

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

Nilotpolo Sarma[†], Anuj Singh Thakur[†], and Chandan Karfa[†]

[†]Indian Institute of Technology Guwahati

[†]{s.nilotpolo, anuj.thakur, ckarfa}@iitg.ac.in

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 RTL. Moreover, existing HLS tools can't place registers at designated places and balance parallel paths in a cryptographic design which is needed to stop glitch propagation. This paper introduces a domain-specific HLS approach tailored to obtain a PSCA secure masked hardware implementation directly from a masked software implementation. It places the registers at specific locations required by the glitch-robust masking gadgets, resulting in a secure RTL. Moreover, our tool 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 the MaskedHLS generated RTLs is also shown with TVLA test.

Index Terms—Embedded System Security, Power Side-Channel Security, High-Level Synthesis, 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. This is possible due to the existence of a direct correlation between the power consumption, which is a result of the overall transistor activity, and the computations being performed by the device under attack. Masking [2] is a countermeasure against such attacks. Masking splits the secret information processed in the algorithm into random shares drawn from a uniformly sampled random distribution. Thereafter, execution proceeds by processing these shares independently, taking care to *re-randomize* computations that cause their recombination. This randomizes the results of intermediate computations, and hence the power consumption becomes random as well. Masking can be applied at the hardware [2]–[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. Secondly, 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 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, software masking is easier to obtain from the algorithmic specification and easily verified [6]. Therefore, ways to obtain masked hardware from the corresponding masked software implementations 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 DOM [2], HPC1 and HPC2 [8], and COMAR [4] have the same structure as their software-masked counterparts with additional registers to stop the propagation of glitches. Thus, in order to generate PSCA-secure masked hardware from masked software, a simple translation of intermediate code to RTL is desired. That can be followed by inserting registers at well-defined locations for state-of-the-art masking gadgets like DOM, HPC and COMAR.

In this regard, High-Level Synthesis (HLS), which automatically generates register transfer level (RTL) hardware from descriptions in C/C++, can be helpful. A few recent works [9] aim to utilize HLS to convert masked software implementations to masked hardware automatically. *In this work, we have shown that all stages of the HLS can hamper the security of masked circuits.* They have been discussed in greater detail in Subsection 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:

- 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).
- 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).
- The correctness of MaskedHLS is shown (in Section VI).
 - 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).

The MaskedHLS tool is generic enough to work for any masking gadget. To the best of our knowledge, this is the first work that provides a complete HLS flow for PSCA secure hardware design from masked cryptographic code written in C/C++.

The rest of the paper is organised as follows. The related works are discussed in Section II. Section III discussed the background material required to understand the working of MaskedHLS. Following this, Section IV illustrates the impact of HLS on the PSCA security of masked designs and the motivation of our work. Section V discusses the flow of the MaskedHLS tool and its steps in great detail. Section VI discusses the correctness of our tool. Section VII discusses the results of using our tool on the selected benchmarks. Finally, Section VIII concludes the paper and discusses possible future directions of work.

II. RELATED WORKS

Several works focusing on HLS of cryptographic implementations have been published [10]–[12]. Works like [12], [13] looked at the effects of various HLS optimizations on the side-channel security of unmasked code. These works were enough to establish that HLS does not possess security evaluation as a part of its design flow and generating secure hardware by HLS requires a case-by-case examination of all the optimizations, which is a very difficult task. However, these works do not consider masked cryptographic implementations and the effect of HLS on masking.

In 2021, Sadhukhan et al. [9] demonstrated how to generate side-channel secure masked hardware in quick time using HLS. For their work, they used a 3-bit DOM-masked AES S-box with bit sliced implementation and generated the Verilog (RTL) for it using the *Bambu* HLS tool [14]. They observed that HLS does not lead to side-channel secure hardware always. Hence, 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 presented remedies for better application of such pragmas. Recently, a study by Pundir et al. [15], highlights the importance of considering security when using HLS for hardware design. They have rightly pointed out that no HLS tool has been found to consider side-channel leakage while performing their code transformation procedures. There does not exist any work that develops domain-specific HLS tool for PSCA secure RTL generation from masked cryptographic design.

III. BACKGROUND

A. Glitch-Resistant masking

Hardware masking of cryptographic algorithms against PSCAs proceeds by breaking the 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 an affine operation, \oplus (XOR), such as in $c = a \oplus b$, can be broken into computations $c0 = a0 \oplus b0$ and $c1 = a1 \oplus b1$. Here a and b are broken 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 $a0, b0, c0$ are 0-shares and $a1, b1, c1$ are 1-shares. Thereafter, $c0 \oplus c1$ gives the correct value of c .

