Logic Locking - Connecting Theory and Practice

Elisaweta Masserova[1*], Deepali Garg[1*], Ken Mai[1], Lawrence Pileggi[1],
Vipul Goyal[1,2], and Bryan Parno[1]

[1] Carnegie Mellon University
[2] NTT Research

Abstract. Due to the complexity and the cost of producing integrated circuits, most hardware circuit designers outsource the manufacturing of their circuits to a third-party foundry. However, a dishonest foundry may abuse its access to the circuit's design in a variety of ways that undermine the designer's investment or potentially introduce vulnerabilities.

To combat these issues, the hardware community has developed the notion of *logic locking*, which allows the designer to send the foundry a "locked" version of the original circuit. After the locked circuit has been manufactured, authorized users can unlock the original functionality with a secret key.

Unfortunately, most logic locking schemes are analyzed using informal security notions, leading to a cycle of attacks and ad hoc defenses that impedes the adoption of logic locking.

In this work, we propose a formal simulation-based security definition for logic locking. We then show that a construction based on universal circuits provably satisfies the definition. More importantly, we explore ways to efficiently realize our construction in actual hardware. This entails the design of alternate approaches and optimizations, and our evaluation (based on standard hardware metrics like power, area, and performance) illuminates tradeoffs between these designs.

1 Introduction

While general-purpose CPUs are perhaps the best known computing devices, a growing share of computation now takes place on application-specific integrated circuits (ASICs) [14]. Compared to a general-purpose CPU, an ASIC can improve performance and reduce energy usage by orders of magnitude. For example, Magaki et al. show ASICs outperforming CPUs in energy effiency by 4676×, 685×, and 8516×, for Bitcoin, Litecoin, and video transcoding respectively [23].

However, manufacturing such ASICs is an increasingly difficult and expensive process [35, 39], and major hardware design houses now save money by outsourcing manufacturing to external foundries. Such outsourcing introduces new risks, since the foundry naturally sees the full details of the design, including the netlist (a description of the hardware nodes and their connectivity) and physical layout information. An unscrupulous foundry can use this information to steal the

[*] The first two authors contributed equally to this work.

designer's clever insights, to produce more ASICs than requested (selling the extras illicitly), or to modify the circuit to, e.g., insert a hardware Trojan. These threats undermine the security of the hardware supply chain [15, 29] and the significant cost associated with developing digital circuits [20].

Of the many solutions the hardware community has explored, one of the most prominent is *logic locking* [30]. Informally, the design house applies an algorithm that takes in the design's original circuit along with a secret key and produces a "locked" circuit, which is provided to the foundry. Intuitively, the foundry can manufacture the locked circuit but should learn nothing about the original one, preventing design theft and making overproduction useless. When the designer receives the manufactured circuit from the foundry, they connect it to memory containing the key and apply standard tamper-proofing to the combined device. With the key present, the device provides the same functionality as the original circuit, but the tamper-proof enclosure prevents an end-user from reverse-engineering the design. Note that the use of a symmetric key distinguishes this setting from traditional work on cryptographic obfuscation [4], opening up new opportunities not possible in that more stringent setting.

Since logic locking's inception [30], a series of proposed schemes and attacks on those schemes have followed a pattern familiar from pre-modern cryptography. For example, early logic-locking constructions inserted XOR/XNOR gates at random locations in the circuit, with one input from the secret key and one input from the original circuit's wire [30, 28]. Using the correct key restores the original functionality, while incorrect key bits flip the corresponding wire value. Later proposals based on inserting AND/OR gates [13] or multiplexers [27] followed a similar approach. All provided heuristic arguments about the difficulty of determining the correct key, given only the locked circuit. In 2015, however, Subramanyan et al. [33] introduced a powerful new attack that broke all of these schemes. Subsequent schemes focused on protecting against this particular attack, and in turn fell victim to new attacks. As we discuss in §2.3, a few attempts were made to introduce more formal definitions of security, but these definitions were either too strict to be achievable, or too weak to rule out obvious attacks.

In this work, to break the cycle of ad hoc designs for logic locking, we provide a new simulation-based security definition. Our definition intuitively matches the security expected for logic locking, and importantly, we show by construction that the definition can be satisfied.

Our construction is based on a well-known cryptographic primitive – *universal circuits*. Introduced by Valiant [37], informally, a universal circuit can be programmed to evaluate any circuit up to a given size. As we formally prove, a universal circuit can be used as a secure locked circuit. The proof is straightforward, but it leaves open the question of how to most efficiently instantiate the scheme in practice.

As we show in §6, directly manufacturing an ASIC that executes Valiant's universal circuit construction imposes significant power, area, and performance overheads relative to the original circuit. Hence, we explore the use of *embedded* *field-programmable gate arrays* (e-FPGAs) [1]. An e-FPGA is integrated into an ASIC where certain logic blocks and the connections between them can be dynamically programmed *after* manufacturing. Note that e-FPGAs allow the ASIC system to retain most of the power and efficiency gains provided by ASICs, which sets them apart from standard FPGAs. e-FGPAs also benefit from decades of research and tool development focused on improving their efficiency.

While it is tempting to think that an eFPGA with N programmable logic blocks *is* a universal circuit for circuits of size N, physical restrictions on the connections between the programmable logic blocks make this unlikely (§5.2). However, we prove that Valiant's universal circuit construction [37] can be mapped onto an eFPGA, despite its constraints, showing that an eFPGA, when properly configured, can be treated as universal. We then discuss ways to optimize the number of programmable logic blocks used in the mapping.

Since the mapping above is an indirect route to universality, as a second approach, we design a direct solution, *topological logic locking*, which directly maps any circuit with a given number of inputs, outputs, and gates to an eFPGA structure. Unlike our mapping of Valiant's UC construction, our topological construction applies even to standard eFPGA designs, which only allow a constant number of connections between each logic block.

As we discover, in both constructions, by increasing the number of connections between the logic blocks, we can reduce the number of logic blocks needed for the mapping. Hence either construction can be optimized for number of connections or number of logic blocks.

In summary, we make the following contributions. We:

- Introduce a formal simulation-based security notion for logic-locking schemes.
- Connect the hardware community's logic-locking problem to universal circuits, a topic with a rich history in the cryptographic community. We prove that universal circuits satisfy our security definition for logic locking.
- Show that Valiant's universal circuit can be mapped onto a eFPGA, and thus such eFPGAs can be used as a provably secure logic-locking solution.
- Propose topological logic locking – a (more direct) eFPGA-based construction which can be programmed to implement any circuit up to a given size.
- Evaluate our proposals, as well as straw-man solutions, in terms of standard power, area and performance metrics.

2 Defining Logic Locking

We begin with an informal description of logic locking. Next, we formally introduce the syntax of a logic locking scheme, as well as a basic correctness definition. We then discuss the limitations of previous notions of security. Finally, we propose our own definition for the security of logic locking.

2.1 Problem Overview

With the rise of outsourced hardware manufacturing, a (simplified) circuit fabrication chain typically looks as follows. A *Design House* spends years developing and optimizing a circuit design. They send the circuit design to the *Foundry* which manufactures the physical chip and sends it to the design house. The design house packages the chip into a device and sells the device to the *End User*.

Logic locking aims to secure the outsourcing process outlined above by hiding the circuit's design from a malicious foundry. Simultaneously, it ensures that a legitimate end user is able to execute the circuit on arbitrary inputs. Such functionality is typically implemented by modifying the circuit (the modified circuit is called a *locked* circuit) and introducing additional *key inputs*. Given the correct key and a normal input, the locked circuit produces the same output as the original circuit. Without the correct key, the hope is that the locked circuit does not reveal any information about the original one.

In the following, we assume that the design house is trusted and that the foundry is untrusted. We assume an honest-but-curious foundry, since a fully malicious foundry can always insert a Trojan that leaks the secret key when activated [17], and foolproof prevention or detection of hardware Trojans remains an open problem. Finally, we assume that the end user is untrusted, but has limited capabilities. In particular, we assume that the final device, which contains the manufactured circuit and key, is physically protected against, e.g., probing or tampering attacks.

2.2 Syntax and Correctness

We now introduce the logic locking syntax. As mentioned above, logic locking transforms an original circuit $c_o : I \to O$ into a locked circuit $c_e : I \times K \to O$ by adding a new key input $k \in K$. The original circuit c_o may be combinational (stateless) or sequential (stateful). However, note that a sequential circuit can be split into combinational blocks, where in addition to the input, each circuit block takes in the current state, and produces not only the output itself (if any), but also the new state. Such combinational function blocks can then be executed one after the other (using memory to buffer the states) and locked separately. Thus for simplicity we will focus on combinational circuits.

A combinational circuit can be thought of as a Directed Acyclic Graph (DAG), where each node represents a gate. We take the following syntax for combinational logic locking on DAG circuit families verbatim from Shamsi et al. [31]:

Definition 1 (Combinational logic locking [31]). *A* combinational logic locking *scheme for a family of stateless DAG circuits $\{C_\lambda\}$ is a probabilistic polynomial time (PPT) algorithm* `Lock` *that takes a security parameter λ and an original circuit $c_o \in \{C_\lambda\}$, and returns a key k^* and a locked combinational circuit c_e, such that the following holds:*

- ℓ **Added key inputs.** We have $c_o : I \to O, I = \mathbb{F}_2^n, O = \mathbb{F}_2^m, c_e : K \times I \to O$, and $K = \mathbb{F}_2^\ell$.
- **Correctness under correct key.** We have $\forall i \in I,\ c_e(k^*, i) = c_o(i)$.

– **Polynomial overhead.** We have $size(c_e) \leq poly(size(c_o))$ and $depth(c_e) \leq poly(depth(c_o))$.

We additionally define the following property restricting the key size of a logic locking scheme, which may be needed if the size of the device's tamper-proof memory is limited.

– **Polynomial key size.** We have $\ell = poly(\lambda)$.

2.3 Limitations of Previous Security Notions

As context, we briefly review prior notions of security, highlighting their strengths and weaknesses.

Since the foundry is untrusted, all security definitions assume that the design of the locked circuit is known to the adversary. However, they differ in whether the adversary also has access to an end-user device. Those that assume such access are known as *oracle-guided* attacks, while those without are *oracle-less*.

Ad Hoc Notions For over a decade, the hardware community evaluated the security of logic locking designs based on their resilience against brute force attacks. Later, this was expanded to include resilience against more advanced attacks, e.g., SAT attacks [34]. A SAT attack uses oracle access to produce input-output pairs, which are in turn used to rule out keys that do not produce the same pairs in the locked circuit. A scheme was considered secure if such attacks required time exponential in the key size.

