GPU Obfuscation: Attack and Defense Strategies

Abhishek Chakraborty
University of Maryland, College Park
abhi1990@umd.edu

Yang Xie
University of Maryland, College Park
yangxie@umd.edu

Ankur Srivastava
University of Maryland, College Park
ankurs@umd.edu

ABSTRACT

Conventional attacks against existing logic obfuscation techniques rely on the presence of an *activated hardware* for analysis. In reality, obtaining such activated chips may not always be practical, especially if the on-chip test structures are disabled. In this paper, we develop an iterative SAT formulation based attack strategy for deobfuscating many-core GPU hardware *without any requirement of an activated chip*. Our experiments on a real testbed using NVIDIA's SASSIFI framework reveal that more than 95% of the application runs on such an *approximately unlocked* GPU result in correct outcomes with 95% confidence-level and 5% confidence-interval. To counter the proposed attack, we develop a Cache Locking countermeasure which significantly degrades the performance of GPGPU applications for a wrong *cache-key*.

1 INTRODUCTION

The increasing trend of outsourcing hardware designs to offshore foundries for fabrication cost reduction has raised several security concerns related to Intellectual Property (IP) piracy, reverse engineering, counterfeiting, etc. [9]. The exposure of chip designs to a potentially *malicious* offshore foundry is of major concern for organizations and hence, there has been an extensive research on security and privacy of IC supply chain in recent past [19]. Several *logic locking techniques* have been proposed to thwart supply chain attacks against designs in untrusted foundries [13–15, 20]. In any standard combinational logic obfuscation approach, the design is locked by inserting additional *key-gates* in combinational blocks of the circuit. The locked circuit exhibits correct functionality only when the *correct key* is loaded into the on-chip *tamper-proof* memory to *activate* the chip after fabrication. The security of logic locking schemes are based on the assumption that the *correct key* cannot be learned by the untrusted foundry within a practical time limit [4, 6, 12–15, 20, 21]. However, in recent literature, various key-learning attacks have been proposed [12, 13, 16, 18] which expose the weaknesses of the existing logic locking schemes.

Till date, the security analyses of logic locking schemes have been confined to only a set of *small-scale* benchmark netlists. These analyses do not necessarily imply that the security standards of the overall hardware implementation is compromised, even if the attacker successfully unlocks certain components of the design. To the best of our knowledge, there has been no study which analyzes the attack vulnerability of any *large-scale processor netlist* locked using standard logic obfuscation techniques. In this paper, we outline the utilization of a SAT formulation based attack methodology [16, 18] against an obfuscated many-core Graphics Processing Unit (GPU) netlist to find an *approx-key* to *approximately* deobfuscate the processor cores. Unlike conventional attacks on logic locking schemes, an adversary can launch our proposed attack **without any activated hardware** and hence, such an attack poses a *major threat* in the supply chain of GPU designs. Subsequently, we evaluate the impact of errors (due to use of the learned *approx-key*) on the outcomes of benchmark applications running on a real GPU.

Modern GPU architectures have been developed to efficiently exploit the data-level parallelism in applications ranging from real-time 3D visualization to high-performance scientific computing. In this work, though we focus on widely used NVIDIA GPUs [2] to outline our attack and defense strategies, the analyses are equally applicable to other GPU architectures as well. In NVIDIA GPUs, hundreds to thousands of Streaming Processors (SPs) or CUDA cores process data in parallel and are the primary components responsible for the superior performance of GPUs in terms of speed and power. In this work, we used the state-of-the-art Anti-SAT based logic locking scheme [20] integrated with Strong Logic Encryption [14] to obfuscate the functionality of the GPU cores. Conventional SAT attack approach [18] requires full scan-chain access to internal registers of an activated chip for analysis, which in practice is an unrealistic assumption. We outline a technique to mount a modified SAT formulation based attack against a GPU core's locked pipelined netlist to *approximately* deobfuscate its functionality without any activated chip requirements. Subsequently, we utilized NVIDIA's architecture-level fault injection tool called SASSI-based Fault Injector (SASSIFI) [8] to analyze the impact of error propagation to *application-level* due to use of the retrieved *approx-key* to unlock the underlying core netlists. The outcomes of our experiments reveal that the benchmark General-purpose GPU (GPGPU) applications are highly resilient to such error propagation effect [7] as the *approx-key* correctly unlocks the functionalities of the CUDA cores for almost all the inputs used in computations. On an average, for all the applications studied, the use of an *approx-key* results in correct outcomes for more than 95% of the application runs with 95% confidence-level and 5% confidence-interval. Therefore, our proposed attack strategy can be utilized by an untrusted foundry to deobfuscate the cores, even *without* an activated GPU.

In this paper, we also outline an architecture-level countermeasure which obfuscates the cache behavior to degrade the performance of applications running on an approximately unlocked GPU netlist. The crux of SAT formulation based attack against any logic encryption scheme is based on exploiting a design's *faulty input-output response pairs* (accessible to an adversary) to iteratively remove subsets of wrong keys. To safeguard against such an attack, we propose a Cache Locking countermeasure which locks the cache block replacement policy in a GPU for wrong *cache-key*. This results in significant performance degradation of applications as evident from experimental results, making the GPU *inefficient* for fast application execution.

