MaskedHLS: Domain-Specific High-Level Synthesis of Masked Cryptographic Designs

Nilotpola Sarma, *Graduate Student Member, IEEE*, Anuj Singh Thakur, and Chandan Karfa, *Senior Member, IEEE*

***Abstract*—The design and synthesis of masked cryptographic hardware implementations that are secure against power side-channel attacks (PSCAs) in the presence of glitches is a challenging task. High-level synthesis (HLS) is a promising technique for generating masked hardware directly from masked software, offering opportunities for design space exploration. However, conventional HLS tools make modifications that alter the guarantee against PSCA security via masking, resulting in an insecure register transfer level (RTL). Moreover, existing HLS tools cannot place registers at designated places and balance parallel paths in a masked cryptographic design. This is necessary to stop the propagation glitches that may hamper PSCA-security. This article introduces a domain-specific HLS tool tailored to obtain a PSCA secure masked hardware implementation directly from a masked software implementation. This tool places registers at specific locations required by the glitch-robust masking gadgets, resulting in a secure RTL. Furthermore, it automatically balances parallel paths and facilitates a reduction in latency while preserving the PSCA security guaranteed by masking. Experimental results with the PRESENT Cipher's S-box and AES Canright's S-box masked with four state-of-the-art gadgets, show that MaskedHLS produces RTLs with 73.9% decrease in registers and 45.7% decrease in latency on an average compared to manual register insertions. The PSCA security of MaskedHLS generated RTLs is also shown with TVLA test.**

***Index Terms*—High-level synthesis (HLS), masking, power side-channel security, retiming.**

I. Introduction

EMBEDDED devices implementing a cryptographic algorithm are susceptible to power side-channel attacks (PSCAs) [1], where an attacker uses the target device's power consumption information to extract the secret values processed by the cryptographic algorithm. These attacks exploit the direct correlation between the device's power consumption, which is a result of the overall transistor activity, and the computations being performed. Masking [2] is a countermeasure against such attacks. Masking splits the secret inputs into random shares drawn independently from a uniform random distribution. Thereafter, all the secret input dependent computations proceed by processing these shares independently, *re-randomizing* computations that cause their recombination. This randomizes the results of intermediate computations, and hence the power consumption. Masking can be applied at the hardware [2], [3], [4] and software levels [5], [6].

Hardware masking must ensure resilience against the asynchronous behavior of circuits, such as those caused by *glitches* that may cause the recombination of shares within the circuit, removing the masking security. There are hardware masking verification tools [3] to verify that a handwritten masked hardware design is secure. However, they are limited in applicability due to gaps in the hardware masking verification theory [7], which prevents the scalability of verification to higher orders. Further, keeping in mind the development of new masking schemes/gadgets, there is an increased need for design-space exploration at the hardware level. Thus, developing secure masked hardware from scratch requires significant expertise in the design, verification, and design-space exploration of masked designs.

In contrast, a software masked design is easier to obtain from the algorithmic specification and easily verified [6]. Therefore, ways to obtain masked hardware from the corresponding masked software implementation would be beneficial. This is indeed a possibility as the glitch-resistant hardware masking properties are a superset of the software masking properties. Also, most glitch-resistant hardware-masked gadgets like domain-oriented masking (DOM) [2], HPCs [8], and COMAR [4] have the same structure as their software-masked counterparts with the primary difference being the inclusion of registers to prevent glitch propagation. Thus, in order to generate PSCA-secure masked hardware from masked software, a translation of gadget-masked intermediate code to register transfer level (RTL) is desired. That can be followed by inserting registers at well-defined locations according to the masking gadget used.

In this regard, high-level synthesis (HLS), which automatically generates RTL hardware from descriptions in C/C++, can be helpful. A few recent works [9] aim to use HLS to convert masked software to masked hardware automatically. *In this work, we have shown that all stages of HLS can alter the security of masked circuits.* They have been discussed in greater detail in Section IV-A. This suggests the need for a domain-specific HLS tool for masked hardware design focussing on the primary objective of keeping the side-channel security of the circuit intact throughout the HLS process.

We propose a domain-specific HLS tool called MaskedHLS, which performs a security-preserving translation of software-level cryptographic implementations into masked hardware. Specifically, the contributions of this work are as follows.

Manuscript received 13 August 2024; accepted 13 August 2024. This work was supported in part by the Semiconductor Research Corporation Project under Grant 2022-IR-3170. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Nilotpola Sarma.)*

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Guwahati 781039, India (e-mail: s.nilotpola@iitg.ac.in).

Digital Object Identifier 10.1109/TCAD.2024.3447223

1937-4151 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

1) We have analyzed the impact of HLS optimizations and the need for domain-specific HLS for PSCA-secure hardware design from masked software (in Section IV).
2) We have utilized the concept of retiming to insert registers in designated locations and balance parallel paths with optimal latency and registers for gadget-based masked design to protect against glitches (in Section V).
3) The correctness of MaskedHLS is shown (in Section VI).
4) A thorough experiment with PRESENT Cipher's S-box and the Canright's AES-256 S-box masked with DOM, HPC1, HPC2, and COMAR gadgets shows the usefulness of our approach (in Section VII).

MaskedHLS is generic enough to work on cryptographic implementations using any masking gadget. To the best of our knowledge, this is the first work that presents a complete HLS flow for masked hardware design from masked software in C/C++.

The remainder of this article is organized as follows. The related works are discussed in Section II. Section III covers the background needed to understand the working of MaskedHLS. Section IV illustrates the impact of HLS on the PSCA security of masked designs and the motivates our work. Section V discusses the flow of MaskedHLS and its steps in greater detail. Section VI discusses the correctness of our tool. Section VII discusses the results of using MaskedHLS on the selected benchmarks. Finally, Section VIII concludes this article.

II. Related Works

Several works on HLS of cryptographic implementations have been published [10], [11], [12]. Studies like [12] and [13] looked at the effects of various HLS optimizations on the side-channel security. However, they do not consider masked cryptographic implementations and the impact of HLS on the security guaranteed by masking.

Sadhukhan et al. [9] demonstrated how to generate side-channel secure masked hardware in quick time using HLS. They used a 3-bit bit-sliced DOM-masked AES S-box and generated the Verilog (RTL) for it using the *Bambu* HLS tool [14]. They observed that HLS does not always lead to side-channel secure hardware. Consequently, they examined the pragmas in the HLS software and came up with certain scenarios where an unguided application of pragmas would lead to side-channel leakage. They then proposed remedies for better application of such pragmas. Recently, a study by Pundir et al. [15], highlighted the importance of considering security when using HLS for hardware design. These works point out that no existing HLS tool considers side-channel leakage while performing their code transformation procedures. Moreover, generating secure hardware with these HLS tools requires a case-by-case examination of all the optimizations, which is a challenging task. Thus, there is no existing work that develops a domain-specific HLS tool for PSCA-secure RTL generation from masked cryptographic designs.

III. Background

A. Glitch-Resistant Masking

Hardware masking of cryptographic algorithms against PSCAs proceeds by splitting the secret inputs into independent random shares. For an affine component of the algorithm, these shares can be computed independently of each other to obtain the output shares. For instance, an $\oplus$ (XOR) operation, as in $c = a \oplus b$, can be split into $c0 = a0 \oplus b0$ and $c1 = a1 \oplus b1$. Here, a and b are split into two shares initially as $(a0,\ a1) : (a \oplus r1,\ r1)$ and $(b0,\ b1) : (b \oplus r2,\ r2)$, where $r1$ and $r2$ are drawn independently from a uniform random distribution. Here, $a0$, $b0$, and $c0$ are 0-shares and $a1$, $b1$, and $c1$ are 1-shares. Thereafter, $c0 \oplus c1$ gives the correct value of c. A nonlinear operation like $\otimes$ (bit-wise multiplication)[1] such as in $c = a \otimes b$, can be performed using shares $a0$, $a1$ and $b0$, $b1$. But to perform the multiplication operation, the four terms $a0 \otimes b0$, $a0 \otimes b1$, $a1 \otimes b0$, and $a1 \otimes b1$ must be calculated. Two of these computations, $a0 \otimes b1$ and $a1 \otimes b0$, can not be performed without combining the 0-shares with the 1-shares, violating the independence of shares required for secure masking. Hence, these operations need to be carefully *remasked*.

Some algorithmic tricks can be used to mask these nonlinear computations to optimize the amount of remasking. For example, the *SecMult* algorithm by Rivain and Prouff [16] proceeds by calculating the term $a0 \otimes b0$ separately and then performing masking with a random variable r as $(a0 \otimes b0) \oplus r$. The other terms are computed as $(((a1 \otimes b1) \oplus (a1 \otimes b0) \oplus (a0 \otimes b1)) \oplus r)$ following the parenthesized order. This requires two remasking operations leading to a correct masked design.