While resistance to SAT attacks is, of course, necessary, it is by no means sufficient. For example, while running the attack to completion may be infeasible, intermediate results may produce keys that are functionally close to the original design. In this case, the adversary need not complete the attack, but rather run until the keys produced exhibit low enough error rates.

Multiple works have attempted to use various circuit parameters of locked design to quantify security under oracle-less attacks. For example, *output corruption* [32] measures the likelihood across all keys and inputs that a locked circuit produces an incorrect answer. However, it is entirely possible that the adversary is interested in the outputs of the circuit only on some application-specific inputs. Furthermore, output corruptibility can be maximized by simply inverting the output bits of the original circuit, and yet such a "locked" circuit would leak the entire original design. Other metrics, such as key sensitivity [36] and structural and functional secrecy factors [38] can expose vulnerabilities in specific locking schemes, but they do not provide a connection to the ultimate security of the scheme.

Formal Notions In the past few years, a few formal notions of security have been introduced. For example, Shamsi et al. [31] propose the following security definitions, where $\mathtt{OG}$ corresponds to oracle-guided attacks, and $\mathtt{OL}$ to oracle-less attacks. An ϵ-approximation of a function f is some function f' that disagrees with f on at most an ϵ fraction of the input space.

Definition 2. *(Approximate Functional Secrecy (AFS) [31]). The adversary $\mathcal{A}$ has c_e, makes up to q chosen input queries to c_o, and then must return an ϵ-approximation of c_o. We say that a scheme is (t, q, ϵ, σ)-*`AFS-OG` *secure if the advantage of any $\mathcal{A}$ bounded by t operations is no more than σ better than the advantage of the adversary $\mathcal{A}$ that makes q queries to c_o and randomly guesses the remaining $2^n - q$ truth-table entries. As for* `OL` *attackers, (t, ϵ, σ)-*`AFS-OL` *$\equiv (t, 0, \epsilon, \sigma)$-*`AFS-OG`*.*

Definition 3. *(Best-Possible Approximate Functional Secrecy* (`BPAFS`) *[31]). The adversary $\mathcal{A}$ has c_e, makes up to q chosen input queries to c_o, and then must return an ϵ-approximation of c_o. We say that a a scheme is (t, q, ϵ, σ)-*`BPAFS-OG` *secure if the advantage of any $\mathcal{A}$ bounded by t operations is no more than σ better than the advantage of any adversary $\mathcal{A}'$ that tries to learn the blackbox c_o with a priori knowledge that depth$(c_o) \leq$ depth(c_e) and size$(c_o) \leq$ size(c_e) through q queries.* [*] *For oracle-less attackers (t, ϵ, σ)-*`BPAFS-OL` *$\equiv (t, 0, \epsilon, \sigma)$-*`BPAFS-OL`*.*

Similar notions have also been proposed by Chhotaray and Shrimpton [11], as well as by Di Crescenzo et al. [12].

Discussion Unfortunately, all of the definitions outlined above are somewhat flawed. For example, as pointed out by Shamsi et al. [31], for some circuit families, AFS-OG is impossible to achieve with an exponentially small adversarial advantage.

More importantly, the intuitive goal of logic locking is that the locked circuit reveals no new information about the original circuit to the adversary. Unfortunately, the definitions outlined above fail to capture this idea – an adversary who can obtain some non-trivial information about c_o (e.g., the last bit of the output for every possible input) but who cannot obtain a representation approximately equivalent to c_o never wins, and thus a scheme which is secure according to this definition might in fact leak information about the original circuit.

2.4 Our Definition of Logic-Locking Security

We now propose our definition of security for logic locking. It is a standard (for the cryptographic community) simulation-based notion. The intuition behind this definition is that having access to the locked circuit does not provide any additional information about the original circuit. In other words, everything the adversary can compute using the locked circuit and oracle access to the original circuit is computable using only oracle access to the original circuit plus any “allowed” leaked information (such as the number of inputs or gates in the original circuit).

Given a family of stateless DAG circuits C_λ, denote the allowed leakage function by $L : C_\lambda \rightarrow \mathbb{F}_2^{ll}$, where ll denotes the length of the leakage. Then, logic locking security can be formally defined as follows.

[*] As the authors note, $\mathcal{A}'$ should be understood as the “strongest learner” of the black-box c_o.

Definition 4. ***Simulation Security.*** *For any PPT adversary $\mathcal{A}$ that has access to $c_e = \texttt{Lock}(\lambda, c_o)$ and is able to make q queries to the oracle $\mathcal{O}_{q,c_o}$ for c_o, there exists a PPT simulator $\mathcal{S}$ that is able to make to q queries to the oracle $\mathcal{O}_{q,c_o}$, such that the following holds*

$$\mathcal{A}^{\mathcal{O}_{q,c_o}}(1^\lambda, c_e, L(c_o)) \approx_c \mathcal{S}^{\mathcal{O}_{q,c_o}}(1^\lambda, L(c_o)),$$

where $X \approx_c Y$ denotes that X and Y are computationally indistinguishable.

Note Concurrent and independent work by Beerel et al. [5] recently appeared. Like us, they propose a simulation-based security definition and suggest instantiating it with a universal circuit. In contrast to our work, however, they do not explore how a universal circuit can be efficiently instantiated in hardware.

3 Building Blocks

We briefly discuss the two main building blocks in our construction (§4). One (universal circuits) comes from the cryptographic community, while the other (eFPGAs) comes from the hardware community.

3.1 Background - Universal Circuits

We can represent any computable function $f(\cdot)$ as a Boolean circuit [26] with n input bits and m output bits. Given an input x to $f(\cdot)$, the corresponding input bits to the circuit are $x = (x_1, \ldots, x_n)$, and likewise for the function's output. Valiant introduced the notion of a universal circuit (UC) [37] that, informally, takes as input both a description of a circuit and an input to that circuit, and then simulates the application of the circuit to the input.

We formalize a UC using a more modern definition from Zhao et al. [40].

Definition 5. *A circuit $\texttt{UC}_s^{n,m}$ is called a* universal circuit*, if for any circuit with n inputs, m outputs, and size (in gates) up to s (denoted by $C_s^{n,m}$), there exists a set of program bits $p \in \{0,1\}^l$ such that $\texttt{UC}_s^{n,m}$ can be programmed to realize $C_s^{n,m}$, i.e., $\forall x \in \{0,1\}^n$, $\texttt{UC}_s^{n,m}(p,x) = C_s^{n,m}(x)$.*

For presentation purposes, we explicitly define the algorithms for generating the UC and for generating a circuit's program bits p.

- $\texttt{ConstructUC}(n,m,s) \rightarrow \texttt{UC}_s^{n,m}$: Given n, m, and s produces a universal circuit for a family $\{C_s^{n,m}\}$.
- $\texttt{ProduceProgram}(n,m,s,C) \rightarrow p$: Given n, m, s, and a circuit $C \in \{C_s^{n,m}\}$, produces a set of program bits p such that for any $x \in \{0,1\}^n : \texttt{UC}_s^{n,m}(p,x) = C(x)$.

In introducing the notion of a UC, Valiant also provided a construction that uses $O(s \log s)$ gates to simulate circuits in $\{C_s^{n,m}\}$, and he proved that this construction is asymptotically optimal [37]. Subsequent work has improved on the constants in the construction [22, 18, 40].

Because one of our solutions depends on Valiant's construction, we briefly summarize it below.

Valiant's Universal Circuit Recall that a circuit can be thought of as a DAG, with circuit gates represented by graph nodes, and circuit wires represented by graph edges. Valiant's construction utilizes this idea and, as a first step, finds a "universal graph", which embeds any DAG of a certain size. Denote DAGs of size z, and fan-in/fan-out d by $\mathtt{DAG}_d(z)$. Then, a "universal graph" is defined as follows.

Definition 6. *(Zhao et al. [40]) An* edge-embedding *σ of $G = (V, E)$ into $G^* = (V^*, E^*)$ is a mapping that maps V into V^* one-to-one, and E into directed paths in G^* (i.e., $(i, j) \in E$ maps to a path from $\sigma(i)$ to $\sigma(j)$) that are pairwise edge-disjoint. A graph G^* is an* edge-universal *graph $\mathtt{EUG}_d(z)$ for $\mathtt{DAG}_d(z)$ if it has z nodes (dubbed* distinguished poles*) $P_1, \ldots, P_z$ such that every $G \in \mathtt{DAG}_{d_0}(z_0)$, with $d_0 \leq d$ and $z_0 \leq z$, can be edge-embedded into G^* by a mapping σ such that $\sigma(i) = P_i$ for each $i \in V$. This should hold for any labeling of G.*

Note that the cryptographic community typically assumes w.l.o.g. that a circuit has fan-in/fan-out $d = 2$. We will follow this approach for now and discuss a possibility for an optimization by changing the fan-in/fan-out in Section 5.3.

Building an $\mathtt{EUG}_2(z)$ involves the following steps.

1. *Construct an $\mathtt{EUG}_1(z)$ given an $\mathtt{EUG}_1(\lceil z/c \rceil - 1)$ for some constant c.*
   For a large z, it is difficult to find a direct EUG construction which avoids large overheads in terms of the size and depth. Thus, constructions typically resort to providing a recursive solution, where an EUG for a larger z is constructed from EUGs for smaller z, repeating recursively until we reach a sufficiently small z for which an EUG can be constructed directly.
2. *Construct an $\mathtt{EUG}_2(z)$ from an $\mathtt{EUG}_1(z)$.*
   Given an EUG for graphs of fan-in/fan-out 1, an EUG for graphs of fan-in/fan-out 2 can be constructed by taking two instances of $\mathtt{EUG}_1(z)$ and merging each pole with its twin. This works because the edge set E of any $\mathtt{DAG}_2(z)$ (V, E) can be split into two edge sets E_1, E_2 such that both (V, E_1) and (V, E_2) are $\mathtt{DAG}_1(z)$. Thus, the original graph can be split into two such sets, each of which can be embedded separately into a single $\mathtt{EUG}_1(z)$. The combination of these two EUGs then results in an EUG which can embed the original graph of fan-in/fan-out 2.

Figure 5 illustrates a recursive EUG construction step.