Similarly, an operation like \otimes (bit-wise multiplication¹), 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, $a0 \otimes b1, a1 \otimes b0$, can not be performed without mixing the 0-shares with the 1-shares, which violates the independence of shares as assumed by masking theory. Hence, these operations need to be carefully remasked to ensure PSCA-security.

Some algorithmic tricks can be used to mask these non-linear 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, resulting in a masked hardware implementation for the multiplication operation.

However, this multiplication 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 in circuits. As demonstrated in [17], assuming that only one share $a1$ arrives later than the others, the number of times all the gates in the *SecMult* circuit change values on different values of b reveals a correlation between the power consumption and the value of b . Thus, masking in a glitchy circuit should be carefully handled. Several masking schemes were designed to be resistant to glitches in hardware [2], [18]–[20].

One of the approaches towards glitch-resistant masking of cryptographic hardware is replacing all non-linear operations with *gadgets* that are provably secure independently as well as in composition. A gadget is a probabilistic algorithm that takes n m-shares as inputs (where n is the number of inputs of the gadget) and returns a single m-shared output. A *gadget-based construction of masked circuits replaces one or more non-linear operations with secure 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 subsection, we briefly introduce those gadgets.

B. Multiplication Gadgets

Gross et al. [2] presented Domain-Oriented Masking (DOM) of hardware implementations of cryptographic algorithms against PSCAs

¹In this paper, \otimes and $\&$ has been used interchangeably to mean bit-wise multiplication. \oplus and \sim has been used interchangeably to mean bit-wise XOR.

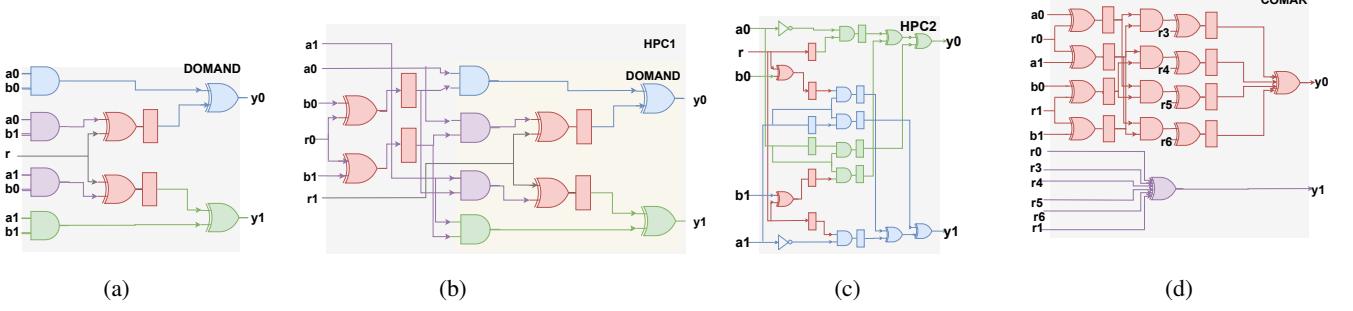


Fig. 1: Masked Multiplication Gadgets (a) DOMAND (b) HPC1 (c) HPC2 (d) COMAR.

where 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, *non-affine* 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. 1a. Here, the products containing cross-domain terms, $a0 \otimes b1$, $a1 \otimes b0$ 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 *non-affine* gadgets were introduced in [21]. The strategy was to refresh the masks of 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 $c = a \otimes b$, a 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. 1b.

In HPC2 [21], 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. 1c.

Figure 1d 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 non-linear 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 2-input COMAR-AND gadgets in a circuit can use the same six random masks.

C. Retiming Basics

Retiming [22] 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 $\nu \in V$ is a design unit and each $e_{u,\nu} \in E$ is the edge corresponding to the flow of signal from the output of design unit u to the input of design unit ν for any $u, \nu \in V$. Each edge $e_{u,\nu} \in E$ has an edge weight $w(e_{u,\nu})$ equal to the number of registers in that edge such that $w(e_{u,\nu}) \geq 0$. Each vertex $\nu \in V$ has a constant computational delay $d(\nu)$ such that $d(\nu) \geq 0$.

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 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 :

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

A retiming label, $r(\nu)$, associated with each vertex $\nu \in V$ indicates the number of registers moved from the outputs to the input of the vertex ν 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 $e_{u,\nu} \in E$ in the original circuit, G , changes to an edge weight $w_r(e_{u,\nu})$ after retiming, then:

$$w_r(e_{u,\nu}) = r(\nu) + w(e_{u,\nu}) - r(u) \quad (2)$$

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

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

$$r(\nu) - r(u) \leq w(e_{u,\nu}), \forall e_{u,\nu} \in E \quad (3)$$

- **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 ν in G . There can be multiple paths from u to ν . The minimum number of registers on any path from u to ν is $W(u, \nu)$. Let the computational delays of all n paths from u to ν having $W(u, \nu)$ registers be $d(p_1), d(p_2), \dots, d(p_n)$. Then $D(u, \nu)$ is:

$$D(u, \nu) = \max_{i=1}^n d(p_i) \quad (4)$$

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

$$r(u) - r(\nu) \leq w(u, \nu) - 1 \quad (5)$$

Thus the objective of retiming is to identify the *retiming labels* r for all vertices that satisfy the constraints in equations (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 comes with various optimizations that help in obtaining an area-latency-optimized RTL from C/C++ code. Given a gadget-based masked software code (without register annotations) of a cryptographic algorithm, HLS can convert it to an RTL design. We observe that the optimizations performed by HLS impact the PSCA security of the masked designs under consideration. Using case studies of VivadoHLS [23] and Bambu [14], we present a few instances where HLS hampers the PSCA security of the masked hardware.

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 made at 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 incorrect recombination of shares within the algorithm. The fact that Bambu HLS does this was identified in [9]. For the input code in Listing. 3 and its interpretation in Figure 3a, the XOR gates $i1$ and $i2$ are required after the cross-domain products $p2$ and $p3$ as specified in the C code. However, LLVM shifts the XOR gates to mask the products $p1$ and $p4$ instead. The absence of these XOR gates masking the outputs of $p2$ and $p3$ results in an unmasked circuit. Specifically, the computation of $y0$ in Listing. 3 is carried out as $y0 = ((a0 \otimes b1) \oplus$

$z) \oplus (a0 \otimes b0)$, ensuring that cross-domain computations are masked before recombination. The reassociation causes the computation to be carried out as $y0 = ((a0 \otimes b0) \oplus z) \oplus (a0 \otimes b1)$ instead. We also could not stop this optimization by LLVM through the Bambu tool version 0.9.6 with `#pragma HLS_INTERFACE <variable> none_registered` as done in [9].

Expression Balancing in GCC: Many times C/C++ code is written with a sequence of operations, resulting in a long chain of operations in 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 and reduces delay. This optimization might hamper the security of the masked circuit. In Listing 1 for example, we have the DOMAND software masked code which gets reassigned 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.

```
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

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 10ns for the example in Listing. 2, all the operations are scheduled in one clock by VivadoHLS as shown in Figure 2b. A single-clock operation-chained datapath will be generated for the single-cycle schedule of Figure 2b. It is pointed out in [24], that such an operation chaining in the datapath also introduces side-channel vulnerabilities in the RTL. However, when the target clock is set to 1ns, the design is scheduled

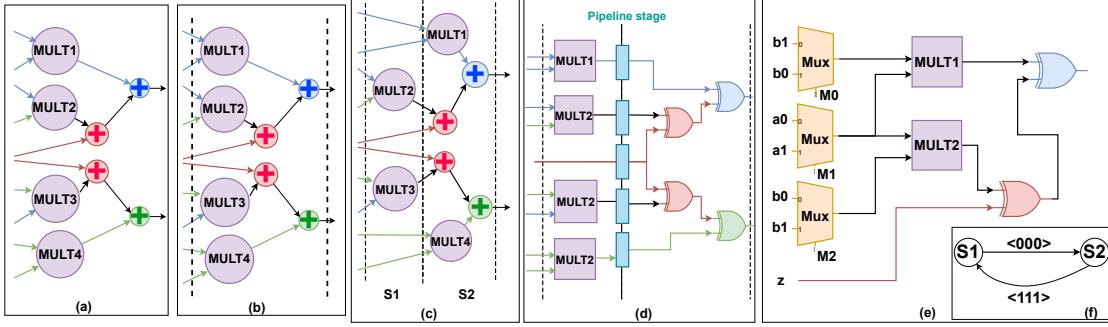


Fig. 2: Example: (a) CDFG of the behavior in Listing. 3, (b) Schedule for 10ns, (c) Schedule for 1ns , (d) Pipelined Design, (e) Resource-Shared Design (f) Controller for Resource-Shared Design

in 2 time steps as shown in Figure 2c. The actual datapath depends on the resource optimization of the HLS tool. By default, most of the HLS tools generate a pipelined design as the one in Figure 2d generated from the schedule in Figure 2c. 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. 2². The number of multiplier instances has been restricted to 2 (line number 10). This results in a circuit with a datapath as shown in Figure 2e where the resources are shared in a time-division multiplexed manner.

To control the execution of the datapath, a controller FSM as shown in Figure 2f 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 [25]. 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 Figure 3b, 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

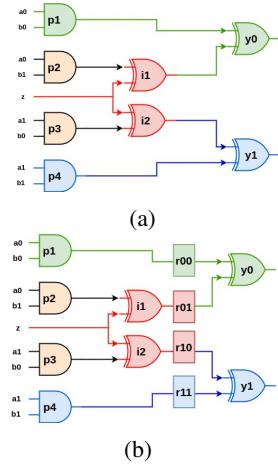


Fig. 3: (a) Software-masked DOMAND hardware realization. (b) Hardware-masked DOMAND circuit with masking and balancing registers.

security must focus on the optimal insertion of registers. In the next Subsection, we discuss how this can be optimally done.

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 HLS tool will 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 Figure 3a. 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 Figure 3b. With only these two registers, the inputs to the gates y_0 / y_1 have different latencies. This will result in incorrect circuit behavior. Thus, the *balancing registers* $r00$ and $r11$

²Listing. 1, 2 and 3 are different representations of the same DOMAND behaviour. We took three variations to illustrate the various security implications of HLS.

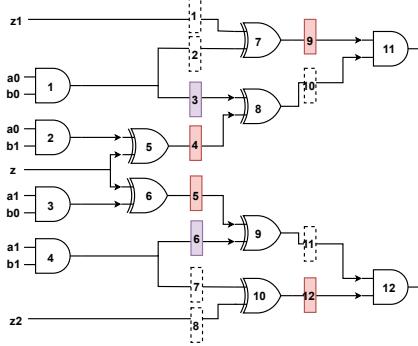


Fig. 4: An example to illustrate the need for optimal register balancing in masked circuits.

must be inserted as well. Figure 3b is the circuit corresponding to an HLS-C input annotated as in Listing. 3. For bigger 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 Figure 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, 2, 3, 6, 7 and 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. The overall latency of the circuit is now 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, 4, 5, 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.

