Towards Provably-Secure Performance Locking

Monir Zaman[1*], Abhrajit Sengupta[2*], Danqing Liu[3], Ozgur Sinanoglu[4],
Yiorgos Makris[1], and Jeyavijayan (JV) Rajendran[3]
[1] The University of Texas at Dallas, [2] New York University, [3] Texas A&M University, [4] New York University Abu Dhabi

***Abstract*—Locking the functionality of an integrated circuit (IC) thwarts attacks such as intellectual property (IP) piracy, hardware Trojans, overbuilding, and counterfeiting. Although functional locking has been extensively investigated, locking the performance of an IC has been little explored. In this paper, we develop provably-secure performance locking, where only on applying the correct key the IC shows superior performance; for an incorrect key, the performance of the IC degrades significantly. This leads to a new business model, where the companies can design a single IC capable of different performances for different users. We develop mathematical definitions of security and theoretically, and experimentally prove the security against the state-of-the-art-attacks. We implemented performance locking on a FabScalar microprocessor, achieving a degradation in instructions per clock cycle (IPC) of up to 77% on applying an incorrect key, with an overhead of 0.6%, 0.2%, and 0% for area, power, and delay, respectively.**

***Index Terms*—IP piracy, performance locking, Boolean satisfiability (SAT), FabScalar**

I. Introduction

A. Motivation

Over the last few decades, the semiconductor industry has seen an exponential growth in integrated circuit (IC) production. With different performance requirements, companies currently build *different* ICs, thereby, inadvertently increasing the time to market. To address these diverse market requirements, Intel proposed a solution known as *Intel Upgrade Service* [1]. The idea was to have a single processor delivering different performances depending on the price paid by the user—we call this *performance locking* (PL).

Intel implemented performance locking in Clarkdale processors by asking their customers to pay an extra $50 to activate one megabyte of extra cache and hyper-threading [2]. The activation was performed by executing an activation code at the software level. Similarly, AMD applied performance locking by disabling one of the cores in their X3 quad-core processor, where the fourth core was only unlocked on applying the correct key, boosting its performance [3]. Currently, this kind of performance locking is realized by speed binning, by leveraging unintended process variations, and by frequency throttling [4].

Unfortunately, none of the above techniques provide any proofs of security, that is, it is not clear if an attacker can possibly boost the performance without paying any extra money. Furthermore, techniques such as speed binning are *non-deterministic* in nature, and hence, do not guarantee *controllable* performance tuning.

B. Business and threat models

Fig. 1 shows the business and threat models of performance locking. The design team synthesizes the design such that the processor can deliver both high and low performances. In this model, the attacker can reside at the foundry or can be the end-user. We assume the attacker has access to the reverse-engineered netlist of the design but does not possess any knowledge of the secret key which is used to lock the performance. To this end, an attacker can buy an IC with high performance from the market, reverse engineer it, and obtain the locked netlist. The aim of the attacker is to extract the secret key from the design such that s/he can unlock the superior performance. Moreover, the user can buy another IC from the market, use it as a black-box to apply input patterns, and obtain the corresponding responses.

In the academic literature, several papers have proposed performance locking or its variants in the past by inserting dummy states, black states, redundant states, and no-op states in the finite state machine (FSM), and retiming, resynthesis, and stuttering [5–10]. Whenever a design enters one of these states, its performance gets degraded. All these techniques are proven to be secure only in the context of preventing an untrusted foundry from stealing and/or overproducing a design. However, they do not discuss security against an attacker who has access to a functional chip. Hence, they cannot be used for performance locking. Recently, delay locking was introduced in [10]. While they assume the attackers have access to a functional chip, they lock the timing to create incorrect outputs. Similarly, performance locking for high-level synthesis was briefly discussed in [11]. These work do not provide any security proofs on why an attacker cannot recover the correct timing either; therefore, none of the previous work can deliver provably-secure performance locking.

C. This paper and its contributions

In this paper, we provide provably-secure performance locking at the microarchitectural level. We selectively lock/unlock certain performance-enhancing modules in the process architecture. However, instead of running an activation code like Intel, here the locking is implemented at the hardware level with a secret key. To this end, we adapt a technique called logic locking[1]. Since logic locking only locks the functionality, it is not suitable for locking the performance of a design. Thus, in this work, we define and implement performance locking.

The contributions of this paper are:

1) Design a performance locking module that is agnostic to the (processor) design;
2) Provide mathematical definitions for the security of performance locking;
3) Provide proofs of security (theoretically and experimentally) of the developed module;

[*] M. Zaman and A. Sengupta contributed equally.

[1] Logic locking enables a design to implement the correct functionality only upon applying the correct key. The design produces incorrect outputs on applying an incorrect key [11, 12].

