAT Solver. The outputs of this circuits are ANDed together. The C_F is a call to the functional circuit, that gives a correct output for every new DI found.

The while loop is executed until no other DI is found. At this point, any key in the SCK set is a correct key. To obtain a correct key, the DIVC circuit is modified to take a single key denoted as KeyGenCircuit. Hence, KeyGenCircuit has input K, and its output is valid if K satisfy all previous constraints imposed by previously found DIs. If the SAT solver does not return a valid key, it means the obfuscation, locking, or camouflaging

technique is invalid. Algorithm 2 shows the SAT Solvers attack, Figure 8 shows a simplified flowchart implementation of Algorithm 2.

Algorithm 2: SAT attack:

1. KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
2. SCKV C = 1;
3. SATC = KDC ∧ SCKVC, LCAC = 1
4. **while** ($(X_{DI}, K_1, K_2, CC) \leftarrow SATF(SATC) = T$) **do**
5. $Y_f \leftarrow C_F(X_{DI})$**;**
6. DIVC = $C(X_{DI}, K_1, Y_f) \wedge C(X_{DI}, K_2, Y_f)$;
7. SCKVC = SCKVC ∧ DIVC;
8. LCAC = LCAC ∧ CC
9. SATC = KDC ∧ SCKV C ∧ LCAC;
10. **end while**
11. *KeyGenCircuit* = SCKVC ∧ (K1 = K2)
12. *Key* ← SATF(*KeyGenCircuit*)

Table 2: The SAT attack algorithm.

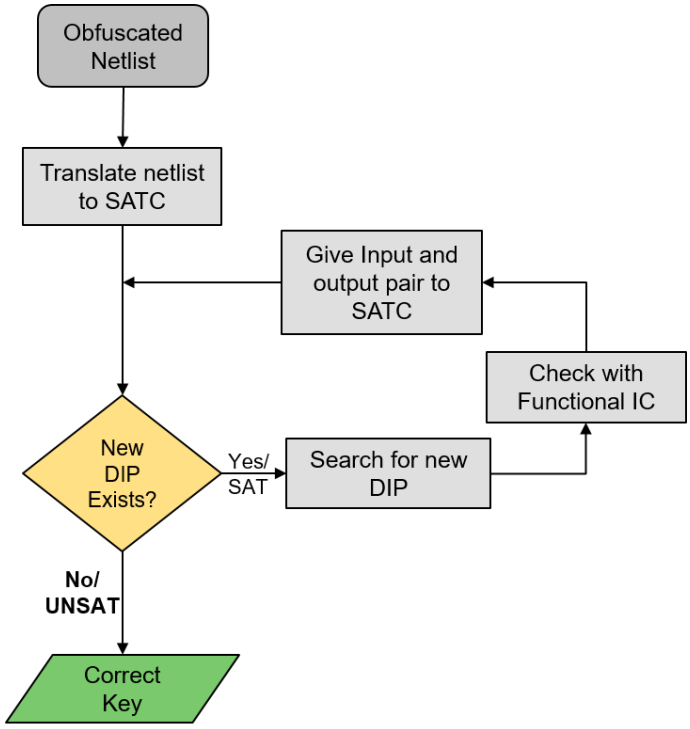


Figure 8: Simplified SAT algorithm flowchart.

3.4 SAT Attack and GPU acceleration:

The SAT attack was defined above. This attack implements a SAT Solver at its core, which for every iteration will return satisfying assignments of the variables, that would be Distinguished Input of the circuit and after the circuit has run for several iterations, i.e., if the circuits is SATC, then the returned assignments are the keys of the circuit for any given input.

As the SAT Solver plays a major role in de-obfuscating the circuits, therefore, accelerating them would be a way to speed up the process.

CHAPTER 4: IMPLEMENTATION

4.1 Implementing the SAT attack:

To implement the SAT attack, a software tool based on C++ was developed, which includes the functionalities both encryption and decryption of the circuit. For the encryption of the circuit, the software tool is required to be provided with, the encryption technique to be used, and the original netlist of the circuit.

The tool does not accept RTL files as input, they must be modified into the format that can parsed by it. They are represented by "*.bench*" files, which contain inputs, outputs, and the nets that are used to connect the gates to each other. The original circuit's RTL files are converted into this format using python scripts that parse the RTL files and, produce the bench files as output. In context, the user must provide this generated bench file and the encryption technique to generate the obfuscated circuit. The encryption techniques include such as random insertion, toc13mux/xor, etc.… These techniques are discussed in the next section. The output of the tool is another bench file with gates that are inserted into the circuit, according to the choice of encryption circuit. The obfuscated circuits are then again translated back into RTL format using another python script which takes bench files as inputs.

The tool also implements a de-obfuscation technique, which employs the SAT attack algorithm discussed in the previous chapter. It requires both the netlists of the

obfuscated and the original circuits, as bench files. The tool then de-obfuscates the encrypted circuit according to the SAT attack algorithm. The output consists of the key which renders the obfuscated circuit to work as the original circuit. The tool to speed up the de-obfuscation of the circuits implement different kinds of SAT Solvers like, minisat, glucose, maple-minisat, maple-glucose, cmsat and lingeling.

4.2.1 Tseytins Transformations:

SAT/Satisfiability Solvers are used to check if for a given Boolean formula, if there is a satisfiable assignment to the variables present in the given formula. The formula provided to the solvers are in Conjunctive Normal Form (CNF) format, as explained in chapter 2.

However, the bench files generated from the RTL files have gates and wires, which does not follow the CNF format, therefore, the tool translates these gates into the CNF format using Tseytins transformation equations. The table below shows the different equations for the basic gates in the circuits. The inputs to the gates are A and B, while C is the output.

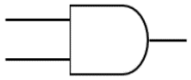
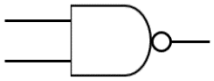
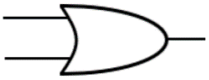
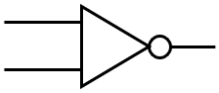
Gate type	Operation	CNF Sub-expression
 **AND**	$C = A\,.\,B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
 **NAND**	$C = \overline{A\,.\,B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee C) \wedge (B \vee C)$
 **OR**	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
 **NOR**	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
 **NOT**	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
 **XOR**	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$

Table 3: CNF sub-expressions for all gate types.