However, this algorithm does not remain secure in a glitchy circuit. Glitches are the phenomenon of different transition times in the signals of a circuit caused by variations in wire lengths and transistor speeds. As demonstrated in [17], assuming that only one share $a1$ arrives later than the others while all other shares arrive simultaneously, the number of times the gates in the *SecMult* circuit change value on different values of b reveals a correlation between the power consumption and the value of b. Thus, masking in glitchy circuits should be carefully handled. Several masking schemes were designed to be resistant to glitches [2], [18], [19], [20].

One approach toward glitch-resistant masking of cryptographic hardware is replacing all nonlinear operations with *gadgets* that are provably secure independently as well as in composition. A gadget is an algorithm that takes n m-shares as inputs (where n is the number of inputs to the gadget) and returns a single m-shared output. *A gadget-based construction of a masked circuit replaces one or more nonlinear operations with gadgets.* Depending on the security guarantees provided by the gadgets, the glitch-robust security of the gadgets in composition can be guaranteed. In the following section, we briefly introduce those gadgets.

[1] In this article, $\otimes$ and & has been used interchangeably to mean bit-wise multiplication. $\oplus$ and $\hat{}$ has been used interchangeably to mean bit-wise XOR.

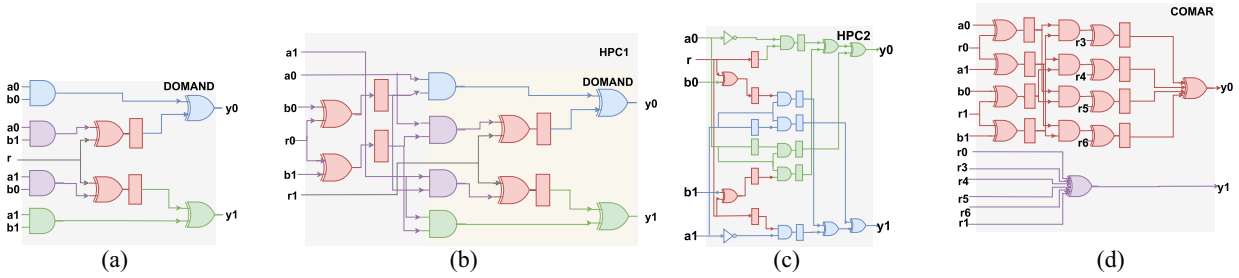

Fig. 1. Masked multiplication gadgets (a) DOMAND, (b) HPC1, (c) HPC2, and (d) COMAR.

B. Multiplication Gadgets

Groß et al. [2] presented DOM of hardware implementations of cryptographic algorithms against PSCAs. In a DOM-based gadget, each input share corresponds to a domain. DOM ensures that the computations corresponding to each share are carried out in their corresponding domain, and domains carry out computations independently of each other. In this context, *nonlinear* operations require computations across domains, and these cross-domain computations require *remasking* using new random values. It was observed that glitches affected the combination of cross-domain shares, and hence, registers are used at those locations. An example of a DOMAND gate (a multiplication gadget for one bit) is shown in Fig. 1(a). Here, the products containing cross-domain terms, $a0 \otimes b1$ and $a1 \otimes b0$ are remasked using the same random value r sampled from a uniform random distribution after which the outputs of the masking gates (XOR) are registered.

A similar class of *nonlinear* gadgets were introduced in [3]. The strategy was to *remask* the inputs to the gadgets before multiplying. The HPC1 gadget, proceeds by refreshing one of the inputs of the DOM gadgets (with two operands) using a refresh (remasking) gadget. For the computation $c = a \otimes b$, an HPC1 gadget masks both the shares of the input b as: $(b0 \oplus r0)$ and $(b1 \oplus r0)$ and puts a register after these masked inputs before being input to the DOMAND circuit. The other inputs $a0$ and $a1$ are put into the DOMAND circuit. The HPC1 multiplication gadget is shown in Fig. 1(b).

In HPC2 [3], all the inputs that have been split into shares of two are registered. Thus, one register each is placed after $a0$, $a1$, $b0$, and $b1$ for the computation $a \otimes b$ in two shares. After that the computation is performed as follows: $c0 = ((a0 \otimes r) \oplus (b1 \otimes r)) \oplus (a0 \otimes b0)$ and $c1 = ((a1 \otimes r) \oplus (b0 \otimes r)) \oplus (a1 \otimes b1)$ with registers being placed at all the input shares and four intermediate locations. The HPC2 multiplication gadget using two shares is shown in Fig. 1(c).

Fig. 1(d) represents the COMAR gadget for $c = a \otimes b$. All the input signals are masked with the same mask bit r for the 0-shares and r' for the 1-shares. Four fresh mask bits $r2$ to $r5$ are used to mask the nonlinear terms. As shown, the shared output is formed as $c0 = (((a0 \oplus r) \otimes (b0 \oplus r')) \oplus r2) \oplus (((a0 \oplus r) \otimes (b1 \oplus r')) \oplus r3) \oplus (((a1 \oplus r) \otimes (b0 \oplus r')) \oplus r4) \oplus (((a1 \oplus r) \otimes (b1 \oplus r')) \oplus r5)$ and $c1 = r2 \oplus r3 \oplus r4 \oplus r5$. This gadget uses six masked bits which is larger than the HPC2 2-input AND gadget. However, all instantiated two-input COMAR-AND gadgets in a circuit can use the same six random masks.

C. Retiming Basics

Retiming [21] is a widely used technique to change the locations of the registers in a design without affecting the input/output functionality of the design. In the following, we formalize the retiming process.

A sequential circuit is represented by a directed graph $G(V, E)$ where each $v \in V$ is a design unit and each $e_{u,v} \in E$ is the edge corresponding to the flow of signal from the output of design unit u to the input of design unit v for any $u, v \in V$. Each edge $e_{u,v} \in E$ has an edge weight $w(e_{u,v})$ equal to the number of registers in that edge such that $w(e_{u,v}) \geq 0$. Each vertex $v \in V$ has a constant computational delay $d(v)$ such that $d(v) \geq 0$.

Given a *circuit* represented by a directed graph $G(V, E)$, a path p is a sequence of alternating vertices and edges such that each edge is a fan-out of the previous vertex in the sequence such that: computational delay of the path $(d(p))$ is the summation of the computational delays of all nodes in the path. Weight of the path p, $(w(p))$, is the summation of the weights of all the edges $e \in E$ in this path. A purely combinational path in a circuit will therefore have $w(p) = 0$. The clock period (c) of a circuit can thus be written as

$$c = \max_{p|w(p)=0} d(p). \tag{1}$$

A retiming label, $r(v)$, associated with each vertex $v \in V$ indicates the number of registers moved from the outputs to the input of the vertex v associated with the retiming label. Retiming is defined as assigning retiming labels $r(u)$ to all the design units $u \in V$ of the circuit. If the edge weights for $e_{u,v} \in E$ in the original circuit, G, changes to an edge weight $w_r(e_{u,v})$ after retiming, then

$$w_r(e_{u,v}) = r(v) + w_(e_{u,v}) - r(u). \tag{2}$$

Given a target clock period c, the minimum period global retiming of a circuit produces a retimed circuit subject to the following constraints on the retiming labels.

1) *Feasibility Constraint (FC):* For each edge $e_{u,v} \in E$, the edge weight $w_r(e_{u,v})$ in the retimed circuit must be non-negative, i.e., $w_r(e_{u,v}) \geq 0 \quad \forall e_{u,v} \in E$. Using (2)

$$r(v) - r(u) \leq w_(e_{u,v}) \quad \forall e_{u,v} \in E. \tag{3}$$

2) *Critical Path Constraint (CPC):* The delay $d(p)$ of all paths p with $w(p) = 0$ should be less or equal to the clock period after retiming.

Consider any two nodes u and v in G. There can be multiple paths from u to v. The minimum number of registers on any path from u to v is $W(u, v)$.

Let the computational delays of all n paths from u to v having $W(u, v)$ registers be $d(p_1), d(p_2), \ldots, d(p_n)$. Then, $D(u, v)$ is

$$D(u, v) = \max_{i=1}^{n} d(p_i). \tag{4}$$

With $D(u, v) > c$ for all paths from u to v, $r(v) - r(u) + w(u, v) \geq 1$ must hold to make the critical path's computational delay $\leq c$. Formally, the *CPC* can be restated as: for all paths from u to v with $D(u, v) > c$

$$r(u) - r(v) \leq w(u, v) - 1. \tag{5}$$

Thus, the objective of retiming is to identify the *retiming labels* r for all vertices that satisfy the constraints in (3) and (5). These can be solved using all pairs' shortest path as described in Section V.

IV. Analysis of the Impact of HLS on PSCA Security

In this section, we first explore the impact of HLS optimizations on the PSCA security of masked hardware implementations. Following that, we discuss the need for automated optimal register insertion during the translation from masked software to masked hardware.