978-3-9819263-0-9/DATE18/©2018 EDAA

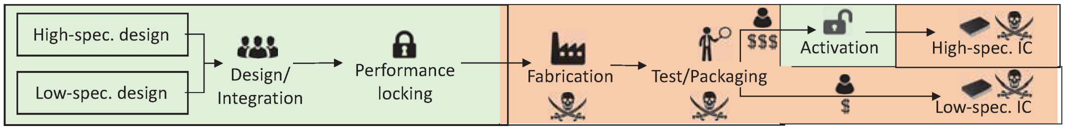
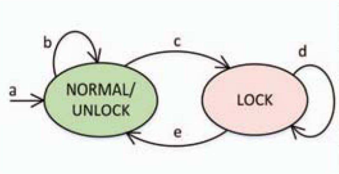
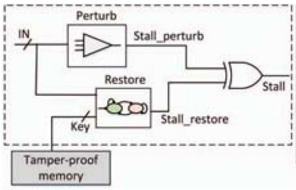


Fig. 1. **Business model:** Premium users can unlock the IC to get superior performance on paying extra money, while regular users get an IC with inferior performance. **Threat model:** The trusted and untrusted entities are shown in green and red, respectively.



Transition	Input	Execution
a	Reset = 1	Normal
b	Key = PDIP	Normal
c	Key ≠ PDIP	Lock if (IN = Key/ IN = PDIP)
d	Count ≤ N	Lock
e	Count = 0	Normal

Fig. 2. The state transition graph of the inserted FSM. N indicates the number of stall cycles. Normal execution indicates the design is in *NORMAL/UNLOCK* state. Lock indicates the design is in *LOCK* state.



IN/Key	000	001	010	011	100	101	110	111	Stall_perturb
000	L	--	--	--	--	--	--	--	✓
001	--	L	--	--	--	--	--	--	✓
010	--	--	L	--	--	--	--	--	✓
011	L	L	L	--	L	L	L	L	✗
100	--	--	--	--	L	--	--	--	✓
101	--	--	--	--	--	L	--	--	✓
110	--	--	--	--	--	--	L	--	✓
111	--	--	--	--	--	--	--	L	✓

Fig. 3. **Design.** *Stall* values controlled by perturb and restore. "L" indicates the *LOCK* state. "–" indicates the *NORMAL/UNLOCK* state. "✓" indicates that *Stall_perturb* has the same value as the original *Stall*, and "×" indicates the opposite.

4) Implement the performance locking on different modules of the FabScalar microprocessor such that performance locking has the maximum impact with provable guarantees and minimal (<1%) overhead.

II. Performance Locking: Design and Analysis

A. Locking mechanism

Idea. To degrade performance, we insert additional states in the FSM. In these additional states, no useful computation is performed. Hence, transitioning through these additional states reduces the performance. On applying the correct key, these additional states are skipped, providing superior performance. Otherwise, the design transits through these additional states, degrading the performance.

While similar approaches have been proposed in the context of IP piracy, they do not necessarily protect against attacker having access to a functional chip [5, 6, 8, 9, 13]. For performance locking, this means that a malicious user is able to recover the key and unlock the superior performance when these techniques are employed, as demonstrated by SAT [14], removal [15], and bypass attacks [16].

Hence, in this work, we design an architecture that can be used for performance locking, and more importantly, provides *provable* security guarantees. To this end, we take a two-step approach. In the first step, we lock the FSM by incorporating the additional states. In the second step, we design an architecture to securely integrate the FSM with a target design, ensuring performance degradation.

Step 1: FSM locking. Performance locking can be implemented using an FSM having two sets of states, namely, *NORMAL/UNLOCK* and *LOCK*. Fig. 2 shows the state transition graph. The original FSM of the system is represented in the *NORMAL/UNLOCK* state, where the system performs as intended without any degradation. Thus, in this state, the design can be considered as "unlocked." *LOCK* is an extra state added for performance locking. Whenever the execution enters this state, it remains there for N clock cycles, and no useful computation is performed during these clock cycles, effectively stalling the system. However, upon applying a correct key these N stalling cycles are skipped, and the execution remains in the *NORMAL/UNLOCK* state.

We now describe how this FSM is embedded into a target design such that it is secure against SAT [14], removal [15], and bypass attacks [16].

Step 2: Architecture. The locking mechanism consists of *restore unit* and an XOR gate as shown in Fig. 3. Here, without loss of generality, we explain the locking mechanism with the help of a *Stall* signal, which is used to stall the execution. In this paper, we modify the internal logic of the *Stall* signal to create another signal *Stall_perturb*. The idea is to set the *Stall_perturb* signal to logic 1 whenever a particular input pattern, called the *performance-degrading input pattern* (PDIP), arrives at the input. The *Stall_perturb* is set high for N clock cycles, effectively stalling the execution for N cycles. The restore unit is used to restore *Stall_perturb* to its original value. It consists of bit comparators between the input bits and the key bits. Thus, whenever the key value equals the correct key, and the PDIP arrives at the input, the *Stall_restore* signal goes high and restores the original value of *Stall*. Note that a similar approach presented in [11] can protect only combinational designs.

