Memory Locking: An Automated Approach to Processor Design Obfuscation

Michael Zuzak and Ankur Srivastava
University of Maryland, College Park
mzuzak@umd.edu, ankurs@umd.edu

***Abstract*—Conventional logic obfuscation techniques largely focus on locking the functionality of combinational modules. However, for processor design obfuscation, module-level errors are tangential to the fundamental adversarial goal: to produce a processor capable of running useful applications. As noted in previous work such as [1], module-level locking poses the following problem: high corruption in a locked module results in a high application-level error rate, but fundamentally leads to SAT attack susceptibility. Therefore, for combinational, module-level locking, increases in application-level error rates are accompanied by a corresponding increase in SAT susceptibility and vice versa. To address this, we introduce an automated and attack-resistant obfuscation technique, called memory locking, which targets on-chip SRAM. We demonstrate the application-level effectiveness of memory locking through system-level simulations of obfuscated processors.**

I. Introduction

Due to the proliferation of custom IC design and the estimated $15-20 billion investment required to setup a fabrication line for the smallest transistor size by 2020 [2], many IC design companies have gone "fabless," relying on unaffiliated foundries to fabricate their intellectual property (IP). These untrusted foundries have raised concerns of piracy as the capability to reverse-engineer GDSII files to pirate, counterfeit, or overproduce ICs has been demonstrated. In response to these capabilities, the hardware security community has proposed logic obfuscation techniques to thwart would be pirates [3].

Previously, research into logic obfuscation has been dominated by logic locking which involves the insertion of key-driven accessory logic such as XOR/XNOR gates, multiplexers, or look-up tables into combinational modules within an IC [3]. In response to these techniques, Subramayan et al. proposed a satisfiability (SAT) attack which, while an NP-hard formulation, was shown to unlock 95% of examined logic locked circuits within 10 hours [4]. The proposed SAT attack leverages a SAT solver to iteratively find a set of special inputs, known as distinguishing inputs (DI), which are used to eliminate the set of incorrect keys. A variant of SAT attack called AppSAT [5] was developed with the objective of finding approximate keys which minimize output corruptibility in a given time by relaxing the guarantee of complete logical correctness. These approximate keys are often extremely accurate, leaving minimal error rates in an improperly keyed IC.

In response to SAT-based attacks, the hardware security community has diverged into two primary approaches: non-digital obfuscation (i.e. delay locking [6]) and SAT resistant logic blocks (i.e. Anti-SAT block [7], SARLock [8], or SFLL [1]). Recent work in [9] proposed the TimingSAT attack which was able to unlock delay based obfuscation techniques such as [6] using a SAT formulation. While Anti-SAT and SARLock are resilient to SAT attack, they succumb to AppSAT [5] as extremely low output corruptibility approximate keys can be found. This led to stripped functionality logic locking (SFLL) [1], a technique enabling a quantifiable trade-off between output corruptibility and attack resilience. The work in [1] shows, using provable methods, that increasing output corruptibility is bound to reduce SAT attack complexity.

In this work, we diverge from the idea that module-level errors are adequate to thwart piracy. We argue that the attacker's goal in processor piracy is not to unlock individual modules, but to sufficiently unlock the system so that it can be used to execute target applications successfully. Pairing this subtle change in attacker with the findings of [1], we arrive at a central theme of this work: a challenging trade-off exists in the fundamental assumptions of proposed combinational logic obfuscation techniques. In order to achieve a high amount of application corruptibility, the module-level corruptibility must be very high. However, a high module-level corruptibility inevitably makes the locked IC susceptible to SAT and AppSAT type attacks. We demonstrate that to achieve the module-level error rates sufficient for the denial of application-level functionality for a processor, one must design a locked circuit which is inherently SAT susceptible.

To address this, we propose memory locking, an automatable logic obfuscation technique capable of denying application-level functionality to the adversary while maintaining SAT resistance. We target on-chip SRAM circuitry due to the 50-90% of transistor count dominated by SRAM-related circuitry in modern processors [10]. This creates significant flexibility in obfuscatable location and functionality. Additionally, the analog effects governing SRAM make it resistant to many proposed attack methodologies such as SAT-based attacks. We then demonstrate the application-level effectiveness of memory locking compared to prior art with system-level simulations using GEM5 [11] to *close the loop* between module-level locking and its architectural impact.

