# Scaling Logic Locking Schemes to Multi-Module Hardware Designs

Dominik Šišejković[1], Farhad Merchant[1], Lennart M. Reimann[1], Rainer Leupers[1], and Sascha Kegreiß[2]

[1] Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Germany
{sisejkovic, merchantf, reimannl, leupers}@ice.rwth-aachen.de
[2] Hensoldt Cyber GmbH, Germany
sascha.kegreiss@hensoldt-cyber.com

**Abstract.** The involvement of third parties in the integrated circuit design and fabrication flow has introduced severe security concerns, including intellectual property piracy, reverse engineering and the insertion of malicious circuits known as hardware Trojans. Logic locking has emerged as a prominent technique to counter these security threats by protecting the integrity of integrated circuits through functional and structural obfuscation. In recent years, a great number of locking schemes has been introduced, thereby focusing on a variety of security objectives and the resiliency against different attacks. However, several major pitfalls can be identified in the existing proposals: (*i*) the focus on isolated and often small circuit components, (*ii*) the assumption of unrealistic attack models that enable powerful attacks on logic locking and (*iii*) the design of very specific locking schemes targeted towards achieving resilience against specific attacks. These observations strongly impair the practicality of logic locking. Therefore, in this paper we present a holistic framework for scaling logic locking schemes to common multi-module hardware designs, thereby showcasing an industry-ready pathway of applying logic locking in a realistic design flow. The framework represents an enhancement of the previously published Inter-Lock methodology, offering several algorithmic improvements as well as toolflow implementation details to facilitate the applicability of the framework to large multi-module designs. The framework is tested and evaluated on a real-life 64-bit RISC-V core.

**Keywords:** Hardware security · Processor cores · IC Design Integrity · Locking Framework · RISC-V.

## 1 Introduction

The Integrated Circuit (IC) design and fabrication flow is nowadays heavily driven by third party Intellectual Property (IP) and outsourcing the fabrication to off-site foundries. This business model reduces the total IC design and fabrication cost, and shortens the time-to-market enabling companies to stay competitive in the semiconductor industry. However, the involvement of untrusted

third parties has raised countless security concerns, ranging from IP piracy to the insertion of hardware Trojans [13].

As a reaction to the security threats, various design-for-trust countermeasures have been introduced, including logic locking, IC camouflaging [14], watermarking [7], split manufacturing [2] and IC metering [4]. Logic locking is identified as a premier technique to protect the integrity of IC designs, as it can protect against adversaries located anywhere in the IC supply chain. The core idea of logic locking is the insertion of additional obfuscation logic into a gate-level netlist in order to make the original design functionally dependent on a secret key [15]. Since the key is only known to the IP owner, the design remains concealed while being in hands of external parties.

**Motivation:** Despite the tremendous amount of proposed logic locking solutions in the past, the ever increasing amount of key-recovery attacks represent a serious challenge to designing practical and resilient locking schemes. Moreover, several major pitfalls can be identified in the existing proposals:

- *Isolation:* Existing proposals mostly focus on isolated and often small circuit components or sequential circuits treated as a single component (e.g., singular gate-level netlist). This has the major drawback that the attack complexity relies on the security of the most vulnerable component. Therefore, all components can be attacked independently.
- *Inflexibility:* Modern designs typically include multiple isolated but functionally interconnected components that we refer to as modules (e.g., controller or decoder in a processor). Treating the complete designs as a single isolated component disables the applicability of expert knowledge about the IP. Often, it is necessary to adapt the security measures for specific components depending on their position, significance or exposure in the design. For example, some components might be more vulnerable to a specific attack, requiring dedicated security enhancements. Moreover, by adapting the overhead of the implied security measure in selected components, the overall power dissipation, chip area or performance can be steered to fulfill the desired customer requirements. If the design is seen as a single isolated unit, the mentioned adaptations become significantly more difficult to implement.
- *Impracticality:* Existing logic locking proposals are focused on thwarting specific attack vectors or achieving particular security objectives [1, 15]. This hampers the practicality of logic locking, as its applicability depends on achieving very specific goals, instead of offering a general solution.