A. Impact of HLS on Power Side-Channel Security

HLS carries out various optimizations that can be used to obtain a area/latency-optimized RTL from C/C++ code. Thus, given a gadget-based masked software code, HLS converts it to an RTL design applying these optimizations. We observe that the optimizations performed by HLS during this process impacts the PSCA security of a masked design. Using case studies of VivadoHLS [22] and Bambu [14], we present a few instances illustrating this observation.

1) HLS Front-End: The HLS front-end consists of the C compilation stage which translates the C/C++ code into an intermediate representation (IR) using a compiler like GCC or LLVM. This phase applies optimizations like expression simplification, code motion, reassociation, etc. that may hamper the security guarantees of the C-level masked implementation. Below we present a few instances of such optimizations and illustrate how they hamper the side-channel security of the IR.

Reassociation: LLVM compiler reassociates some of the intermediate computations causing insecure recombination of shares within the algorithm. [9] identified this in the Bambu HLS tool. For the C code in Listing 3 and its interpretation in Fig. 3(a), the XOR gates $i1$ and $i2$ are required after the cross-domain products $p2$ and $p3$ for secure masking. However, LLVM shifts the XOR gates to mask the products $p1$ and $p4$ instead. The absence of these XOR gates at the outputs of $p2$ and $p3$ results in an insecure circuit. Specifically, $y0$ in Listing 3 is computed as $y0 = ((a0 \otimes b1) \oplus z) \oplus (a0 \otimes b0)$, ensuring that cross-domain computations are masked before recombination. Reassociation causes the computation to be carried out as $y0 = ((a0 \otimes b0) \oplus z) \oplus (a0 \otimes b1)$ instead. We were unable to stop this optimization by LLVM using the Bambu tool version 0.9.6 with *#pragma HLS _interface ⟨variable⟩ none_registered* as done in [9].

```
1. #include "ap_int.h"
2. ap_int<9> domand (ap_int<9> a0, ap_int<9> a1,
3. ap_int<9> b0, ap_int<9> b1, ap_int<9> z,
4. ap_int<9> *y0, ap_int<9> *y1) {
5.     *y0 = ((a0 & b1) ^ z ) ^ (a0 & b0);
6.     *y1 = ((a1 & b0) ^ z ) ^ (a1 & b1);
7. return 0;}
```

Listing 1. DOMAND expression

```
1. #include "ap_int.h"
2. ap_int<9> multiply (ap_int<9>a0, ap_int<9>a1) {
3. #pragma HLS INLINE off
4.     return a0 & a1;
5. }
6. ap_int<9>domand (ap_int<9>a0, ap_int<9>a1,
7. ap_int<9>b0, ap_int<9>b1, ap_int<9>z,
8. ap_int<9>*y0, ap_int<9>*y1) {
9. #pragma HLS EXPRESSION_BALANCE off
10. #pragma HLS allocation instances=multiply limit=2
11. function //above pragma enables resource sharing
12. *y0 = (multiply(a0, b1) ^ z ) ^ multiply(a0, b0);
13. *y1 = (multiply(a1, b0) ^ z ) ^ multiply(a1, b1);
14. return 0;}
```

Listing 2. Resource-shared DOMAND

Expression Balancing in GCC: C/C++ code is often written as a sequence of operations, resulting in a long chain of operations at RTL after HLS. This can increase the delay in the design. By default, VivadoHLS rearranges the operations using associative and commutative properties. This rearranges operators to construct a balanced tree, reducing delay. However, such an optimization might hamper the security of the masked circuit. In Listing 1 for example, we have the DOMAND software masked code which gets reassociated into the expression: $y0 = ((a0 \otimes b0) \oplus z) \oplus (a0 \otimes b1)$ as a result of these optimizations by GCC. For integer operations expression balancing is enabled by default but can be disabled using the *#pragma HLS EXPRESSION_BALANCE off* directive as shown in Listing 2. For floating-point operations, expression balancing is disabled by default but may be enabled using the *#pragma HLS EXPRESSION_BALANCE*.

Thus, it is clear that the designer needs precise knowledge of all the optimizations to avoid such consequences.

2) HLS Backend—Scheduling and Resource Allocation: After the preprocessing stage, based on the target clock period, the scheduler decides the number of time steps and the scheduled time of each operation. For example, if the target clock is 10 ns for the example in Listing 2, all the operations are scheduled in one clock by VivadoHLS as shown in Fig. 2(b). A single-clock operation-chained datapath will be generated for the single-cycle schedule of Fig. 2(b). It is pointed out in [23], that such an operation chaining in the datapath also introduces side-channel vulnerabilities in the RTL. However, when the target clock is set to 1 ns, the design is scheduled in 2 time steps as shown in Fig. 2(c). The actual datapath depends on the resource optimization of the HLS tool. By default, most of the HLS tools generate a pipelined

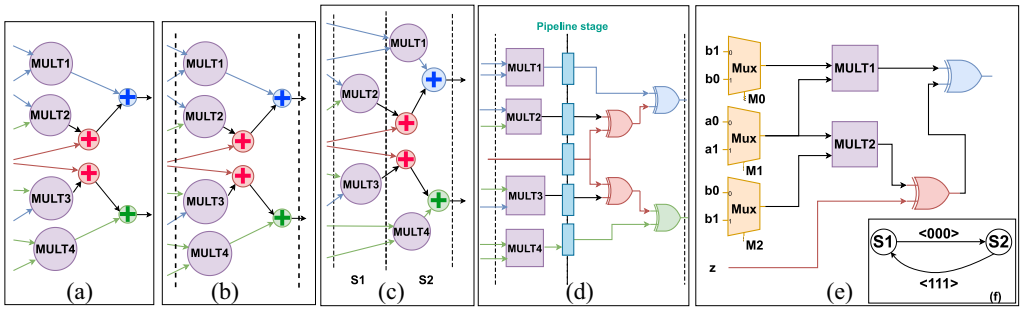

Fig. 2. Example: (a) CDFG of the behavior in Listing 3, (b) schedule for 10 ns, (c) schedule for 1 ns, (d) pipelined design, (e) resource-shared design, and (f) controller for resource-shared design.

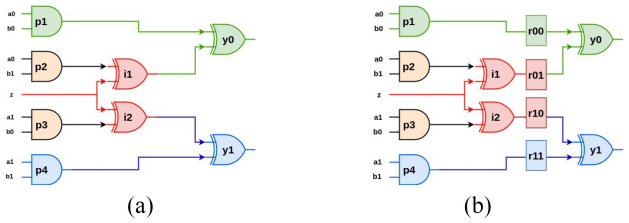

Fig. 3. (a) Software-masked DOMAND hardware realization. (b) Hardware-masked DOMAND circuit with masking and balancing registers.

```
1. int domand (bool a0,
2. bool a1, bool b0,
3. bool b1, bool r01,
4. bool *i1, bool *i2,
5. bool z,
6. bool *y0, bool *y1)
7. {p2 = a0 * b1;
8. i1 = p2 ^ z;
9. p3 = a1 * b0;
10. i2 = p3 ^ z;
11. p1 = a0 * b0;
12. p4 = a1 * b1;
13. *y0 = *i1 ^ p1;
14. *y1 = *i2 ^ p4;
15. return 0;}
```

Listing 3. DOMAND C code

design as the one in Fig. 2(d) generated from the schedule in Fig. 2(c). On the other hand, a user can specify resource constraints to restrict the area of the generated hardware. VivadoHLS allows the specification of resource bounds using pragmas like *#pragma HLS resource_allocation* for a function or operation to restrict its number of instances. Consider the Listing 2.[2] The number of multiplier instances has been restricted to 2 (line number 10). This results in a circuit with a datapath as shown in Fig. 2(e) where the resources are shared in a time-division multiplexed manner.

To control the execution of the datapath, a controller FSM as shown in Fig. 2(f) will also be generated by the HLS tool. Here, in state S1, the controller will assign $\langle M0M1M2\rangle = 000$ to execute the operations scheduled in state S1. Similarly, it will assign $\langle M0M1M2\rangle = 111$ in S2 to execute the operations scheduled in S2. Such controller FSM may further introduce glitches in the datapath as shown in [24]. The PSCA security of the generated RTL may be compromised due to these glitches. Thus, additional analysis is needed for such a controller.

3) Discussion: Thus, it is evident that masked designs are restrictive in terms of allowing for design-space exploration via rearrangement and resource sharing. Additionally, the yet unexplored security vulnerabilities of the HLS optimizations on various other cryptographic implementations present a vast range of possible security vulnerabilities. However, the steps in converting masked software to masked hardware for state-of-the-art masking schemes like DOM [2], HPC [8], COMAR [4] primarily require an operation by operation conversion into RTL from the IR. This should be followed by the insertion of registers at proper locations in the design to stop leakage due to glitches. For the DOMAND circuit in Fig. 3(b), the registers $r01$ and $r10$ are required as shown in Section III. Thus, it may not be advisable to use a generic HLS tool to directly generate PSCA secure masked hardware. Instead, the HLS process seeking to leverage the software-level masking security must focus on the optimal insertion of registers. In the next section, we discuss how this can be optimally done.