2 BACKGROUND

2.1 Existing attacks on logic locking

The security objective of logic locking techniques is to increase the output *corruptibility* given an incorrect key, hence countering key-learning attacks. Given a set of correct I/O patterns observed from the activated chip, key-searching based attacks intend to find a key that can satisfy the correct I/O patterns [10, 12, 13]. Based on circuit

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196058

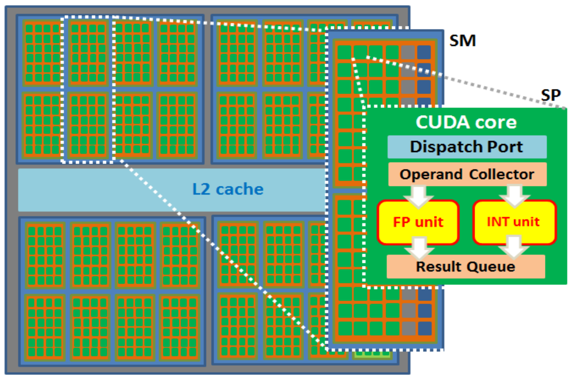

Figure 1: Block diagram of an NVIDIA GPU architecture

SAT formulation, Subramanyan *et al.* [18] proposed a satisfiability-checking (SAT) based attack that iteratively finds specific correct I/O patterns which can be used to eliminate a subset of wrong keys until none exists. The SAT attack effectively breaks most logic locking techniques proposed [4, 13–15] within a few hours even for a reasonably large key-size. Countermeasures to the SAT attack have been proposed which inserts a SAT-attack resilient logic block such as Anti-SAT [20] and SARLock [21] in the netlist. The number of SAT attack iterations to retrieve the secret key of a netlist consisting of an Anti-SAT block is an exponential function in terms of the key size, thereby making SAT attack infeasible in practice. Meanwhile, a new attack called AppSAT [16] has been proposed which determines an *approx-key* to *approximately* unlock a netlist.

2.2 Overview of GPU architecture

In a NVIDIA GPU, blocks of threads are executed in Streaming Multiprocessors (SMs), which primarily consists of groups of streaming processors (SPs) or CUDA cores. The unit of execution flow in the SM is a collection of 32 threads, called *warp*. The threads in a warp follow the Single Instruction Multiple Thread (SIMT) mode, i.e., they execute the same instruction sequence but with different data. SPs are the primary computing elements of GPUs and corresponds to cores that perform scalar calculations. An SM, in addition to SPs, consists of other different types of functional modules such as load/store (LD/ST) units, Special Functional Unit (SFU), on-chip memory (instruction cache, configurable shared memory/ L1 cache, register files) and instruction control units (dispatcher, scheduler). In figure 1, we present a structural overview of an NVIDIA GPU architecture. In this work, we propose an attack against a locked GPU netlist to retrieve an *approx-key* and study the application-level impact of error propagation (due to the use of *approx-key*) utilizing NVIDIA's SASSIFI framework [8] on a real GPU.

2.3 Instrumentation of GPGPU applications

In this paper, we focus on GPGPU applications based on the widely adopted NVIDIA's Compute Unified Device Architecture (CUDA) framework [1]. The CUDA programming framework adopts the SIMT model in hierarchies consisting of kernels, blocks, and threads. The CPU spawns the multithreaded kernels onto the GPU, which subsequently allocates the blocks of threads to available SMs using internal schedulers. The parallel programs written in high-level language such as CUDA is compiled by a front-end compiler (NVIDIA's NVVM) to generate intermediate code in a virtual ISA called parallel thread execution (PTX). PTX abstracts the GPU as a data-parallel computing platform, but the PTX code does not run directly on the GPU. Another backend compiler optimizes and translates the PTX instruction in native machine code by either using ahead-of-time compilation of compute kernels via PTX assembler (ptxas) or using just-in-time compiler in the display driver to compile PTX representation of kernel available in binary format. In this work, we used NVIDIA's SASSIFI framework [8] to study the application-level error impact in a real GPU due the use of learned *approx-key* to unlock its core functionalities. The SASSI-based Fault Injector (SASSIFI) framework utilizes ahead-of-time backend compilation as the SASSI instrumentation is embedded in ptxas. SASSI is implemented as the final compiler pass in ptxas and uses *nvlink* to link instrumented applications with instrumentation handlers. The SASSI based application instrumentation requires two things to be specified: (i) *where* to insert instrumentation and (ii) *what* information to extract from each instrumentation site.

3 PROPOSED ATTACK ON OBFUSCATED GPU

3.1 Obfuscation of GPU cores

In the inset of figure 1, we outline the structure of an NVIDIA GPU architecture's SP module which primarily consists of *inorder* integer and floating point pipelines. The SPs or CUDA cores are the most abundant computational elements in a GPU and are primarily responsible for its high throughput performance. Hence, as a natural choice, we assume that the designer inserts key-gates in the gate-level netlist between various SP pipeline stages to lock the overall functionality of the GPU, following the steps of any standard combinational logic locking approach [13, 15, 20, 21]. We also consider that all the SP modules in the GPU are locked using a *single key* so that the layouts of the cores are identical, thus having optimal fabrication cost. Moreover, the use of separate keys for different cores will lead to an impractical key size as the number of cores in modern GPUs can be very large (for example NVIDIA's GeForce GTX Titan Z consists of 5760 CUDA cores).