Consider the example shown in Fig. 3 having $n = k = 3$, where n and k denote the input and key sizes, respectively. The *Stall* signal is modified for the input pattern 011 to logic 1 as shown in *Stall_perturb* column in the table. However, it is restored to its correct value (logic 0) only on applying the correct key 011 in the restore unit. Otherwise, for an incorrect key, it goes high. This keeps the *Stall* signal at logic 1, degrading the performance.

B. Security analysis

Without loss of generality, we assume n input and k key bits, where $k \leq n$. The security definitions and conditions for security against many state-of-the-art attacks for logic locking are proposed in [17]. However, no notions of security exist for sequential designs that are required for our performance locking. Hence, we follow the security notions introduced in [17] to develop security properties for sequential designs. For brevity, in this section we will use "PL" in place of "performance locking".

Definition 1. *A sequential circuit ckt consists of combinational logic C and memory elements* M^2*, where the output at the* t^{th} *clock cycle is given by* $o_t = ckt(C, M_{t-1}, i_t)$*, where* o_t, i_t*, and* M_{t-1} *denote the output at the* t^{th} *clock cycle, input at the* t^{th} *clock cycle, and memory elements at the* $(t-1)^{st}$

[2] Here, the memory elements refer to the registers in the FSM of the design.

clock cycle, respectively. Thus, a PL technique $\mathcal{L}$ is a triplet of algorithms, (`Gen, Lock, Activate`), *where:*

1) `Gen` *is an algorithm to generate random keys,* `Gen` : $\{0,1\}^k \rightarrow z$*, where k denotes the key size,*
2) `Lock` *is an algorithm to implement PL,* $ckt_{lock} \leftarrow \texttt{Lock}_z(ckt)$ *such that* $\forall i \in P$, $ckt_{lock}(C, M_{t-1}, i_t) \neq ckt(C, M_{t-1}, i_t)$*, where P denotes the set of performance-degrading input patterns (PDIPs), and*
3) `Activate` *is an algorithm to unlock the circuit's superior performance,* $ckt_{actv} \leftarrow \texttt{Activate}_z(ckt_{lock})$ *such that* $\forall\ t,\ \forall\ i\ \in I,\ ckt_{actv}(C, M_{t-1}, i_t) = ckt(C, M_{t-1}, i_t)$*, where I is the set of all input patterns.*

Threat model. We consider the same threat model described in [12, 14–16, 18, 19]. The attacker has access to an oracle, denoted by $ckt(\cdot)$, which is a high-performance functional chip with the correct key (bought from the market). The attacker can apply certain input patterns to the chip and observe the corresponding responses. Also, the reverse-engineered netlist ckt_{lock}, locked using a PL technique $\mathcal{L}$, is available to the attacker. In this setting, the attack success for an attacker $\mathcal{A}^{\Delta}$, following an attack strategy Δ, implies recovering a circuit ckt_{rec} such that

$$\begin{aligned} \forall t, \forall i \in\ I, ckt_{rec}(C, M_{t-1}, i_t) &= ckt(C, M_{t-1}, i_t); \\ \mathcal{A}^{\Delta} &: ckt_{lock} \rightarrow ckt_{rec} \quad (1) \end{aligned}$$

SAT attack resilience. SAT attack is the strongest form of attack against logic locking that iteratively prunes the key space [14]. At each iteration of the attack, the oracle $ckt_{lock}(\cdot)$ is queried with an input pattern, called a distinguishing input pattern (DIP), which eliminates multiple incorrect keys. A DIP is an input pattern which when applied to the oracle produces different outputs for two different sets of keys. The attack terminates when no further DIP can be found and returns a key z'. An attacker $\mathcal{A}^{\mathbb{SAT}}$ uses it to reconstruct the circuit $ckt_{rec} \leftarrow \texttt{Activate}_{z'}(ckt_{lock})$ such that Eq. (1) is satisfied.

Definition 2. *PL technique $\mathcal{L}$ is λ-secure against a probabilistic polynomial time (PPT) attacker $\mathcal{A}^{\mathbb{SAT}}$, making a polynomial number of queries $q(\lambda)$ to the oracle, if s/he cannot reconstruct ckt_{rec} with probability greater than $\epsilon(\lambda)$, where $\epsilon(\lambda)$ is a negligible*[3] *quantity in λ [17].*

Theorem 1. *PL is k-secure against SAT attack.*

Proof. PL ensures that the number of DIPs required by the SAT attack is exponential in the key size. This is achieved by restricting each DIP to eliminate *only one* incorrect key value. However, in a fortuitous attempt, the attacker may hit the PDIP, thereby, eliminating all incorrect keys immediately. But, as the attacker is oblivious to the PDIP, the probability of such an event is exponentially small in the key size.