4.2 Obfuscation Techniques:

A random obfuscation scheme was proposed by Roy et al. we refer to this obfuscation as rnd. A major weakness of this scheme was the ability of an attacker to sensitize the circuit, by application of carefully selected inputs, and to propagate the obfuscation keys to the primary outputs of the circuit. Rajendran et al. proposed a more sophisticated obfuscation mechanism to avoid such sensitization attacks. We refer to this scheme as dac12. An important metric in logic obfuscation is increasing the output

corruption when a wrong key is used. Rajendran et al. also proposed an obfuscation method that uses fault propagation analysis. We refer to these obfuscation schemes as toc13xor and toc13mux. Wires with low controllability are susceptible to Trojan insertion. To obfuscate the degree of controllability of wires in a netlist, Dupuis et al. tried to minimize the wires with low controllability. We refer to this obfuscation method as iolts14. These obfuscation techniques are briefly explained,

- **rnd –** In this obfuscation technique, key gates are randomly inserted into the original circuit, to which only when the correct key applied, functions like the original circuit. [7]
- **dac12 –** This technique involves in the insertion of XOR/XNOR gates to the original circuit, in order to, reduce the probability of the fault-based attacks on the obfuscated circuits. These gates are inserted into the obfuscated circuit based on an algorithm that carefully selects locations for insertion. [5]
- **toc13 –** In this technique, a logic element is inserted into the original circuits at specified locations in the circuit, such that, the hamming distance between the correct and in-correct output increases. This algorithm is implemented using 2 logic elements as logic blocks, XOR gates and MUX's. [5]
- **iolts14** – Controllability of wires in a circuit can be used to find if there are any Hardware trojans inserted into the circuit. Therefore, to increase the number of controllable wires in the circuit, this obfuscation technique was introduced.

4.3 SAT Solvers implemented:

The de-obfuscation tool employs a SAT Solver at its core, the algorithm 2 of chapter 3 defines its use. A total of 6 SAT Solvers were interfaced to the de-obfuscation tool, to analyze the limitations and capabilities of them. The motivation for analyzing is to enlighten researchers in the hardware security field to use a suitable SAT Solver, according to their application, that would help them test their proposed obfuscation schemes. The observed limitations and capabilities of each of these SAT Solvers are presented in detail below.

- Minisat – It is developed as a modular SAT Solver that implements conflict-driven backtracking, watched literals and dynamic variable ordering. Most of the modern off-the -shelf SAT Solvers are a version of this solver. For this survey of limitations and capabilities, minisat can be used. As to identify features used in new solvers and evaluate their effect on the de-obfuscation of circuits. Minisat is implemented using several user-defined methods and functions, which are optimized in terms of access time and memory usage.

- Glucose – This solver is an extension of Minisat which remove clauses that are already satisfied and does not affect the search. It also implements dynamic restarts strategy, called as Literal Block Distance (LBD). LBD is used to stop the solver to be stuck in just one branch of the search, rather restart at different parts of the search and save the learnt clauses, from conflicts for every restart.

- Lingeling – It is based on the idea of interleaving search and pre-processing. It uses various techniques to reduce the size of the search space. Binary and ternary clauses are stored separately from large clauses. Large clauses are kept using literal stacks and references to them are simplified from pointers to stack position. Binary and ternary clauses are kept in occurrence lists. Occurrence lists are defined using stacks and are referenced by stack position. It also uses a modified version of restart mechanism used in Glucose. Number of variables and clauses are also monitored during execution and the number of learned clauses is controlled using their variance.

- Maple-(minisat/glucose) – Maple uses a new branching heuristic in place of Variable State Independent Decaying Sum (VSIDS) called Learning Branching Heuristic (LRB). Two variants of MapleSat are Maple-minisat and Maple-glucose, respectively based on Minisat and Glucose. Maple-glucose uses LRB for 2500 seconds of the execution, and then switches to VSIDS. In MapleMiniSat, VSIDS is replaced with LRB.

- CryptoMiniSat – This is a SAT solver that compiled from SatELite, PrecoSat, Glucose and MiniSat features. It has special mechanisms for XOR clause handling and separates watch lists for binary clauses. It can detect distinct subproblems in clause list and try to solve them with sub-solvers.

4.4 Attack Algorithm explained:

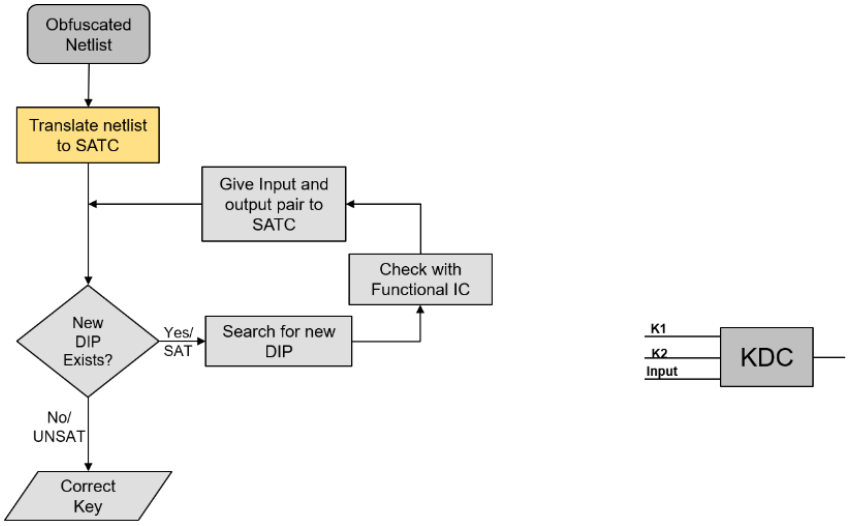


Figure 9: Translation stage of the algorithm.

The SAT attack algorithm to be implemented, the circuit in the input bench files is converted into the CNF format, and translated to generate the intermediate stage, KDC, which is provided with an input and two randomly generated keys. Here the SCKVC does not contain any logic, therefore, it is evaluated assuming it to be true. Figure 9 shows the algorithms first stage.

After translation, the de-obfuscation tool iterates over the while loop providing the SAT Solver with different inputs for each iteration as the DIVC. Each DIVC is an instance of the circuit with different valid inputs, from the search space. The output of the DIVCs are ANDed together. Therefore, this results in building of the SCKVC for each iteration. The while loop iterates until no other DI is found. Figure 10 depicts the generation of the SCKVC for each iteration of the while loop.

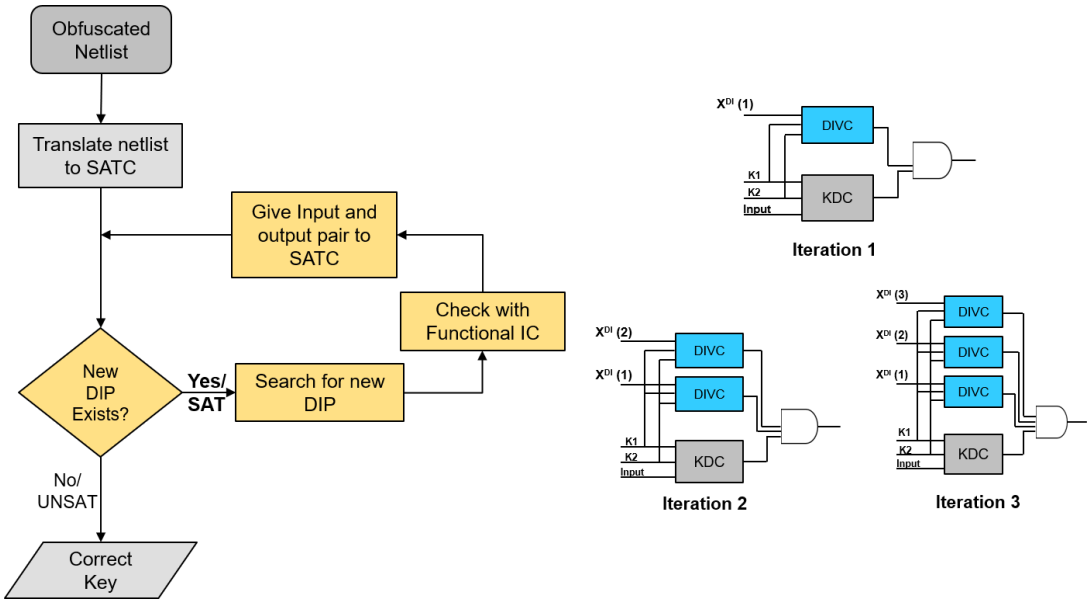


Figure 10: SATC generation over multiple iterations.