[2]Listings 1–3 are different representations of the same DOMAND behavior. We took three variations to illustrate the various security implications of HLS.

B. Motivation of Our Work

Given a software-level masked implementation of a cryptographic algorithm, we need to add registers in specific places in order to stop the propagation of glitches. HLS tools have *pragma* directives to allow such annotation.

However, there is no guarantee that an HLS tool will not choose to enforce these pragmas due to the other constraints. For example, the Bambu HLS Version 0.9.6 ignored the *#pragma HLS none_registered* when applied on our example in Listing 3. Further, a design may have many parallel paths. To preserve the latency of the circuit, after the insertion of registers in specific paths, the parallel paths would also require register insertion. For example, consider the circuit given in Fig. 3(a). This circuit corresponds to the DOMAND software specification in Listing 3. DOM-masked hardware requires the insertion of registers $r01$ and $r10$, as shown in Fig. 3(b). With only these two registers, the inputs to the gates $y0$/$y1$ have different latency. This will results in incorrect circuit behavior. Thus, the *balancing registers* $r00$ and $r11$ must be inserted. Fig. 3(b) is the circuit corresponding to an HLS-C input annotated as in Listing 4. For bigger circuits,

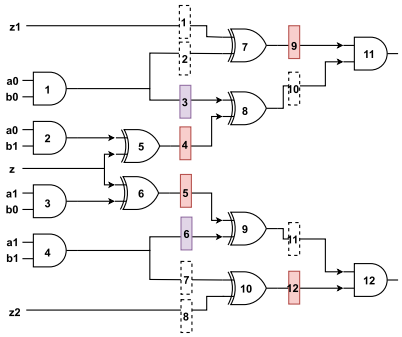

Fig. 4. Example to illustrate the need for optimal register balancing in masked circuits.

there may be many parallel paths. Therefore, register annotations that facilitate register insertion in parallel paths need automation.

For register insertion in multiple locations, the number of *balancing registers* and the design latency must be minimized as well. Consider the circuit in Fig. 4. Let us assume that the registers numbered 4, 5, 9, and 12 are required by a masking scheme for security. To insert registers 4 and 5, registers 1–3 and 6–8 need to be inserted to balance the parallel paths. Now, inserting registers 9 and 12 will require the insertion of registers 10 and 11 to balance the paths at gates 11 and 12. Thus, a total of 12 registers need to be inserted, resulting in an overall latency of 2. However, careful examination of the circuit reveals that registers 1, 2, 7, 8, 10, and 11 can be optimized out. With the other 6 registers (3–6, 9, and 12) the circuit has an overall latency of 1 and all the parallel paths in the circuit are balanced. This illustrates the need for optimal register balancing in masked circuits.

In our opinion, modern HLS tools perform too many optimizations which are counter-productive for PSCA secure hardware generation. There should be one-to-one translation from the C code to RTL. Moreover, register insertion and balancing are the most important measures to stop the propagation of glitches while maintaining minimum register usage and latency. None of the existing HLS tools can do these tasks in an automated way. *This calls for a domain-specific HLS tool for masked designs*. Such a tool would not create vulnerabilities due to HLS and retain the security properties required while performing register balancing automatically. In this work, we have developed an automated register balancing approach at behavioral level using the concept of retiming [21]. The concept of retiming is presented in Section III-C.

V. Proposed MaskedHLS Flow

A common approach toward masking cryptographic implementations is to replace the unmasked operations in the overall implementation with *masked gadgets*. The state-of-the-art masked gadgets like DOMAND, HPC1, HPC2, and COMAR discussed in Section III-B have locations for the insertion of registers. These masking gadgets at hardware and software differ only in the presence of registers for the hardware case. The registers ensure glitch-resistant masking. Thus, in conversion from software to hardware-masked designs we need to annotate the software-masked code with directives for the placement of registers. In VivadoHLS we could realize this by defining a template class $template < classT > T\ reg(Tx)$ and using it akin to a function call. In addition to inserting registers in specific locations, we also need to identify an optimal number of pipelined states and the registers required to balance parallel paths as discussed above. Our proposed MaskedHLS does exactly that while ensuring the PSCA-security of its output.

The input to MaskedHLS is a software implementation of a cryptographic algorithm that has been masked using gadgets. The gadgets have locations for register insertion for glitch-robust masking and these locations are indicated in the input software implementation using annotations. Given these inputs, MaskedHLS identifies the minimum possible pipeline stages in the circuit satisfying all necessary register requirements specified by the annotations. In the next step, the register balancing module of MaskedHLS identifies all locations in parallel paths where registers need to be added. This produces an annotated C code on which MaskedHLS performs a one-to-one translation into RTL code with registers inserted in all the places as required by masking as well as balancing. Finally, MaskedHLS creates a pipelined RTL design. The overall flow of MaskedHLS is shown in Fig. 6. The steps are discussed in detail below.

A. Register Balancing at Behavioral Level

Given an unmasked software implementation in C/C++, the masked software is obtained by replacing the nonlinear operations with the corresponding gadgets according to the masking scheme. The masking gadget/scheme specifies where registers must be inserted to maintain PSCA security in the corresponding hardware. These locations are indicated by annotations in the C/C++ input as $\langle lhs\ of\ operation \rangle = reg(\langle rhs\ of\ operation \rangle)$. We need to put a register in those locations and balance parallel paths automatically, with minimum pipelined stages. To do so, the annotated C code is converted into an abstract syntax tree (AST). This AST has the same structure as the graph definition of the sequential circuits described in Section III-C. We develop a method that creates a special model of the AST and utilizes retiming logic on it to achieve the above goal.

Let us consider that the target clock period is c in hardware implementation. For a given software code, the target clock period is always known. The AST is modified as follows to create the *HLS model*.

1) *Adding Source and Sink Nodes:* A source node is added to the AST for all the inputs with edge weights 0, and a sink node is added for all the output nodes with edge weights 0.
2) *Adding a Back-Edge:* A directed edge is added from the sink node to the source node.

Such a model will allow us to add the additional registers in the back edge, and later, balancing will move them into the desired locations. In the created *HLS model*, we make the following changes to enable *register balancing*.

1) *Adding Dummy Nodes:* After each node $v \in V$, which has an annotation for a register insertion succeeding it, a *dummy-node* v' is inserted.

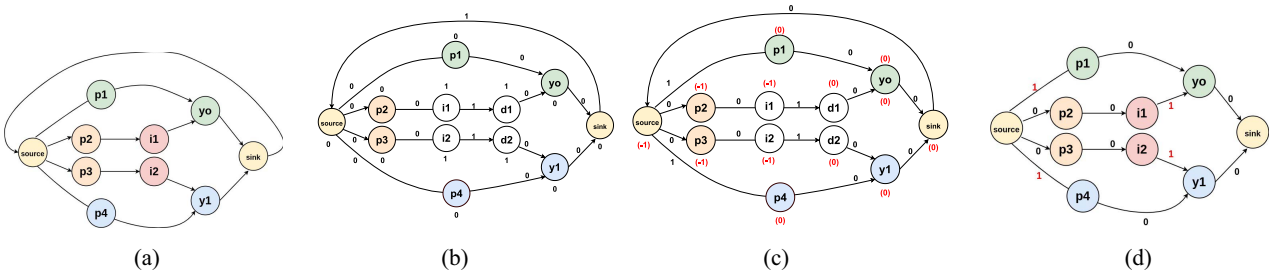

Fig. 5. (a) AST before retiming. (b) HLS-model with back edge. (c) HLS-model after retiming (retiming labels shown in parenthesis). (d) Final circuit after removing dummy nodes and back edge.

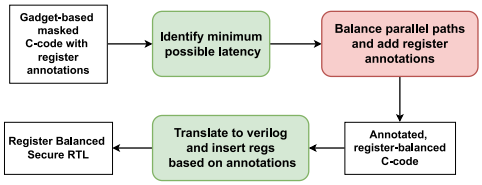

Fig. 6. Flow of MaskedHLS tool.

2) *Assigning Computational Delay to Nodes:* The nodes v, after which registers must be added, and the dummy nodes v' are assigned computational delay of $d(v) = c$ and $d(v') = c$, respectively. All other nodes $u \in V$ apart from the ones assigned a computational delay of c in the previous step are assigned computational delay $d(u) = 0$.

Any path p involving the edge $e_{v,v'}$, will have a delay of $2c$. Therefore, such a path will fail to meet the target clock period of c. Hence, a register must be inserted in that path at the location between v and v' to meet the CPC required for minimum period global retiming as defined in Section III-C. By virtue of retiming, a register will be added in the parallel paths as well.