II. Motivation

We motivate our work with a demonstration of the underlying trade-off involved in the combinational obfuscation of processor ICs. At its core, the module-level error rates sufficient for the denial of application-level functionality yields a circuit which is inherently SAT susceptible. We arrive at this conclusion experimentally, leveraging derivations in [1], [7].

A. Attacker Model

The attacker is an untrusted foundry with the goal of obtaining a key enabling the correct functionality of a logically obfuscated processor through the use of a locked gate-level netlist and an activated, black-box oracle IC. This is a standard attacker model [5]–[7], however, in each cited work, it is applied to combinational circuits. In this work, we enlarge our

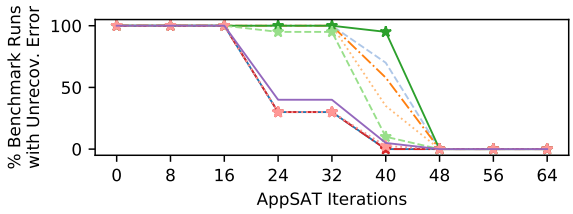

Fig. 1: Application failure rate of PARSEC benchmarks on SFLL locked, AppSAT attacked netlist.

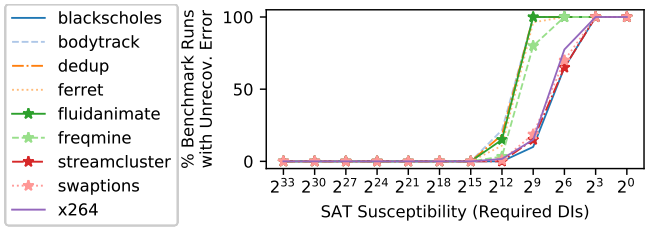

Fig. 2: Relationship between SAT susceptibility and application-level error rates in obfuscated processor IC.

view to consider the processor architecture as a whole, rather than individual modules. Using module-level error rates to evaluate the effectiveness of architectural piracy is insufficient. Other fields have extensively studied a similar concept with respect to "soft errors" such as [12], which states that many soft errors occurring in processors are innocuous when viewed architecturally and have no effect on application-level performance. We argue that this is also the case in logic obfuscation.

B. Relationship Between SAT Susceptibility and Error Rate

The work in SFLL [1], Anti-SAT [7], and SARLock [8] mathematically demonstrates that increases in module-level error rates directly correspond to decreases in the complexity of a SAT attack against the circuit. This is the case because increasing the number of module-level errors for any given wrong key leads to a necessary corresponding increase in the overlap of incorrect primary inputs between multiple wrong keys. When several wrong keys share common incorrect primary inputs, the adversary is able to eliminate several keys by selecting any shared primary input as a DI. The runtime of a SAT attack is dictated by the number of DIs which must be found to eliminate all wrong keys, therefore, increasing the number of keys which can be eliminated with a given DI inherently reduces SAT attack runtime because it reduces the number of DIs which must be found. We are able to create SAT resilience by limiting the number of wrong keys which can be eliminated with any given DI, hence limiting our module-level error rate. We refer the interested reader to the derivations in [1], [7] for further details of this relationship.

C. Attacking Combinationally Locked Processors

We begin by demonstrating successful attack results on real MIPS and Motorola 68000 (M68000) processor netlists obfuscated using both SFLL [1] and Anti-SAT [7] to prevent IC piracy. By choosing one RISC and one CISC architecture, we hope to provide a reasonable cross section of processor netlists. As SFLL and Anti-SAT have been proposed for use in both processor control and data path logic [13], we will obfuscate circuitry in each path.

For data path circuitry, we rely on ALU locking, locking both the adder and multiplier circuits using both SFLL and Anti-SAT. We have targeted adder and multiplier circuits as

Control Circuit	SAT Att. Runtime	Mod. Att. Runtime
SAT Susceptibility: SFLL Locked Control Logic		
MIPS Inst. Dec. (RISC)	148493 sec	0.0123 sec
M68000 Inst. Dec. (CISC)	99331.4 sec	2884.95 sec
SAT Susceptibility: Anti-SAT Locked Control Logic		
MIPS Inst. Dec. (RISC)	132867 sec	0.0292 sec
M68000 Inst. Dec. (CISC)	102054 sec	2921.36 sec

TABLE I: SAT attack runtime for processor control logic.

they are sufficiently large netlists to provide SAT resistance and are critical for processor functionality. We have locked both netlists (RISC and CISC) with the configurations proposed in their respective work ([1] and [7]). SFLL locked netlists were locked with 128-bits of conventional logic locking [14] and the maximum size SFLL block [1], using all available input bits to the netlist. The SFLL locked adder and multiplier netlists had a single stripped functionality I/O pair, as was done in the fabricated processor IC example in [1]. The Anti-SAT block was similarly configured with 128-bits of conventional logic locking [14] alongside the maximum size Anti-SAT block [7], using all netlist inputs. For control path circuitry, we selected the largest control circuit (by gate count) to be targeted for locking. In both netlists (RISC and CISC), this was the instruction decoder stage of the pipeline.