For proof, we define two sets, P and $\widehat{P}$ which denote the set of PDIPs and non-PDIPs, respectively. Now, for performance locking $|P| = 1$ and $|\widehat{P}| = 2^k - 1$, as P is a singleton set.

As described the above the attacker can recover the key, and thus, the high-performance design if s/he can find any PDIP in the set P. However, the probability of finding this PDIP with a polynomial number of queries $q(k)$ to the oracle is

$$\frac{|P|}{2^k} + \frac{|P|}{2^k - 1} \cdots \frac{|P|}{2^k - q(k)} \approx \frac{q(k)}{2^k} < \epsilon(k)$$

So, from Definition 2, PL is k-secure against SAT attack. □

Removal attack resilience. Removal attack is a structural attack, where the attacker removes the protection unit from the locked netlist [15]. It can be viewed as a transformation $T : ckt_{lock} \rightarrow ckt_{rec}$ such that Eq. 1 is satisfied without any knowledge of the key value z. However, for a removal attack against PL $ckt_{rec}(C, M_{t-1}, i_t) \neq ckt(C, M_{t-1}, i_t), \forall t, \forall i \in P$, where P denotes the set of PDIP.

Definition 3. *PL technique $\mathcal{L}$ is λ-resilient against a removal attack, where λ denotes the number of input patterns for which the output of ckt_{rec} differs from that of the oracle [17].*

Theorem 2. *PL is 2^{n-k}-resilient against removal attack.*

Proof. The high bias of the *restore* signal towards zero can be easily identified and subsequently, removed to recover the modified design [15]. However, for PL, the recovered circuit is different from the original one, because recovered circuit stalls the design for the PDIP. Note that for the general case of $n > k$, each PDIP in P has $n - k$ don't care bits, thereby, representing 2^{n-k} input patterns. Let Γ denotes the set of all input patterns represented by all the PDIPs in P,

$$\begin{aligned} ckt_{rec}(C, M_{t-1}, i_t) &\neq ckt(C, M_{t-1}, i_t), \quad \forall t, \forall i \in \Gamma \\ |\Gamma| &= |P| \times 2^{n-k} = 1 \times 2^{n-k} = 2^{n-k} \end{aligned}$$

From Definition 3, PL is 2^{n-k}-resilient against removal attack. □

Bypass attack resilience. Bypass attack randomly selects an incorrect key, finds the corresponding DIP for which it produces an incorrect output, and then uses a bypass logic to flip the incorrect output to the correct value for that DIP [16]. Hence, it can be viewed as a transformation $T : ckt_{lock} \rightarrow ckt_{rec}$ such that Eq. 1 is satisfied.

Definition 4. *PL technique $\mathcal{L}$ is λ-secure against an attacker $\mathcal{A}^{\mathbb{BP}}$ if the probability of finding all the incorrect outputs for an incorrect key is less than $\epsilon(\lambda)$.*

Theorem 3. *PL is k-secure against bypass attack.*

Proof. The bypass attack terminates only after *one* iteration of SAT attack by finding an incorrect key and its corresponding DIP. However, as the attacker is oblivious to the PDIP, s/he fails to reconstruct the bypass logic for the PDIP. Moreover, the probability of the DIP being the PDIP is $1/2^k < \epsilon(k)$. Thus, from Definition 4, PL is k-secure against bypass attack. □

III. Case study: Performance Locking in FabScalar Microprocessor

We implement the performance locking as a case study on an open-source microprocessor named FabScalar [21]. FabScalar toolset generates synthesizable RTL code of parameterizable superscalar microprocessor. In this work, we implement performance locking by modifying the internal

[3] Here, "negligible" indicates it is asymptotically smaller than any polynomial function [20].

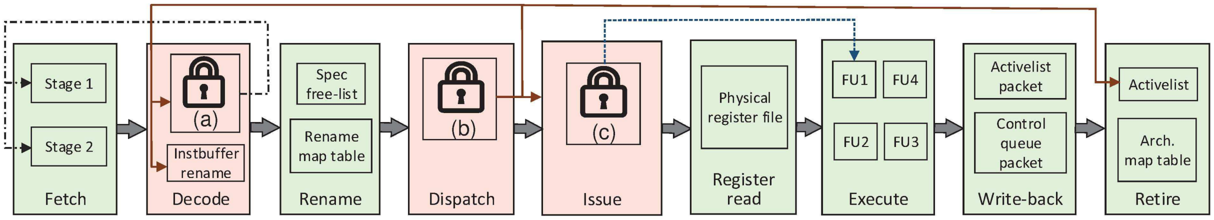


Fig. 4. Performance locking implemented in three modules of the FabScalar: (a) Instruction buffer (IB), (b) Dispatch (DP), and (c) Issue queue (IQ).

control logic of three modules: instruction buffer (IB), dispatch (DP), and issue queue (IQ) modules.