For the example in Listing 3, a *HLS model* is constructed in Fig. 5(a). One dummy node is inserted following each white-colored node (cross-domain nodes) and colored white as in Fig. 5(b). Assuming the target clock period is 1, all white-colored nodes are assigned the computational delay $d(v) = 1$. For all other nodes, the computational delay $d(u) = 0$.

B. Finding the Maximum Number of Register Annotations in Path

Among all the paths in the *HLS model* between the source node and the sink node, the maximum number of annotations for register insertion, thereafter referred to as *maximum extra regs*, is identified using a depth first search (DFS). Before being translated to RTL, the *HLS model* should contain these many registers in all parallel paths between source and sink. Therefore, *maximum extra regs* will determine the latency of the generated RTL. These extra registers are added as weight in the back edge between the source and the sink. For all other edges, the edge weight is assigned to zero. The *HLS model* is obtained from the C code, which initially had no registers.

C. Calculation of Retiming Constraints

For each node, we consider the retiming label $r(v)$. FCs are calculated for each edge $e_{u,v}$ and CPCs are calculated for each path from u to v such that $D(u, v) > 1$. For the HLS model in Fig. 5(b), the FCs are shown in Table I and some of the CPCs are shown in Table II.

TABLE I
FCs FOR THE CIRCUIT IN FIG. 5(B)

$r(i2) - r(p3) <= 0$	$r(i1) - r(p2) <= 0$
$r(y1) - r(p4) <= 0$	$r(y0) - r(p1) <= 0$
$r(p4) - r(source) <= 0$	$r(p3) - r(source) <= 0$
$r(p2) - r(source) <= 0$	$r(p1) - r(source) <= 0$
$r(sink) - r(y1) <= 0$	$r(sink) - r(y0) <= 0$
$r(source) - r(sink) <= 1$	$r(d2) - r(i2) <= 0$
$r(d1) - r(i1) <= 0$	

TABLE II
CPCs FOR FIG. 5(B)

$r(p4) - r(d2) <= 0$	$r(p4) - r(d1) <= 0$
$r(p3) - r(p4) <= 1$	$r(p3) - r(p2) <= 1$
$r(p3) - r(p1) <= 1$	$r(p3) - r(i1) <= 1$
$r(p3) - r(y1) <= -1$	$r(p3) - r(y0) <= 1$
$r(p3) - r(source) <= 1$	$r(p3) - r(sink) <= -1$
$r(p3) - r(d2) <= -1$	$r(p3) - r(d1) <= 1$
$r(p2) - r(p4) <= 1$	

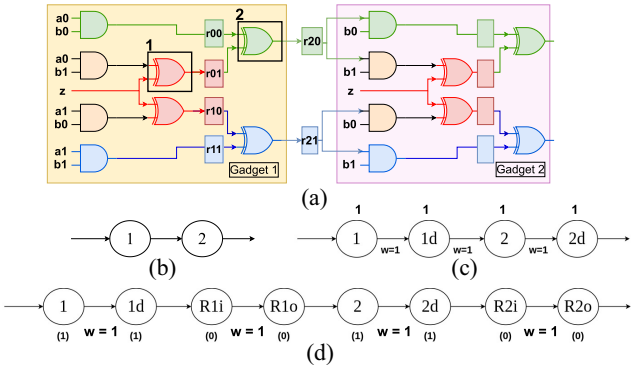

Fig. 7. (a) DOMAND-composed circuit. (b) Part of a circuit with two register insertions in series. (c) Register insertion in series. (d) Register insertion in series using an extra register.

D. Inserting Registers in Series

In gadget based masking a situation may thus arise where a single path contains two locations where registers need to be inserted. Consider the two DOMAND gadgets composed with each other in Fig. 7(a). To make the Gadget1-Gadget2 combination *composable*, the registers $r20$ and $r21$ must be inserted as shown in Fig. 7(b). The target clock is $c = 1$. If we want to insert registers after gates 1 and 2 then we have to specify retiming constraints on both of them, thus, inserting

a dummy node after each of them, as shown in Fig. 7(c). Now the $d(v)$ values for the nodes become $d(1) = 1$, $d(1d) = 1, d(2) = 1$, and $d(2d) = 1$. The $D(u, v)$ values are now: $D(1, 1d) = 2, D(1d, 2) = 2$, and $D(2, 2d) = 2$. Thus, these 3 edges $e_{1,1d}$, $e_{1d,2}$, and $e_{2,2d}$ violate the *CPCs*. Hence, 3 registers are placed into the circuit at locations where $w = 1$ in Fig. 7(c). Here, the register between nodes $1d$ and 2 is not needed, and as a result, the circuit is not balanced with minimum latency. The *adding dummy nodes* step from Section V-A is updated by the addition of these steps to address this issue as follows.

1) A redundant register at the edge between $1d$ and 2 is deliberately inserted into the HLS model. This causes the critical path from $1d \rightarrow 2$ to break into two paths that meet the target clock $c = 1$. This register is later removed after retiming.
2) To ensure that this register is not moved by retiming, it is *locked* with two nodes at its input and output, respectively. After the dummy node $1d$, two other nodes $R1i$ and $R1o$ are inserted. Similarly for node $2d$.
3) *Register-lock* constraints are added for each R. r(RIn) == r(ROut). This constraint ensures that the number of registers moved into the edge $e_{\text{RIn,ROut}}$ equals the number of registers moved out of this edge. This means registers can move across this edge without affecting the existing edge weight; thus, it locks the register.

This results in the AST in Fig. 7(d) and the subsequent steps can be performed on it.

E. Finding the Retiming Labels

To find the values of *retiming labels* that satisfy these constraints, we construct a constraint graph as follows.

1) For each *retiming label* $r(v)$, a node v' is created.
2) If N is the number of nodes in the circuit, a $N + 1$th node is created.
3) For each inequality $r(u) - r(v) \leq k$, an edge $v' \rightarrow u'$ from the nodes v' to u' of weight k is drawn. It is possible that $k < 0$ for some set of retiming labels.
4) For each node $v' \in V'$, an edge $N + 1 \rightarrow v'$ from the nodes $N + 1$ to the v' with weight 0 is drawn. At this point, the graph is guaranteed to not contain any negative edge cycle as shown in Lemma 2.

Using Lemma 1, the shortest path from $N + 1$ to any node v' will give the correct *retiming label* corresponding to v'. Since, there are no negative weight cycles in the constraint graph G' (as shown in Lemma 2), we can apply the Dijkstra's single-source shortest path algorithm to obtain the *retiming labels* $r(v)$. The retiming labels obtained as a solution for the HLS model in Fig. 5(b) are shown (within parenthesis) for each vertex in Fig. 5(c). The retiming labels satisfying all the constraints will give us the correct locations in the circuit where registers have to be inserted. The register balanced design obtained using these $r(v)$ values is shown in Fig. 5(d). After retiming, all dummy nodes and edges are removed. MaskedHLS will generate a register balanced C code from this HLS model. The register balanced, annotated C code, corresponding to Listing 3 is shown in Listing 4. It may be noted that registers are added in the designated locations and in all parallel paths.

```
1. int domand (bool a0,
2. bool a1, bool b0,
3. bool b1, bool z,
4. bool *i1, bool *i2,
5. bool *y0, bool *y1)
6. {p2 = a0 * b1;
7.  i1 = reg(p2 ^ z);
8.  p3 = a1 * b0;
9.  i2 = reg(p3 ^ z);
10. p1 = reg(a0 * b0);
11. p4 = reg(a1 * b1);
12. *y0 = *i1 ^ p1;
13. *y1 = *i2 ^ p4;
14. } return 0;}
```

Listing 4. DOMAND with register annotations.

F. Generating Pipelined RTL Design

The final phase of MaskedHLS takes the register-annotated C-code obtained from the previous step and generates RTL from it. This translation of C-code to Verilog is done via a one-to-one mapping from the AST at C to RTL. The tool places registers according to the annotations in the C-code. Our tool does not apply any optimizations in the process. This effectively creates a pipelined RTL design with the number of pipeline stages equal to the maximum number of registers in a path (as identified in Section V-B).

VI. Correctness of MaskedHLS

In this section, we prove the correctness of our register balancing approach in MaskedHLS. We also show that MaskedHLS add minimum number of pipelined stages.

Lemma 1: The shortest path from $N + 1$ to v' in the constraint graph will give the *retiming label* satisfying the constraints.

Proof [(By Induction) Base]: There is a direct edge form $N+1$ to each vertex v' with weight 0. If this edge is the shortest path from $N + 1$ to v', then the retiming label $r(v') = 0$. It means there will be no registers moved across v'.