```

1. int demand (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

Listing 4: DOMAND with register annotations

In our opinion, modern HLS tools perform too many optimizations which are counter-productive for PSCA secure hardware generation through HLS. 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 as proposed in this work.* Such a tool would not create vulnerabilities due to HLS and retain the security properties required while performing register balancing in an automated way. *In this work, we have developed an automated register balancing at behavioural level using the concept of retiming [22].* The basic concept of retiming is presented in Section III-C.

V. THE PROPOSED MASKEDHLS FLOW

A common approach towards masking cryptographic implementations is to replace the unmasked operations in the overall implementation with *masked gadgets*. The state-of-the-art masked gadgets in DOMAND, HPC1, HPC2, and COMAR as 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 masking case. These registers ensure glitch-resistant masking security. Thus, in conversion from software to hardware-masked circuits we need to annotate the software-masked code with directives for the placement of registers. In VivadoHLS we could realize this using a template class *template < classT > T reg(Tx)* and later 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 need to be placed to balance parallel paths as discussed above. This can't be done by any existing HLS tools.

Hence, we propose an HLS tool called MaskedHLS that makes this preservation of side-channel security the primary focus.

The input of 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 indicated by the annotations. In the next step, the register balancing procedure in MaskedHLS identifies all locations in parallel paths where registers need to be added. This produces an annotated C code. On this annotated C code, MaskedHLS performs a one-to-one translation into RTL code with registers inserted in all the places required by masking as well as balancing. Finally, MaskedHLS will create a pipelined RTL design. The overall flow of MaskedHLS is shown in Figure 6. The steps are discussed in detail below.

A. Register Balancing at Behavioural Level

Given an unmasked software implementation in C/C++, the masked software is obtained by replacing the non-linear 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 *<lhs of operation> = reg(<rhs of operation>)*. We need to put a register in

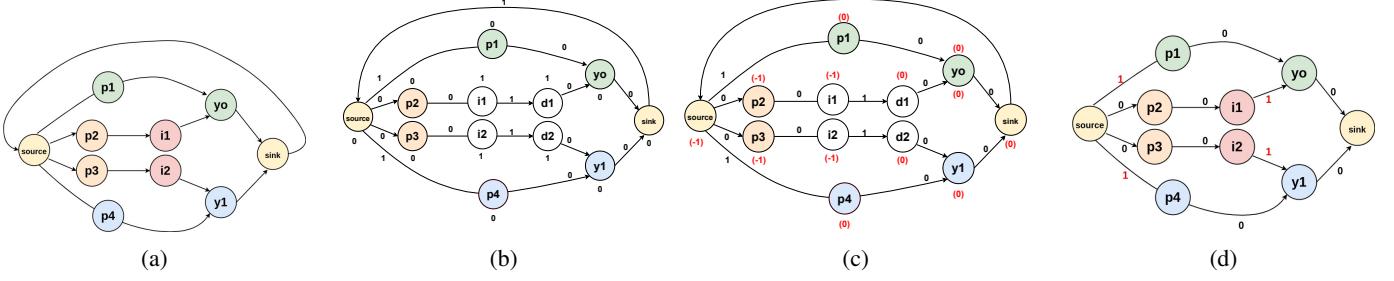


Fig. 5: (a) AST before retiming, (b) HLS-model with back edge, (c) HLS-model after retiming (retiming labels shown in parenthesis), (d) Final circuit after removing dummy nodes and back edge

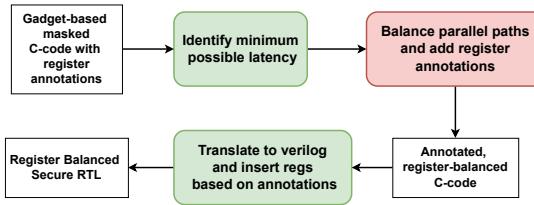


Fig. 6: Flow of MaskedHLS tool

those locations and balance parallel paths automatically, with minimum pipelined stages. For this, 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.