**Contribution:** To address the mentioned pitfalls, in this paper we introduce a holistic framework that enables the applicability of any logic locking scheme to modern multi-module hardware designs. Hereby, the focus of the work is not to design a specific locking scheme, but rather to present a methodology of *scaling logic locking* to multi-module hardware designs in a practical design flow, thereby taking the *complexity* and the *interdependent* nature of modern designs into account. The main contributions of this work include the following:

- Based on the proposed Inter-Lock methodology [10], we present and discuss multiple enhancements of the framework in regard to optimizing the inte-

gration of security features into a multi-module design, thereby focusing on the exploitation of existing design interdependencies.
– We showcase the framework in a realistic and industry-ready scenario based on a 64-bit RISC-V processor, thereby discussing the implementation, configuration and realization of the toolflow from a practical point of view.
– We evaluate the security-cost trade-off implied by the framework on the selected real-life case study.

The rest of this article is organized as follows. Section 2 introduces the background on logic locking. The framework improvements, setup and application are presented in Section 3. The evaluation results are discussed in Section 4. Related work is introduced in Section 5. Finally, the paper is concluded in Section 6.

## 2 Preliminaries

### 2.1 Logic Locking

A major type of logic locking is referred to as *combinational* logic locking. This locking type performs design manipulations of the combinational path of integrated circuits. The core idea is the extension of Boolean functions with redundant logic that is bound to an activation key. If the correct key is provided, the design performs as originally intended. Otherwise, an incorrect key ensures the generation of faulty outputs for at least some input patterns. Combinational logic locking is typically applied to a gate-level netlist representation of a design by inserting different types of key-controlled gates into specific locations in the netlist. These gates are referred to as *key gates*.

As an example, let us consider the random locking scheme known es EPIC [9]. This scheme is based on the insertion of XOR and XNOR (XOR + INV) gates at random locations in the design. An XOR gate buffers the second input when its first input is fixed to 0. Same is true for an XNOR gate and the fixed input value 1. Following this rule, the XOR/XNOR gates can be disseminated in the netlist, thereby preserving the original functionality when a correct key is given. Due to the presence of INV, an adversary has to guess if the INV is part of the locking or original functionality, i.e., removal attacks are mitigated.

In the past years, a great variety of combinational locking schemes has been introduced, including locking strategies based on AND/OR, XOR/XNOR and MUX gates. A comprehensive overview of the historical evolution of combinational locking schemes can be found in [15].

In the current literature, logic locking is also referred to as "logic encryption" or "logic obfuscation". In this work, the term "logic locking" implicitly refers to combinational logic locking.

### 2.2 Logic Locking in the IC Design Flow

The IC design and fabrication flow including logic locking is presented in Figure 1. The flow consists of two regimes: the trusted and the untrusted. The

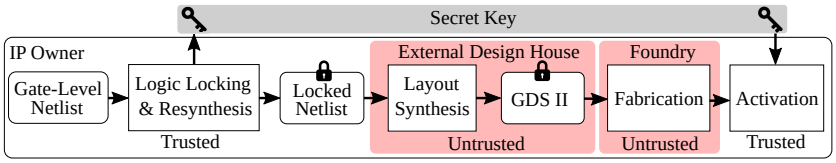



Fig. 1: Logic Locking in the IC Design Flow

trusted regime incorporates the tools and personnel involved in the design of the original IP. This includes the design of the register-transfer level and the initial logic synthesis (not shown in Figure 1) as well as the application of logic locking to the gate-level netlist. Note that after locking, the netlist typically needs to be resynthesized once more to incorporate the changes. The untrusted regime includes an external design house (for the layout synthesis) and the foundry. Hereby, logic locking protects the integrity of the original IP by binding its functionality to a secret key.

**Attack Model:** Finding the secret key is the first barrier that needs to be overcome by an adversary to successfully unlock and understand the design. Typically, it is assumed that an adversary has the locked netlist as well as an activated IC with oracle I/O access (available from the semiconductor market) at his disposal. This combination enables a great variety of powerful key-recovery attacks [1, 11]. However, it has recently been shown that the key storage itself can be compromised by probing attacks or key-extraction hardware Trojans, thereby gaining access to the key without the necessity to formally attack the locking scheme [3, 8]. Moreover, most attacks rely on having full access to a scan chain. However, genuine IC vendors typically do not leave a scan chain open (especially in security-critical IPs) or simply use a secured scan chain [5]. Based on these observations, we limit our approach to the following *realistic* attack assumption: the adversary has only access to the locked netlist. Therefore, the locking mechanism is effective only for the first batch of produced ICs before an activated IC with the *identical* locking mechanism and key is available on the market.

## 3 The Inter-Lock Framework for Processor Cores: A Practical Approach

In this work, we present the practical implementation details of the critical steps of the Inter-Lock [10] flow as well as several algorithmic improvements that enable a targeted applicability of the approach to any hardware design. With the provided details, we bridge the gap between a theoretical locking scheme and its applicability to a large-scale multi-module design.