On completion of *D* iterations, the SAT Solver returns UNSAT to the tool, this indicates the search has completed and there are no assignments to be removed from it. At this point the circuit is called the SAT Circuit (SATC). Therefore, the remaining inputs in the search space are only the correct keys. Which can be extracted from the SAT Solver by providing any random input to the SAT Circuit (SATC). Figure 11 shows the final iteration. To obtain a correct key, the DIVC circuit is modified to take a single key denoted as KeyGenCircuit. Hence, KeyGenCircuit has input K, and its output is valid if K satisfy all previous constraints imposed by previously found DIs. If the SAT solver does not return a valid key, it means the obfuscation, locking, or camouflaging technique is invalid.

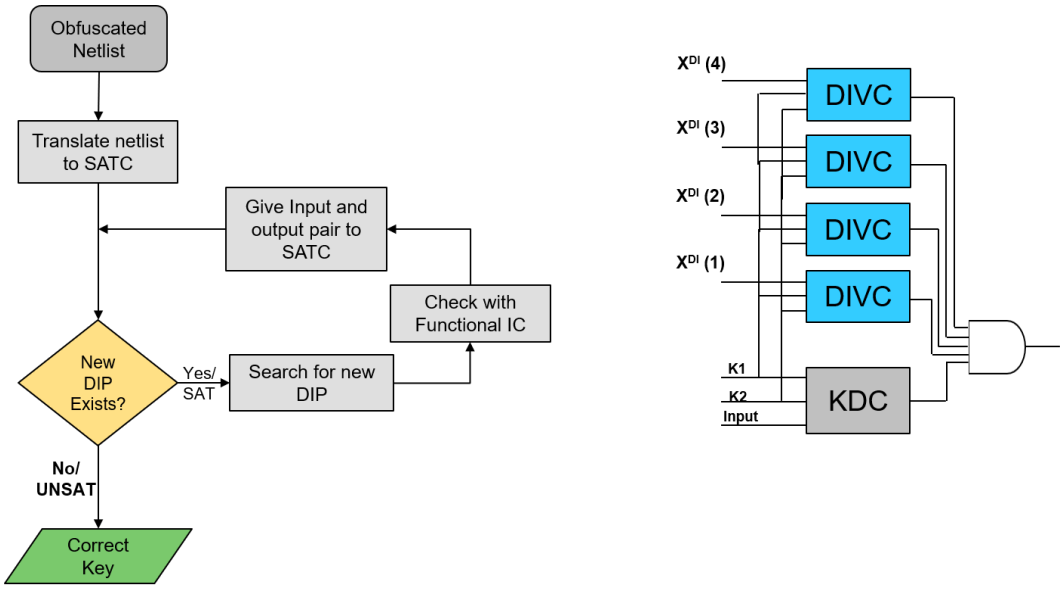


Figure 11: Final Iteration of the de-obfuscation tool.

4.4 Minisat on GPU:

For GPU acceleration of the SAT Solvers, Minisat was selected due its modular implementation of the code. Also, this solvers implementation was clear and extension documentation was also available. The implementation involved in accelerating a core method in the DPLL based solver named “propagate”, this method is based on Boolean Constraint Propagation. In simple terms, BCP prunes unsatisfiable branches as the SAT solver traverses the tree of all possible variable assignments. The computation pattern of BCP is as follows:

1. For each variable that is assigned, fetch a list of all clauses that contain this variable. This was improved to a smaller “watched literals” list.

2. For each of these clauses, fetch the variable assignments. If all variables except for one are assigned such that the clause is not satisfied, we know we must assign the last unassigned variable to satisfy the clause.
3. As BCP implies new variables, repeat step 1 for each.

The BCP procedure takes up around 80% to 90% of the solvers run time. This is because the modern solvers spend less time assigning new variables and more time propagating the implied assignment of a variable to other effected clauses in the problem. Therefore, the solver only spends 10 % to 20% of the time in assigning, backtracking, conflict analysis and other methods.

The approach to accelerate the SAT Solvers was inherently the possibility of accelerating BCP/propagate stage of the solvers. Two approaches are addressed here, one, parallel implementation of the BCP procedure, as in *cudasat*, and parallelizing the original sequential BCP procedure in minisat.

Cudasat is a GPU based SAT Solver that implements a parallel of the DPLL algorithm and the idea of parallelizing the BCP procedure was proposed and tested. On the other hand, the BCP procedure was parallelized by launching multiple threads of the internal nested *for* loops. The cudasat implementation involved several modes of operations, each mode represented a combination of the features implemented in the solver, for both CPU and GPU solving. After the analysis of both solvers, the results are discussed in the next chapter.

CHAPTER 5: BENCHMARKING & RESULTS

5.1 Benchmarking:

Benchmarking of the obfuscation schemes was done as to contribute to the hardware security community to understand the capabilities and limitations of the SAT Solvers when dealing with hardware obfuscation problems. This would provide an insight on the capabilities and performance of the SAT Solvers, therefore, helping the researchers in selecting the most able SAT solver for evaluating the effectiveness and hardness of their proposed obfuscation schemes. Hence, preventing the researchers from selecting a poor choice of the SAT solver solution for their obfuscation scheme and generalizing the failing results of the SAT solvers to break the proposed obfuscation scheme.

For this purpose, a total of 6 SAT Solvers were chosen from the SAT competition, which tested the hardness of the obfuscation schemes discussed in the previous chapters. The benchmarks circuits were used from the ISCAS'85 and MCNC suites. The circuits are obfuscated with an area overhead of 1%, 2%, 3%, 5%, 10% & 25%. To account for a run-to-run variation in performance, we ran the tool for 15 times for each obfuscated benchmark. The benchmarking platform included a farm of 20 Dell Latitude-7010 desktops equipped with Intel-i5 processor and 8GB of RAM, running in shell mode of the Ubuntu 16.04 LTS operating system.