We utilize a SAT based methodology to attack both the data and control path locks of the SFLL [1] and Anti-SAT [7] secured M68000 and MIPS netlists. We begin by unlocking the control circuitry through mounting a SAT attack [4] on each of the locked instruction decoder netlists. Note that the control circuitry was locked using techniques which provide strong theoretical guarantees. However, by its very nature the controller is also rather small thereby making an exhaustive search of the input space feasible. Moreover the controller opcodes are publicly known in many processors [15] and tend to only utilize a small portion of the controller input space. Hence, a "modified" SAT attack which exclusively searches for a key which unlocks these inputs can be easily launched.

We mounted both of these attacks (one in which SAT is directly applied and one in which SAT is modified to unlock only valid instruction opcodes) on the locked decoder netlists and recorded the results in Table I. We were able to determine a key fully unlocking each netlist, regardless of locking technique, within 1 hour using the modified SAT formulation described above. Because decoder circuitry was the largest control circuit in both cases, it appears that control circuitry is not complex enough to provide resistance to SAT-based attacks, even in the presence of SAT-resistant techniques.

We now turn our attention to the data path. SAT resistant locking paired with the size and complexity of ALU circuits, compared to control logic, leaves us unable to mount a traditional SAT attack in a timely fashion. Instead, we mount an AppSAT attack to find an approximate key that unlocks most netlist functionality. While the AppSAT attack weakens the effectiveness of obfuscation over the un-attacked netlist, it does not fully unlock the circuit into a zero-error state.

The persistence of module-level error leads to a central question: are these module-level errors sufficient to provide application-level security? We approach this question by leveraging the GEM5 [11] simulator to perform system-

level simulations of PARSEC [16] benchmarks on the locked netlist. We aggregate the architectural simulation results to estimate the likelihood of encountering an unrecoverable error after a certain number of AppSAT attack iterations. We have displayed the results for the highest error case, the SFLL locked M68000 ALU netlist, in Figure 1. The remaining results have been omitted for brevity.

By utilizing approximately correct keys from the AppSAT attack, we achieved 0% benchmark error rates within 48 AppSAT iterations in all cases. Despite module-level error remaining in the circuit, the approximate key yielded a piratable IC capable of running useful workloads, the adversary's primary goal. By successfully attacking each of these netlists, obfuscated in 3 locations using state-of-the-art logic obfuscation techniques, we show that combinational locking as proposed is limited in preventing processor piracy in practice.

D. Limitations of Combinational Obfuscation

These successful attacks demonstrate the limitations of combinational logic obfuscation: high module-level corruptibility, while effective at denying application-level functionality, leads to increased SAT susceptibility. We attempt to quantify this trade-space using system-level simulations by varying the module-level error rate remaining in the locked processor to determine the application-level failure rate of the PARSEC benchmark suite. We then related the module-level error rate to the required number of DIs and therefore SAT iterations necessary to locate the correct key. This transformation allows us to directly relate the application-level error rate to the SAT susceptibility of state-of-the-art locking techniques.

From Figure 2, notice that while application-level errors begin to occur around a SAT susceptibility of 2^{12}, significant application-level error rates regardless of workload, are not achieved by module-level locking until between 2^9 and 2^6 DIs must be found to successfully SAT attack the circuit. 64 to 512 SAT iterations to locate the correct key is a quite feasible complexity and would lead to successful SAT termination in all but the largest combinational modules. For context, the SAT resistant 64-bit SFLL configuration used to secure the processor IC in [1] would require 2^{63} SAT iterations to locate the correct key on average. The massive difference between the SAT susceptibility required for application-level errors and the SAT susceptibility present in SAT resistant circuitry highlights that module-level locking as proposed is inherently unable to provide both SAT resistance and application-level errors.