B. Creating the HLS Model from the AST

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*:

- *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.
- *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*:

- *Adding dummy nodes:* After each node $\nu \in V$, which has an annotation for a register insertion succeeding it, a *dummy-node* ν' is inserted.
- *Assigning computational delay to nodes:* The nodes ν , after which registers must be added, and the dummy nodes ν' are assigned computational delay of $d(\nu) = c$ and $d(\nu') = c$ 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 $e_{\nu,\nu'}$, will have a delay of $2c$. Therefore, such a path will fail to meet the target

TABLE I: Feasibility constraints for the Circuit in Figure 5b

$$\begin{array}{ll}
 r(i2) - r(p3) \leq 0 & r(i1) - r(p2) \leq 0 \\
 r(y1) - r(p4) \leq 0 & r(y0) - r(p1) \leq 0 \\
 r(p4) - r(source) \leq 0 & r(p3) - r(source) \leq 0 \\
 r(p2) - r(source) \leq 0 & r(p1) - r(source) \leq 0 \\
 r(sink) - r(y1) \leq 0 & r(sink) - r(y0) \leq 0 \\
 r(source) - r(sink) \leq 1 & r(d2) - r(i2) \leq 0 \\
 r(d1) - r(i1) \leq 0 &
 \end{array}$$

TABLE II: Critical path constraints for Figure 5b

$$\begin{array}{ll}
 r(p4) - r(d2) \leq 0 & r(p4) - r(d1) \leq 0 \\
 r(p3) - r(p4) \leq 1 & r(p3) - r(p2) \leq 1 \\
 r(p3) - r(p1) \leq 1 & r(p3) - r(i1) \leq 1 \\
 r(p3) - r(y1) \leq -1 & r(p3) - r(y0) \leq 1 \\
 r(p3) - r(source) \leq 1 & r(p3) - r(sink) \leq -1 \\
 r(p3) - r(d2) \leq -1 & r(p3) - r(d1) \leq 1 \\
 r(p2) - r(p4) \leq 1 &
 \end{array}$$

clock period of c . Hence, a register must be inserted in that path at the location between ν and ν' to meet the Critical Path Constraint (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 Figure 5a. One dummy node is inserted following each white-colored node (cross-domain nodes) and colored white as in Figure 5b. Assuming the target clock period is 1, all white-colored nodes are assigned the computational delay $d(\nu) = 1$. For all other nodes, the computational delay $d(u) = 0$.

C. Finding the maximum number of register annotations in a path

Among all paths in the *HLS model* between the source node and the sink node, the maximum number of annotations for register insertion is identified using a Depth First Search (DFS). We call it the *maximum extra regs* in a path. Once the *HLS model* is converted to RTL by our HLS tool, we need to add these many registers in all parallel paths between source and sink. Therefore, this *maximum extra regs* will determine the latency of the generated RTL. We will add those extra registers as weight in the back edge between the source and the sink. For all other edges, the edge weight is assigned to zero. It may be noted that the *HLS model* is obtained from the software code, which initially had no register.

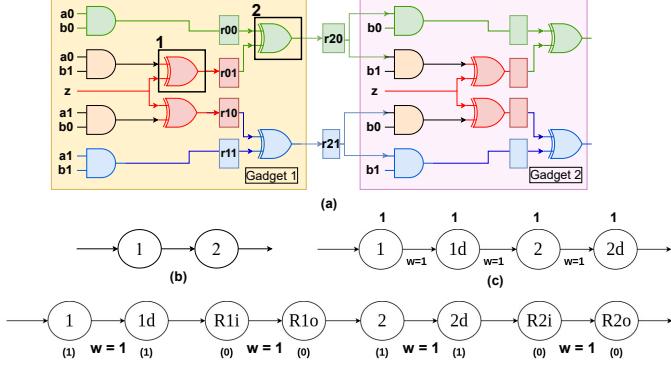


Fig. 7: (a) The DOMAND-composed circuit. (b) Part of a circuit with two register insertions in series. (c) Register insertion in series. (d) Register insertion in series using an extra register.