Now, assume the shortest path to v' is through u', i.e., $N + 1 \xrightarrow{0} u' \xrightarrow{-k} v'$ is the shortest path. The edge $e_{u',v'}$ came from the retiming constraint $r(v') - r(u') \leq -k$. The retiming label of u' must be 0. So the value of the shortest path to v', i.e., $-k$ will satisfy the constraint.

Inductive Step: Now assume we have another vertex u' in the constraint graph with a direct edge to v' with the edge weight $w_{u',v'} = l$. Let the shortest path to u' of length $-m$ (from the base case $m \geq 0$) already exist and be equal to the value of the retiming label of u': $r(u') = -m$. Therefore, given this node u', the shortest path from $N + 1$ to v' either passes through u' or does not.

Case I: The shortest path from $N + 1$ to v' is the path $N + 1 \xrightarrow{0} v'$. The direct edge from $N + 1$ to v' is the shortest path. Therefore, $w(N+1 \xrightarrow{-m} u' \xrightarrow{l} v') \geq w(N+1 \xrightarrow{0} v') \implies -m + l \geq 0$. Putting the value of $r(u') = -m$ in this equation we get: $l + r(u') \geq 0 \implies 0 - r(u') \leq l \implies r(v') - r(u') \leq l$ which is the constraint on the vertex v'. Hence, the constraint is satisfied in this case.

Case II: The shortest path from $N+1$ to v' is $N+1 \xrightarrow{-m} u' \xrightarrow{l} v'$. So the shortest path's weight is $-m+l$. Here, $r(v') = -m+l$. Now, we have to show that the constraint on v', $r(v') - r(u') \leq l$ is satisfied with the retiming constraints. Putting the value $r(v') = -m+l$ in the constraint's RHS we have $r(v') - r(u') \implies -m+l-(-m) \implies l$ which is $\leq l$. Hence, the constraint for this case is satisfied. ■

To find the correct set of retiming labels, a solution to the constraint graph must be found. The shortest path algorithm can be used for that purpose. For shortest path to give a solution, which is a correct set of retiming labels, the graph should contain no negative weight cycles as otherwise no solution can be reached using shortest path.

Lemma 2: The constraint graph contains no negative weight cycles.

Proof Idea: We start by considering a hypothetical negative weight cycle C in the constraint graph, the weight of which is: $w_C = \sum_i w_{i,i+1}$, where $w_{i,i+1}$ denotes the weight of the edge $e_{i,i+1}$ in C. We observe that for any cycle C in the *HLS model*, there is one or more (due to CPCs there may be multiple edges in the constraint graph corresponding to one edge between two nodes in the *HLS model*) cycle in the constraint graph derived from that cycle. Also, since there are no loops in the input circuit, thus the only cycles in the *HLS model* will contain the edge $e_{\text{sink},src}$. Hence, for each cycle in the constraint graph, there exists an equivalent path in the HLS model from $src \rightarrow$ sink. The weights along this path represent the number of registers moved across the vertices in the path, which is the total number of registers contained in the path. Since a circuit cannot have a path from input to output with a negative number of registers, the sum of weights along the path must be non-negative. By removing the edge $e_{\text{sink},src}$ from cycle C, we obtain a path from source to sink in the HLS model. The sum of weights along this path must also be non-negative. However, the weight of cycle C is negative. This leads to a contradiction. Thus, our initial assumption of the existence of a negative weight cycle in the constraint graph is false.

At this point, it is noteworthy that our *register balancing* procedure will always terminate with a solution. It will never be the case that an infeasible set of constraints is generated for which there is no solution possible. As discussed in Section V-B, we identify the maximum number of registers needed in a path and assign that as the weight of the edge between the source and the sink. These registers adequately satisfy all constraints. In Lemma 3 we prove the termination of our procedure.

Lemma 3: Register balancing will always terminate with a solution resulting in the same latency as the number of registers inserted into the back edge.

Proof: Let the circuit obtained via register balancing using our method be C. Let the *maximum extra regs* value we have obtained after the DFS of the AST with the annotations be m. For a minimal latency circuit, we need to have a circuit with m registers in all parallel paths. Say our circuit C has $m+k$ registers in a parallel path after register-balancing. Then, our circuit C will have an un-optimal latency. We have to prove that such a scenario will never be reached by our *register balancing* procedure. So let us assume there is a path p after retiming with a weight $w(p) = m+k$ for some $k > 0$. Then, for this path $src \rightarrow v_i \rightarrow v_{i+1} \rightarrow \cdots \rightarrow v_N \rightarrow$ sink, following from the convention of retiming rules in Section III-C where weight of each edge before retiming is $w(e_{j,j+1})$ and after retiming are $w_r(e_{j,j+1})$, we have:

$$w_r(e_{src,v_i}) = r(v_i) - r(src) + w(e_{src,v_i})$$
$$w_r(e_{v_i,v_{i+1}}) = r(v_{i+1}) - r(v_i) + w(e_{v_i,v_{i+1}})$$
$$\cdots$$
$$w_r(e_{v_N,\text{sink}}) = r(\text{sink}) - r(v_N) + w(e_{v_N,\text{sink}}).$$

Adding them all, we get the weight of the path $w(p)$ to be,

$$w_r(e_{src,v_i}) + w_r(e_{v_i,v_{i+1}}) + \cdots + w_r(e_{v_N,\text{sink}}) = w(p) = r(\text{sink}) - r(src)$$
$$\implies r(\text{sink}) - r(src) = m+k.$$

Since we must move the registers from the sink $\rightarrow src$ edge into the path p via retiming, therefore $r(src) = -m$. $r(\text{sink}) = 0$. Therefore, following from above, $r(\text{sink}) - r(src) = m+k \implies 0-(-m) \neq m+k$, which is a contradiction. Thus, our initial assumption is wrong. Hence, retiming results in a circuit with an optimal latency. ■

Lemma 4: Retiming does not change the PSCA security of the circuit.

Proof: The retiming procedure only inserts registers at the locations annotated in the input C code and the locations requiring balancing. Introducing registers at locations other than the locations annotated (balancing registers) does not compromise the security. Since we lock all the existing registers using register locking constraints, retiming will not move any existing registers. Therefore, there is no removal, insertion or movement of any circuit components during register-balancing that can impact the security guaranteed by masking. Hence, retiming does not impact the PSCA security of the circuit. ■

A. Complexity Analysis

The complexity of MaskedHLS is upper bound by the complexity of the register-balancing procedure. The calculation of the D and W matrices together takes $O(n^3)$ time where n is the number of nodes in the AST of input. This is because they can be obtained using all pairs shortest-path. Following that, the FCs are obtained for each edge of the graph in $O(n^2)$ [as the number of edges in a graph is $O(n^2)$] and CPCs for each edge $e_{u,v}$ where $D(u,v) > c$ which is at most $O(n^2)$. These constraints are then modeled using a constraint graph which is linear in the number of constraints which is $O(n^2)$. These constraints are solved again using the Dijkstra's algorithm on the constraint graph which takes $O(n^3)$ (i.e., $V+E$, $|V| = n$) here n is the number of nodes in the original HLS model. Therefore, the time complexity of the balancing procedure is $O(n^3)$ in the number of nodes in the retiming model n.

TABLE III
RESULTS FOR MASKEDHLS

Design	#ann_regs	#bal_regs	#total_regs	#C	#nodes	#RTL	Runtime (s)
PRESENT_DOMAND	16	36	52	83	105	299	0.33
PRESENT_HPC1	32	68	100	84	169	454	0.40
PRESENT_HPC2	48	82	130	91	168	420	0.33
PRESENT_COMAR	56	38	94	94	209	515	0.88
AES_DOMAND	72	999	1071	485	1308	5307	11.22
AES_HPC1	216	1689	1905	515	1668	7707	25.77
AES_HPC2	432	1587	2019	481	1884	7875	64.87
AES_COMAR	468	2486	2954	495	2322	10261	119.17

VII. EXPERIMENTAL RESULTS

A. Implementation and Benchmark Details

MaskedHLS makes use of Pycparser [25] to parse the AST of the input C-code on which the balancing procedure and the one-to-one transformation to RTL are performed. We have tested MaskedHLS on four different variants of the PRESENT Cipher's 4-bit S-box [26] and Canright's AES-256 S-box [27] masked using four different gadgets: 1) the DOMAND gadget; 2) the HPC1 gadget; 3) the HPC2 gadget; and 4) the COMAR gadget, respectively. The source code of MaskedHLS is available on github.[3]

B. MaskedHLS Synthesis Results

Table III presents the results of MaskedHLS on all the eight test-cases. The runtime of MaskedHLS is dependent on the number of nodes being processed. Specially, MaskedHLS takes an average of 54 s on the AES S-box designs with an average of 1795 nodes; and an average of 0.48 s on the PRESENT S-box designs with an average of 422 nodes. AES_COMAR took a significantly longer time to synthesize using MaskedHLS due to the higher number of constraints generated during register balancing due to a higher number of critical paths in the design for AES_COMAR compared to the other AES S-box designs. The major part of the time is taken in register balancing.