Based on the results in this section, we argue that the trade-off between SAT susceptibility and application-level error rate is a real one. Additionally, in the specific trade-space characterized by Figure 2, there does not exist a configuration which guarantees the IC designer both SAT resistance and application-level security using current state-of-the-art combinational locking techniques. In order to provide piracy protections for processor ICs, we believe these results highlight that as a community, we must explore methodologies beyond conventional combinational logic obfuscation techniques.

III. Memory Locking

To remedy the issues underlying combinational obfuscation presented in the previous section, we propose memory locking, a logic obfuscation technique focused on denying on-chip SRAM functionality to the adversary. The SRAM circuit is targeted for 3 reasons. 1) SRAM circuitry dominates processor area and is involved in most processor functionality. This provides flexibility in obfuscatable location and functionality. 2) SRAM contains a delicate analog/timing balance. This leads to discrete-domain attack resilience and easily induced errors. 3) SRAM arrays are generated with design automation which can be leveraged to incorporate memory locking.

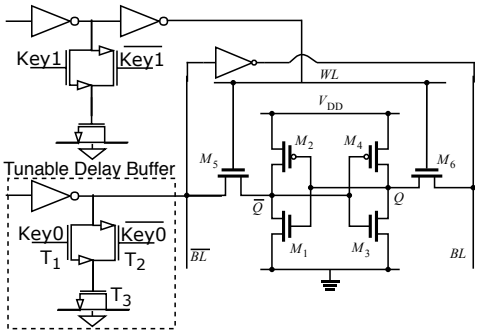

Fig. 3: Memory locked SRAM cell.

More specifically, memory locking is the insertion of tunable delay buffers (TDB), contained in the box in Figure 3, within buffer step-up chains of the bit-lines and word-lines of on-chip SRAM arrays. These TDBs enable the key-driven modification of the parasitic capacitance on these lines thereby altering the analog signature (drive strength, power leakage, timing, etc.) of any attached SRAM cells.

Throughout the remainder of this work, we will rely on an abstraction called the ϕ value of an SRAM cell, which characterizes the current state of analog parameters a cell is operating in. We use this because an SRAM cell is designed for multiple interrelated parameters including relative word/bit-line timing, power leakage, cell sizing, drive strength, and cycle timing. The raw values of each of these variables are irrelevant as memory locking relies primarily on the divergence of these variables from the values the SRAM cell was designed for. Fundamentally, memory locks act to upset the SRAM cell state from the unique designed for state, $\phi_{correct}$, to an non-unique incorrect state, $\phi_{incorrect}$, when improperly keyed.

A. Memory Locking Example

A basic example of a memory locked circuit is contained in Figure 3. First, we focus on the functionality of the labeled TDB circuit, T1-T3, located on the bit-line of Figure 3. When a '1' is applied as key0 to the pass transistors (T1 and T2), a parasitic capacitance is created through the gate/source-drain of T3 and added to the bit-line. Applying '0' as key0 would not connect the parasitic capacitance (T3) to the circuit. At a fundamental level, by modifying the key, memory locking acts to modify parasitics on the bit/word-line. This added parasitic alters the bit/word-line drive-strength, leakage, timing, and other analog parameters. Utilizing this, when a correct key is applied to the IC, the signal modifications created by memory locks induce a designed for state. In an improperly keyed IC, an unintended parasitic is introduced which upsets the analog equilibrium of the SRAM array and induces errors.

Now, let us assume the memory locked SRAM circuit displayed in Figure 3 was designed to have a correct key value of '10.' If an untrusted foundry were to fabricate this circuit and incorrectly apply the key '11,' both a write error and read error state would be induced. In the case of a write, no value would be stored in the SRAM cell because the pass transistor

(M6) is no longer able to overpower the value held in the SRAM cell inverter (M3 and M4) due to the increased bit-line capacitance. Note that this error is due to the drive-strength of the bit-line rather than bit-line timing. While relaxing timing might help to alleviate this error, a sufficiently large TDB would restrict the bit-line charge from ever overpowering the SRAM cell, inducing an error regardless of timing. In the case of a read, a read error would occur as the SRAM cell would be unable to pull the bit-lines apart rapidly enough to ensure accurate sense amplifier functionality, once again due to the increased capacitance. Again, as is the case for a write, a sufficiently large TDB would yield a bit-line too capacitive to be overpowered by the SRAM cell, inducing an error regardless of timing. Each alternate wrong key results in similar analog and timing issues within the locked cell.