**RISC-V Case Study:** All details and improvements are presented through a case study based on the open-source 6-stage in-order Ariane processor [16]. This core implements the 64-bit RISC-V instruction set [12].

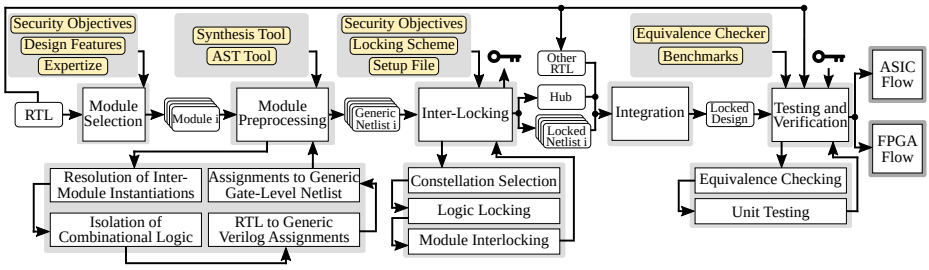



Fig. 2: The Inter-Lock Flow

**Framework Setup and Implementation:** Inter-Lock is designed to exploit the inherent interdependent nature of common hardware designs; different modules (components) communicate with each other through forward and backward connections (similar to forwarding in pipelined processors). To utilize this existing feature for security purposes in addition to locking, Inter-Lock *adapts* the original functionality of every selected module in a design to generate a subset of the activation key for other modules. This creates a *security dependence* between selected modules; only if one module is correctly activated, its co-dependent modules can be unlocked as well. This functional dependence has the following consequences:

- Since part of each module key is internally derived, an adversary is not able to distinguish between common and key inputs.
- The overall design functionality depends on the correct activation of all components. A single incorrect key in one module creates a chain reaction of functional failures throughout all security-dependent modules.
- To correctly unlock a design, the adversary needs to attack all modules at once. This holds as well in case of attacks that include an activated IC with a closed scan chain.

The complete Inter-Lock flow is presented in Figure 2. The input is a hardware design description on Register-Transfer Level (RTL). In our case, this is the complete Ariane core available in System Verilog. The output is the same core with the embedded security features. All intermediate steps are described in the following sections in more detail, thereby following the flow in Figure 2.

### 3.1 Module Selection

The module selection incorporates the selection of design modules that will be included in the locking procedure. A module defines an enclosed (System) Verilog module. For example, a common processor design typically consists of modules such as a decoder, ALU, controller and others. Any number of modules can be selected for the locking. However, at this stage, it makes sense to only select the modules that are critical in terms of security to mitigate the area/power/delay overhead implied by the locking scheme. For a module to be eligible for further

processing, it must not contain further instantiations of other modules or any sequential elements (registers). The reason for these requirements is that logic locking typically works on combinational paths. Therefore, we focus our flow on purely combinational modules. Even though this decision seems fairly limiting, with a few simple adjustments, any module can be transformed to a combinational one, as discussed in the next section. For the purpose of the case study, we selected all Ariane modules shown in Table 1.

### 3.2 Module Preprocessing

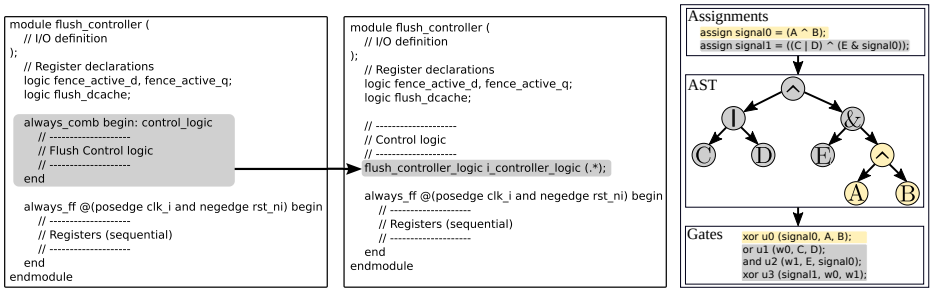
This step prepares a set of selected modules for the locking procedure. The input to this stage is a set of RTL modules, while the output is a set of the same modules in a generic gate-level netlist format. In our case, the generic netlist is represented with simple gate primitives that are defined in the Verilog standard. The preprocessing consists of the following steps: (*i*) the resolution of inter-module instantiations, (*ii*) the isolation of combinational logic, (*iii*) the transformation from RTL to generic Verilog assignments and (*iv*) the transformation from assignments to a generic gate-level netlist.