In a logic obfuscated SP pipelined netlist, wrong inputs to key-gates will result in errors in outcomes of threads which utilize such *faulty* key-gates for their computations. Depending on locations of *faulty* key-gates and data, such errors will have varying impacts on multithreaded kernel executions in SIMT mode as follows:

(i) Datapath error: A *wrong* key bit input to a key-gate located in the datapath of pipeline will have an error propagation effect only in the fan-out cone of the *faulty* key gate. In other words, such a fault will have *thread localized* effects in computations, i.e., impacting only the threads which execute on that erroneous datapath.

(ii) Controlpath error: In SIMT mode, the decoder module of a SM decodes the opcode for all the active threads in a warp and individual threads execute the *same* decoded operations on different SPs or cores but with *different* data operands. Hence, a *wrong* key bit input to a key-gate located in the decoder module or controlpath will have an error propagation effect in the datapaths of *all the active threads* in a warp. Hence, controlpath errors will have *warp wide* effects in computations.

We consider a simple multithreaded *sum* application to study the effect of datapath and controlpath errors on an actual application-level output using NVIDIA's SASSIFI framework (more details in section 5.2). Based on the *key* inputs chosen, we will have different

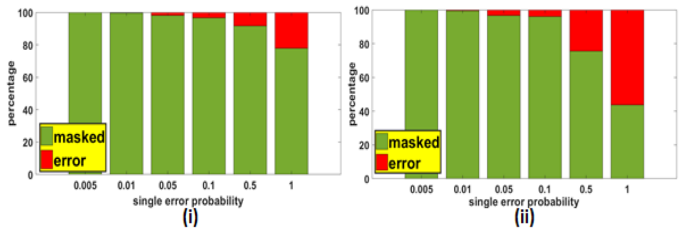

Figure 2: Application outcome vs. probability of single inst. being faulty for (i) datapath & (ii) controlpath errors in core

probabilities of errors occurring in instructions. In figure 2, we illustrate the percentage of the application outcomes (out of several runs) being faulty (red) or correct/masked (green) due to a single *randomly selected* instruction being executed faulty (with different probabilities) for datapath and controlpath errors. From the plots it is evident that even error in a single thread due to a *wrong* key input may lead to faulty application outcomes. However, we observed that the *difference* in the number of faulty outcomes for datapath and controlpath errors is significant when the probability of an instruction being faulty is high, where as the *difference* becomes quite negligible with a decrease in the probability. We observe this effect on the experimental results for benchmark applications also as detailed in section 5.2. Hence, a smartly selected *approx-key* which injects very small error in instructions can indeed result in very accurate application outcomes despite not unlocking the hardware in its entirety.

As NVIDIA GPU's SP pipeline architecture/netlist details are proprietary, we instead consider a locked netlist of standard MIPS 5-stage inorder pipeline as substitute of the GPU netlist to perform our experimental analyses. We obfuscated the the control and data paths of MIPS pipeline using state-of-the-art Anti-SAT based logic locking scheme [20] integrated with Strong Logic Encryption [14].

3.2 Attack on locked GPU cores

3.2.1 Attack Model. In *conventional* SAT attack [18], in addition to the netlist, full scan-chain access to an activated hardware is also required to deobfuscate a locked hardware. This is because the formulation of an iterative SAT attack utilizes the input-output truth tables of each of the locked modules. However, this is a *very strong* assumption as such privilege is *not available* in practice. First, when the untrusted foundry is trying to unlock a chip, the actual activated chip may not have been marketed yet. Second, even if they are in possession of the unlocked chip, the attacker needs to have full scan chain access into the internal combination modules. The designer who wishes to secure his design may just disable on-chip test structures before marketing the unlocked chip. **In our attack model, we allow the adversary to only possess a locked netlist, and do not grant her privileges to have full scan chain access to internal pipeline latches of SP modules in an activated chip.**

3.2.2 SAT formulation based attack. The primary challenge to deobfuscate the functionality of a locked SP netlist using *conventional* SAT attack approach [18] is the lack of knowledge of internal pipeline latch contents as per our attack model. In this section, we demonstrate how an adversary *can still successfully devise an iterative SAT formulation based attack* to effectively learn the *key* without an activated GPU hardware. The crux of the ensuing attack strategy is the observation that the internal pipeline latches are *only responsible for performance speed-up* by dividing the long latency single-cycle datapath into low latency multi-cycle pipelined datapath. These pipeline registers play *no role* in determining the overall functionality of the pipelined netlist. Hence, for the sake of analyzing the functionality of the locked SP module, the adversary can model an equivalent netlist by transforming the multi-cycle pipelined datapath to a single-cycle datapath design. This equivalent netlist can be constructed easily by logically removing the pipeline latches and then simply connecting the input wires to corresponding output wires of the removed latches as shown in figure 3. The outcome of this transformation is the conversion of a locked sequential SP netlist to a *functionally equivalent locked combinational SP netlist*, which we analyze next using an iterative SAT formulation. Before we outline the details of our attack methodology, we define the following terminologies for convenience:

(i) f_{lock}**:** Functionally equivalent locked combinational SP netlist
(ii) *PI***:** Primary input to the locked SP netlist, consisting of opcode contents, source and destination register addresses, etc. as obtained from instruction binary (details later)
(iii) *PO***:** Primary output of the locked SP netlist, consisting of destination register contents (for R-type or I-type MIPS instructions) or jump/branch address (for J-type MIPS instructions). From the locked netlist, the functional relationship among *PI*, key-gate inputs (K) and *PO*, i.e., $PO = f_{lock}(PI, K)$ is known to the attacker.