Given an $\mathtt{EUG}_2(z)$, where $z = s + n$, the universal circuit $\mathtt{UC}_s^{n,m}$ can be constructed by replacing graph nodes by gates and edges by wires. In more detail, the replacement strategy makes use of the restriction of the EUG to fan-in/fan-out two. It replaces each distinguished pole of the EUG with a *universal gate*, which implements any of the 16 functions from two inputs bits to one output bit. While each of the remaining nodes (dubbed *common* nodes) could be implemented by a universal gate as well, there is a cheaper solution. If a common node's in-degree is two, it can be implemented by a switching gate (which requires at most 4 binary gates, in contrast to the 9 required for a universal gate). If the node's in-degree is one, and out-degree is two, it can be implemented by

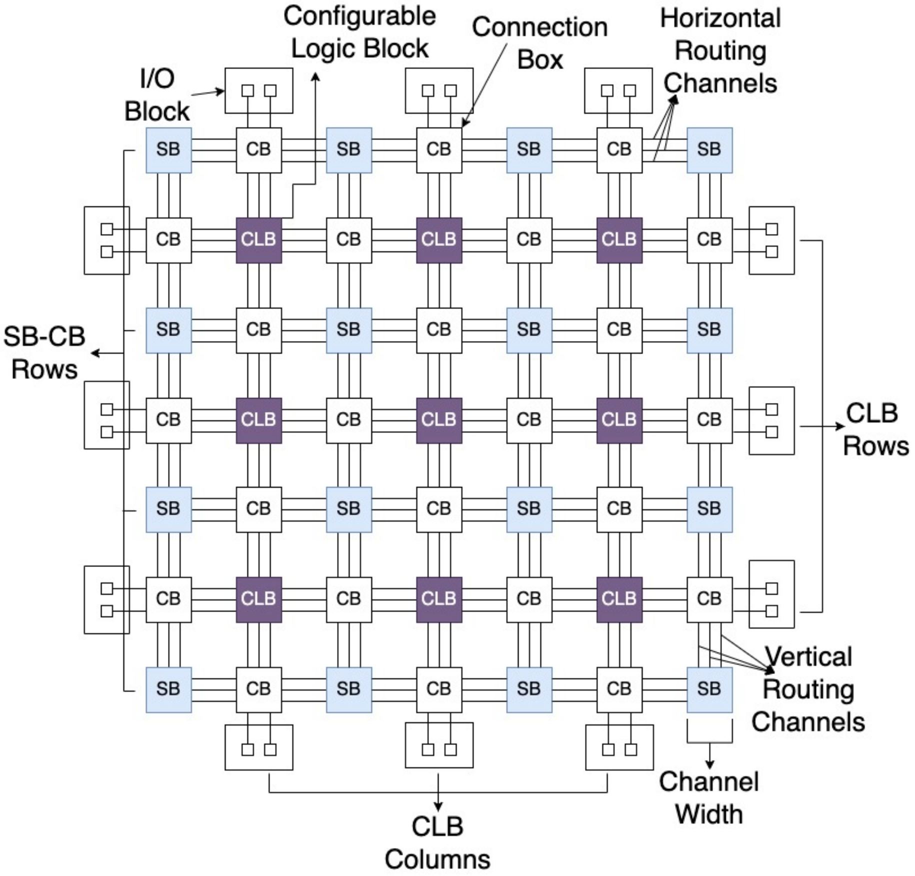


Fig. 1: Overview of Island-style eFPGA architecture

two wires carrying the same signal. If both in-degree and out-degree are one, it can implemented simply with a wire. Each edge in the EUG is then implemented by a wire between the corresponding gates in the universal circuit.

Intuitively, this results in a circuit which can embed any circuit with n inputs, m outputs, and s gates. Each input and each gate of the original circuit can be simulated by one of the $z = s + n$ universal gates. Since an EUG can embed any DAG and the original circuit can be thought of as a DAG, there exists a connection between the embedding of any two gates of the original circuit (implemented using universal gates), as there is a connection between the corresponding two poles of the EUG. These connections are furthermore *guaranteed to be pairwise edge-disjoint*, as the same holds for the corresponding connections in the EUG. Thus, we never use the same wire to forward two different signals.

Overall, this approach produces a universal circuit with size $O(s \log s)$ and depth $O(s)$ [37].

3.2 Background - eFPGAs

In the hardware world, an intuitive analog of a universal circuit is a Field-Programmable Gate Array (FPGA). After manufacturing, an FPGA can be dynamically configured into particular circuit configuration. This can provide significant performance and power efficiency improvements compared to a general-purpose CPU [2], particularly for data-parallel workloads. However, the programmability comes at a cost: for a given configuration, an FPGA is typically 3-4x slower and consumes about 14x more power and area than a corresponding ASIC [21].

An embedded FPGA (eFPGA) [1], however, aims to provide provide the best of both worlds, namely the flexibility of an FPGA and the performance and efficiency of an ASIC. eFPGA is a small FPGA IP core, which can be tuned to a small size and integrated into the ASIC SoC, this integration allow the overall system to maintain high performance and energy efficiency.

The most common eFPGA architecture consists of a two-dimensional array of configurable logic blocks that are interconnected through a programmable routing network. Figure 1 illustrates the typical "island-style" design, which consists of the following building blocks.

1. Configurable Logic Blocks (CLBs): A CLB (shown in Figure 2) implements the basic logic for an application design. It consists of an array of logic elements called Look-up Tables (LUTs). An n-input LUT can be configured to compute any function $C \in L : (0,1)^n \rightarrow (0,1)$, depending of its configuration $p \in \{0,1\}^{2^n}$. For example, a 2-input LUT, with (x_1,x_0) inputs, and (p_3,p_2,p_1,p_0) configuration implements logic L as $L(x_1, x_0)_{(p_3,p_2,p_1,p_0)} = x_1 \cdot x_0 \cdot p_3 + x_1 \cdot \bar{x}_0 \cdot p_2 + \bar{x}_1 \cdot x_0 \cdot p_1 + \bar{x}_1 \cdot \bar{x}_0 \cdot p_0$.
2. I/O Blocks: The input/output blocks connect the eFPGA to the outside world. The I/O blocks are located around the periphery of the chip, providing programmable I/O connections and support for various I/O standards.
3. Routing Network: The routing resources comprise vertical and horizontal channel tracks (routing wires) which run through the length and width of the eFPGA fabric. These channels connect the CLBs to each other and to the I/O blocks. The connections are facilitated through connection boxes and switch-boxes which are used to re-direct signals. The routing is entirely reconfigurable subject to the constraint that two signals can never be merged. That is, each wire can only carry one signal at a time.
   (a) Connection Box (CB): The connection boxes connect the CLBs and I/O blocks within the horizontal channel or vertical channels they lie on. An example of a connection box for channel width two is shown in Figure 3, where each of the channel tracks (T_1, T_2) can be programmed to connect to either of the CLBs through their pins (P_{11}, P_{12}) and (P_{21}, P_{22}), respectively. A graphical representation of these connections is shown on the right-hand side. An example connection is shown in green, where pin P_{11} of CLB_1 is connected to pin P_{22} CLBs using track T_2. This would imply that (T_2) can't be further used for any connections, whereas, (T_1) would be available.

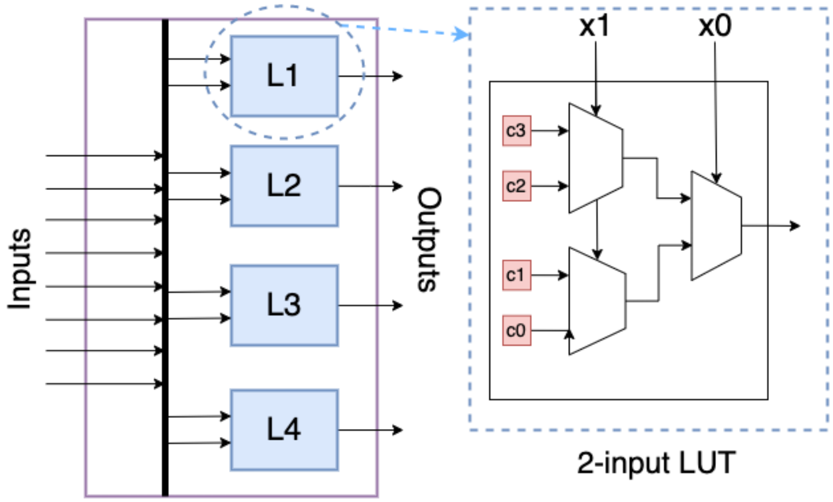


Fig. 2: Configurable Logic Blocks

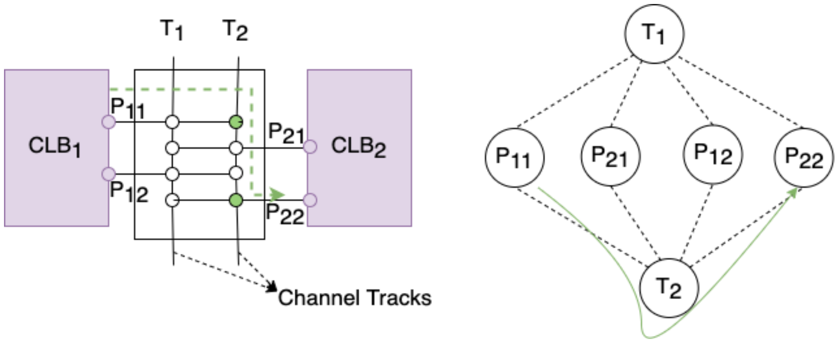


Fig. 3: Connection Box

(b) Switch Box (SB): Switch boxes lie at the intersection of the horizontal and vertical channels, facilitating connections in all directions. An example of a switch box of width two is shown in Figure 4, where each terminal (say, T_{11}) can be connected to any of the other 3 sides of the switch box, through terminals on that side (here, T_{21}, T_{31} and T_{41}). All possible connections among these terminal sets (e.g., $T_{11}, T_{21}, T_{31}, T_{41}$) form a clique, as shown in the graph representation.

Nomenclature: In addition to the standard notions of CLBs, connection boxes and switch boxes discussed above, for presentation purposes, we introduce the following terms: Whenever we refer to a *row* or *column*, we mean an eFPGA row (column) that contains CLBs, enumerated from top to bottom (left to right). eFPGA rows and columns which do not contain CLBs we dub *SB-CB rows* (*SB-*

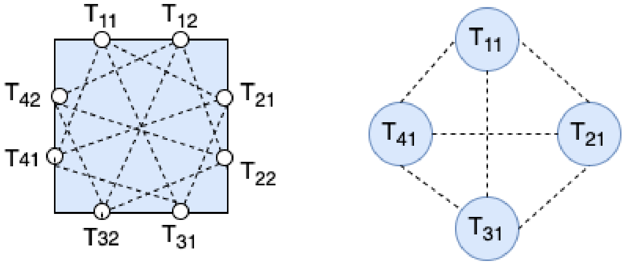


Fig. 4: Switch Box

CB columns), similarly enumerated from top to bottom (left to right). We let the *height* of the eFPGA be the number of (CLB) rows and the *width* be the number of (CLB) columns.