1) Instruction buffer (IB): This module is placed in between the fetch and rename stages. In every cycle, a maximum of four instructions can be placed in the IB queue, later to be forwarded to the rename stage. Whenever the number of incoming instructions exceeds the available empty slots in the IB queue, a stall signal is asserted to prevent new fetches by the fetch module. During the stalled cycles, current instructions in the IB queue are processed to free the slots for new instructions. Once enough empty slots are available, the stall signal is set low to fetch new instructions.

We exploit the stall feature to implement our performance locking, shown in Fig. 4. For the correct key, the functionality and the performance of the IB remain the same as the original one. When the user provides an incorrect key, the performance locking stalls the module whenever the module receives a PDIP at its inputs. If the output signal is low, the stall signal is set high, creating intentional stalls in the IB module. This is performed by stopping the fetch module from fetching any new instructions. The overall performance is degraded as no new instruction is allowed to enter the pipeline, effectively stalling the processor.

2) Dispatch (DP): DP holds the instructions to be processed by the next pipeline stages and is responsible for checking for empty slots in the load/store, retire, and issue queues. If enough empty slots are available, the new instructions are released by the DP module. During this time, the stall signal for the DP is kept low, indicating a normal pipeline operation. If any of the queues are full, the stall signal goes high and stalls various modules in the pipeline (shown in Fig. 4). The pipeline processes existing instructions in the queues to free up spaces, after which the DP releases the instructions by setting the stall signal low.

We leverage this stall signal to incorporate our performance locking in the DP module. Whenever the module receives a PDIP at its inputs, the performance locking unit checks for the stall signal and checks if the user has provided the correct key. With a wrong key and a low stall signal, the lock is activated to set the stall signal high. This informs the other modules in the pipeline that the queues are not empty to accept new instructions, degrading performance.

3) Issue Queue (IQ): IQ is placed in the issue stage to hold instructions from the dispatch module, which are then forwarded to the execution stage, where there are multiple functional modules operating in parallel. Each functional module also receives a *valid* signal from IQ, confirming that the instruction is ready to be executed. If the *valid* signal becomes low, the functional module refrains from executing the instruction until the signal is set high.

TABLE I

MICROARCHITECTURAL FEATURES USED IN THE FABSCALAR CORE-1

Microarchitecture feature	Value	Microarchitecture feature	Value
Fetch to dispatch width	4	Fetch depth	2
Issue to retire width	4	Rename depth	2
Fetch queue	16	Issue depth	2 / 2
Issue queue	32	Register read depth	1
Load/store queue	32/32	Fetch-to-execute pipeline depth	10

Performance locking unit (in Fig. 4) checks if the IQ module received a PDIP, and if the user provided key is correct. For a wrong key and a high *valid* signal, the lock resets the *valid* signal, stalling the pipeline, and thus, degrading performance.

IV. RESULTS

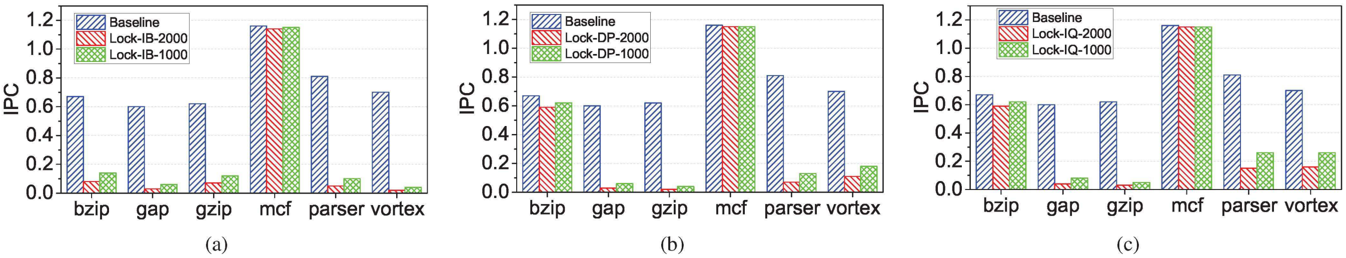
A. Experimental setup

We used the modified core-1 version of the FabScalar processor to implement performance locking. Table I lists its microarchitectural details. The toolset runs 10M instructions, which are selected by the Simpoints tool from each of the six SPEC2000 integer benchmarks [21]. This toolset uses Cadence NC-Verilog for RTL simulation and C++ for functional simulation. The design was synthesized using Synopsys EDK 90nm library [22]. The security analyses have been performed on a 64-core Intel Xeon processor running at 2.2 GHz with 264 GB of RAM. We implemented two versions of performance locking on FabScalar microprocessor: one with $N = 1K$ and the other with $N = 2K$, where N is the number of stall cycles.