B. Relationship to Prior Work

Notice that memory locking relies on similar structures to delay locking [6], but utilizes them differently. TDBs are utilized in memory locking to modify the analog signature of an SRAM array (i.e. power leakage, drive strength, etc.) rather than to create timing violations within combinational paths as was the case in [6]. Memory locking targets bit/word-lines because these lines must have sufficient drive strength to deliver power to overwrite internal SRAM transistors on a write, but also capable of being overpowered by these same transistors on a read for accurate functionality. By modifying analog parameters with TDBs and SRAM ϕ values, memory locking creates security through inducing a challenging analog design problem with many interdependent variables rather than the combinational timing focus of [6].

C. Locking Large Scale SRAM Arrays

As we scale memory locking, a security limitation presents itself. Assuming the SRAM array was symmetric, the ϕ value for every cell is identical and therefore each SRAM cell will have the same correct key. An adversary could reverse-engineer the correct key for one cell and replicate this key for all other cells, unlocking the whole SRAM array. This is due to *lack of diversity* in ϕ values. Because of this simple intuition and the symmetry of SRAM arrays, memory locking as described would be limited in key-space and security.

We can create diversity through ensuring multiple unique $\phi_{correct}$ values throughout the SRAM array. By placing non-keyed parasitics within SRAM cells through small, but *random* deviations in cell parameters, the designer can create unique $\phi_{correct}$ values for each cell (or set of cells). Each of these $\phi_{correct}$ values have a unique correct key which should be at least 1 bit, but could be longer. As we linearly increase the number of unique $\phi_{correct}$ values, the attacker will face an exponential increase in the searchable key-space, therefore an exponential increase in the required reverse engineering effort. Finding the correct key for a cell in isolation does not guarantee a solution for cells with alternate $\phi_{correct}$ values.

By designing for multiple $\phi_{correct}$ values throughout the array, the adversary is forced to redesign the SRAM array in its entirety in the process of solving for the correct key. This is unrealistic given that 50-90% of the transistor count on modern CPUs is devoted to SRAM circuitry [10]. In Figure 4 we show the topology of a memory locked SRAM array containing a unique $\phi_{correct}$ value for each SRAM cell.

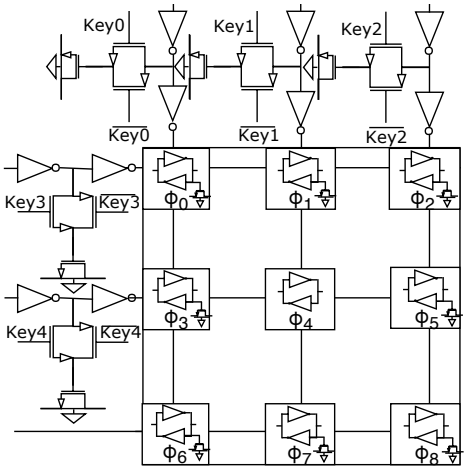

Fig. 4: 100% unique ϕ value memory locked 3x3 SRAM.

D. Memory Locking Implementation

Memory layout is generally performed with an SRAM compiler. These tools strive to lessen the time required to layout the large area of SRAM cells that span modern ICs. We propose a methodology for the design automation of a memory locked SRAM array leveraging the feature set of openRAM [17], an open-source SRAM compiler. Despite our focus on openRAM, nearly every modern SRAM compiler shares the functionality necessary to implement this methodology.

To automate a memory locked layout, two additional custom blocks, a TDB and a parasitic capacitance with parameterizable sizing, must be included in the SRAM compiler. First, the IC designer would tile an array of SRAM cells for a specific architectural block (i.e. register file). Depending on the level of security desired, a certain set of unique ϕ values are chosen. These values are distributed randomly across the SRAM cells. Based on the ϕ value, each cell is redesigned to include a certain internal parasitic capacitance.

Following array layout, TDBs are added on bit/word-lines utilizing the compiler's timing analysis tool to size these locks. Note that each unique ϕ value corresponds to a unique memory locking formulation and requires at least 1 key bit. The total number of key-bits corresponds to the total number of unique ϕ values in the SRAM array. Following said modifications, the IC designer proceeds with a standard design flow, adding peripheral circuitry as required. This automated methodology yields a memory locked SRAM with a given key.

Leveraging the proposed methodology, we designed a 6x6 SRAM array to explore the effects of memory locking in a slightly larger circuit. This array was designed using FreePDK15 [18], an open-source, 15nm, predictive-process library and simulated using HSpice. With these tools, we were able to design a 15nm SRAM cell, tile it into a memory locked 6x6 array, and add peripheral circuitry to the array. By cycling the inputs to our decoder, we were able to simulate memory traffic with accurate timing and control signals.