4 Using UCs for Logic Locking

We now introduce our first provably secure logic-locking solution. Intuitively, universal circuits satisfy our security definition simply because they can be programmed to realize *any* circuit up to a certain size with a fixed number of inputs and outputs. Thus, when using a universal circuit as the locked circuit, no information is leaked as to which specific circuit will be executed by the end user.

We define our logic-locking scheme as follows.

UC-based Logic Locking

- **Inputs:**
  Original circuit C with number of inputs n, number of outputs m, size of the circuit s, security parameter λ.
- **Locking algorithm:**
  $\texttt{Lock}(\lambda, C) \rightarrow (p, \texttt{UC}_s^{n,m})$, where
  $\texttt{UC}_s^{n,m} \leftarrow \texttt{ConstructUC}(n, m, s)$ and
  $p \leftarrow \texttt{ProduceProgram}(n, m, s, C)$.

We show that this construction is a secure logic locking scheme, where the allowed leakage function L outputs the number of inputs, outputs, and gates of the original circuit C.

Proof. The correctness of this construction follows directly from the correctness of the underlying universal circuit construction. To prove security, we provide a simulator $\mathcal{S}$ which does not have access to the locked circuit. In this case, constructing simulator S is trivial, as the locking algorithm does not actually use the original circuit C (only the information provided by the leakage function L, to which the simulator has access according to the definition). Thus, $\mathcal{S}$ can simply follow the actual locking procedure and use the algorithm $\texttt{ConstructUC}(n, m, s)$ to obtain $c_e = \texttt{UC}_s^{n,m}$. Then, S starts an internal adversary $\mathcal{A}$ with c_e as input. Whenever $\mathcal{A}$ sends a query to its oracle, $\mathcal{S}$ forwards the query to its own oracle, and forwards the oracle's answer to $\mathcal{A}$.

We define the following hybrids:

$\mathbf{Hybrid}_0$: This hybrid corresponds to the execution in the real world. The simulator honestly follows the protocol, including following the locking procedure for the given circuit.

$\mathbf{Hybrid}_1$: In this hybrid the simulator works as outlined in the description above.

Note that nothing changed between $\mathbf{Hybrid}_0$ and $\mathbf{Hybrid}_1$. Thus, the distributions are identical. Furthermore, note that in $\mathbf{Hybrid}_1$ the simulator does not require access to the description of the original circuit c_o. Thus, no information about the original circuit (besides the allowed leakage consisting of n, m, and s) is leaked.

While our logic locking definition (Def. 4) requires only computational security, note that the construction above is, in fact, information-theoretically secure.

Key length optimization When used with non-trivial circuits, the construction above produces a large key, i.e., the set of program bits needed to describe the original function. However, it is possible to obtain a small key and thus satisfy the additional polynomial-size key requirement by adjusting the construction above as follows. The UC's program bits for the original circuit is encrypted with a key from a symmetric encryption scheme (e.g., AES). The ciphertext is included as part of the locked circuit, while the symmetric encryption key serves as the logic-locking key. During execution, the symmetric encryption key is used to decrypt the ciphertext, and the UC is executed using the resulting program bits. This is a generic way of satisfying the polynomial key requirement. However, since it is orthogonal to our constructions, for simplicity, in the following we focus on the information-theoretic version.

5 Manufacturing Efficient UCs

While we have established that universal circuits are a secure logic-locking solution, a key question remains: how can we *manufacture* a universal circuit that retains (most of) the performance and energy efficiency of the original, unlocked circuit design?

Below, we consider three possible approaches: **(1)** directly manufacture an existing UC construction, **(2)** map an existing UC construction to an eFPGA, **(3)** construct a new UC scheme that maps more naturally to the structure of an eFPGA. As discussed qualitatively below and evaluated quantitatively in §6, the latter two approaches are able to take advantage of the tooling and optimizations the hardware community has already invested in eFPGA design, leading to better performance, energy use, and integration with existing design pipelines.

5.1 Directly Manufacturing a UC

The simplest approach to manufacturing a UC would be to feed an existing UC construction, e.g., Valiant's [37] or a more recent optimization [22, 40, 18], to an ASIC design tool. While simple, this approach has several drawbacks.

First, existing UC constructions are primarily designed and optimized for use in cryptographic protocols and theory proofs, rather than optimizing for the performance and energy use of a physical circuit.

Second, modern circuits have increasingly complex designs, which are fine-tuned to a particular application. Mapping such complex designs efficiently to a programmable substrate like a UC, is a hard problem, and hardware equivalents, such as FPGAs and eFPGAs, rely on high-quality automated design tools [10, 19] specific to those technologies. Comparable tools do not (yet) exist for existing UC constructions, and it is unclear if sufficient market investment would support their development.

Finally, logic-locked circuits are typically embedded in a larger circuit consisting of less sensitive components. Hence, it is beneficial for the design tools for the logic-locked circuit integrate with those for the rest of the design.

Despite these drawbacks, we use this construction as a baseline in our evaluation, where we find that our other two approaches, described below, outperform it in most dimensions.

5.2 Mapping a UC to an eFPGA

Our next approach to manufacturing a UC is to map an existing UC construction to the structure of an eFPGA. This approach is intuitively appealing, since eFPGAs are already designed for general programmability, suggesting we could think of the eFPGA's configuration bits as analogous to the key bits of a logic-locked circuit. Indeed, prior work has explicitly proposed using eFPGAs for logic locking [24, 16].

On the plus side, this prior work shows that it is feasible to incorporate eFPGAs into a larger system design without disrupting the typical ASIC design flow. It also shows that this approach is compatible with optimizations that can help achieve reasonable power, and performance overheads.

Unfortunately, this prior work offers no proofs of security for their design, and there is reason to believe that simply treating an eFPGA with s CLBs and $n + m$ I/O blocks as a $UC_s^{n,m}$ is unsound. In particular, even if we count the switch and connection boxes that constitute the eFPGA's routing network, an eFPGA with s CLBs can be represented as a circuit with $O(s)$ gates. If such a circuit were a universal circuit $UC_s^{n,m}$, it would violate Valiant's lower bound of $z \log z = (n + s) \log (n + s)$.

Indeed, the news gets worse: It is not immediately clear if even that lower bound is achievable on an eFPGA! In particular, in Valiant's graph construction of the EUG, any two nodes can be connected "for free". However, on an eFPGA, the horizontal and vertical channels typically have a (small) constant width, and the physics of the circuitry require that a new connection must take a path on the eFPGA grid which has not been previously used and *does not interfere with the connections that have been established already.*

As an initial step, we show (§5.2) that by carefully embedding an existing UC construction in an eFPGA, we can provably use $O(z^2) = O((n+s)^2)$ CLBs as a universal circuit $\mathtt{UC}_s^{n,m}$. Despite the optimizations in §5.2, this construction requires that the channel width grow logarithmically with the circuit size. Since most eFPGA designs offer only a (small) constant channel width, this construction either requires a custom eFPGA design, or a limit on the maximum circuit size.

Mapping Valiant's UC We focus on mapping Valiant's UC construction to an eFPGA, since it is arguably one of the simplest constructions, and it remains asymptotically optimal. Experimenting with more recent constructions [22, 40, 18] to see if their improvements to Valiant's constants translate to the eFPGA setting would be interesting future work.

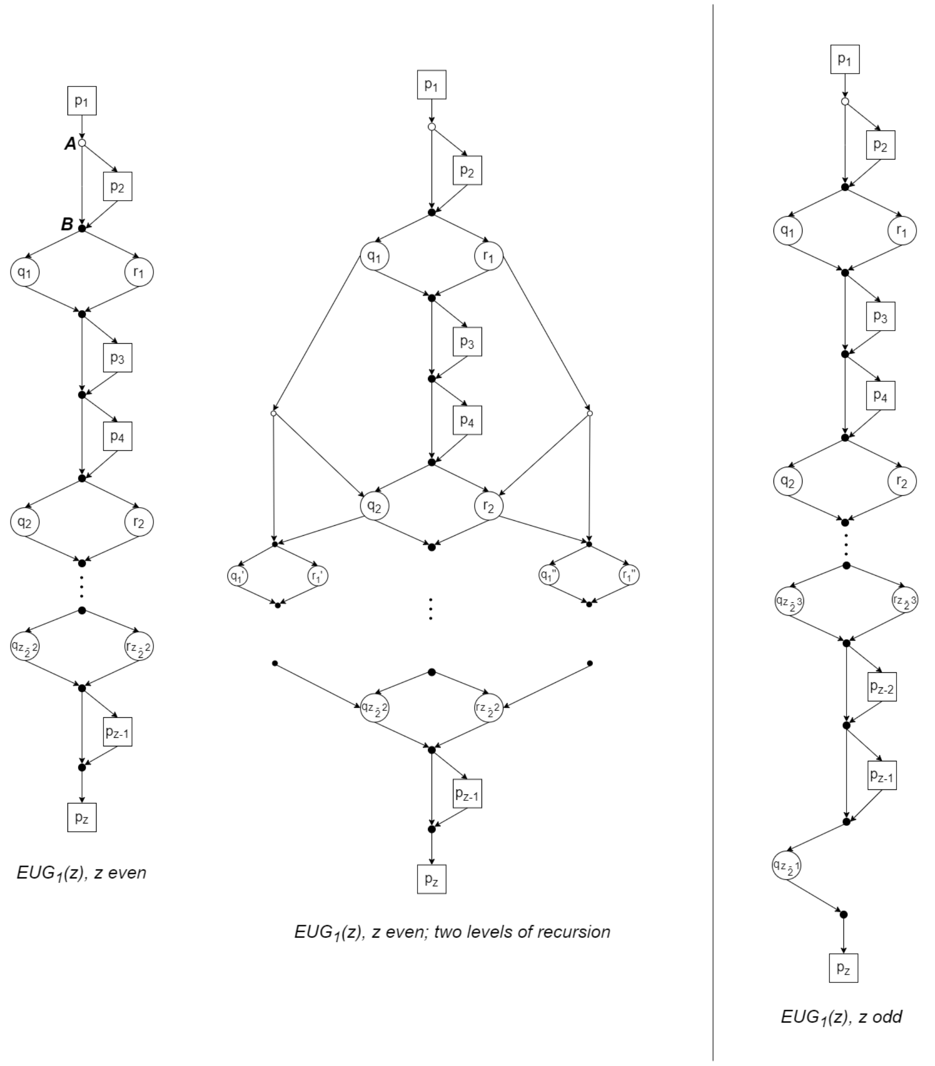
Recall that Valiant's construction proceeds recursively (§3.1). It relies on an edge-universal graph (EUG) for fan-in/fan-out 1 as depicted in Figure 5. Here, the distinguished poles of the EUG of size z are shown as squares, and the additional nodes are depicted as dots and circles. Consider the recursive construction of $\mathtt{EUG}_1(z)$, where z is even. The nodes $\{q_1, \cdots, q_{\frac{z-2}{2}}\}$ as well as $\{r_1, \cdots, r_{\frac{z-2}{2}}\}$ are poles of two $\mathtt{EUG}_1(\frac{z-2}{2})$. These two subgraphs can still be quite large and might in turn be built out of multiple levels of recursion (see Figure 5 for an example of two such levels).