**Resolution of Inter-Module Instantiations:** One module can include multiple instantiations of other modules within its body, especially since we are still operating on RTL at this point. The construct of an instantiation is not compatible with logic locking. Therefore, we need to resolve it. Two viable options are available. The first option includes flattening the module during the process of mapping to a generic library (addressed in the next section). The second one includes temporarily commenting the instantiation while the module proceeds in the framework flow. Afterwards, the instantiation can be re-embedded into the code. In our flow, we proceed with the first option.

**Isolation of Combinational Logic:** Typically, a single module consists of a combinational path and sequential elements. To simplify the locking procedure that is drafted for combinational logic, we propose the structural isolation by creating internal wrappers for the purely combinational path. An example is shown in Figure 3 (a). Here, the flush controller logic of the Ariane core is extracted

Table 1: Ariane Combinational Modules

| IC | Abbreviation | #Inputs | #Gates | #Outputs |
|---|---|---|---|---|
| flush_controller_logic | f_ctrl | 145 | 22 | 11 |
| csr_buffer_logic | c_buff | 222 | 134 | 90 |
| instruction_scan | i_scan | 32 | 240 | 139 |
| instruction_realigner_logic | i_real | 183 | 629 | 276 |
| compressed_decoder | c_dec | 32 | 848 | 34 |
| commit_stage | commit | 985 | 1584 | 417 |
| branch_unit | br_unit | 342 | 1655 | 328 |
| decoder | decoder | 518 | 2169 | 362 |
| pc_select | pc_sel | 521 | 3333 | 128 |
| branch_prediction | br_pred | 814 | 4669 | 333 |
| alu | alu | 206 | 7412 | 65 |




(a) (b)

Fig. 3: Examples: (a) Isolation of Combinational Logic and (b) Assignment to Gate-Level Transformation

and separated from the sequential part into the module file *flush_controller_logic*. This step is repeated for the following modules as well: *csr_buffer_logic* and *instruction_realigner_logic*.

**Transformation from RTL to Generic Verilog Assignments:** To enable a smooth transition from RTL to a generic gate-level netlist, we rely on the utilization of an intermediate verilog assignments format. These assignments enable the generation of a generic netlist, thereby decoupling the design from any specific technology. To generate a gate-level netlist, the RTL design must be synthesized according to a technology library. Afterwards, the produced netlist can be processed either as technology dependent or stored in a technology-independent format. The latter has the benefit of remaining independent of any technology or tool specifications, i.e., no specific technology library is necessary. Therefore, the design can proceed with any design flow (e.g., ASIC or FPGA). In our case study, we utilize the Synopsys Design Compiler (DC) to map the RTL to a *generic* library. Afterwards, DC can be instructed to store the generated netlist in a Verilog format that results in simple assignments. In principle, any synthesis tool and technology library can be used for this step.

**Transformation from Assignments to Generic Gate-Level Verilog:** The final preprocessing step transforms the assignment-level Verilog into a generic gate-level netlist. To perform this task, we utilize the open-source PyVerilog library. This library is able to parse a Verilog file and represent it as an Abstract Syntax Tree (AST). Through a simple traversal of the AST, we map the assignments to primitive Verilog gates. An example transformation is shown in Figure 3 (b).

### 3.3 Inter-Locking

Once the generic gate-level netlists are prepared, the next step includes the setup and execution of the Inter-Lock procedure. The procedure consists of three major parts: (*i*) constellation selection, (*ii*) application of a logic locking scheme and

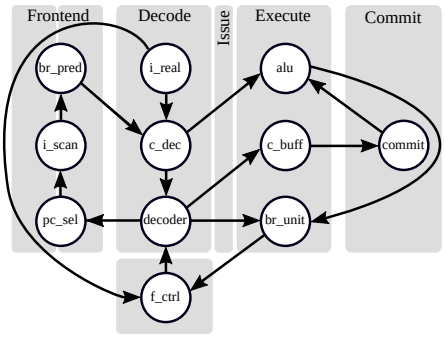



Fig. 4: Example: Constellation Selection

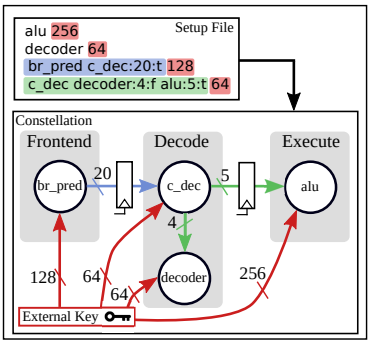



Fig. 5: Example: Constellation Setup File

(*iii*) module interlocking (dependence creation). All three steps are described in the following.