In Table III, the number of registers annotated initially for gadgets (#ann_regs) and the number of additional registers inserted by MaskedHLS for balancing (#bal_regs), the total number of registers (#total_regs) and the lines of code in input C (#C) and RTL (#RTL) are also shown. As seen in Table III, the runtime of MaskedHLS on a 6-core Intel i7-8700 CPU operating at 3.20 GHz is less than 1 s for all PRESENT S-boxes and less than two min for all AES S-boxes.

The generated RTLs from MaskedHLS were synthesized to netlists using Synopsys design compiler (DC) using the TSL18FS120 cell library from Tower Semiconductor Ltd. at 180 nm technology node. To ensure that the downstream synthesis tool does not impact the security of the generated RTL via optimizations, we added commands (like *set_dont_touch*) in the synthesis script. To compare the area and latency overhead due to balancing, the gadget-based masked c-codes for all designs sans the registers were synthesized to RTL. These, too, were converted to netlist using the Synopsys DC with the same library. The area and latency data from the DC's synthesis report were obtained for both versions of the designs while constraining the circuit to use only and, xor and invert gates and registers wherever necessary. Table IV shows the comparison of total area and timing for all the designs against the versions without registers. It may be observed that the area has increased by 2.97 and 6.85× on an average for PRESENT S-boxes and AES S-boxes, respectively, after inserting the register. We have also added area and timing results for the AES S-box and PRESENT S-box designs in their native form (without masking) to show the area overhead due to masking (first row in each set of results in Table IV). The area overhead of Masking is 5.9, 9.4, 11.5, and 9.5× for PRESENT S-box masked using DOM, HPC1, HPC2, and COMAR, respectively. For the Canright's AES S-box masked using DOM, HPC1, HPC2, and COMAR, the area overhead due to masking is 33.1, 56.3, 41.3, and 48.9×, respectively. This increase in area is because of the additional registers added by HLS. This is also due to the fact that the technology mapping for a pipelined design does not allow for much area optimization versus the combinatorial circuits of the designs without registers which get largely optimized. The clock period (in ns) for designs generated by MaskedHLS is less due to the pipeline stages added through registers.

[3]https://github.com/nilotpolas/MaskedHLS

TABLE IV
AREA AND TIMING OVERHEAD COMPARISON WITH DESIGNS WITHOUT REGISTERS

Design	Area(wo_reg)	Area(w_reg)	Timing(wo_reg)	Timing(w_reg)
PRESENT_unmasked	940.11	NA	1.11	NA
PRESENT_DOMAND	2639.22	5546.05	1.64	0.93
PRESENT_HPC1	2614.83	8815.18	1.69	0.83
PRESENT_HPC2	3220.92	10788.05	1.85	0.94
PRESENT_COMAR	2892.56	8936.05	1.82	0.98
PRESENT_average		2.97x		0.52x
AES_unmasked	55728.13	NA	18.99	NA
AES_DOM	1002202.72	1841877.19	28.01	5.72
AES_HPC1	1004612.10	3136636.19	28.94	4.12
AES_HPC2	1028632.47	2305685.39	31.60	4.50
AES_COMAR	134810.27	2727581.43	31.12	2.44
AES_average		6.85x		0.13x

wo_regs corresponds to the gadget based masked circuit without registers, w_reg corresponds to the output of MaskedHLS(gadget based masked circuit with registers according to the masking scheme and balancing registers in parallel paths). Timing is in nanoseconds.

TABLE V
COMPARISON OF REGISTER AND LATENCY SAVINGS USING MASKEDHLS AND MANUAL METHODS

Design	Registers			Latency		
	MaskedHLS	Manually	Saving(%)	MaskedHLS	Manually	Saving(%)
PRESENT_DOMAND	52	168	69.0	3	5	40
PRESENT_HPC1	100	290	65.5	5	9	44.5
PRESENT_HPC2	130	398	67.3	5	10	50
PRESENT_COMAR	94	570	83.5	5	9	44.5
AES_DOMAND	1071	4752	77.4	5	9	44.5
AES_HPC1	1905	6578	71.0	7	13	46.1
AES_HPC2	2019	8901	77.3	7	13	46.1
AES_COMAR	2954	14987	80.2	13	25	50
Average			73.9			45.7

C. Register Balancing Results

MaskedHLS optimizes balancing registers and hence leads to a decrease in the number of registers in the RTL versus the circuit derived via conventional methods as discussed in Section IV-B. As can be seen in Table V, on an average over both PRESENT S-box and AES S-box designs combined, MaskedHLS results in an RTL with 73.9% lesser number of registers and 45.7% less latency while ensuring PSCA-security versus the conventional approach where registers are placed in all parallel paths manually without any optimization as

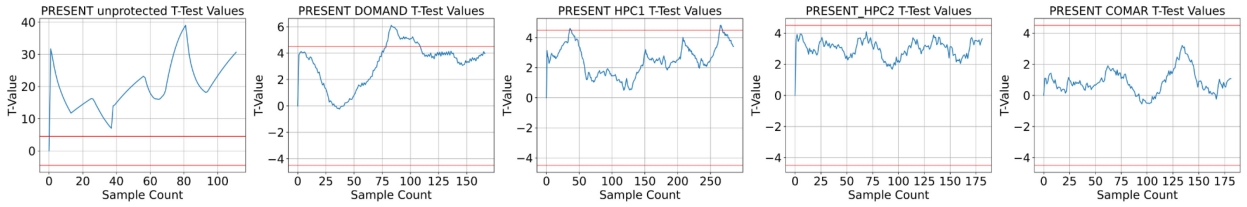

Fig. 8. T-values for: (a) PRESENT_unmasked. (b) PRESENT_DOMAND. (c) PRESENT_HPC1. (d) PRESENT_HPC2. (e) PRESENT_COMAR. (For each design, the *x*-axis contains T-values and the *y*-axis contains the number of sample points per plaintext.)

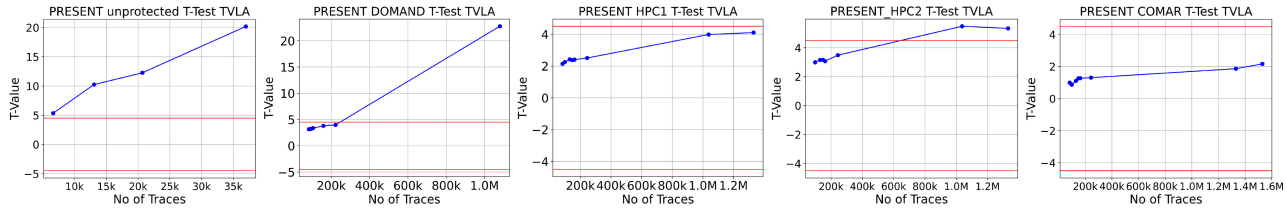

Fig. 9. TVLA values versus the number of traces: (a) PRESENT_ unmasked. (b) PRESENT_DOMAND. (c) PRESENT_HPC1. (d) PRESENT_HPC2. (d) PRESENT_COMAR. (For each design, the *x* axis contains T-values and the *y*-axis contains the number of traces for which that TVLA value was observed.)

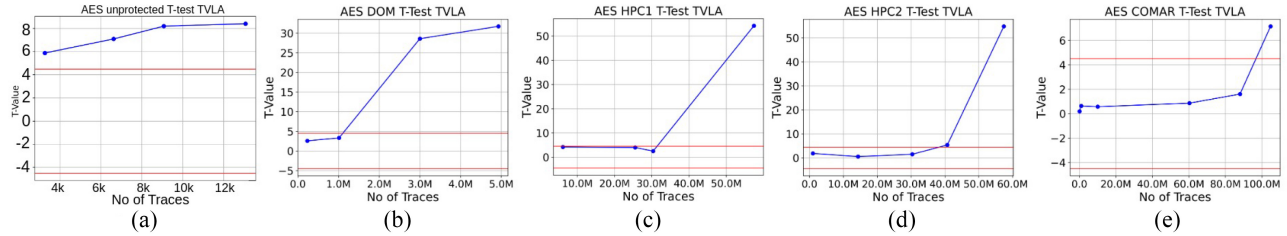

Fig. 10. TVLA values versus the number of traces. (a) AES_unmasked. (b) AES_DOMAND. (c) AES_HPC1. (d) AES_HPC2. (e) AES_COMAR. (For each design, the *x*-axis contains T-values and the *y*-axis contains the number of sample points per plaintext.)

proposed in this work. This result affirms our objective of obtaining minimum latency and registers.