As a result, it is not immediately clear how such a construction can be embedded onto the grid-like structure of the eFPGA. Specifically, in Valiant's graph construction, when the new level of recursion is added, the new poles can simply be connected to the old ones by introducing a new edge. In contrast, given the limited channel width of the eFPGA, we must ensure that whenever a new connection is introduced, it does not physically interfere with any previous connections.

Below, we describe how to perform a physically compatible embedding of Valiant's EUG into an eFPGA, which allows us to show the following result:

Theorem 1. *An eFPGA with $O((n+s)^2)$ CLBs and channel width $O(\log_2(n+s))$ can embed all stateless DAG circuits with fan-in/fan-out 2, s gates and n inputs bits.*

As a first step, Figure 6 shows how an eFPGA which is supposed to embed a $\mathtt{EUG}_1(z)$ can be built out of eFPGAs which embed EUGs $\mathtt{EUG}_1(\frac{z-2}{2})$ (if z is even) or $\mathtt{EUG}_1(\frac{z-1}{2})$ (if z is odd). Below, we elaborate on each of the steps from Figure 6.



$EUG_1(z)$, z even

$EUG_1(z)$, z even; two levels of recursion

$EUG_1(z)$, z odd

Fig. 5: Valiant's UC – Edge-Universal Graph (EUG) for graphs of fan-in/fan-out 1 and z distinguished poles.

Embedding Valiant's UC onto an eFPGA

- **Inputs:**
  Number of poles z in the EUG.
- **Construction:**
  1. Embed $\texttt{EUG}_1(1)$ into an eFPGA of height $h = 1$, width $w = 1$, and channel width $cw = 1$ (Figure 7). Denote the resulting eFPGA chip as $\texttt{eFPGA}_1$.
  2. For each i from 2 to $log_2(z)$:
     - Depending on whether the number of poles z' in step i is even or odd, use the corresponding structure for $\texttt{EUG}_1(z')$ as follows:
       - $*$ Let $cw = cw + 1$.
       - $*$ If z' is even, let $h = 2 * h + 2$. Otherwise, let $h = 2 * h + 1$.
       - $*$ Let $w = 2 * w + 1$.
       - $*$ Double the number of rows in $\texttt{eFPGA}_{i-1}$ and place each CLB that was previously on row i on row $2 * i - 1$ (each CLB's column remains the same). Connect the CLBs exactly the same as they were connected in $\texttt{eFPGA}_{i-1}$, except that if a column wire segment was used to connect CLB (SB-CB) row i to SB-CB (CLB) row j, this wire segment now connects CLB (SB-CB) row $2 * i - 1$ to SB-CB (CLB) row $2 * j - 1$. Denote this "stretched" version of $\texttt{eFPGA}_{i-1}$ by $\texttt{S-eFPGA}_{i-1}$.
       - $*$ Let $\texttt{eFPGA}_i$ be an eFPGA fabric of height h and width w. Then, the CLB on row i of the middle column corresponds to the pole p_i. The left-hand side of $\texttt{eFPGA}_i$ (the first $(w-1)/2$ columns) is used to embed $\texttt{S-eFPGA}_{i-1}$, the right-hand side (the last $(w-1)/2$ columns) is used to embed $\texttt{S-eFPGA}_{i-1}$ as well.

Fig. 6: Mapping of EUG used in Valiant's UC onto eFPGA, Fan-in/Fan-out One

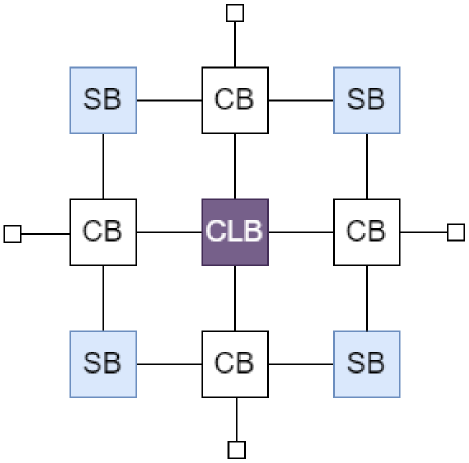


Fig. 7: FPGA with height $h = 1$, width $w = 1$, channel width $cw = 1$.

Connecting new poles to old ones First, note that Figure 6 does not specify how the common nodes (white and black dots in Figure 5) are implemented or how the "middle part" of an $\texttt{eFPGA}_i$ is connected to the CLBs representing the poles of $\texttt{S-eFPGA}_{i-1}$. Consider a common node such as A in Figure 5. Its only purpose is to connect the new pole p_1 to p_2 and B. As for B, its purpose is to enable any connection from A (thus effectively p_1) and p_2 to the old poles q_1 and r_1. The other common nodes are similarly used to connect new and old poles which are close to each other. Instead of explicitly implementing these nodes on the eFPGA, we rely on the internal eFPGA structure (of corresponding channel width) to show that any required connection from the new poles to the poles of the previous recursion level is possible.

Now, consider a node q_i (r_i is analogous). According to Valiant's construction, one of the following four nodes can provide input to q_i: p_{2i-1}, p_{2i}, q_{i-1}, r_{i-1}. If p_{2i-1} is used, one simply uses the SB-CB row $2i-1$ (note that q_i is on row $2i-1$ in $\texttt{S-eFPGA}_{i-1}$) to connect q_i and p_{2i-1}. If p_{2i} is used, one uses the w-th SB-CB column to go from row $2i$ to the SB-CB row $2i-1$, and then uses this row to connect to q_i. If q_{i-1} is used, the SB-CB row $2(i-1)$ is used to connect to the w-th SB-CB column, and then SB-CB row $2i-1$ is used to connect to q_i. Finally, if r_{i-1} is used, the SB-CB row $2(i-1)$ is used to connect to the $w+1$-th SB-CB column, and then SB-CB row $2i-1$ is used to connect to q_i (see Figure 8 for an example).

Similarly, a node q_i can provide outputs to one of the following four nodes: p_{2i+1}, p_{2i}, q_{i+1}, r_{i+1}. We already described how the last two cases are handled. For p_{2i+1}, the SB-CB row $2i$ is used connect to the w-th SB-CB column and go to row $2i+1$. Similarly, for p_{2i}, the SB-CB row $2i$ is used connect to the w-th SB-CB column and go to row $2i$.

These connections are always possible because $\texttt{eFPGA}_1$ does not use any connections on the outer SB-CB columns and in the following iterations, new connections never utilize new outer columns. Thus, the outer SB-CB columns of $\texttt{S-eFPGA}_{i-1}$ are "free", and hence we can utilize SB-CB columns w and $w+1$ to connect the new poles to the old ones (at most two connections use the same segment of each of these columns simultaneously). Additionally, note that at most a single new connection is used on each SB-CB row to the left and to the right side of the middle column (these two connections do not intersect). Thus, by increasing channel width by one at each level i, we account for all new connections.

CLBs Acting as Switches Note that in recursive step, we use CLBs to represent the poles. However, in the next recursive step these poles q_i and r_i are no longer poles (see Figure 5, the graph in the middle). Instead, they are simply nodes with at most two inputs and at most two outputs and can thus act either as switches, or simply forward the incoming signal. All of these functionalities can be performed by a CLB. As described in Section 3.2, a CLB consists of LUTs, and a 2-input LUT can be programmed to perform any function from two input bits to one output bit. Two LUTs can then be programmed to perform any

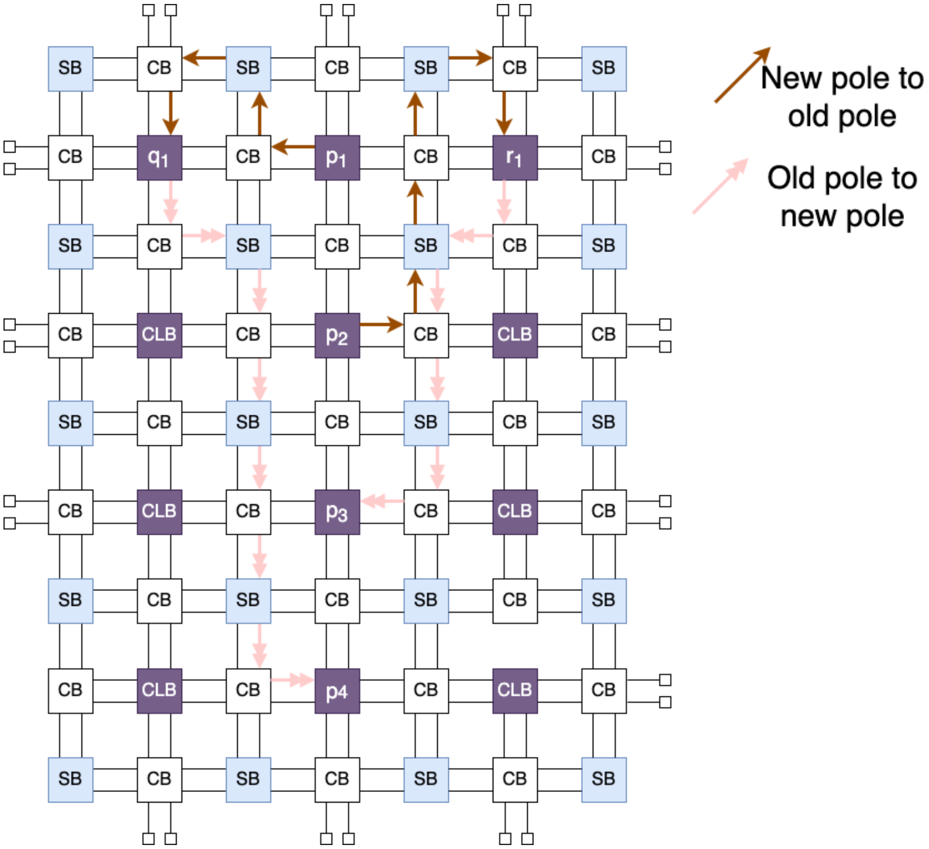


Fig. 8: Embedding of recursive step $i = 2$, where $z' = 4$ ($\texttt{EUG}_1(1)$ to $\texttt{EUG}_1(4)$). CLBs which are assigned to both new and old poles are labeled as their counterparts in Figure 5. CLBs which are labeled simply as "CLB" are not assigned to any nodes. Connections using arrows indicate how a node p_1 can be connected to q_1, q_1 to p_4, p_2 to r_1, and r_1 to p_3.

function from two inputs to two outputs. Thus, we do not need to change our construction when moving to the next recursive level.