D. Calculation of retiming constraints

For each node, we consider the retiming label $r(\nu)$. Feasibility constraints for each edge $e_{u,\nu}$ are calculated. For the HLS model in Fig. 5b, the Feasibility Constraints are shown in Table I. Critical Path constraints for each path from u to ν such that $D(u, \nu) > 1$ are computed. For the HLS model in Fig. 5b, some of the critical path constraints are shown in Table II.

E. Inserting Registers in Series

In a gadget based masking a situation may thus arise where in a single path there are two locations where registers need to be inserted. Consider the two DOMAND gadgets composed with each other in Figure 7a. To make the Gadget1-Gadget2 combination *composable*, the registers r_{20} and r_{21} must be inserted as shown in Figure 7b. The target clock is $c = 1$. If we want to insert registers after gate 1 and gate 2 then we have to specify retiming constraints on both of them, thus, inserting a dummy node after each of them, as shown in Figure 7c. Now the $d(\nu)$ values for the nodes become $d(1) = 1, d(1d) = 1, d(2) = 1, d(2d) = 1$. The $D(u, \nu)$ values are now: $D(1, 1d) = 2, D(1d, 2) = 2, D(2, 2d) = 2$. Thus, these 3 edges $e_{1,1d}, e_{1d,2}$ and $e_{2,2d}$ violate the *Critical Path Constraints*. Hence 3 registers are placed into the circuit at locations where $w = 1$ in Figure 7c. 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 Subsection V-B is updated by the addition of these steps to address this issue:

- 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.
- 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 R_{1i} and R_{1o} are inserted. Similarly for node $2d$.
- *Register-lock* constraints are added for each R : $r(RIn) == r(ROut)$. This constraint ensures that the

number of registers moved into the edge $e_{RIn, ROOut}$ 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 Figure 7d and the subsequent steps can be performed on it.

F. 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* $r(\nu)$, a node ν' 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 node v' to node u' of weight k is drawn. It is possible that $k < 0$ for some set of retiming labels as shown in the above example.
- 4) For each node $v' \in V'$, an edge $N + 1 \rightarrow v'$ from the node $N + 1$ to the node v' with weight 0 is drawn. At this point, the graph is guaranteed to not contain any negative edge cycle as shown in Lemma VI.2.

Using Lemma VI.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 VI.2), we can apply Dijkstra's single-source shortest path algorithm to obtain the *retiming labels* $r(\nu)$. The retiming labels obtained as a solution for the HLS model in Fig. 5b are shown (within parenthesis) for each vertex in Fig. 5c. 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 $r(\nu)$ values is shown in Fig. 5d. 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.

G. Generating Pipelined RTL Design

The last phase of maskedHLS takes this register-annotated C-code obtained in the previous step and generates RTL from it. The translation of this C-code to Verilog is done via a one-to-one mapping from the AST of the C to the RTL. In addition, the tool places registers according to the annotations in the C-code. Our tool does not apply any optimization in this phase. This effectively creates a pipelined RTL design with the number of pipeline stages equal to the maximum number of registers that have been added in a path (as identified in Sub-Section V-C).

VI. CORRECTNESS OF MASKEDHLS

Here, we prove the correctness of our register balancing approach in MaskedHLS. We also show that MaskedHLS adds the minimum number of pipelined stages.

Lemma VI.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 from $N + 1$ 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, $w(N + 1 \xrightarrow{-m} u' \xrightarrow{l} v') \geq w(N + 1 \xrightarrow{0} v')$. $\Rightarrow -m + l \geq 0$. Putting the value of $r(u') = -m$ in this equation we get: $l + r(u') \geq 0 \Rightarrow 0 - r(u') \leq l \Rightarrow r(v') - r(u') \leq l$ 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 $r(v') - r(u') \Rightarrow -m + l - (-m) \Rightarrow l$ which is $\leq l$. Hence, the constraint for this case is satisfied. \square

To find the correct set of retiming labels, we have to find a solution to the constraint graph using the shortest path. For the shortest path to give a solution which is the correct retiming labels, the graph should contain no negative weight cycles as otherwise no solution can be reached using the shortest-path approach.

Lemma VI.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 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 critical path constraints 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 $e_{sink,src}$. 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 $e_{sink,src}$ 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. \square

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-C, we identify the maximum number of registers are needed in a path and add that as weight in the edge between the source and sink. These registers are sufficient to satisfy all constraints. Below in Lemma VI.3 we prove the termination of our procedure.