By sweeping over the percentage of unique ϕ values present in the array, we are able to quantify the timing overhead of memory locking. Because SRAM cycle time is dictated by the capacitance of the circuit, increases in unique ϕ values cause a degradation of the minimum cycle time of the array. We quantified the relationship between unique ϕ values and circuit timing degradation in Figure 5. Note that a linear increase in

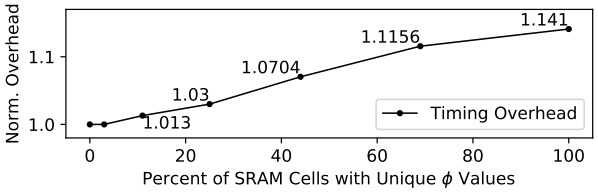

Fig. 5: Memory locking security/timing overhead trade-off.

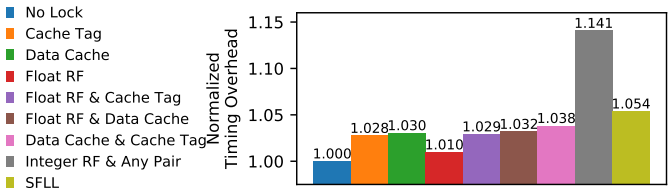

Fig. 6: Average obfuscation timing overhead on 8086 core running PARSEC.

the percent of unique ϕ values yields a linear increase in timing overhead while exponentially increasing key-space.

E. Security Analysis of Memory Locking

Tool-Driven Approach: The automated nature of memory locking might drive concerns of an adversary using similar tools to determine the correct key for a memory locked SRAM. While the attacker has access to circuit details, they would need to perform detailed row/column simulations for any SRAM cell with a unique ϕ value and a memory lock on an associated bit/word-line. For each cell, the attacker must apply a key to the array and then read/write both a '1' and a '0' at each locked cell to verify it. To verify that a read stability error does not occur, the attacker must perform two consecutive reads for each word-line while storing both a '1' and a '0'. To verify the write capability from any state, the adversary must write each cell from '1' to '0' and '0' to '1'. This process would require 9 read/write operations and hence 9 clock cycles in the best-case to achieve. This series of read/writes must take place for each locked cell in the SRAM array for each key guess. This implies that unlocking a register file, one of the smallest on-chip SRAM arrays, with a 64-bit key, corresponding to 64 memory locks with unique ϕ values, and 64, 64-bit registers would require over $6.8 * 10^{23}$ cycles to unlock. This is unrealistic.

SAT Based Approach: SAT-based formulations, including those which target timing based locking, such as TimingSAT [9], do not apply to memory locking. This is due to the feedback loop and internal state present in each SRAM cell. This feedback loop leads to a recursive and thus unresolvable SAT formulation. We direct the reader to Lemma 1 from [19] which states that if a feedback loop in a circuit is stateful, SAT attack will enter an infinite loop. Given that memory is inherently stateful, memory locking becomes SAT-unresolvable.

Removal Attack: In this case, we assume the adversary has removed all TDBs from SRAM circuitry. This creates a non-functional circuit as the SRAM array was designed to function given only the correct key configuration which includes added parasitic capacitance from the memory locks.

Redesign Based Attacks: One can argue that locked, on-chip SRAM arrays could be selected from the layout and replaced by an unlocked SRAM array of the same size. However, such an attack is also unrealistic. The primary concern is that 50-90% of transistor count within modern processors is involved in SRAM circuitry [10]. Even with the help of SRAM automation tools, searching through, removing, and then redesigning this portion of IC transistor count is a massive undertaking. On-chip memory is distributed in several smaller modules such as pipeline registers, register files, branch predictors, etc. A complete "find and replace" of all these locked modules with unlocked modules while matching the timing, area, etc. characteristics is quite a challenging endeavor. We also emphasize that conventional locking approaches are subject to such "find and replace" attacks as the functionality of targeted combinational modules, such as an adder, multiplier, instruction decoder, etc., is generally known.