To obtain the baseline performance values, we executed all six benchmarks on the original FabScalar microprocessor. For IB, we protected the four incoming instructions. For DP, one of the four incoming instructions is protected, and for IQ, we protected the first incoming instruction. Each of these protected entities is 80-bits long except for DP, where it is 76-bits. Note that for modern security standards 80-bits key is considered sufficient [23]. Using simulations, we selected the instructions (i.e., PDIPs) that occur predominantly, so as to activate the performance locking often. The number of occurrences of these instructions (PDIPs) at IB, DP, and IQ is 139K, 157K, and 138.5K, respectively, on an average across all the six benchmarks.

B. Impact on performance

Correctness. Using functional simulation, we verified that embedding performance locking into the FabScalar microprocessor did not change its functionality. The outputs matched with those of the baseline processor. To check whether performance locking impacts performance, we first collected the instruction per cycle (IPC) for each benchmark for the FabScalar microprocessor. Next, we reran the simulation for the FabScalar microprocessor with performance locking with the correct key in place. The IPC of the latter case is the same



(a)

(b)

(c)

Fig. 5. Performance degradation in terms of IPC for three locked modules with N=1K and N=2K compared with the baseline. (a) instruction buffer (IB), (b) dispatch (DP), and (c) issue queue (IQ).

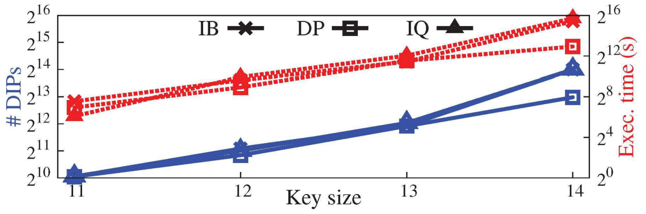


Fig. 6. Results for performance locking for k={11,12,13,14}; # DIPs required and execution time in seconds for the SAT attack [14].

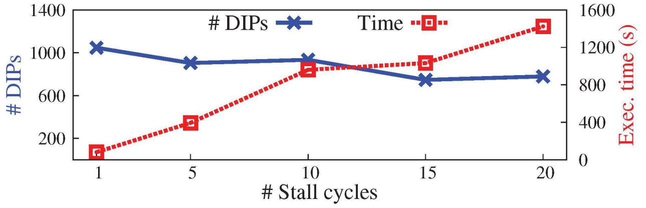


Fig. 7. Effect of stall cycles on security for a fixed key size of 14 bits.

as that of the baseline. This indicates that performance locking does not impact IPC when the correct key is in place.

Fig. 5(a) shows IPC for the locked IB module. IPC degrades on an average by 72% and 77% for 1K and 2K stall cycles, respectively. The `mcf` benchmark shows an average degradation of only 1.29%, because its PDIP occurs only 46 times, thus, triggering the lock only 46 times out of 10M.

An average IPC degradation of 58% and 63% for 1K and 2K stall cycles, respectively is shown in Fig. 5(b) for the locked DP module. The `mcf` and `bzip` benchmarks show an average degradation of only 0.86% and 9%, respectively, because their PDIP occur only 12 and 3K times out of 10M, respectively.

The locked IQ module shows an average IPC degradation of 53% and 60% for 1K and 2K stall cycles, respectively shown in Fig 5(c). The `mcf` benchmark shows little degradation because its PDIP occurred only 12 times out of 10M.

C. Security analysis results

In this section, we analyze the effectiveness of our technique against the state-of-the-art attacks [14–16, 18, 19].

1) SAT attack resilience: Fig. 6 reports the number of DIPs and the execution time required for a SAT attack for key sizes, 11, 12, 13, and 14, respectively. For our experiments, we have repeated the experiments 100 times for each module, and show the average value in Fig. 6. This way, we cancel out the effect of any outliers. Although these key sizes will not be used in real-life applications, we use them to show the trend in terms of number of DIPs and the execution time required by the SAT attack, which is exponential in the size of key bits.

Impact of the key size. Fig. 6 shows that the number of DIPs required increases *exponentially* with the key size. This is in accordance with our theoretical claim that the SAT attack requires 2^{k-1} DIPs on average to find the secret key.