5.2 Results and Analysis:

The difficulty of the obfuscation techniques are tested and the results are shown in terms of *Execution Time* and *Percentage Area Overhead*. To generate this graph, the execution times for finding the keys to all obfuscated benchmarks are added together at each obfuscation overhead percentage point. The figure 12 illustrates that the complexity of benchmarks obfuscated by dac12 is considerably higher than that for all other investigated obfuscation schemes. We should also note that the time needed for obfuscating a design using the dac12 methodology is considerably longer than the time required by earlier obfuscation methods. The simulation results confirm that increasing the controllability of internal signals, as done in iolts14, or increasing the output corruption, as implemented in toc13, significantly reduces the strength of obfuscation scheme against SAT attacks. Hence, obfuscation schemes that produce the lowest possible output corruption, or reduce the controllability of internal signals pose a harder problem for SAT solvers.

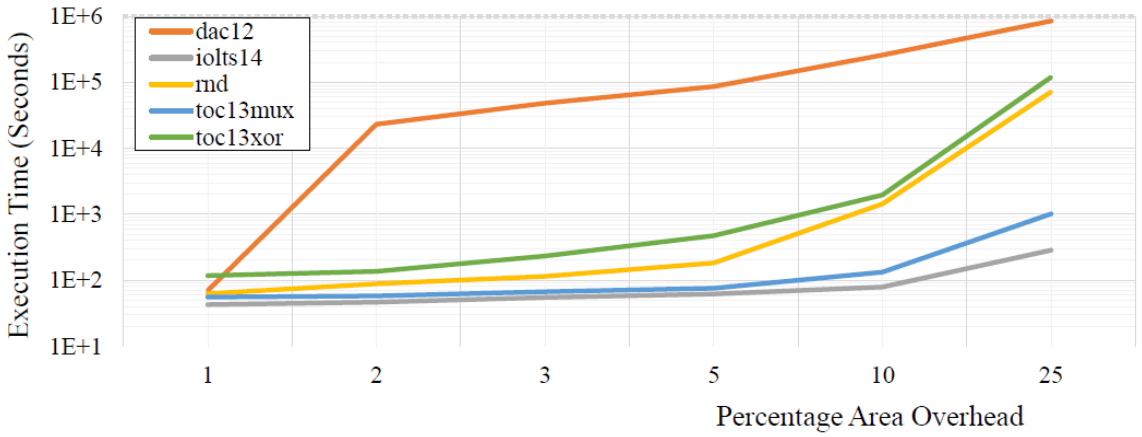


Figure 12: Difficulties of all obfuscation techniques.

Low-to-Mid complexity:

Figure 13 (left) illustrates the ability of SAT solvers in defeating the obfuscation scheme across all low-complexity obfuscation schemes (all obfuscation methods except dac12). As illustrated in this figure, the relationship between execution time and area overhead is exponential. However, note that the execution time grows at a very different pace for different SAT solvers, leading to a significant difference in runtime at higher obfuscation percentages. This is illustrated in figure 13 (right), where the runtime of SAT solvers under study, benchmarked at 25%, is plotted. As shown in this figure, MapleGlucose, although not the best SAT solver at smaller percentages, outperforms all other solvers by a considerable margin for high percentages, to the point that its runtime is about 3x smaller than that of CryptoMiniSat.

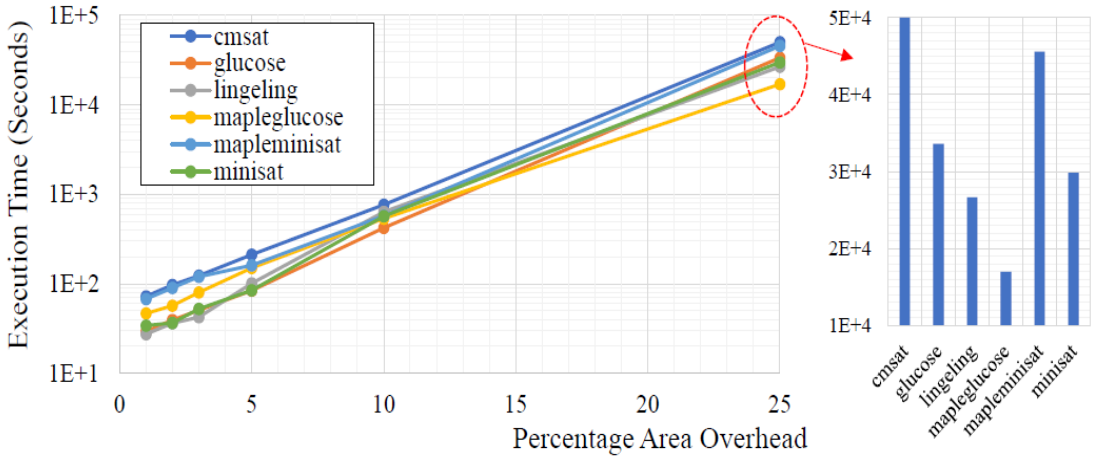


Figure 13: Execution time for finding the keys of low-to-mid complexity problems.

Mid-to-High Complexity:

Figure 14 shows the ability of the SAT Solver to break the obfuscated benchmarks of mid-to-high complexity, the obfuscation technique used is dac12. As the obfuscation complexity increases, the runtime of SAT solvers widely varies. In this experiment, a 24-hour limit was imposed on the SAT solvers to break the obfuscated benchmarks. Maple-Glucose outperformed all other SAT solvers in this experiment.

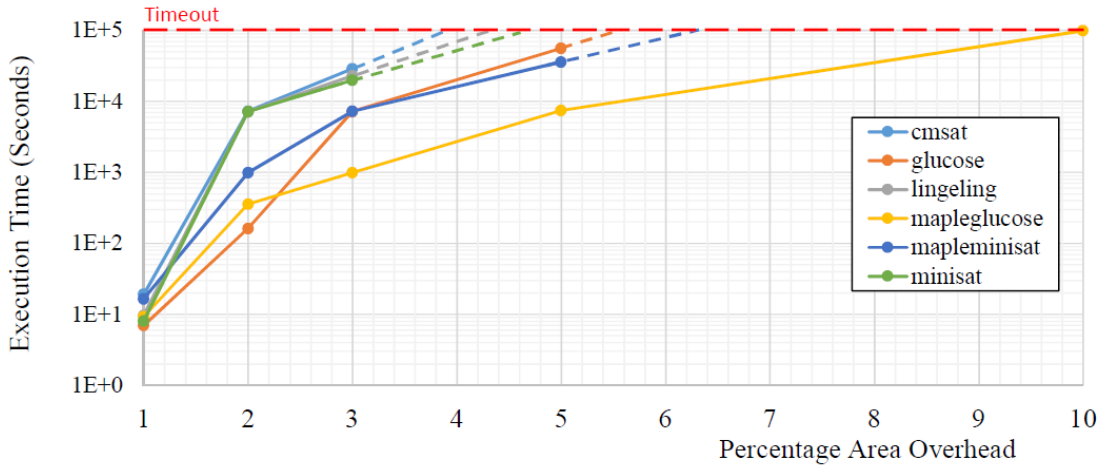


Figure 14: Execution time for finding the keys of mid-to-high complexity problems.