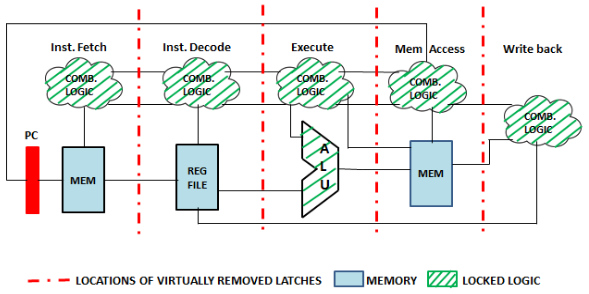

Figure 3: Multi-cycle to Single cycle datapath transformation of the locked pipelined netlist

The primary difference between *conventional* IC obfuscation and the obfuscation of a GPU core netlist is that the *correct* $PI - PO$ pairs are not known in the former case *without* an activated chip, whereas in the later case, the attacker can deduce the *correct* $PI - PO$ pairs for a SP netlist as explained below:

- The *PI* corresponding to each instruction is obtained by relating the human readable assembly instructions to binary information of the assembled application. For example, in case of NVIDIA GPUs, it is possible to successfully extract the PTX or SASS from a *cubin* or *executable* using the *cuobjdump tool* in CUDA Toolkit [1]. In addition, the attacker can utilize the NVIDIA's Nsight Visual Studio Edition to correlate between lines of CUDA C, PTX, and SASS [3]. Therefore, using the publicly available instruction set architecture (ISA), the adversary can determine *PI* for every instruction.
- Again, *PO* for each instruction is obtained from the corresponding *PI* and the ISA information because the *PO* depends on the result of operation (*known* from ISA) carried out on the source register contents (*known* from *PI*).

Therefore, the adversary has *prior knowledge* of *correct* $PI - PO$ pairs for every instruction of a compiled application being executed on a GPU core. For example, let us consider a simple assembly-level program fragment executed by a thread on a SP:

$$\begin{array}{ll} & \cdots \\ \text{I1: ADD } R1, R2, R3 & //R1{=}R2 + R3 \\ \text{I2: ADD } R4, R1, R3 & //R4{=}R1 + R3 \\ \text{I3: MUL } R5, R2, R4 & //R5{=}R2 * R4 \\ \text{I4: SUB } R3, R5, R4 & //R3{=}R5 - R4 \\ & \cdots \end{array}$$

Let us suppose that the initial contents (prior to instruction I1 execution) of registers $R1$,$R2$,$R3$,$R4$, and $R5$ are $1, 2, 3, 4$ and 5 respectively. In instruction I1, the contents of registers $R2$ and $R3$ are added and written to register $R1$. Hence, using the *PI* information the adversary can easily calculate the expected value at the destination register, i.e., PO:$[R1]$=$[R2]$+$[R3]$=2+3=5. Now that the *correct* $PI - PO$ pair is known for each instruction, it is equivalent to having in possession an unlocked chip. Hence, SAT formulation based attack strategies [16, 18] can be utilized which use this information to iteratively identify distinguishing input-output (*DI*) pairs. As noted in [18], each *DI* pair eliminates a subset of unique *wrong keys* for that SAT iteration, till we converge to the *correct key*. We can write an iterative SAT formulation for locked SP netlist as follows:

$$\mathbf{F_i} := \mathrm{C}(\mathbf{PI}, \mathbf{K_1}, \mathbf{PO_1}) \wedge \mathrm{C}(\mathbf{PI}, \mathbf{K_2}, \mathbf{PO_2}) \wedge (\mathbf{PO_1} \neq \mathbf{PO_2})$$
$$\Big(\bigwedge_{j=1}^{j=i-1} \mathrm{C}(\mathbf{PI^d_j}, \mathbf{K_1}, \mathbf{PO^d_j})\Big) \wedge \Big(\bigwedge_{j=1}^{j=i-1} \mathrm{C}(\mathbf{PI^d_j}, \mathbf{K_2}, \mathbf{PO^d_j})\Big) \tag{1}$$

where, $\mathbf{F_i}$ denotes the i^{th} SAT iteration formulation, $\mathbf{C(PI, K, PO)}$ is the SAT formula for a locked circuit and $(\mathbf{PI^d_{\{1...i-1\}}}, \mathbf{PO^d_{\{1...i-1\}}})$ are the distinguishing input-output pairs that are found in previous $i-1$ iterations. Following such an attack strategy, if the adversary finds the *correct key* used to lock the original synthesized SP netlist, then all such *PO* responses for different instructions will be consistent with corresponding *correct PO* responses. In context of the aforementioned program fragment, the *correct key* will result in the contents of registers $R1, R2, R3, R4, R5$ being updated with values $5, 2, 8, 8, 16$ respectively just after the execution of instruction I4. To make the process of finding new distinguishing input-output pair *more efficient*, the adversary may develop customized microbenchmark applications consisting of a targeted set of operations carried out by the instructions.