Lemma VI.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 $src \rightarrow v_i \rightarrow v_{i+1} \rightarrow \dots \rightarrow v_N \rightarrow sink$, 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:

$$\begin{aligned} w_r(e_{src,v_i}) &= r(v_i) - r(src) + w(e_{src,v_i}) \\ w_r(e_{v_i,v_{i+1}}) &= r(v_{i+1}) - r(v_i) + w(e_{v_i,v_{i+1}}) \\ &\dots \\ w_r(e_{v_N,sink}) &= r(sink) - r(v_N) + w(e_{v_N,sink}) \end{aligned}$$

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

$$\begin{aligned} w_r(e_{src,v_i}) + w_r(e_{v_i,v_{i+1}}) + \dots + w_r(e_{v_N,sink}) &= w(p) = \\ r(sink) - r(src) \\ \Rightarrow r(sink) - r(src) &= m + k \end{aligned}$$

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

Lemma VI.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

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

requiring balancing. Introducing registers at locations other than the locations annotated (balancing registers) does not compromise the security. Since we lock all existing registers using register locking constraints, the 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. \square

A. Complexity Analysis

Our MaskedHLS tool's complexity 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 feasibility constraints 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 critical path constraints for each edge $e_{u,v}$ where $D(u,v) > c$ which is at most $O(n^2)$. These constraints are then modelled using a constraint graph which is linear in the number of constraints which is $O(n^2)$. These constraints are solved again using 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 (n). Therefore the time complexity of the balancing procedure is $O(n^3)$ in the number of nodes in the retiming model n .

VII. EXPERIMENTAL RESULTS

A. Implementation and Benchmark Details

The MaskedHLS tool makes use of PyCparser [26] to parse the abstract syntax tree of the input C-code on which the balancing procedure and the one-to-one transformation to RTL are performed. We have tested our MaskedHLS tool on four different variants of the PRESENT Cipher's 4-bit S-box [27] and Canright's AES-256 S-box [28] masked using four different gadgets: the DOMAND gadget, the HPC1 gadget, the HPC2 gadget, and the COMAR gadget. The codebase, examples and scripts of MaskedHLS and the flow are available in github³.

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 seconds on the AES S-box designs with an average of 1795 nodes; and an average of 0.48 Seconds on

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_{reg}s corresponds to the gadget based masked circuit without registers, w_{reg}s 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.

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 3.20GHz is less than 1 sec for all PRESENT S-boxes and less than two minutes 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 180nm 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 2.97x and 6.85x 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 5.9x, 9.4x, 11.5x, 9.5x 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 33.1x, 56.3x, 41.3x, 48.9x respectively. This increase in area is because of the additional registers added by HLS and the

³Will be made public upon acceptance

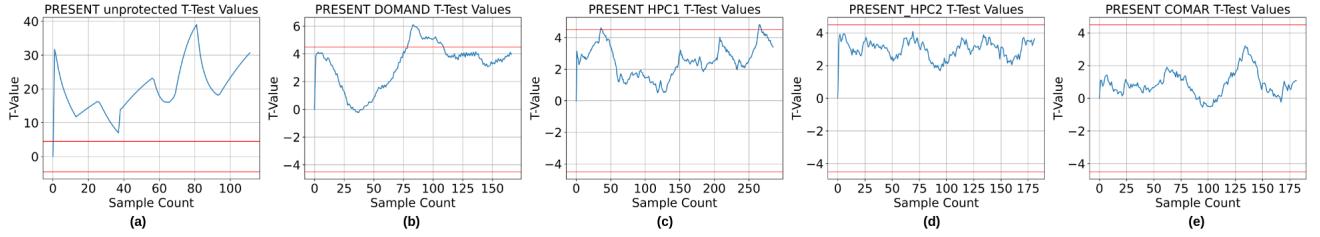


Fig. 8: T-values for: (a) PRESENT_unprotected. (b) PRESENT_DOMAND. (c) PRESENT_HPC1. (d) PRESENT_HPC2. (e) PRESENT_COMAR. (For each design, the x-axis contains T-values, y-axis contains the number of sample points per plaintext.)

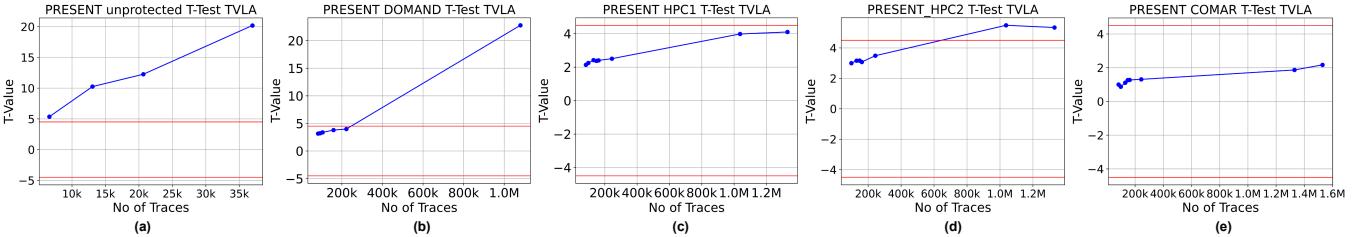


Fig. 9: TVLA values versus the number of traces: (a) PRESENT_unprotected. (b) PRESENT_DOMAND. (c) PRESENT_HPC1. (d) PRESENT_HPC2. (e) PRESENT_COMAR. (For each design, x axis contains T-values, y-axis contains the number of traces for which that TVLA value was observed.)

technology mapping for a pipelined design does not allow for much area optimization versus the combinatorial circuits of the designs without registers that get largely optimized. The clock period (in ns) for designs generated by MaskedHLS is less due to the pipeline stages added through registers.