**Constellation Selection:** The first step defines the security interdependence between modules, i.e., which modules influence the correct activation of other modules in a design. The goal of defining a constellation is to create a cyclic interdependence, meaning that every module depends on the activation of every other module. To understand the principle, we represent all modules as a dependency graph in Figure 4. A node represents a *single module* and an arc represents a *security dependency*. This dependency is defined by a source and a sink module, *where the activation of the sink depends on the activation of the source*. In other words, once a source module is correctly activated, it generates the correct internal keys for the sink module. The nodes use the abbreviated naming scheme defined in Table 1. Moreover, the nodes in the example are placed according to their position in the processor pipeline. To ensure that an adversary has to consider all modules at once in an attack, all nodes must be included in the dependency graph. This can be done by selecting a constellation in which every node has at least one input and one output arc. If this is achieved, starting from any node, a dependency chain can be traced back, covering all other nodes in the constellation.

Besides the security dependencies, the constellation selection also includes the selection of the external key length for each module. To support a simple setup of a selected constellation, we propose a setup file consisting of multiple entries of the following format:

$< source >$ $[< sink : num\_interlocks : use\_reg >]$ $< key\_len >$, where:

- $< source >$: Name of source module.
- $[< sink : num\_interlocks : use\_reg >]$: Optional list of all sink modules for which the source module generates internal keys (known as interlocks), where:
  - *sink*: Name of sink module.
  - *num_interlocks*: Number of internal key inputs to be generated.

  - *use_reg*: Defines if a register should be placed between the source and sink module for every interlock.
- $< key\_len >$: Total external key length for the source module.

Using this format, every selected module needs to be described in the file. In other words, the file must contain as many lines as there are modules, as every module must be listed as a source at least once. A simple example is provided in Figure 5.

*Key Length:* Note that if a particular module is dependent on others, its *total* key length equals the sum of its *external* key length and the number of interlocks (internal key) from all other modules it is influenced by. For example, the total key length of *c_dec* is $20 + 64 = 84$ bits. This implies that the total *external* key length of the whole design is, in fact, smaller than the actual key, as the internal keys are hidden through interlocks. Compared to the fixed constellations of the previous work [10], this format enables a more flexible application of the locking mechanisms drafted specifically for a selected architecture. Hereby, the designer can follow a few simple rules while drafting a setup (in this case, biased towards processor designs):

- Create dependencies between modules that are near each other, e.g., at most one pipeline stage apart. This follows the natural implementation of a pipelined core, without raising suspicion.
- Place registers between modules that are naturally divided by a pipeline stage (e.g., between *br_pred* and *c_dec*).
- Register placement is not required if the source and the sink are both in the same pipeline stage and a functional connection already exists; for example, if the output of the source is directly driving the sink in the original design (e.g., between *c_dec* and *decoder*).
- Register placement is not required if the source and the sink are in different pipeline stages where a dependency creates a forwarding path (e.g., between *decoder* and *pc_sel*).

**Application of Logic Locking:** This step includes the application of a selected locking scheme to all preprocessed input modules. Since the framework itself is independent of the actual locking scheme that is applied, any scheme can be selected at this point. To perform the locking, the toolflow has to calculate the correct total key length for each source module (sum of external and all internally-derived keys). Afterwards, a selected locking scheme is applied using this particular key length, where the key itself can be randomly generated or predefined. Moreover, the process of creating interdependencies (internal keys) does not interfere with the actual locking mechanism, since the locking only cares about the key input itself rather than how the key is derived. A modular implementation enables a simple switching of locking schemes and the targeted application of specific schemes to specific modules. This can especially be useful in the case when a module is more exposed (e.g., to the primary inputs or outputs), thereby being more susceptible to selected attacks. For the case study, we selected the simple random locking scheme described in Section 2.1. This

scheme is a superset of other XOR-based locking schemes as it disseminates XOR/XNOR gates on random locations, i.e., without making biased decisions. Therefore, it is a valid selection for the cost evaluation.