To counter the feasibility of such a SAT attack, *point-function* based obfuscation approaches like Anti-SAT [20] and SARLock [21] have been proposed. Though the *point-function* based obfuscation scheme makes the SAT solving time exponential to obtain the correct key, a recent technique called the AppSAT attack [16] can retrieve an *approx key* to unlock the functionality of such a locked netlist for almost all the primary inputs. In section 5.1, we present the results of the AppSAT attack against an Anti-SAT block based obfuscated netlist of MIPS pipelined design, which we considered as a functional substitute of the GPU core's netlist.

4 CACHE LOCKING COUNTERMEASURE

The crux of the our proposed attack against the locked SP netlist is the iterative elimination of *wrong keys* based on evaluation of new *DI* pairs which satisfy the SAT formulation. The primary motivation behind our proposed Cache Locking countermeasure is that a *wrong cache-key* will result in slowdown of the GPU hardware even though it exhibits correct functionality, and thus being resistant to SAT formulation based attacks (including AppSAT [16]). The cache block replacement *protocols* of modern many-core GPUs are proprietary and hence, not known to an untrusted foundry. For example, several research related to performance analysis of NVIDIA GPUs has led to the conclusion that the cache block replacement protocols is neither of the standard ones commonly studied [11]. Therefore, we analyzed the effect of obfuscating *cache block replacement policy* to lock the overall performance of the GPU for wrong *cache-key* guesses. It is to be noted that locking the cache block replacement policy does not alter the expected (PI, PO) pairs corresponding to the SP units as the overall GPU still performs the correct functionality with an *approx-key* for the cores (retrieved using our proposed attack), but the overall application performance will suffer *significantly* due to drops in cache hit rates with a wrong *cache-key*. This is due to the fact that a wrong *cache-key* will lead to higher number of data fetch requests from slow off-chip memory which require a *significantly* large number of additional clock cycles [11].

In this work, to demonstrate our countermeasure, we assume that the cache replacement policy in place is *least recently used* (LRU). However, the proposed Cache Locking scheme can be applied to any other replacement policies as well and will continue to be immune to SAT type attacks. In order to lock the cache block replacement policy, we considered a hardware implementation of standard clock algorithm [17] which approximates LRU policy by augmenting an extra clock bit to a cache block to keep track of whether or not a block was accessed recently. If the i^{th} cache block was recently accessed the corresponding clock bit ($clock\_bit(i)$) is set to 1, whereas, on the other hand $clock\_bit(i)$ is reset to 0 if the block wasn't accessed recently. The cache blocks are assumed to be arranged as a circular queue with a current pointer or "clock hand" which cycles through this queue on every memory access. If the clock hand is currently pointing to $clock\_bit(i) = 1$, then as it moves to the next cache block the $clock\_bit(i)$ is reset to 0. The status of clock bit of i^{th} cache block is updated in a periodic manner as follows:

- On a **cache hit**, the $clock\_bit(i)$ is set to 1.
- On a **cache miss**, the clock hand moves to next available i^{th} block with $clock\_bit(i)$ set to 0 and replaces it by a data block fetched from lower memory hierarchy, followed by setting of $clock\_bit(i)$ to 1 to designate the recently written cache block.

To implement the Cache Locking scheme, we modified the aforementioned standard clock algorithm to a ***cache-key* dependent** block replacement policy. As per the modification, the i^{th} cache block will have an associated clock bit ($clock\_bit(i, K_i)$, $K_i \in \{0, 1\}$), which is set to K_i if it was recently accessed, whereas it is reset to $\overline{K_i}$ if the block wasn't accessed recently. Now if the input key matches the correct key then this approach is basically equivalent to the aforementioned LRU policy. However, if the input key bit for i^{th} cache block mismatches it's corresponding actual key bit, we invert this policy of setting and resetting $clock\_bit(i, K_i)$. This would end up scrambling the designation of the least recently accessed status for those cache blocks with wrong key-bit inputs. For every wrong key-bit (K_i) guess there will be either of the two *faulty scenarios* for the i^{th} cache block: **(i)** instead of the i^{th} cache block, some other j^{th} cache block will be replaced from the cache whose associated clock bit $clock\_bit(j, K_j)$ is set to $\overline{K_j}$ **(ii)** instead of replacing some other cache block with $clock\_bit(j, K_j)$ is set to $\overline{K_j}$, the i^{th} cache block is replaced. Therefore, this will result in drops of cache hit rates as such *faulty* cache block replacements will not be suitable for applications utilizing the cache locality principles. To have a practically reasonably *key-size*, the designer can also associate a single key-bit to multiple cache blocks.

This entire Cache Locking scheme is simple enough to be implemented in a lookup table (LUT) which can be configured after fabrication at test time by the designer. Hence, the attacker cannot simply remove the proposed countermeasure implementation since she is not aware of the locking mechanism as well as the original cache block replacement policy. In section 5.3, we present the experimental results highlighting the slowdown of applications due to Cache Locking countermeasure.