IV. Architectural Obfuscation and Results

A. Simulator Overview

For this work, we leveraged the GEM5 simulator [11] to *close the loop* between module-level obfuscation and its application-level impact. To do so, we incorporated various locking techniques within an 8086 processor netlist. A fault analysis of the netlist was performed for a given key and the errant minterms remaining in the circuit were incorporated into the GEM5 simulator model of the core. Workloads could then be run on the GEM5 model of the locked netlist to evaluate the application-level impact of module-level locking.

We configured our simulator to mimic a logically obfuscated 8086 core running a Linux operating system. We performed our architectural benchmarking using benchmarks from the PARSEC benchmark suite [16] and aggregated the results of 40 monte-carlo simulation runs of each benchmark and processor configuration to quantify the impact logic obfuscation had on timing overhead, error rate, and error severity. We define error severity as the number of operations successfully executed until an unrecoverable error occurs in the processor, indicating the amount of work completed before obfuscation related errors derail the core. We define error rate as the percent of benchmark runs which do not complete successfully due to errors injected from logic obfuscation.

B. Simulation Results

We have investigated the architectural impact of both state-of-the-art combinational locking and memory locking. In this section, we present the simulation results derived using the methodology laid out in Section IV-A. For space, we only include results for the optimal configuration of each technique in Figures 1 and 7. In our consideration of memory locking, we target on-chip memory modules used for the integer register file, floating point register file, data cache tag, and data cache data field. We model memory locking using randomly selected keys of various lengths at each of these locations. We initially consider memory locking in a single location.

Stand Alone Memory Locking: Based on simulation results for each memory locking location, locking the data cache tag performed optimally. As seen in Figure 7, any key length greater than 32-bits yielded 100% application-level error rates for each workload within the first 100 operations executed. This implies that minimal useful work could be performed. Compare this to the performance of conventional logic obfuscation in Figure 1 where 0% application-level error rates were recorded after a SAT-based attack despite 3 locking locations.

In addition to the application-level security of memory locking, the incurred timing overhead was also much lower than the SFLL locked processor as shown in Figure 6. For brevity,

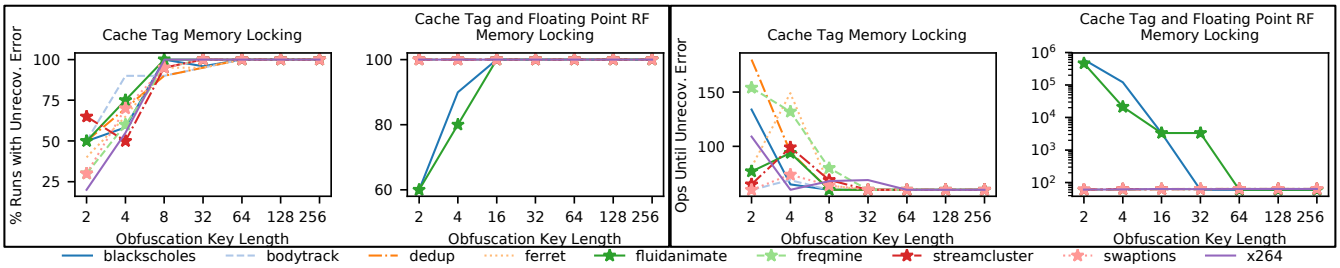

Fig. 7: Error rate and severity results for optimal memory locking configurations of 8086 core running PARSEC suite.

we have only shown results for 100% unique ϕ value memory locking, the worst case timing overhead. These significantly reduced overheads are due to the inclusion latency hiding techniques (i.e. out-of-order processing, cache-banking, etc.) within the IC. These techniques, architected to hide memory latency, were able to amortize the 14.1% latency overhead from memory locking for all locations except integer register file locking. This is unsurprising as the integer register file latency dictates the cycle time of this processor; therefore, no overhead could be amortized. Data cache tag memory locking exhibited only a 2.8% runtime overhead compared to the 5.4% overhead reported by SFLL, a significant improvement.

Paired Location Memory Locking: We performed pairwise simulations of all combinations of 2 memory locking locations with an equal key distribution to investigate the effect of locking diversity on application-level security. We found the optimal pairwise configuration to be cache tag and floating point register file memory locking. Multiple locking locations as a whole appear to perform better than their single location counterpart. This is likely due to the diversity in processor functionality locked, thereby inducing more diversity in errors. When considering the optimal stand alone and pairwise memory locking configuration, as shown in Figure 7, the benefit of diversity seems reduced. Cache tag and floating point register file locking with a 128-bit key caused 100% of workloads to encounter an unrecoverable error within the first 100 operations. This performance is similar to stand alone cache tag locking; however, notice that the pairwise application-level error rate is higher regardless of key length. When considering the amortized timing overhead in Figure 5, the optimal pairwise and stand alone memory locking configurations perform similarly, with paired locking exhibiting a 2.9% overhead compared to the 2.8% overhead of stand alone data cache tag locking. As before, memory locking appears to significantly outperform SFLL regardless of configuration.