Increasing Fan-in/Fan-out Recall that the construction in Figure 5 is an EUG for graphs with fan-in/fan-out one. Recall that it is possible to use two copies of such a construction and merge the respective poles in order to obtain an EUG for graphs with fan-in/fan-out two. We follow the same approach and simply double the channel width to obtain an EUG for fan-in/fan-out two.[†]

Analysis As the recursive step is repeated $O(\log_2(z))$ times, and each step increases the channel width by one, the resulting channel width of the eFPGA is $O(\log_2(z))$. The height is $O(z)$, and the width is also $O(z)$. Thus, unfortunately, while the UC is of size $O(z \log z)$, the construction uses area $O(z^2)$. While this discrepancy might seem surprising initially, it is directly caused by the limited channel width of the eFPGA. Hence, we cannot simply connect a new pole to an old one as Valiant's construction does. Intuitively, at each recursive step we must decide between a higher overhead in terms of the channel width or a higher overhead in terms of the resulting area. In the construction above, we optimize for channel width by increasing it only by one in each step. However, as we show below, other trade-offs are indeed possible.

Finally, recall that the number of poles z in Valiant's EUG construction for circuits with n inputs and s gates is $z = n + s$. Thus, the number of CLBs in the construction is equal to $O(z^2) = O((n + s)^2)$, which gives us exactly the result in Theorem 1.

Optimization: Channel Width - Area Trade-off The mapping above effectively prioritizes channel width over area – in each recursive step, channel width is increased only by one, while both the height and the width increase by at least a factor of two. We ask ourselves whether it is possible to improve the area (possibly by sacrificing channel width), and the answer is yes.

In the following, for simplicity assume that in each recursive step we use the EUG construction for $\texttt{EUG}_1(z)$ where z is even. Consider the construction which follows an approach similar to the one introduced above, except that now instead of using a single column in the middle for the new poles, in recursive step i, we place the CLBs which represent the new poles in *two* such columns (p_{2k+1}, p_{2k+2} are placed on row $2k + 1$ for each $k \in \{0, \cdots, \frac{i-2}{2}\}$). Then, by an argument analogous to that in Section 5.2, the channel width will need to increase by two – each old pole q_i (r_i) requires one connection to obtain input from a new pole, and one connection to forward its output to another new pole.

[†] It is also possible to instead double both the left and the right side of the eFPGA (by placing the new left side to the left of the original one, and the new right side to the right of the original) and then connect the poles in the middle to the old poles on the new left and right side. This requires an increase of the channel width only by one, but almost doubles the area.

The width increases by two instead of one (since we did not double the height and thus might need to use same SB-CB row to forward output of a previous pole, and forward input to another previous pole); however, the height increases only by two instead of a *factor* of two. Thus the area increases only slightly compared to our earlier construction.

For the same reasoning to apply to the next step $i+1$, we need to use four middle columns instead of two (otherwise the height would increase by a factor of two in comparison with step i). This way, the width increases by four, and the height increases by two in comparison with step i. However, the channel width will increase by four (to connect old poles from two columns to the new ones). In general, in order to ensure that each connection from old to new poles is possible, we need to increase the channel width by $2 * W_{poles}$, where W_{poles} denotes the number of columns used for the mapping of the poles of the previous recursive step.

Thus, at each step one can make a decision whether to prioritize area or channel width by deciding how many columns to use for the new poles (more columns result in higher channel width, but when the column number is chosen carefully it can help to reduce the area of the final construction).

We propose the following heuristic: Starting with an efficient mapping of some small EUG onto the eFPGA fabric, use as many new middle columns as are needed to make the height of the new eFPGA the same as that of the old one. Additionally, before using the eFPGA constructed in step $i-1$ for step i, rotate this eFPGA by 90° whenever the number of columns needed to embed the new poles of level $i-1$ are larger than the number of rows needed to embed these poles. As the channel width in each step increases by $2 * W_{poles}$, this rotation allows us to save channel width. As we show in Section 6, this optimized version indeed produces better results.

5.3 Topological Logic Locking

We now introduce our final approach, *topological logic locking*, which relies on a new UC construction that is tailored to the constraints of the eFPGA setting. Compared to the approach based on embedding Valiant's UC, this construction allows us to achieve a more general result.

Theorem 2. *An eFPGA with $\lceil \frac{(n+s)^2}{w'} \rceil$ CLBs, where $w \geq 2$ is the channel width and $w' \leq \lfloor \frac{w}{2} \rfloor$, can embed all stateless DAG circuits with fan-in/fan-out 2, s gates and n inputs bits.*

We prove this result by giving an embedding of any stateless DAG circuit onto an eFPGA with these parameters in Figure 9.

We illustrate this mapping with an example in Figure 10; the corresponding circuit is given in Figure 11.

Analysis The connections between CLBs in the construction outlined in Figure 9 are always possible, since we never user more wires on each column and

eFPGA to UC, Topological Logic Locking

- **Input:**
  Original circuit C with number of inputs n, number of outputs m, size of the circuit s.
- **Finding an embedding:**
  1. Sort inputs and gates of C in topological order. Let C' denote the resulting circuit.
  2. Assign column i of the eFPGA to the i-th element (input or gate) in C'.
  3. Following the topological ordering, process every edge (x, y) which connects two elements in C':
     (a) Check whether a row was already assigned to x. If not, assign the first row j that was not yet used to x. Program the CLB on row j and column x (that was assigned to x in Step 2) to implement x; denote this CLB as $\texttt{CLB}_x$.
     (b) Check whether a row was already assigned to y. If not, assign the first row j that was not yet used to y. Program the CLB on row j and column y (that was assigned to y in Step 2) to implement y; denote this CLB as $\texttt{CLB}_y$.
     (c) $\texttt{CLB}_x$ and $\texttt{CLB}_y$ can be connected by first going up or down to the row that was assigned to y and then following this row until it reaches $\texttt{CLB}_y$.
  4. Any outputs can be forwarded to the I/O blocks of the eFPGA by using the same row as the gate that computes the output.

Fig. 9: Topological Logic Locking

row than the eFPGA channel width. This can be seen by the following argument. Since the fan-in of $\mathcal{C}$ is two, and we assign each input and each gate a new row, there are at most two connections that use each row. Additionally, since the fan-out of $\mathcal{C}$ is two, and each column is used only to route the output of a *single* CLB to the correct row, there are at most two connections that use each column. This is clearly supported by a width $w \geq 2$.

This construction clearly gives us a correct embedding of a circuit $\mathcal{C}$, since the CLBs can be programmed to implement any input or gate, and each connection (wire) in $\mathcal{C}$ is implemented on the eFPGA structure by construction.

For this construction, we use an eFPGA with $n + s$ CLB columns (as each column is assigned to a single input/gate), and $n + s$ rows (again, since each row is assigned to a single input/gate). Hence, in total the construction requires an eFPGA with $(n + s)^2$ CLBs.

Optimizations There are a few ways we can optimize the construction introduced above. As described, it only uses two connections in each row and column. If the eFPGA width is larger than two (for simplicity, we assume that the channel width $w = 2w'$ is even), then we can improve the number of rows used in the construction by using a single row (with w wires) to provide inputs to and

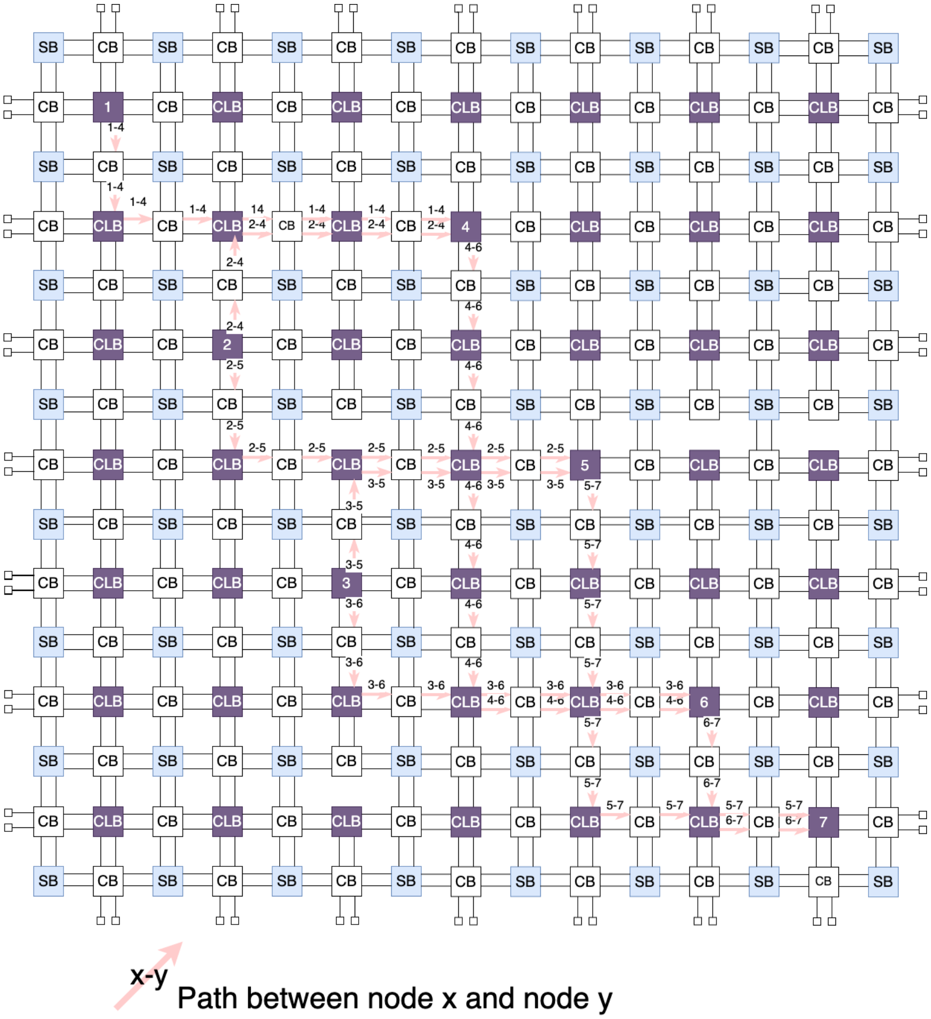


Fig. 10: Topological Construction – Embedding the circuit in Figure 11. A label x in a CLB denotes $\texttt{CLB}_x$.