5 EXPERIMENTAL RESULTS

5.1 AppSAT attack on single-cycle netlist

The datapath and controlpath of the MIPS pipelined netlist were obfuscated using Anti-SAT scheme [20] integrated with Strong Logic Encryption (SLE) [13]. We used 5% XOR/XNOR key-gate overhead for locking the original synthesized netlist using SLE, and used additional key-gate inputs for obfuscation with Anti-SAT block, total key-size being 364 bits. Subsequently, we launched the AppSAT attack on the functionally equivalent locked single-cycle netlist (as outlined in section 3.2.2) with following parameters: a total of $5,000$ iterations of the SAT attack was performed, and at each iteration $10,000$ randomly generated patterns were queried to estimate the error rate $\mathcal{E}$, storing the distinguishing input/output pairs as constraints for successive iterations. In figure 4, we show the decreasing trend of $\mathcal{E}$ with the progress of SAT attack iterations. The *approx key* returned by the AppSAT attack consists of inputs to functional

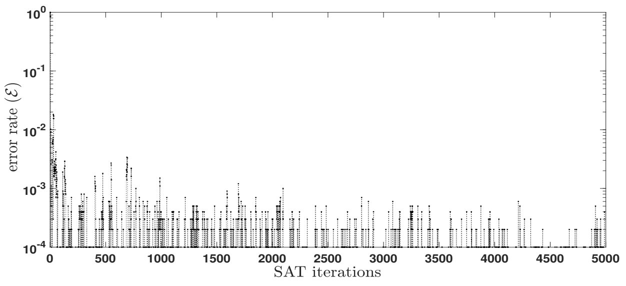

Figure 4: Error rate ($\mathcal{E}$) vs SAT attack iterations

key-gates (inserted using SLE scheme) and key inputs for Anti-SAT block. We observed that there was an exact match between the portions of the original key and the *approx key* that correspond to the functional key inputs, while there were mismatches in the portions of keys that correspond to the Anti-SAT block key-gates. However, as the output corruptibility of Anti-SAT block is *very low*, it has very limited effect on overall functionality retrieved by the *approx key*.

5.2 Application-level impact of *approx-key*

5.2.1 Experimental framework. We utilized the NVIDIA's SASSIFI framework [8] to capture errors on the application-level manifested due to the use of the retrieved *approx key* used to approximately unlock the inorder pipelined core netlist. As the SASSIFI tool injects errors in the architectural state, the outcomes of the injections are not dependent on specific GPU used, provided the binary file is not modified. We used NVIDIA's Maxwell architecture based GeForce GTX950M, CUDA 6.5 toolkit, and display driver version 352.63 for our experiments. We used 5 applications from Rodinia benchmark suite (version 2.3) [5] which include diverse workloads.

5.2.2 Error probability of faulty instructions. We applied the AppSAT attack as outlined in subsection 5.1 to deduce an *approx-key* for unlocking the pipelined cores of a GPU after converting the pipelined design to a functionally equivalent single-cycle one. For every application, we first noted the number of GPU assembly-level instructions that write to a General Purpose Register (GPR), $N = \#inst_{GPR}$. In order to estimate the number of GPR type instructions which are faulty for a benchmark application, we simulated the approximately unlocked SP core netlist with rN number of inputs. These inputs had the same opcode set as the benchmark instructions thereby capturing the spirit of the application instruction mix. Note that each benchmark has a different number and combination of instructions. Hence the error rate of each benchmark when executed on an approximately unlocked core could be different. For our experiments, we set $r = 20$ which resulted in simulating around hundreds of millions of inputs to the approximately unlocked core. Even for such a large test case, we found *no error* in *PO* values when compared with the correct responses for each instruction that we simulated. This illustrates the effectiveness of approximate unlocking the GPU chip using our proposed technique which does not require an unlocked chip. While our AppSAT based approximately unlocked GPU caused no measurable errors, we still wish to analyze the worst case scenario where 1 out of rN instructions are faulty to capture the *application-level* impact of the retrieved *approx-key*, though in practice the error probability will be even lower. We assumed that the number of faulty GPR instructions executed due to the use of *approx-key* follows Binomial distribution with error probability $\mathcal{E}_{AK} = 1/rN$ (because we assume that 1 out of rN instructions is faulty). Therefore, the probability that k number of GPR instructions are executed faulty can be expressed as follows:

$$P(X = k) = \binom{N}{k} \mathcal{E}_{AK}^{k} (1 - \mathcal{E}_{AK})^{N-k} \qquad (2)$$

Table 1: Application-level impact of datapath errors due to *approx-key* applied to key-inputs

Application	Outcomes of injections (percentage)			
	Masked	DUEs	Pot. DUEs	SDCs
BFS	95.57	1.82	0.00	2.60
gaussian	99.47	0.52	0.00	0.00
hotspot	97.92	0.00	0.78	1.30
nw	96.09	1.30	0.00	2.60
pathfinder	95.57	2.34	0.00	1.82

Table 2: Application-level impact of controlpath errors due to *approx-key* applied to key-inputs