Impact of the number of stall cycles. Fig. 7 illustrates the effect of the number of stall cycles on the resiliency against the SAT attack. We report the number of DIPs and execution time for the DP module for a fixed key size of 14 bits. Even if the number of stall cycles increases, the execution time increases only linearly. Nonetheless, it has no effect on the number of DIPs required, which remains almost constant across a different number of stall cycles. This shows that the number of stall cycles does not guarantee security but only impacts the performance of the attack.

AppSAT attack recovers an approximate netlist, thereby, improving the run-time of SAT attack against SAT-resilient techniques [18]. Approximate netlist differs from the original netlist for a polynomial number of input patterns. The terminating criterion is dictated by a predetermined error rate[4], which acts as a threshold. Since for performance locking the error rate is $\frac{1}{2^k}$, the attack terminates within a few seconds. Yet, it fails to extract the correct key.

In functional locking, the key returned by AppSAT can render the recovered netlist produce correct outputs for many input patterns. However, in performance locking, the key returned by AppSAT fails to boost the performance of the locked FabScalar. Fig. 8 shows the IPC results on using the key returned by AppSAT. On average, the IPC degrades by 77%, 63%, and 60% for IB, DP, and IQ, respectively. This result is similar to the one on using a random incorrect key. This experimentally validates our claim that performance locking is resilient to AppSAT.

Double DIP attack requires that at least two incorrect keys be eliminated with each DIP [19]. Since performance locking can only eliminate one incorrect key per DIP, as shown in Fig. 3, a Double DIP attack would immediately terminate.

2) Removal attack resilience: As shown in Fig. 8, the IPC degradation for a removal attack is 77%, 63%, and 60% for IB, DP, and IQ, respectively. This is because if an attacker recovers a netlist, s/he only has access to the modified one with the same performance as the locked design.

D. Overhead analysis

Table II shows the overhead of performance locking. Both the IB and IQ has $< 2\%$ overhead for area and delay. However, the overhead for DP module is high because the original DP is a combinational circuit with less than hundred gates and performance locking inserts registers into this module. However, when compared with baseline FabScalar microprocessor, the

[4] Error rate is the fraction of input patterns for which the output differs from that of the original design.

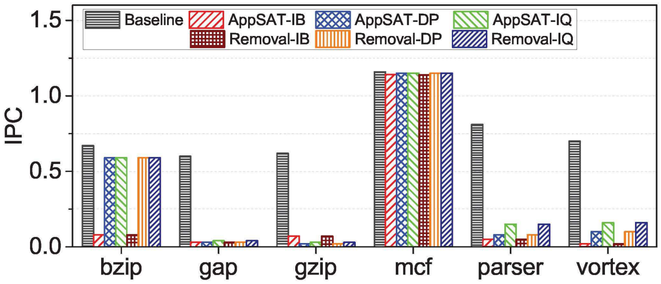


Fig. 8. Performance locking vs. AppSAT [18] and removal attacks [15]. Both the attacks return keys which does not improve the IPC, confirming that performance locking is resilient against AppSAT and removal attacks.

TABLE II
AREA, POWER, AND DELAY OVERHEAD

Modules	# Inputs	# Outputs	# Gates	Overhead (%)		
				Area	Power	Delay
IB	1021	509	38423	1.3	-0.6	0
DP	614	818	76	1110	244	1.7
IQ	715	590	39272	1.7	6.3	0
FabScalar	291	33	277252	0.6	0.2	0

overhead for locking all three modules is 0.6%, 0.2%, and 0% for area, power, and delay, respectively.

E. Discussion

Why not dynamic voltage and frequency scaling (DVFS) and speed binning? To implement different performances, manufacturers typically use speed binning, DVFS techniques, or design a completely different microprocessor [4]. While speed binning depends on process variations, it is hard for the designer to accurately predict the process variations for a target chip. DVFS enables different performance levels in microprocessors during runtime. Typically, this technique is used for power savings [24]. DVFS techniques can be easily extended to provide different performance levels by tuning the voltage and frequency. However, DVFS techniques are not secure because an attacker can control the voltage and frequency, thereby, increasing the performance.

V. Conclusion

The performance locking presented in this work enables a designer to design a secure and controllable single chip capable of having different performances. This approach can significantly reduce the time to market associated with IC design. We experimentally and theoretically proved that our performance locking technique is resilient to all the state-of-the-art attacks [14–16, 18], while entailing ∼1% overhead in power, area, and delay.

We also showed that through careful PDIP selection, significant performance degradation can be achieved. In future, we plan to incorporate the locking in more modules with minimum 80-bits of PDIP.