Combinational Obfuscation: In Section II, we used SFLL [1] and Anti-SAT [7] to lock both a M68000 and MIPS netlist in 3 locations using the methodologies proposed by the authors. We demonstrated a successful attack methodology to unlock the control path circuitry and an approximate attack to partially unlock the data path circuitry. Even with module-level error remaining in the data path circuitry, simulation results demonstrated a 0% application-level error rate after attack in all 4 cases, as seen in Figure 1.

V. Conclusion

In this work, we demonstrated the limitations of combinational, module-level logic obfuscation techniques for processor design obfuscation. In response to these limitations, we have proposed a logic obfuscation technique to obfuscate SRAM circuitry, called memory locking, and a corresponding automated implementation methodology. Using system-level simulations, we evaluated memory locking and state-of-the-art combinational logic obfuscation at the architecture-level. Based on our results for the simulated architecture, we found 64-bit cache tag/64-bit floating point register file combined memory locking, or 128-bit cache tag memory locking to perform optimally. However, other architectures and techniques may result in different optimal configurations.

VI. Acknowledgements

This work was supported by a joint University of Maryland and Northrop Grumman Corporation seedling grant along with the Air Force Office of Scientific Research under grant FA9550-14-1-0351.

References

[1] M. Yasin *et al.*, "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 1601–1618.

[2] S. S. Technology. (2012). Why node shrinks are no longer offsetting equipment costs, [Online]. Available: http://electroiq.com/blog/2012/10/why-node-shrinks-are-no-longer-offsetting-equipment-costs/.

[3] M. Rostami *et al.*, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, pp. 1283–1295, 2014.

[4] P. Subramanyan *et al.*, "Evaluating the security of logic encryption algorithms," in *2015 Hardware Oriented Security and Trust (HOST)*, IEEE, 2015, pp. 137–143.

[5] K. Shamsi *et al.*, "Appsat: Approximately deobfuscating integrated circuits," in *Hardware Oriented Security and Trust (HOST)*, IEEE, 2017, pp. 95–100.

[6] Y. Xie *et al.*, "Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction," in *Proceedings of the 54th Annual Design Automation Conference*, ACM, 2017, p. 9.

[7] ——, "Mitigating sat attack on logic locking," in *Conference on Cryptographic Hardware and Embedded Systems*, 2016, pp. 127–146.

[8] M. Yasin *et al.*, "Sarlock: Sat attack resistant logic locking," in *Hardware Oriented Security and Trust (HOST)*, IEEE, 2016, pp. 236–241.

[9] A. Chakraborty *et al.*, "Timingsat: Timing profile embedded sat attack," in *International Conference on Computer-Aided Design*, 2018, 6:1–6:6.

[10] J. Rabaey, *Low power design essentials*. Springer Science & Business, 2009.

[11] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] X. Li *et al.*, "Application-level correctness and its impact on fault tolerance," in *High Performance Computer Architecture*, 2007.

[13] A. Chakraborty et al. *et al.*, "Gpu obfuscation: Attack and defense strategies," in *Design Automation Conference*, ACM, 2018.

[14] J. Rajendran *et al.*, "Fault analysis-based logic encryption," *IEEE Transactions on computers*, vol. 64, no. 2, pp. 410–424, 2015.

[15] K. Asanović *et al.*, "Instruction sets should be free: The case for risc-v," *University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[16] C. Bienia *et al.*, "The parsec benchmark suite: Characterization and architectural implications," in *international conference on Parallel architectures and compilation techniques*, ACM, 2008, pp. 72–81.

[17] M. R. Guthaus *et al.*, "Openram: An open-source memory compiler," in *International Conference on Computer-Aided Design*, 2016.

[18] K. Bhanushali *et al.*, "Freepdk15: An open-source predictive process design kit for 15nm finfet technology," in *International Symposium on Physical Design*, ACM, 2015, pp. 165–170.

[19] H. Zhou *et al.*, "Cycsat: Sat-based attack on cyclic logic encryptions," in *International Conference Computer-Aided Design*, 2017, pp. 49–56.