Memory Usage:

The memory usage for each SAT solver across all benchmarks and at each obfuscation area overhead percentage point is shown in figure 15. We observe that lingeling has the lowest memory usage among all the solvers and it is also the fastest solver in the low-to-mid complexity category. Therefore, lingeling can be used for circuits with high memory constraints environment and low overhead obfuscation

techniques. On the other hand, CryptoMinisat is the most demanding solver across all the obfuscation overheads.

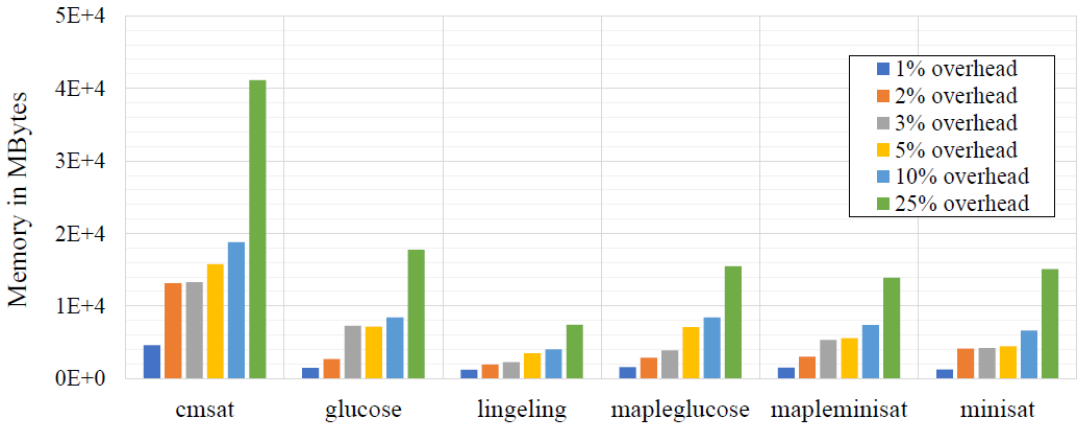


Figure 15: Memory usage of each SAT Solver with increasing overhead in benchmarks.

Exceptions in the trend:

However, there were certain exceptions regarding the analysis of individual benchmarks, which have prevented in generalizing the results of benchmarking. For example, the execution time of all the SAT Solvers for finding keys to benchmarks C2670 and C3540 (being a part of the ISCAS- 85 benchmark suite). The toc13xor obfuscation in circuit C3540 produces a much harder problem for SAT solvers across different obfuscation overheads when compared to dac12, whereas in C2670 the behavior is the opposite. Hence, the netlist characteristic (number of inputs, number of gates, connectivity, topology, number of outputs) plays a significant role in the strength of the

applied obfuscation, suggesting the use of hybrid obfuscation methods to defend various netlists.

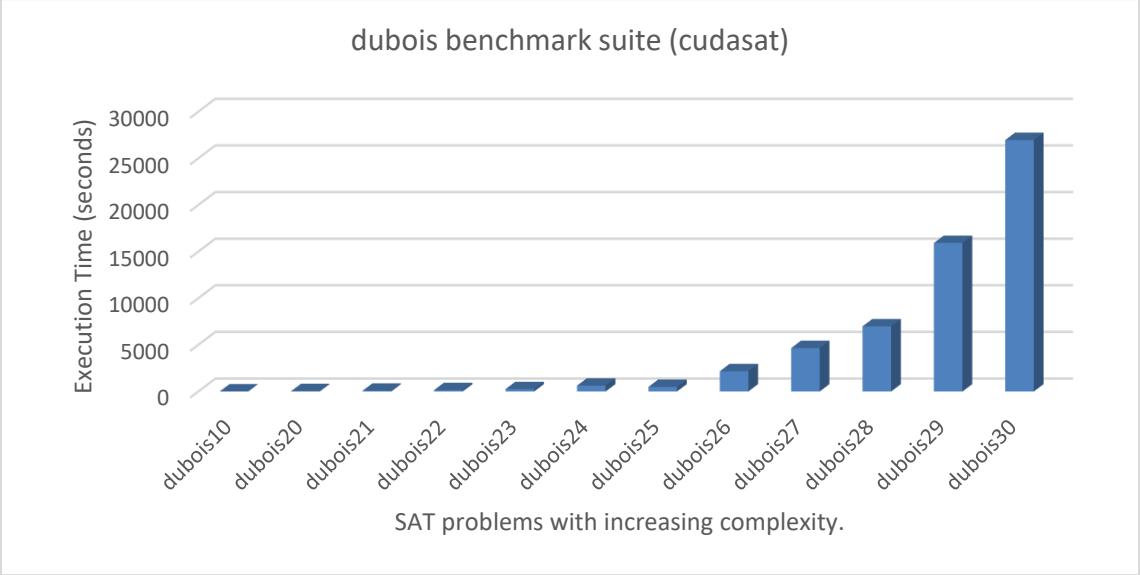
5.3 MinisatGPU vs CUDASAT:

To analyze the performance and limitations of both the solvers, they have been run on several benchmark SAT problems. The MinisatGPU as discussed in the previous chapter implemented the approach of using both the GPU and CPU, i.e., running the Unit Implication/BCP procedure on the GPU and other procedures like VSIDS, non-chronological backtracking etc… on the CPU. However, Cudasat had implemented several modes of operation with a combination of different procedures on both CPU and GPU. In context, mode 5 was selected as it included unit propagation with watched literals on both CPU and GPU, also the GPU is used for searching the lower parts of the search tree.

For running both the implementations, the benchmark suites used were taken from the *satlib* and the SAT competition websites, where 2 benchmark suites of different complexities were selected. For running the solver instances, they were provided to a research computing cluster of George Mason University named Argo [31], consisting of 8 Nvidia's Tesla K80 GPGPUs on a single node. Each of the benchmark suites were assigned to one GPU each and were run for 5 times to average out the differences between run-time performances. The timeout for these runs was given after 50 minutes (3000 seconds).

Cudasat Results:

Figure 16 shows the average run times of different problems in the dubois and the pigeon-hole benchmark suites for the Cudasat implementation, The X-axis shows different SAT problems with differing complexities of the benchmark suites and Y-axis shows the average execution time for each the SAT problems in both benchmark suites.