**Module Inter-Locking:** Inter-locking is defined as the procedure of adapting the functionality of the source and sink modules to generate the security dependencies defined in the selected constellation. The input to this stage is a set of already locked modules. The idea is as follows. A source module must generate correct and constant output keys for all its sink modules once activated. Note that this is true only for a correct key; otherwise, the output keys are changing based on the circuit input. This internal key bits are referred to as *interlocks*. To perform this task, the inter-locking procedure integrates an additional *Inter-Locking Circuitry (ILC)* to the source module. The properties of the ILC can be summarized as follows:

- If the source itself is not correctly activated, the ILC generates incorrect outputs. On the contrary, if the source is activated, the output keys must be correct and constant. This is achieved through a random Boolean function whose output depends on a subset of the key inputs of the source.
- The ILC is indistinguishable from the rest of the source implementation. This is achieved by binding the functionality and structure of the ILC to the original source functionality. More details can be found in [10].

*Activation Procedure:* An important part of Inter-Lock is the activation procedure of the whole design. Compared to the existing work, we introduce several implementation details that enable a smooth design activation in terms of logic locking. As previously, we focus on the Ariane core. For a correct activation of the core, the correct external key must be provided to all locked modules before the execution starts. This is performed by setting the *reset* signal for a given amount of cycles. Afterwards, the reset is lifted and the core starts executing. Depending on how the inter-locking procedure is implemented, the activation can fail if not facilitated with multiple "free" cycles. We propose the following activation sequence strategies:

- Cycle-Preventive Insertion: Let us assume that two modules influence each other, i.e., they both act as source and sink. This constellation can create hazardous combinational loops and an unstable activation sequence. The latter can occur if both activations are at all times incomplete because both modules never become fully activated (one is waiting for the other and vice versa). This can be prevented by avoiding the usage of key outputs (interlocks) of the first module in the input cone of the ILC of the second module. On one hand, this cycle-preventive insertion is impairing the unrestricted dissemination of key gates, thereby having a negative impact on the underlying locking scheme. On the other hand, this insertion does not necessarily require register placement as cycles are prevented by design. In both cases, if longer security dependencies exist, the core needs *multiple cycles* until a stable activation is reached. This can be achieved by blocking the core for multiple cycles through the *reset* input or by providing multiple NOP instructions before the actual code execution. This activation sequence mimics

a sequential locking mechanism, as the core has to move through different cycles until reaching the correct activated state. Moreover, an interesting observation can be made: it only works if the reset state of the core is *correctly* implemented. We noticed this in different versions of the same Ariane core. One had a faulty reset which led to changed states in the core regardless of the activated reset. Such a behavior impacts the activation procedure as it changes signal values that in turn lead to faulty ILC outputs even if the locking mechanism is correctly implemented. This showcases the tight interleaving of the locking mechanism with the functionality of the core.

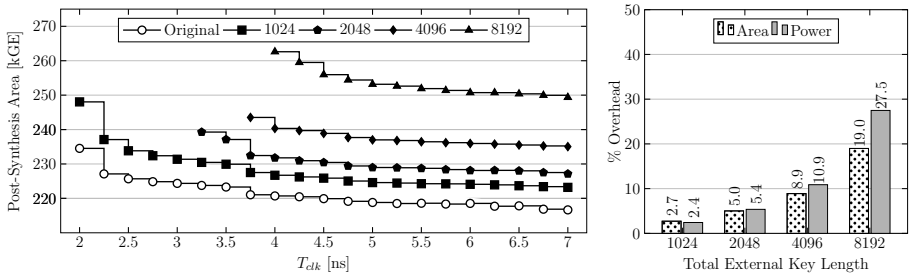
– Exclusive Insertion: This insertion adapts the inter-locking procedure by ensuring that the ILC of every module exclusively depends on external keys. If this is the case, the activation of all modules is performed instantly, i.e., within the *first cycle*. The security of this approach is not affected, since we assume that the ILC is indistinguishable by design. So an adversary has no advantage in detecting the ILC even if exclusive insertion is used.

### 3.4 Integration

Once the inter-locking is performed, the resulting locked modules must be reintegrated into the original design. To facilitate this procedure, the toolflow generates a *hub*; a Verilog module defining the correct wiring based on the selected constellation. The input of this module are all external keys as well as all internal key outputs (interlocks) generated by all source modules. The output consists of all final (internal and external) keys for every module. All these connections might be routed through multiple levels of module instantiations, depending on the location of the locked module. Note that this step also includes the integration of RTL modules that have not been modified by the locking procedure.