Application	Outcomes of injections (percentage)			
	Masked	DUEs	Pot. DUEs	SDCs
BFS	96.09	1.82	0.00	2.08
gaussian	98.41	0.52	0.00	1.04
hotspot	97.92	0.00	0.78	1.30
nw	95.31	1.82	0.00	2.86
pathfinder	95.83	1.30	0.26	2.60

As $\mathcal{E}_{AK} << 1$ and $N >> 1$, we get $P(X = 1) \simeq N\mathcal{E}_{AK} = 1/r = 0.05$. It is to be noted that the probability that multiple GPR instructions will be executed faulty , i.e., $P(X >= 2)$, due to the use of *approx-key* is practically negligible. It is to be noted that in an actual scenario, the error propagation effect due to an *approx-key* will be restricted to only a few number of *low probability netlist paths*, and hence, the expected application-level error impact is even lesser. In our experiments, we *randomly* selected a GPR type instruction, to estimate effect of error propagation for different error injection sites.

5.2.3 Error impact on benchmark applications. Based on the analysis in previous subsection, we only considered the scenario where a single GPR type instruction is faulty with a probability of $p = 0.05$ for our experiments. For studying the application-level impact of errors due to the use of *approx-key* in data path and control path key-gates of the core netlist, we considered these cases separately. We used the SASSIFI framework to run error injections on 5 Rodinia benchmark applications [5] in Instruction Output Value (IOV) mode. In IOV mode, SASSIFI uses instrumentation handlers to inject errors into the destination register values of an instruction after they are executed. We performed 384 error injection runs for each of our application workloads so that the injection results have maximum error bars of 5% at 95% confidence level. In each error injection run, we *randomly* selected a dynamic instruction among all the GPR type instructions and either (i) *randomly* updated the destination register value of a thread for studying the impact of a datapath error or (ii) *randomly* updated all the destination register values of all the threads in a warp for studying the impact of a controlpath error. The results of the injections were categorized [8] as follows:
(i) Masked: No error symptom detected and the application output with fault injection run is same as the original *error free* output.
(ii) DUEs: The application terminated with a non-zero exit status or application runtime crossed the *timeout threshold.*
(iii) Potential DUEs: Symptoms of unsuccessful kernel execution (detected by comparison of kernel exit status with *cudaSuccess*), explicit application error messages can be found in *stderr/stdout.*
(iv) SDCs: Application execution terminates without any crashes, hangs, or failure symptoms but output file/*stdout* is different compared to fault-free run.

In tables 1 and 2, we report the results of such injection runs on the benchmark applications for datapath and controlpath errors. As evident from the statistics of the resulting outcomes, *almost all* of the injected errors are *masked* (95% or more depending on applications), implying that the *approx-key* is good enough to deobfuscate the functionalities of the locked SP or core pipelines such that there is very

Table 3: Benchmark apps slowdown due to Cache Locking (CL) with $\delta_{hit\ rate}$=0.5, α_{mem} = 0.5, and #*penalty*=500

Application	#*inst*($*10^6$)	#*mem*(%)	Runtime (secs)		*slowdown*
			$t_{original}$	t_{CL}	
BFS	424.2	12	9.37	19.43	2.07
gaussian	246.3	5	0.80	3.30	4.13
hotspot	440.1	7	13.44	19.58	1.46
nw	123.2	32	0.79	8.55	10.82
pathfinder	436.9	18	4.23	19.54	4.62

Table 4: BFS slowdown due to Cache Locking (CL) vs. $\delta_{hit\ rate}$ with penalty cycles (#*penalty*)=500 and α_{mem} = 0.5

$\delta_{hit\ rate}$	Runtime (secs)		*slowdown*
	$t_{original}$	t_{CL}	
0.3	9.37	15.33	1.64
0.4	9.37	17.41	1.86
0.5	9.37	19.49	2.08
0.6	9.37	21.35	2.28
0.7	9.37	23.35	2.49
0.8	9.37	25.37	2.71

Table 5: BFS slowdown due to Cache Locking (CL) vs. penalty cycles (#*penalty*) with $\delta_{hit\ rate}$ = 0.5 and α_{mem} = 0.5

#*penalty*	Runtime (secs)		*slowdown*
	$t_{original}$	t_{CL}	
400	9.37	17.34	1.85
450	9.37	18.35	1.96
500	9.37	19.43	2.07
550	9.37	20.50	2.19
600	9.37	21.40	2.28

low effect of gate-level error propagation impact at the application-level. As highlighted in figure 2 earlier, the *difference* in erroneous application outcomes is negligible for datapath and controlpath errors due to a very low probability of an instruction being faulty.