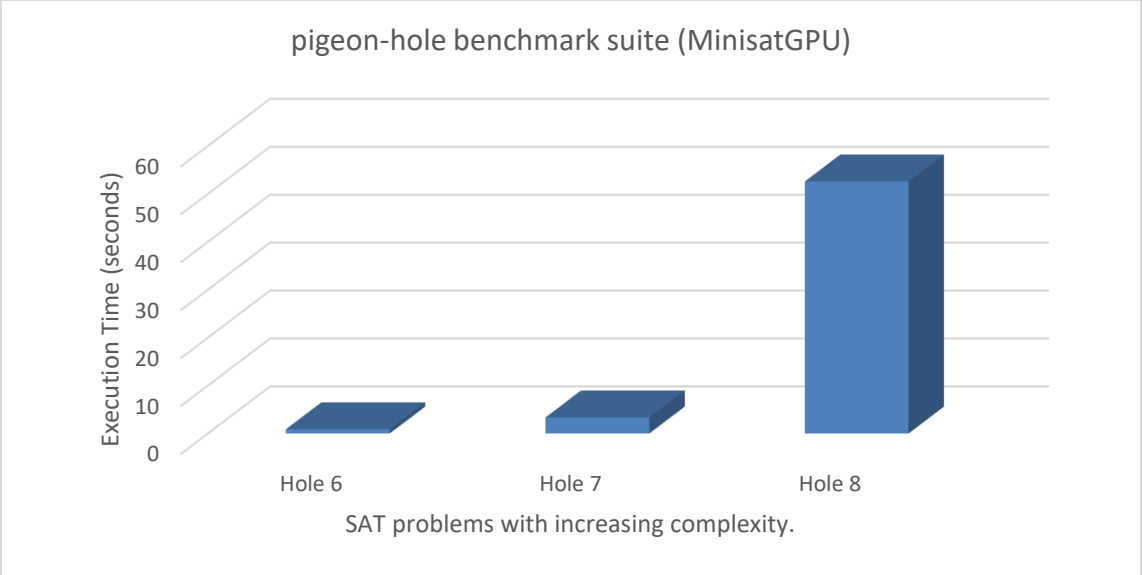
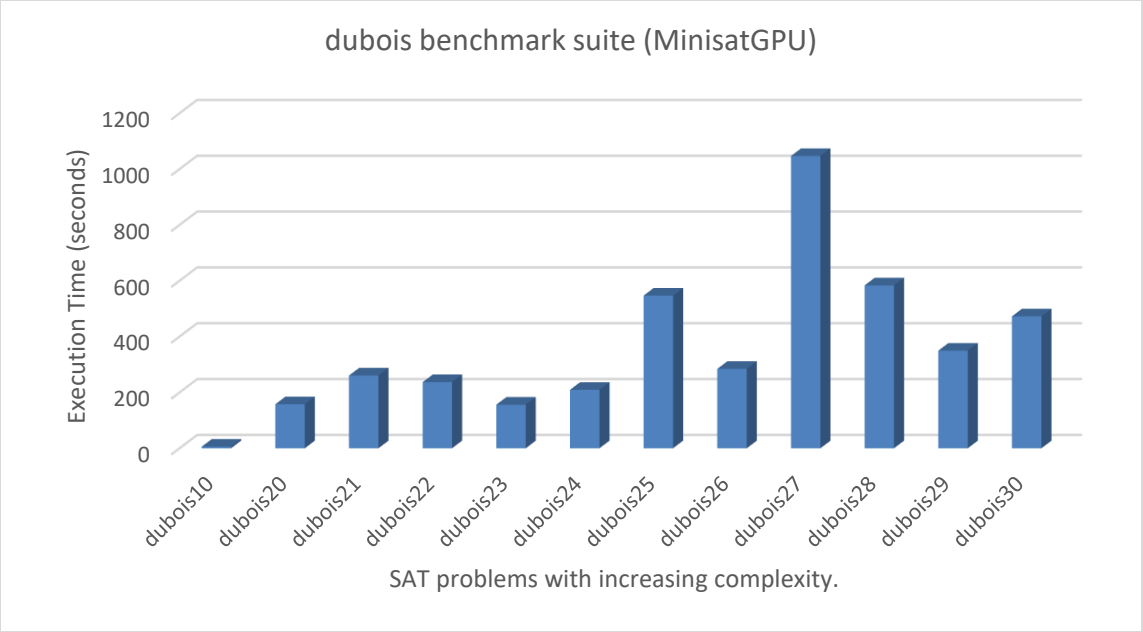


Figure 16: Execution time for cudasat (above) dubois (below) pigeon-hole.

We observe that, as the complexity of the problem increases of both the benchmark suites the execution time increases in respect to it (maybe exponential or linear depending on the type of the problem should be considered).

MinisatGPU Results:

For the MinisatGPU implementation, figure 17 shows the average run times for different problems in the dubois and the pigeon-hole benchmark suites for the MinisatGPU implementation.



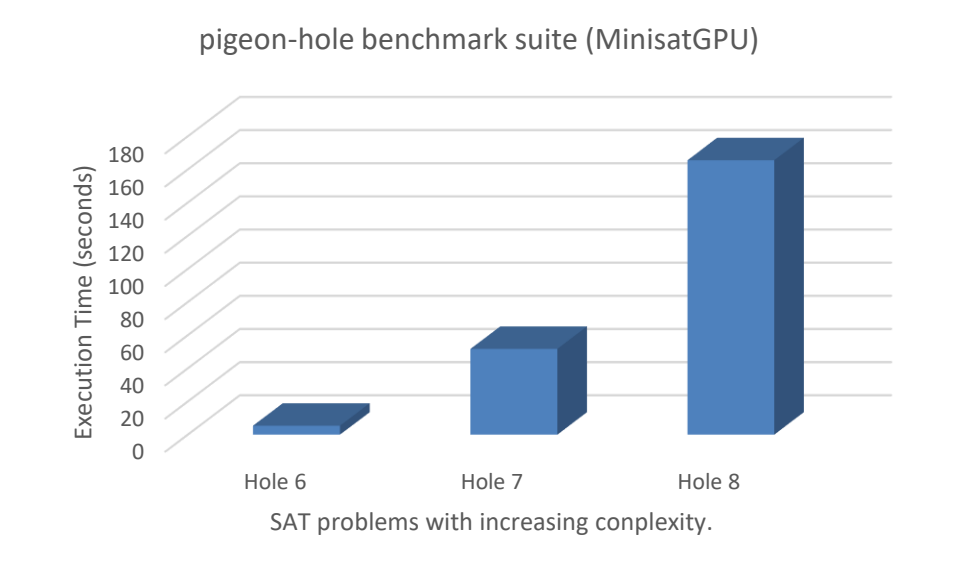
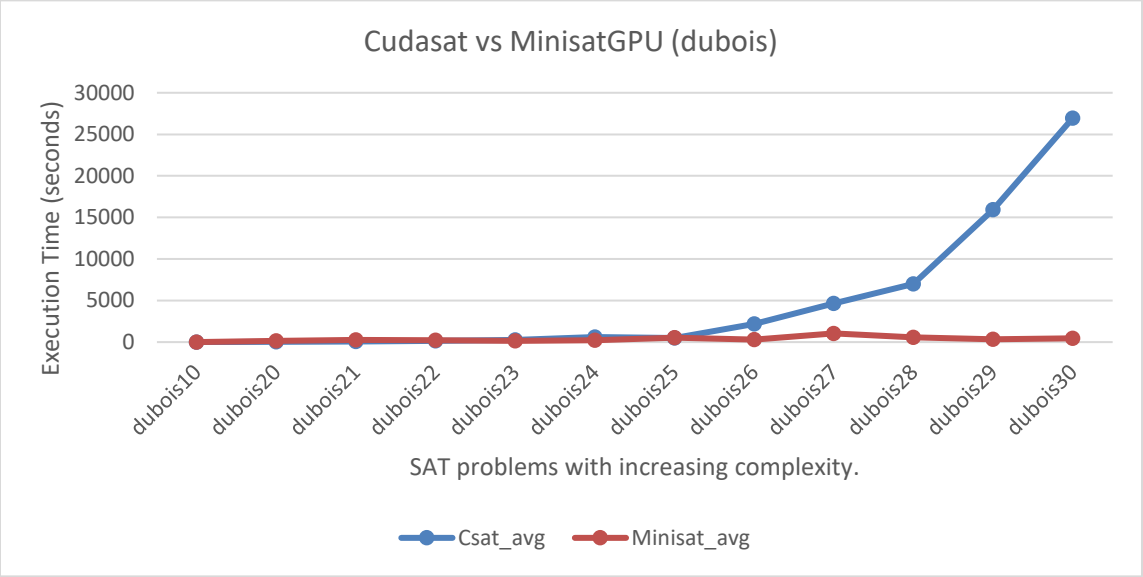