forward outputs from not only a single CLB, but w' such CLBs (as each CLB requires only two wires). To ensure that the output signal of a CLB does not interfere with the other CLBs on the path, to connect $\texttt{CLB}_x$ and $\texttt{CLB}_y$ in Step 3c), as well as to forward the output in Step 4 of Figure 9, we will now use the SB-CB rows directly above the CLB row that was assigned to a particular gate (instead of the CLB row itself). We call this approach *packing*. This changes our embedding as follows: When assigning a row to an input/gate, instead of assigning the first row that was not yet used, we assign the first row that is *not yet saturated*. This allows us to use each (SB-CB) row for w' inputs/gates,

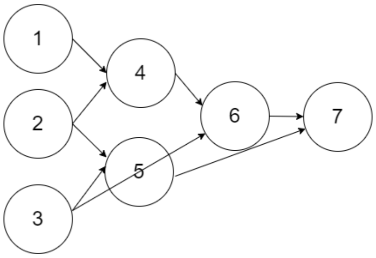


Fig. 11: Topological Construction – Cicruit

instead of only one and thus decrease the total number of rows to $\frac{n+s}{w'}$. The number of CLBs used in our construction thus becomes $\frac{(n+s)^2}{w'}$. Generalizing w' to $w' \leq \lfloor \frac{w}{2} \rfloor$ trivially gives us the result in Theorem 2.

We can further reduce the number of CLBs by using CLBs which can compute functions of larger input sizes. Instead of computing functions of two input bits to two output bits, typical commercial CLBs can compute functions from, e.g., five input bits to two output bits. Our construction using such CLBs remains largely the same: $\mathcal{C}$ must be brought into a form where gates accept as many input bits as the CLBs, and the rows must be packed according to the new input sizes (if each CLB supports five input bits, $w/5$ rows can be packed into one).

Channel Width – Area Trade-off An interesting aspect of our construction with packing enabled is that it allows us to easily optimize for the parameters of the eFPGA. If the channel width is constant, the number of CLBs in this construction is $O((n+s)^2)$. If we allow the channel width to depend on n and s, then even the *asymptotics* for the number of CLBs in this construction can improve. For instance, if the channel width is $O(\frac{n+s}{\log(n+s)})$, then the number of CLBs becomes $O((n+s)\log(n+s))$ – equivalent to the number of gates in Valiant's UC construction.

6 Evaluation

We evaluate the efficiency of each of our proposed approaches to manufacturing a UC: directly manufacturing a UC, mapping an existing UC construction onto an eFPGA, and our topological construction. To assess the cost of security, we compare these provably secure approaches with both the original circuit (which provides no security) and with a standard eFPGA design, which offers programmability but does not guarantee security.

We compare these approaches using standard VLSI metrics: performance, power, and area. The physical area of the chip affects its one-time manufacturing cost, while operational costs stem from the chip's power and delay. Higher power consumption or lower performance can undermine the design efforts that went into the original circuit, which is designed and optimized to meet the specifications of a particular application.

6.1 Evaluation Methodology

Here we try to mimic the flow a designer would have to follow in order to implement our suggested logic locking schemes.

1. Logic-Gate Implementations: The process of locking a circuit starts with a Verilog description of the design. The locking algorithm takes this design as input and produces a Verilog description of the locked circuit. For our constructions, we use the following tool flow for locking.
   (a) *Valiant's Universal Circuit Tool Flow*: We use a modern open-source implementation of Valiant's UC construction [18]. We develop Python-based parsers to map the original circuit Verilog to the Bristol format expected by the tool. We use another Python script to map the UC tool's output to Verilog.
   (b) *eFPGA Tool Flow*: We use the popular open-source FPGA-CAD tool VPR (Versatile Place and Route) [25] to map our original circuit onto an eFPGA. We parse the graph description of the eFPGA used, and the placement and routing produced for the original circuit. We parse these into a Verilog description and eFPGA configuration using Chisel [3] and Python scripts, respectively.
2. *Synthesis*: Using the Cadence Genus logic synthesis tool[8], we map the Verilog for the locked circuit (produced above) to standard cells (a collection of technology-optimized logic gates), which are each sized and repacked through an iterative process to maximize the design's frequency (which correlates with performance). This is the locked design, which which the foundry will manufacture.

Given the locked circuit produced by the process above, we analyze its efficiency as follows.

1. *Static Timing Analysis* [9]: This is standard practice in an ASIC design flow. The design is broken down into timing paths, and based on the signal probability constraints (if any) applied to the inputs, it calculates the delay along each path. The result is a measure of the worst-case delay in calculating a new output for a given input, which corresponds to the highest frequency of operation the design can support. Higher frequencies, in turn, correspond to better application performance. For our constructions, we do this analysis by setting the signal probability to the value of the correct key-configuration bits.
2. *Power Analysis* [7]: Similarly, we assert the correct constant signal values and zero toggle rate for the key configuration bits (meaning they are held fixed). All other inputs are toggled at the operational frequency, in order to determine the average power consumption for operating the circuit at its highest possible frequency.

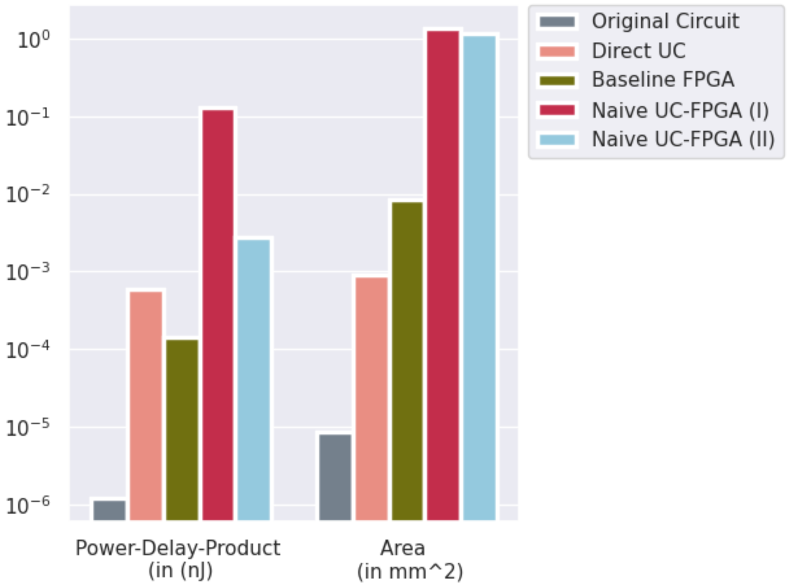


Fig. 12: Baseline evaluation over c17. Note the log-scale y-axis. Lower is better.

3. *Evaluation Metric*: In hardware designs, higher performance can always be achieved by consuming more power, and lower power can be achieved by accepting a lower operational frequency. Hence, to reasonably compare the efficiency of different circuit architectures, we use a standard energy efficiency metric *Power-Delay Product (PDP)*. A lower PDP implies better energy efficiency, implying that the given architecture can run faster (lower delay) in the same amount of power, or will consume lower power at the same operational frequency.

6.2 Baseline Costs of Universality

Our initial experiments measure the cost of supporting programmability compared to simply manufacturing the (insecure) original circuit. For the original circuit we use a small combinational circuit, ISCAS'85 benchmark c17 [6].

We evaluate the cost of instantiating this circuit using a directly manufactured (secure) UC construction and the base eFPGA design (which offers programmability but does not security). As a sanity check, we also measure two naive approaches to mapping Valiant's UC construction onto an eFPGA. In both approaches, we start by simply feeding the standard UC circuit to the standard eFPGA synthesis pipeline. This obviously retains the universality of the UC circuit. When it comes time to unlock the circuit, we can do so in two ways.

- Version I: Program the eFPGA to function as a UC whose program bits are another set of inputs to the eFPGA, which are held constant (to the values corresponding to c17) throughout the operation of the design.
- Version II: Program the eFPGA to function as the original c17 circuit. From an adversary's perspective, since the eFPGA was constructed to be large enough to execute Version I, this approach retains the same security, but potentially offers better run-time performance.

Figure 12 shows the energy efficiency (PDP) and area of each of these constructions. We highlight a few important points.

1. The cost of programmability, via Valiant's UC or the eFPGA, is high. However, as discussed in §5.1, a logic-locked circuit is typically incorporated into larger circuit design. Hence, the overhead paid for locking may overestimate the overhead for the system as a whole.
2. The direct UC implementation has worse efficiency than the base eFPGA but uses less area.
3. The efficiency of the *naive versions* of mapping Valiant's UC to an eFPGA is particularly bad. This is expected as a much larger (than required to map the original design) eFPGA is configured to emulate a circuit (the universal circuit) which introduces a second layer of inefficiency compared to the original circuit.
4. By design, both versions of the naive mapping of a UC to the eFPGA use the same area, but Version II over an order of magnitude more efficient. This suggests that even with a relatively large eFPGA construction, the tools can find an energy-efficient configuration to implement the original logic.

Since the direct UC instantiation dominates the naive UC-to-eFPGA embeddings, in the experiments below, we use the direct UC instantiation as our baseline.

6.3 Comparison of Our Constructions

We now compare the energy efficiency of each of our constructions as we increase the size and complexity of the original circuit. To control these aspects of the circuit, we create a synthetic circuit benchmark suite. Circuits in the suite are DAGs of varying size and connectivity. Each node has fan-in/fan-out ≤ 2 and is programmed to a random logic gate.