5.3 Cache Locking countermeasure results

We denote the drop in cache hit rate ($\delta_{hit\ rate}$) due to wrong *cache-key* guess as: $\delta_{hit\ rate} = hr_{original} - hr_{faulty}$, where, $hr_{original}$ and hr_{faulty} denote the cache hit rates corresponding to the designer's intended block replacement policy and attacker's faulty block replacement policy respectively. As an outcome of such a faulty policy, the application will incur additional clock cycles (#*add cycles*) which is estimated as follows:

$$\#add\ cycles = \delta_{hit\ rate} * \alpha_{mem} * (\#mem) * (\#penalty) \quad (3)$$

where, #*mem* denotes the number of memory access instructions (load or store) in the GPU assembly-level, #*penalty* is the number of penalty cycles to access slower off-chip memories, and α_{mem} corresponds to fraction of memory access instructions executed in parallel across multiple GPU cores. The value of the parameter α_{mem} will depend not only on the application workloads but also on the number of GPU cores as well as on the thread scheduling policies. To evaluate the effect of cache misses on an actual NVIDIA GPU, we modified the CUDA codes of the applications to introduce #*add cycles* number of sleep cycles in the device. In table 3, we report the relative slowdowns of various benchmark applications for a wrong *cache-key* setting parameters $\delta_{hit\ rate} = 0.5$, $\alpha_{mem} = 0.5$, and #*penalty* = 500. It can be observed that the proposed Cache Locking countermeasure results in slowdowns ranging from factors of 1.46 to as high as 10.82 depending on applications. In table 4, we report the variations in slowdowns of BFS application due to wrong *cache-key* with $\delta_{hit\ rate}$. The results from this table show that even if the attacker guesses a significant fraction of the *cache-key* correctly, resulting in small $\delta_{hit\ rate}$ (say 0.3), then also there is a notable slowdown (by a factor of 1.64) of the BFS application. In table 5, we report the variations in slowdowns of BFS application due to wrong *cache-key* with #*penalty* (ranging over 400−600 clock cycles) in order to capture the impact of Cache Locking scheme across different GPU architectures (with different off-chip memory configurations [1, 11]). As evident from the trends in the experimental results (see table 3), the impact of the proposed countermeasure will become *even more prominent* for practical applications having large number of memory references (#*mem*), thus defeating the *efficient utilization* of the GPU for high performance computing purposes for wrong *cache-key*.

6 CONCLUSION

In this paper, we outline an iterative SAT formulation based attack to *approximately* unlock the functionalities of pipelined GPU cores which are obfuscated using state-of-the-art logic locking scheme. The experimental results (obtained using NVIDIA's SASSIFI framework) reveal that the benchmark GPGPU applications exhibit high resiliency to error propagation effect due to use of a retrieved *approx-key* for unlocking the core netlists. Our proposed attack technique can be effectively utilized by an untrusted foundry to successfully deobfuscate GPU core netlist, even *without any requirement of an activated hardware*. Subsequently, we propose the Cache Locking scheme as a low-overhead countermeasure which significantly degrades the performance of GPGPU applications for a wrong *cache-key*.

7 ACKNOWLEDGMENT

This work was supported by AFOSR under Grant FA9550-14-1-0351.

REFERENCES

[1] NVIDIA CUDA. http://www.nvidia.com/cuda/. Accessed: 2017-11-15.
[2] NVIDIA Maxwell. https://developer.nvidia.com/maxwell-compute-architecture. Accessed: 2017-11-15.
[3] NVIDIA Nsight assembly correlation. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-view-assembly-code-correlation-nsight-visual-studio-edition/. Accessed: 2017-11-15.
[4] A. Baumgarten, A. Tyagi, and J. Zambreno. Preventing IC piracy using reconfigurable logic barriers. *IEEE Design & Test of Computers*, 2010.
[5] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, IEEE International Symposium on*, pages 44–54, 2009.
[6] S. Dupuis et al. A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*, pages 49–54. IEEE, 2014.
[7] B. Fang et al. Evaluating error resiliency of gpgpu applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1502–1503. IEEE, 2012.
[8] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 249–258. IEEE, 2017.
[9] R. Karri et al. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46, 2010.
[10] Y.-W. Lee and N. A. Touba. Improving logic obfuscation via logic cone analysis. In *LA Test Symposium 2015*, pages 1–6. IEEE, 2015.
[11] X. Mei and X. Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel & Distributed Systems*, 28(1):72–86, 2017.
[12] S. M. Plaza and I. L. Markov. Solving the third-shift problem in IC piracy with test-aware logic locking. *CAD, IEEE Transactions on*, 34(6):961–971, 2015.
[13] J. Rajendran et al. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 83–89. ACM, 2012.
[14] J. Rajendran et al. Fault analysis-based logic encryption. *Computers, IEEE Transactions on*, 64(2):410–424, 2015.
[15] J. A. Roy, F. Koushanfar, and I. L. Markov. Epic: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 1069–1074. ACM, 2008.
[16] K. Shamsi, M. Li, T. Meade, Z. Zhao, Y. Jin, and D. Pan. Appsat: Approximately deobfuscating integrated circuits. In *Proc. IEEE Symp. Hardware-Oriented Security and Trust. IEEE*, 2017.
[17] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, 1978.
[18] P. Subramanyan, S. Ray, and S. Malik. Evaluating the security of logic encryption algorithms. In *Hardware Oriented Security and Trust, 2015*, pages 137–143, 2015.
[19] K. Xiao et al. Hardware trojans: lessons learned after one decade of research. *ACM TODAES*, 22(1):6, 2016.
[20] Y. Xie and A. Srivastava. Mitigating sat attack on logic locking. In *CHES 2016*, pages 127–146. Springer, 2016.
[21] M. Yasin er al. Sarlock: Sat attack resistant logic locking. In *Hardware Oriented Security and Trust (HOST), 2016*, pages 236–241. IEEE, 2016.