Figure 17: Execution time for MinisatGPU (above) dubois (below) pigeon-hole.

Comparing the results:

For the analysis of the performance of both solvers, their results are compared with each other to find the solver that performs better for these set of benchmark suites.



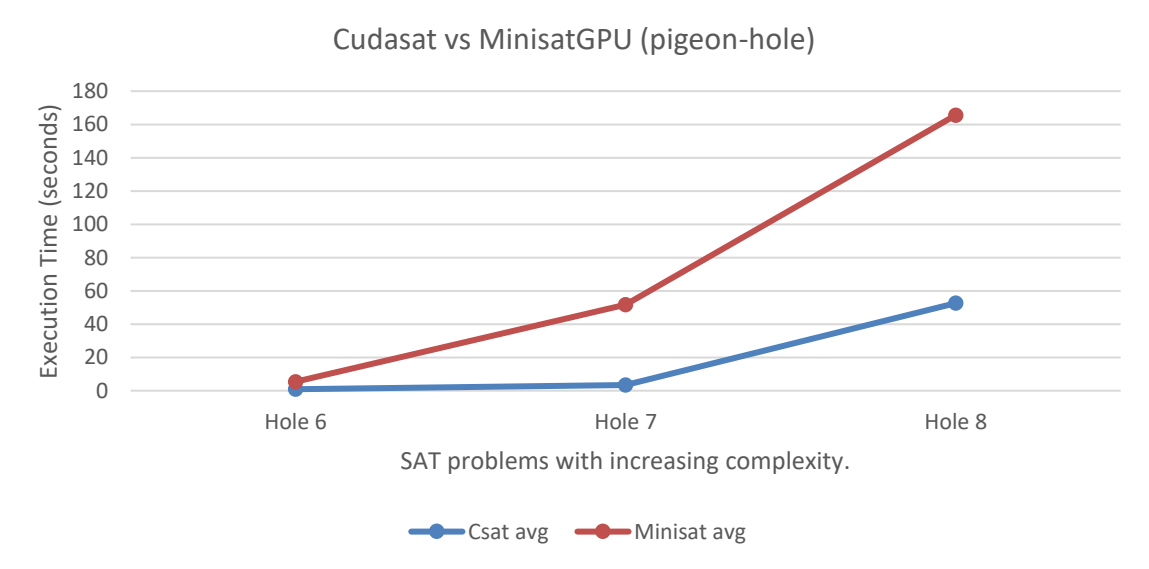


Figure 18: Cudasat vs MinisatGPU (above) dubois and (below) pigeon-hole.

In figure 18 we observe, for the dubois benchmark suite, we observe that for half (6) of the problems in the benchmark suite have a similar execution time for both the solvers and while the complexity of the problem increases for different benchmarks, the execution time of Cudasat increases exponentially and reaches close to the time-out zone.

For the pigeon-hole benchmark suite, also has the pattern of execution time is similar to that of the dubois benchmark suite, however, cudasat has lower execution time of both the solvers. This is because the complexity of the pigeon-hole benchmark suite is increasing in an exponential order. However, this increase in complexity of the problems is the factor of the selected benchmark suites, i.e., every benchmark suite is unique, and they cannot be generalized. In the above case, although the size of the problems of pigeon-hole benchmark suite are similar to that of the dubois benchmark suite, the formers complexity increases in accordance with the number of variables and clauses.

CHAPTER 6: CONCLUSIONS & FUTURE WORK

6.1 Benchmarking Conclusion:

The benchmarking results reveal that the Glucose and Lingeling solvers are best suited for small to midsize k-obfuscation problems, while MapleGlucose provides the best execution time for large k-obfuscation problems. When dealing with extremely large k-obfuscation problems, Lingeling again becomes the best choice due to its efficient and less memory demanding database implementation.

In terms of testing the hardness of k-obfuscation methods, especially for mid-to-hard size problems, we observed that the increase in the k-obfuscation difficulty affects the runtime of each solver quite differently. Hence, although the increase in difficulty could be verified by one SAT solver, a pace of the increase in difficulty is dependent on the choice of a SAT solver and the results from one solver cannot be generalized.

Finally, from a defender's perspective, the results of this benchmarking study suggest that targeting the clause-learning process by means of k-obfuscation, to increase the size of each learned conflict clause, directly affects the effectiveness of SAT solvers in pruning the search space and is a possible promising area for further investigation.

6.2 GPU implementation Conclusion:

The benchmarking results of cudasat and MinisatGPU on different benchmarks revealed their characteristics like complexity and difficulty of the problems. We observe that, every benchmark suite is unique as they are derived from real world applications and simulations. Also, their complexity and difficulty in solving them is in relation with the number of variables and clauses in the problem.

In terms of the execution time of the solvers, we observe that cudasat is capable and fastest for solving benchmarks of smaller size and high complexity. Therefore, cudasat can be used for solving smaller benchmarks. On the other hand, MinisatGPU is faster for low complexity and low-to-mid sized benchmarks. Therefore, MinisatGPU can be used for finding the solutions to the medium and less complex benchmarks.

Comparing these results with purely CPU based solvers, there was no speedup observed when comparing both cudasat and MinisatGPU with the CPU based solvers. This is because, the later implement several data structures, complex interleaving algorithms that are inherently sequential and parallelizing them would involve in rethinking them in the parallel perspective. The data structures implemented are efficient in terms of access latency and storing the clauses based on their sizes. Also, in terms of acceleration, there is a requirement of a translation stage which translates the in-build data structures to GPU domain.

In terms of communication, the CPU and GPU communicate with each other through the PICe interface, though it is the fastest communication interface on the hardware, however, it adds an additional overhead to execution time.

Finally, in conclusion, the GPU architecture being SIMT, for gaining decent acceleration, running applications with low thread divergence and low inter thread communication should be done. For example, matrix multiplication, image processing, neural networks etc… As the threads are executed in warps (batch of 32 threads) on the GPU, the programmer cannot take advantage of the individual CUDA cores which make the GPU's compute core.