### 3.5 Testing and Verification

The result of the integration is a locked design containing RTL and generic technology-independent Verilog netlists. To verify the correctness of the Inter-Lock flow, we resort to equivalence checking and functional unit testing. The equivalence checking is performed using Synopsys Formality, thereby formally comparing the original RTL to the synthesized core with the correct activation key applied. Note that this step is still performed in-house; therefore, the key is known. The equivalence checker proves that the module preprocessing, interlocking as well as integration have not introduced functional errors to the core; i.e., with the correct key, the design is functionally equivalent to the original. Functional testing can further be performed to check the desired functionality of the core. Hereby, we applied the open-source RISC-V test suite containing a set of assembly and benchmark tests [16]. Once the design is verified, it can proceed with either the ASIC or FPGA flow since the Inter-Lock toolflow provides technology-independent locking.




(a) (b)

Fig. 6: (a) Post-Synthesis AT-Plot for Different Total External Key Lengths and (b) Area and Power Overhead at $T_{clk} = 4\ ns$ (250 MHz)

## 4 Cost Evaluation

### 4.1 Experimental Setup

The cost evaluation is performed on the Ariane core, including all modules from Table 1. The main objective of this evaluation is to show the cost impact of locking a variety of critical processor modules. Hereby, it becomes difficult to select all necessary Inter-Lock properties (constellation, number of interlocks, key size and others). Therefore, we propose the following setup. We evaluate a set of fixed total external key lengths (1024, 2048, 4096 and 8192). This implies that, e.g., a key of 1024 bits is divided among all selected modules (similar to the example in Figure 5). For each key length, the constellation is *fixed* to the one shown in Figure 4. The register insertion is done using exclusive insertion, as discussed in Section 3.3. The external key is divided among the modules linearly to the module size (number of gates). As a rule of thumb, we fixed the total number of interlocks generated by a source module to 5% of the amount of its original outputs. In case a source has multiple sink modules, the interlocks are evenly divided among them. Logic synthesis was done with Synopsy DC using the standard-performance cell library for the UMC 90 nm CMOS process operating under typical conditions (1 $V$, 25°$C$). QuestaSim was used for RTL and gate-level simulation. A security analysis of the approach is available in [10].

### 4.2 Evaluation Results

We performed an Area vs Time (AT) evaluation to compare the influence of various total external key lengths to the original design. The results are presented in the AT-plot in Figure 6 (a). The area is shown in Gate-Equivalent (GE) and the clock period ($T_{clk}$) in $ns$. One GE is the area of one 2-input drive-1 NAND gate. The original design achieves a minimum $T_{clk}$ of 2 ns (500 MHz). As expected, the design area as well as the minimum $T_{clk}$ are increasing with larger keys. The 1024-bit key design is able to achieve $T_{clk} = 2\ ns$, resulting

in 0% delay overhead. However, the 2048-bit, 4096-bit and 8192-bit key designs achieve $T_{clk} = 3.25\ ns$, $T_{clk} = 3.75\ ns$ and $T_{clk} = 4\ ns$ respectively. This implies a delay overhead between 62.50% and 100%.

The second evaluation concerns the cost comparison of the locked variants for $T_{clk} = 4\ ns$. This clock period is achieved for all key lengths. By fixing the clock period, we can take a closer look at the area and power cost differences, as shown in Figure 6 (b). The area overhead ranges from 2.7% (1024-bit) to 19% (8192-bit). As expected, the area overhead doubles (approximately) when doubling the key length. The power overhead increases in line with the area (approximated with DC); from 2.4% (1024-bit) to 27.5% (8192-bit). At $T_{clk} = 4\ ns$, the original design area is 220.71 $kGE$, while the total power is 43.37 $mW$.

The presented results provide us with a closer look at the cost of applying locking schemes to a practical processor design at a larger scale, thereby considering multiple modules, their interdependencies as well as large external keys. Moreover, based on the evaluation, one can choose an appropriate hardware-secured processor design, thereby balancing the area, power and delay overhead against the key length.

## 5 Related Work

A similar framework-based approach is known as MIRAGE [6]. This framework can be used for design space exploration as well as obfuscation strength analysis. However, the focus of the framework lies within the selection and evaluation of specific logic locking schemes applied to isolated components. In comparison, our approach takes a more abstract view of the locking procedure for multi-module designs, regardless of the specific locking scheme. In that regard, MIRAGE can be used for the dedicated selection of locking schemes for each specific component of the overall design within Inter-Lock. Therefore, in this paper, we do not focus on algorithmic details of prior locking schemes, as our approach is decoupled from any algorithmic specifications. An extensive overview of locking schemes can be found in [15], as well as in the prior work [10].