Figure 13 shows our results. We include data for both our original embedding of Valiant's UC into an eFPGA (§5.2), and a version with the optimizations from §5.2. Both our optimized embedding of a UC construction into the eFPGA and our topological locking construction outperform the baseline direct UC implementation, with topological locking performing particularly well. Further, the gap between our constructions and the direct UC grows as circuit size increases, suggesting that they will provide even larger savings for more realistic circuits.

We attribute the improved energy efficiency both to our more tailored design and to the eFPGA CAD tools, which can find a relatively optimal configuration to execute the original circuit's functionality.

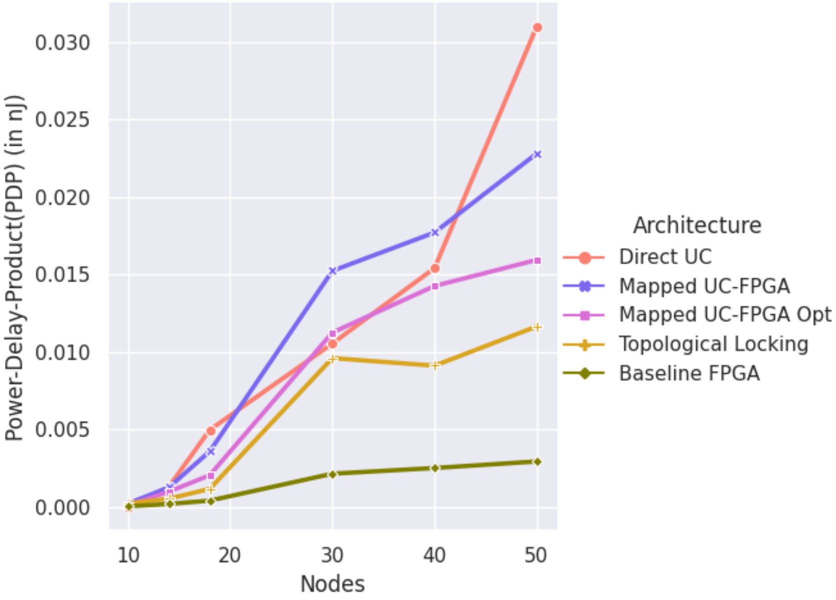


Fig. 13: Comparison of Our Logic-Locking Approaches. Lower is better.

6.4 Evaluation Over Arithmetic Circuits

Although our evaluation above with synthetic circuits provides insight into how our constructions' energy efficiency scales, we also wish to evaluate more realistic circuits. Hence, we measure their efficiency on the most common arithmetic circuits, an adder and a comparator.

Figure 14 shows our results, which indicate similar trends to those seen in §6.3. Although the efficiency of the direct UC approach is worse than the base eFPGA, our optimized topological construction helps bridge the gap.

7 Future Work

While our constructions provide provably-secure logic locking and improve over the performance of a direct UC solution, the locking overhead is still non-trivial. There are several possible directions for future exploration.

- Our constructions could be easily extended to support logic locking for stateful circuits, which could increase computation capacity per unit area.
- We could explore the use of spatial parallelism for our constructions.
- Increasing the information provided by the leakage function L (e.g., to reveal the number of layers in the circuit) could improve performance while still offering a provable security guarantee.

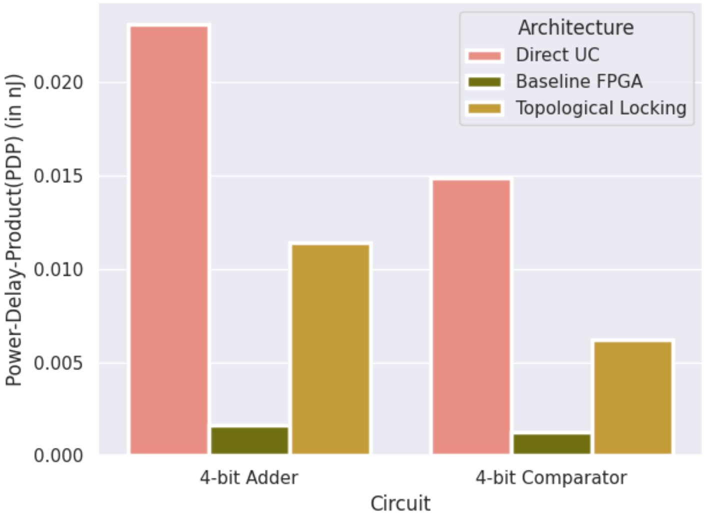


Fig. 14: Comparison of Locking over Arithmetic Circuits. Lower is better.

8 Conclusion

To stop the cycle of attacks and ad hoc defenses for logic locking, we provide a well-grounded formal definition of logic-locking security. We present a solution which meets this definition, and we explore constructions which can realize the solution as an ASIC design. While we propose several efficient constructions, there are likely other interesting points within this design space that can offer additional improvements and tradeoffs.

Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency under contract FA8750-17-1-0059 “Obfuscated Manufacturing for GPS (OMG)”. Additionally, Bryan Parno and Elisaweta Masserova were supported by a fellowship from the Alfred P. Sloan Foundation, and a gift from Bosch. Vipul Goyal was supported by the NSF award 1916939, the DARPA SIEVE program, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.

Bibliography

[1] V. Aken'Ova and R. Saleh. A "soft++" efpga physical design approach with case studies in 180nm and 90nm. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 6 pp.–, 2006.

[2] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131, 2009.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design automation conference 2012*, pages 1212–1221. IEEE, 2012.

[4] Boaz Barak. Hopes, fears, and software obfuscation. *Commun. ACM*, 2016.

[5] Peter A. Beerel, Marios Georgiou, Ben Hamlin, Alex J. Malozemoff, and Pierluigi Nuzzo. Towards a formal treatment of logic locking. *To Appear in IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022.

[6] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In *Proceedings of IEEE Int'l Symposium Circuits and Systems (ISCAS 85)*, pages 677–692. IEEE Press, Piscataway, N.J., 1985.

[7] Cadence. Cadence genus power analysis.

[8] Cadence. Cadence genus synthesis solution.

[9] Veena S Chakravarthi. Static timing analysis (sta). In *A Practical Approach to VLSI System on Chip (SoC) Design*, pages 99–116. Springer, 2020.

[10] Deming Chen, Jason Cong, and Peichen Pan. Fpga design automation: A survey.

[11] Animesh Chhotaray and Thomas Shrimpton. Hardening circuit-design IP against reverse-engineering attacks. *IACR Cryptol. ePrint Arch.*, 2021.

[12] Giovanni Di Crescenzo, Abhrajit Sengupta, Ozgur Sinanoglu, and Muhammad Yasin. Logic locking of boolean circuits: Provable hardware-based obfuscation from a tamper-proof memory. In *Innovative Security Solutions for Information Technology and Communications*, 2019.

[13] Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In *2014 IEEE 20th International On-Line Testing Symposium*, pages 49–54, 2014.

[14] Norman Einspruch. *Application specific integrated circuit (ASIC) technology*, volume 23. Academic Press, 2012.

[15] Ujjwal Guin, Ke Huang, Daniel DiMase, John M. Carulli, Mohammad Tehranipoor, and Yiorgos Makris. Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain. *Proc. IEEE*, 2014.

[16] Bo Hu, Jingxiang Tian, Mustafa Shihab, Gaurav Rajavendra Reddy, William Swartz, Yiorgos Makris, Benjamin Carrion Schaefer, and Carl Sechen. Functional obfuscation of hardware accelerators through selective partial design extraction onto an embedded fpga. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, page 171–176, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Ayush Jain, Ziqi Zhou, and Ujjwal Guin. TAAL: tampering attack on any key-based logic locked circuits. *ACM Trans. Design Autom. Electr. Syst.*, 2021.

[18] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In *International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9665 of *Lecture Notes in Computer Science*, 2016.

[19] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. Fabulous: An embedded fpga framework. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–56, 2021.

[20] KPMG. Managing the risks of counterfeiting in the information technology industry, 2006.

[21] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[22] Helger Lipmaa, Payman Mohassel, and Seyed Saeed Sadeghian. Valiant's universal circuit: Improvements, implementation, and applications. *IACR Cryptol. ePrint Arch.*, 2016.

[23] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 178–190, 2016.

[24] Prashanth Mohan, Oguz Atli, Joseph Sweeney, Onur Kibar, Larry Pileggi, and Ken Mai. Hardware redaction via designer-directed fine-grained efpga insertion. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1186–1191, 2021.

[25] Kevin E Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G Graham, Jean Wu, Matthew JP Walker, et al. Vtr 8: High-performance cad and customizable fpga architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(2):1–55, 2020.

[26] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 1979.

[27] Stephen M. Plaza and Igor L. Markov. Solving the third-shift problem in IC piracy with test-aware logic locking. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2015.

[28] Jeyavijayan Rajendran, Youngok K. Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *The 49th Annual Design Automation Conference 2012*, 2012.

[29] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.

[30] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. EPIC: ending piracy of integrated circuits. In *Design, Automation and Test in Europe*, 2008.

[31] K. Shamsi, D. Z. Pan, and Y. Jin. On the impossibility of approximation-resilient circuit locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[32] Kaveh Shamsi, Travis Meade, Meng Li, David Z. Pan, and Yier Jin. On the approximation resiliency of logic locking and ic camouflaging schemes. *IEEE Transactions on Information Forensics and Security*, 14(2):347–359, 2019.

[33] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *IEEE International Symposium on Hardware Oriented Security and Trust*, 2015.

[34] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. *Proceedings of the 2015 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2015*, pages 137–143, 2015.

[35] Can Sun and Thomas Rose. Supply chain complexity in the semiconductor industry: Assessment from system view and the impact of changes. *IFAC-PapersOnLine*, 48(3):1210–1215, 2015. 15th IFAC Symposium onInformation Control Problems inManufacturing.

[36] Joseph Sweeney, Marijn J. H. Heule, and Lawrence Pileggi. Modeling techniques for logic locking. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[37] Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*, 1976.

[38] Dominik Šišejković, Rainer Leupers, Gerd Ascheid, and Simon Metzner. A unifying logic encryption security metric. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '18, page 179–186, New York, NY, USA, 2018. Association for Computing Machinery.

[39] B. Zahiri. Structured asics: opportunities and challenges. In *Proceedings 21st International Conference on Computer Design*, pages 404–409, 2003.

[40] Shuoyao Zhao, Yu Yu, Jiang Zhang, and Hanlin Liu. Valiant's universal circuits revisited: an overall improvement and a lower bound. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 401–425. Springer, 2019.