6.3 Future Work for GPU implementation:

This investigation of accelerating the SAT Solvers on GPU would not be feasible for larger problems with hundreds of variables and thousands of clauses, which are easily solved by the CPU based solvers. To gain a significant speedup in accelerating the SAT Solvers, the best proposed solution would involve parallelizing the DPLL algorithm for the modern multi-core CPUs. The advantages like, lack of translation of the data structures, stronger CPU cores that can handle high thread divergence would render in running separate solver instances on each of the CPU cores.

In conclusion, the SAT solvers were implemented on the GPUs, however, they are highly suitable for very small problems with low complexity and the path to speed up the solvers would be to take advantage of the multi-core CPU architectures.

REFERENCES

[1] Yasin, Muhammad & Sinanoglu, Ozgur. (2017). Evolution of Logic Locking. 10.1109/VLSI-SoC.2017.8203496.

[2] R. P. Cocchi, J. P. Baukus, L. W. Chow, and B. J. Wang, “Circuit Camouflage Integration for Hardware IP Protection,” in 2014 51st IEEE Design Automation Conf. (DAC), June 2014, pp. 1–5.

[3] M. Rostami, F. Koushanfar and R. Karri, "A Primer on Hardware Security: Models, Methods, and Metrics," in Proceedings of the IEEE, vol. 102, no. 8, pp. 1283-1295, Aug. 2014. doi: 10.1109/JPROC.2014.2335155

[4] Shervin Roshanisefat, Harshith K. Thirumala, Kris Gaj, Houman Homayoun, Avesta Sasan, “Benchmarking the Capabilities and Limitations of SAT Solvers in Defeating Obfuscation Schemes.” Online: https://arxiv.org/pdf/1805.00054.pdf

[5] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, “Security Analysis of Logic Obfuscation,” in Proceedings of the 49th Annual Design Automation Conf. ACM, 2012, pp. 83–89.

[6] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, “Fault Analysis-Based Logic Encryption,” IEEE Trans. on Computers, vol. 64, no. 2, pp. 410–424, Feb 2015.

[7] J. A. Roy, F. Koushanfar and I. L. Markov, "EPIC: Ending Piracy of Integrated Circuits," 2008 Design, Automation and Test in Europe, Munich, 2008, pp. 1069-1074. doi: 10.1109/DATE.2008.4484823.

[8] Shenoy, Gaurav. (2016). Implementation and Evaluation of SAT-based Attacks on Hybrid STT-CMOS Circuits for Reverse Engineering. 10.13140/RG.2.1.1008.9207.

[9] J. Zhang, “A Practical Logic Obfuscation Technique for Hardware Security,” IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 3, pp. 1193–1197, March 2016.

[10] Shervin Roshanisefat, Hadi Mardani Kamili, Avesta Sasan, “SRCLock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware”. Online: https://arxiv.org/pdf/1804.09162.pdf

[11] El Massad, Mohamed & Garg, Siddharth & Tripunitara, Mahesh. (2015). Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes. 10.14722/ndss.2015.23218.

[12] P. Subramanyan, S. Ray, and S. Malik, “Evaluating the Security of Logic Encryption Algorithms,” in 2015 IEEE Int. Symp. on Hardware Oriented Security and Trust (HOST), May 2015, pp. 137–143.

[13] J. Davis, N. Kulkarni, J. Yang, A. Dengi, and S. Vrudhula, “Digital IP Protection Using Threshold Voltage Control,” in 2016 17th Int. Symposium on Quality Electronic Design (ISQED), March 2016, pp. 344–349.

[14] M. El Massad, S. Garg, and M. V. Tripunitara, “Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes,” in NDSS, 2015.

[15] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu, “SARLock: SAT Attack Resistant Logic Locking,” in 2016 IEEE Int. Symp. on Hardware Oriented Security and Trust (HOST), May 2016, pp. 236– 241.

[16] Y. Xie and A. Srivastava, “Mitigating SAT Attack on Logic Locking,” in Int. Conf. on Cryptographic Hardware and Embedded Systems. Springer, 2016, pp. 127–146.

[17] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NPCompleteness. W. H. Freeman & Co., New York, NY, USA (1979).

[18] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7) (July 1962) 394–397

[19] Marques-Silva, J., Sakallah, K.: GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5) (1999) 506–521

[20] N. E´en and N. S¨orensson, “An Extensible SAT-solver,” in Int. Conf. on theory and applications of satisfiability testing, 2003, pp. 502–518.

[21] G. Audemard and L. Simon, “Glucose and Syrup in the SAT Race 2015,” SAT Race, 2015.

[22] A. Biere, “Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013,” Proceedings of SAT Competition, vol. 2013, 2013.

[23] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, “Learning Rate Based Branching Heuristic for SAT Solvers,” in Int. Conf. on Theory and Applications of Satisfiability Testing. sSpringer, 2016, pp. 123–140.

[24] M. Soos, K. Nohl, and C. Castelluccia, “CryptoMiniSat,” SAT Race solver descriptions, 2010.

[25] S. Dupuis, P. S. Ba, G. D. Natale, M. L. Flottes, and B. Rouzeyre, “A Novel Hardware Logic Encryption Technique for Thwarting Illegal Overproduction and Hardware Trojans,” in 2014 IEEE 20th Int. On-Line Testing Symposium (IOLTS), July 2014, pp. 49–54.

[26] Jason Sanders, Edward Kandrot, (2011) “CUDA by Example”, Publisher: Pearson Education.

[27] Domenic Forte, Swarup Bhunia, Mark M. Tehranipoor (Editors) “Hardware Protection through Obfuscation”, Publisher: Springer.

[28] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano & Enrico Pontelli (2015) CUD@SAT: SAT solving on GPUs, Journal of Experimental & Theoretical Artificial Intelligence, 27:3, 293-316, DOI: 10.1080/0952813X.2014.954274

[29] Carlos Filipe Costa (May 2014), “Parallelization of SAT algorithms in GPUs” Online:https://pdfs.semanticscholar.org/71cc/b79bc57aa687865498486643d006805c28cd.pdf

[30] Nvidia Tesla K80 GPU Accelerator and Kepler Architectures (whitepaper), Online: https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf https://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[31] Information page of GMU ARGO cluster, http://wiki.orc.gmu.edu/index.php/About_ARGO

BIOGRAPHY

Harshith Kumar Thirumala is a Master of Science student at the Electrical & Computer Engineering Dept. at the Volgenau School of Engineering, George Mason University. He received his bachelor's degree in Electronics & Communication Engineering, Jawaharlal Nehru Technological University, Hyderabad, India. He was previously an intern at Defense Electronics Research Laboratory (DLRL), Telangana, India. He has been a member of Dr. Avesta Sasan's Green, Accelerated & Trustworthy Engineering (GATE) laboratory for his master's thesis since June 2016. He specializes in "Low Power & Secure VLSI design". His research interests are Low Power design, Embedded systems, Computer Architecture and Hardware-level functional safety & security.