## 6 Conclusion

This paper presents the application of Inter-Lock from a practical point of view, thereby addressing an important security challenge; scaling logic locking mechanisms to multi-module designs under the consideration of their complexity and interdependent nature. We presented multiple framework improvements and provided an in-depth overview of the setup and implementation of the underlying toolflow. All framework components were showcased through a practical case study based on a 64-bit RISC-V processor. Furthermore, we evaluated the cost impact of the approach in terms of area, power and delay overhead compared to various key lengths. The insights provided in this paper offer a first look into the procedure and cost of comprehensively locking a modern processor design. In future work, we plan to perform an evaluation on a multi-core environment.

## References


1. Azar, K.Z., Kamali, H.M., Homayoun, H., Sasan, A.: Threats on logic locking: A decade later. GLSVLSI 2019 pp. 471–476 (2019). https://doi.org/10.1145/3299874.3319495
2. Imeson, F., Emtenan, A., Garg, S., Tripunitara, M.: Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In: 22nd USENIX. pp. 495–510. USENIX, Washington, D.C. (2013)
3. Jain, A., Zhou, Z., Guin, U.: Taal: Tampering attack on any key-based logic locked circuits. ArXiv **abs/1909.07426** (2019)
4. Koushanfar, F.: Provably secure active IC metering techniques for piracy avoidance and digital rights management. IEEE TIFS **7**(1), 51–63 (Feb 2012). https://doi.org/10.1109/TIFS.2011.2163307
5. Lee, J., Tebranipoor, M., Plusquellic, J.: A low-cost solution for protecting IPs against scan-based side-channel attacks. In: 24th IEEE VTS. pp. 6 pp.–99 (April 2006). https://doi.org/10.1109/VTS.2006.7
6. Menon, V.V., Kolhe, G., Schmidt, A., Monson, J., French, M., Hu, Y., Beerel, P.A., Nuzzo, P.: System-level framework for logic obfuscation with quantified metrics for evaluation. In: 2019 IEEE SecDev. pp. 89–100 (Sep 2019). https://doi.org/10.1109/SecDev.2019.00020
7. Newbould, R.D., Irby, D.L., Carothers, J.D., Rodriguez, J.J., Holman, W.: Watermarking ICs for IP protection. Electronics Letters **38**(6), 272–274 (Mar 2002). https://doi.org/10.1049/el:20020143
8. Rahman, M.T., Tajik, S., Rahman, M.S., Tehranipoor, M., Asadizanjani, N.: The key is left under the mat: On the inappropriate security assumption of logic locking schemes. Cryptology ePrint Archive, Report 2019/719 (2019), https://eprint.iacr.org/2019/719
9. Roy, J.A., Koushanfar, F., Markov, I.L.: EPIC: Ending piracy of integrated circuits. In: 2008 DATE. pp. 1069–1074 (March 2008). https://doi.org/10.1109/DATE.2008.4484823
10. Šišejković, D., Merchant, F., Leupers, R., Ascheid, G., Kegreiß, S.: Inter-Lock: Logic encryption for processor cores beyond module boundaries. In: 2019 IEEE ETS. pp. 1–6 (May 2019). https://doi.org/10.1109/ETS.2019.8791528
11. Šišejković, D., Leupers, R., Ascheid, G., Metzner, S.: A unifying logic encryption security metric. In: SAMOS 2018. pp. 179–186. SAMOS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3229631.3229636
12. Waterman, A., Lee, Y., Patterson, D., Asanovic, K.: The RISC-V instruction set manual. volume I: User-level ISA, version 2.0, Tech. Rep. UCB/EECS-2014-54 (2014)
13. Xiao, K., Forte, D., Jin, Y., Karri, R., Bhunia, S., Tehranipoor, M.M.: Hardware trojans: Lessons learned after one decade of research. ACM Trans. Design Autom. Electr. Syst. **22**(1), 6:1–6:23 (2016). https://doi.org/10.1145/2906147
14. Yasin, M., Sinanoglu, O.: Transforming between logic locking and IC camouflaging. In: 2015 IDT. pp. 1–4 (Dec 2015). https://doi.org/10.1109/IDT.2015.7396725
15. Yasin, M., Sinanoglu, O.: Evolution of logic locking. In: 2017 IFIP/IEEE VLSI-SoC. pp. 1–6 (Oct 2017). https://doi.org/10.1109/VLSI-SoC.2017.8203496
16. Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit RISC-V core in 22-nm FDSOI technology. IEEE TVLSI **27**(11), 2629–2640 (Nov 2019). https://doi.org/10.1109/TVLSI.2019.2926114