Next, we implemented performance locking with 1K and 2K stall-cycles. Our future work will include selecting the optimum number of wait cycles for the locked modules in a processor. This will be done by analyzing the performance impact of each module and the overhead they entail for protection. Finally, it is possible that during synthesis, the "KEY" logic checking circuit might add "and" logic tree depending how the synthesis tool is optimized [15]. This tree can then be traced and removed by an attacker. In our future work, we plan to implement the logic synthesis such that these traces are provably "hidden" from the attacker.

Nevertheless, our proposed performance locking technique is agnostic to the type of module to be locked and can be implemented to any design.

Acknowledgement

This work was supported in part by the National Science Foundation Computing and Communication Foundations (NSF/CCF) under Grant 1319841, the National Science Foundation, Division of Computer and Network Systems (NSF/CNS), under Grant number 1652842; and the New York University/New York University Abu Dhabi (NYU/NYUAD) Center for Cyber Security (CCS).

References

[1] C. Doctorow, "Intel + DRM: a crippled processor that you have to pay extra to unlock," https://boingboing.net/2010/09/19/intel-drm-a-crippled.html, 2010.

[2] A. Kingsley, "Facepalm of the Day: Intel charges customers $50 to unlock CPU features," https://goo.gl/ZNY6Z4, 2013.

[3] M. Buchanan, "AMD Phenom X3 Triple Core Processors Are Crippled Quad Cores in Disguise," http://goo.gl/Uj4CBM, 2008.

[4] S. Herbert and D. Marculescu, "Variation-aware dynamic voltage/frequency scaling," *IEEE International Symposium on High Performance Computer Architecture*, pp. 301–312, 2009.

[5] R. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, pp. 1493–1502, 2009.

[6] Y. Alkabani, F. Koushanfar, and M. Potkonjak, "Remote activation of ICs for piracy prevention and digital right management," *IEEE/ACM International Conference on Computer-aided Design*, pp. 674–677, 2007.

[7] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," *USENIX Security Symposium*, pp. 20:1–20:16, 2007.

[8] L. Li and H. Zhou, "Structural transformation for best-possible obfuscation of sequential circuits," *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 55–60, 2013.

[9] T. Meade, Z. Zhao, S. Zhang, D. Pan, and Y. Jin, "Revisit Sequential Logic Obfuscation: Attacks and Defenses," *IEEE International Symposium on Circuits & Systems*, 2017.

[10] Y. Xie and A. Srivastava, "Delay Locking: Security Enhancement of Logic Locking Against IC Counterfeiting and Overproduction," *IEEE/ACM Design Automation Conference*, pp. 9:1–9:6, 2017.

[11] M. Yasin, A. Sengupta, B. Schafer, Y. Makris, O. Sinanoglu, and J. Rajendran, "What to Lock?: Functional and Parametric Locking," *ACM Great Lakes Symposium on VLSI*, pp. 351–356, 2017.

[12] J. Roy, F. Koushanfar, and I. Markov, "Ending Piracy of Integrated Circuits," *IEEE Computer*, vol. 43, pp. 30–38, 2010.

[13] Y. Alkabani and F. Koushanfar, "Active control and digital rights management of integrated circuit IP cores," *ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 227–234, 2008.

[14] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the Security of Logic Encryption Algorithms," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 137–143, 2015.

[15] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Removal Attacks on Logic Locking and Camouflaging Techniques," *IEEE Transactions on Emerging Topics in Computing*, vol. PP, pp. 1–1, 2017.

[16] X. Xu, B. Shakya, M. Tehranipoor, and D. Forte, "Novel Bypass Attack and BDD-based Tradeoff Analysis Against all Known Logic Locking Attacks," Cryptology ePrint Archive, 2017, https://eprint.iacr.org/2017/621.

[17] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," *ACM/SIGSAC Conference on Computer & Communications Security*, pp. 1601–1618, 2017.

[18] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Pan, and Y. Jin, "AppSAT: Approximately Deobfuscating Integrated Circuits," *IEEE International Symposium on Hardware Oriented Security and Trust*, pp. 95–100, 2017.

[19] Y. Shen and H. Zhou, "Double DIP: Re-Evaluating Security of Logic Encryption Algorithms," Cryptology ePrint Archive, 2017, https://eprint.iacr.org/2017/290.

[20] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2014.

[21] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 11–22, 2011.

[22] R. Goldman, K. Bartleson, T. Wood, K. Kranen, C. Cao, V. Melikyan, and G. Markosyan, "Synopsys' open educational design kit: Capabilities, deployment and future," *IEEE International Conference on Microelectronic Systems Education*, pp. 20–24, 2009.

[23] N. Smart, "ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012)," http://www.ecrypt.eu.org/ecrypt2/documents/D.SPA.20.pdf, 2012.

[24] R. Teodorescu and J. Torrellas, "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors," *IEEE International Symposium on Computer Architecture*, pp. 363–374, 2008.