D. PSCA Security Analysis

It is necessary to verify that the output produced by MaskedHLS is indeed secure. We performed the test-vector leakage analysis (TVLA) [28] of the power traces of RTL obtained through MaskedHLS and compared them with those of unprotected (unmasked) design. Each RTL design was compiled into netlist using Synopsys DC and TSL18FS120 cell library. Then, the netlist was simulated using a testbench in the Synopsys VCS simulator. The switching activity of the circuit was dumped into the value change dump (VCD) file. We then used Synopsys PrimeTime, which used netlist generated through DC and VCD file generated through VCS compiler, giving the power traces in fast signal database (FSDB) format. After that, Synopsys Custom WaveView tool was used to extract power traces in CSV format from the FSDB file. On this data, we applied the conventional TVLA method [28] to obtain the t-values. The t-value corresponding to one plaintext for all PRESENT designs is shown in Fig. 8. Clearly, the unprotected design is leaking. Among the PRESENT S-box designs, PRESENT_HPC1 and PRESENT_COMAR are more secure compared to PRESENT_DOMAND and PRESENT_HPC2 whose t-value exceeded $\|4.5\|$ in 18% and 16% cases, respectively.

We extracted power traces ranging from 5000 to 100 million. The objective was to check how good the protection was. The higher the number of traces for which the t-value does not cross the threshold of $\|4.5\|$, the more secure the design is. Fig. 9 shows the trend of TVLA-values for this experiment for the PRESENT designs. The unprotected design crosses the $\|4.5\|$ mark for around 6000 traces. Whereas the COMAR and HPC1-masked designs are secure upto 1.3 million traces. The threshold value crosses the $\|4.5\|$ mark for around 220 thousand traces for DOMAND and around 600k for HPC2. HPC2 uses lesser random variables as compared to HPC1 as shown in Fig. 1(c). The design with COMAR is the most secure among all gadgets available. The experimental results are aligned with the theoretical analysis of the gadgets.

We also performed TVLA on the output of VivadoHLS [22] on the DOMAND and COMAR masking gadget protected - PRESENT S-box. It can be seen in the results in Fig. 11(a) and (b) that the security is significantly lesser (20k and 800k traces, respectively), in terms of number of traces to obtain a correlation, compared to MaskedHLS output ($\geq$ 200k traces and $\geq$ 1.3 million traces, respectively). We observed similar results for PRESENT S-box using other gadgets (HPC1 and HPC2). However, due to space limitations, we could not add

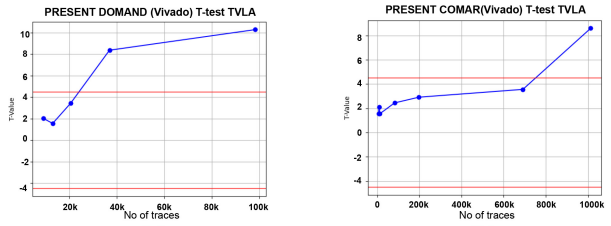

Fig. 11. TVLA values versus number of traces for: (a) PRESENT_DOMAND and (b) PRESENT_COMAR, both synthesized using VivadoHLS.

all results. This reaffirms our motivation for a domain-specific tool for PSCA-secure designs. Also, to test the efficacy of our tool on bigger benchmarks, we have tested MaskedHLS on the Canright's AES S-box [27] masked using the DOM, HPC1, HPC2, and COMAR gadgets. The TVLA results show that the key could not be revealed up to 1 million traces for AES_DOMAND, 30 million traces for AES_HPC1, 40 million traces for AES_HPC2, and 100 million traces for COMAR as shown in Fig. 10. The result for COMAR corresponds to the claims reported in the original proposal of the COMAR gadget [4]. Thus, our experiments clearly show that MaskedHLS generates PSCA-secure RTL from the masked software code.

VIII. Conclusion

Secure masked hardware design is a nontrivial task that requires significant time and expertise. Therefore, obtaining masked hardware from masked software using HLS is beneficial. We have shown that the existing HLS actually does not guarantee the PSCA security of the generated RTL. To address this shortcoming, we have developed MaskedHLS to generate PSCA secure RTL from the masked software version of the cryptographic designs. Experiments with two S-boxes for four gadgets show that MaskedHLS save on an average 73.9% of registers and 45.7% of latency as compared to the conventional processes. The TVLA analysis affirms the PSCA security of generated RTLs. The state-of-the-art PSCA-secure hardware design [7] focuses on reducing the number of registers, design latency and randomness. In this regard, having minimum balancing registers is crucial. MaskedHLS generates RTL that uses minimum latency and registers to achieve PSCA security. In future, we plan to integrate randomness optimization strategies into MaskedHLS.

Acknowledgment

The authors would like to thank Sujeet Narayan Kamble and Mr. Thockchom Birjit Singha for their help in the implementation of some parts of MaskedHLS.

References

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. AICC*, 1999, pp. 388–397.
[2] H. Groß, S. Mangard, and T. Korak, "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order," Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Rep. 486/2016, 2016.
[3] G. Cassiers, B. Grégoire, I. Levi, and F.-X. Standaert, "Hardware private circuits: From trivial composition to full verification," *IEEE Trans. Comput.*, vol. 70, no. 10, pp. 1677–1690, Oct. 2021.
[4] D. Knichel and A. Moradi, "Composable gadgets with reused fresh masks—First-order probing-secure hardware circuits with only 6 fresh masks," Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Rep. 1141/2023, 2023.
[5] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of AES," in *Proc. SAC*, 2004, pp. 69–83.
[6] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Proc. CHES*, 2012, pp. 58–75.
[7] T. Moos, A. Moradi, T. Schneider, and F.-X. Standaert, "Glitch-resistant masking revisited: Or why proofs in the robust probing model are needed," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 2, pp. 256–292, 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7392
[8] G. Cassiers and F.-X. Standaert, "Trivially and efficiently composing masked gadgets with probe isolating non-interference," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 2542–2555, 2020.
[9] R. Sadhukhan, S. Saha, and D. Mukhopadhyay, "Shortest path to secured hardware: Domain oriented masking with high-level-synthesis," in *Proc. ACM/IEEE DAC*, 2021, pp. 223–228.
[10] S. Inagaki, M. Yang, Y. Li, K. Sakiyama, and Y. Hara-Azumi, "Examining vulnerability of HLS-designed Chaskey-12 circuits to power side-channel attacks," in *Proc. 23rd ISQED*, 2022, p. 1.
[11] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Syst. Lett.*, vol. 12, no. 4, pp. 133–136, Dec. 2020.
[12] L. Zhang, D. Mu, W. Hu, Y. Tai, B. Jeremy, and R. Kastner, "Memory-based high-level synthesis optimizations security exploration on the power side-channel," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2124–2137, Oct. 2020.
[13] S. C. Konigsmark, D. Chen, and M. D. Wong, "High-level synthesis for side-channel defense," in *Proc. IEEE 28th ASAP*, 2017, pp. 37–44.
[14] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proc. 23rd FPL*, 2013, pp. 1–4.
[15] N. Pundir, S. Aftabjahani, R. Cammarota, M. Tehranipoor, and F. Farahmandi, "Analyzing security vulnerabilities induced by high-level synthesis," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 3, pp. 1–22, 2022.
[16] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *Proc. CHES*, 2010, pp. 413–427.
[17] B. Bilgin, "Threshold implementations: As countermeasure against higher-order differential power analysis," Ph.D. dissertation, Res. Inf., Univ. Twente, Enschede, The Netherlands, May 2015.
[18] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Proc. ICICS*, 2006, pp. 529–545.
[19] E. Prouff and T. Roche, "Higher-order glitches free implementation of the AES using secure multi-party computation protocols," in *Proc. CHES*, 2011, pp. 63–78.
[20] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Higher-order threshold implementations," in *Proc. ASIACRYPT*, 2014, pp. 326–343.
[21] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. New York, NY, USA: Wiley, 2007.
[22] "AMD; Vivado HLS," 2022. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[23] S. Inagaki, M. Yang, Y. Li, K. Sakiyama, and Y. Hara-Azumi, "Power side-channel attack resistant circuit designs of ARX ciphers using high-level synthesis," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 5, pp. 1–17, 2023.
[24] A. Raghunathan, S. Dey, and N. K. Jha, "Register transfer level power optimization with emphasis on glitch analysis and reduction," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 8, pp. 1114–1131, Aug. 1999.
[25] E. Bendersky, "PyCParser C parser and AST generator written in Python," 2012. [Online]. Available: https://pypi.org/project/pycparser/
[26] A. Bogdanov et al., "PRESENT: An ultra-lightweight block cipher," in *Proc. 9th Int. Workshop CHES*, Vienna, Austria, Sep. 2007, pp. 450–466.
[27] D. Canright, "A very compact S-box for AES," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2005, pp. 441–455.
[28] G. Becker et al., "Test vector leakage assessment (TVLA) methodology in practice," in *Proc. ICMC*, 2013, p. 13.