TABLE V: Comparison of Register and Latency savings using MaskedHLS and Manual Methods

Design	Registers MaskedHLS	Registers Manually	Saving(%)	Latency MaskedHLS	Latency 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 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) [29] 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 Wave View tool was used to extract power traces in CSV format from the FSDB file. On this data, we used the conventional TVLA method [29] to obtain the t-values. The t-value corresponding to one plaintext for all PRESENT designs is shown in Figure 8. Clearly, the unprotected design is leaking. Among the PRESENT S-box designs, PRESENT_HPC1, PRESENT_COMAR are more secure compared to PRESENT_DOMAND and PRESENT_HPC2 whose t-value exceeded the Threshold of $\|4.5\|$ in 18% and 16% cases, respectively.

We extracted power traces, the number of which ranged between 5 thousand and 100 million. The objective was to check how good the protection was. The higher the number of traces where the t-value is not crossing the threshold of $\|4.5\|$, the more secure the design is. Figures 9 shows the trend of TVLA-values for this experiment for the PRESENT designs. The unprotected design crosses the $\|4.5\|$ mark for around 6 thousand traces. Whereas the COMAR and HPC1-masked designs are secure to 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 Figure 1c.

The design with COMAR is the most secure among all gadgets available. The experimental results are aligned with

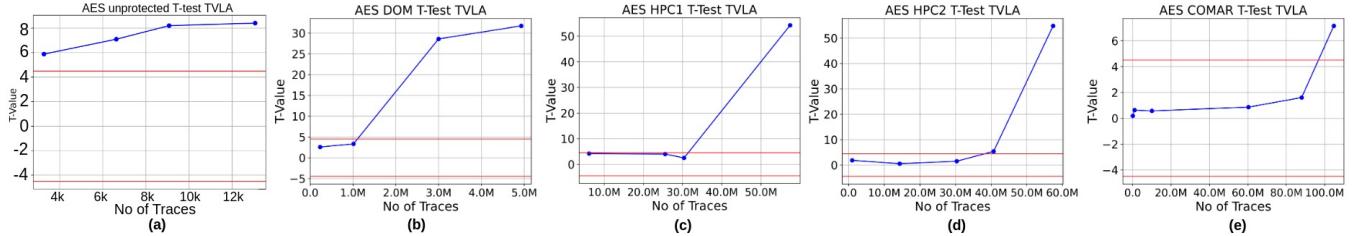


Fig. 10: TVLA values versus the number of traces: (a) AES_unmasked. (b) AES_DOMAND. (c) AES_HPC1. (d) AES_HPC2. (e) AES_COMAR. (For each design, the x-axis contains T-values, y-axis contains the number of sample points per plaintext.)

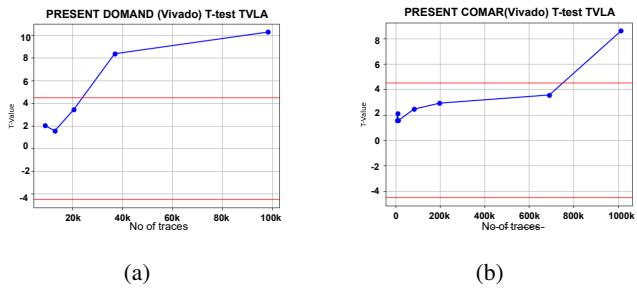


Fig. 11: TVLA values versus number of traces for: (a) PRESENT_DOMAND (b) PRESENT_COMAR, both synthesized using Vivado HLS

the Theoretical analysis of the gadgets.

We also performed TVLA test on the output of Vivado HLS [23] on the DOMAND and COMAR masking gadget protected - PRESENT S-box. It can be seen in the results in Figure 11a and 11b 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 ≥ 1.3 million traces respectively). We observed similar results for PRESENT S-box using other gadgets (HPC1, HPC2). However, due to space limitations, we could not add all results. This reaffirms our motivation of domain specific MaskedHLS tool for PSCA-secure designs. Also, to test the efficacy of our tool on bigger benchmarks, we have tested our MaskedHLS tool with a Canright's AES S-box [28] masked using the DOM, HPC1, HPC2 and COMAR gadgets. The TVLA results show that the key can 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 Figure 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 non-trivial task that requires significant time and expertise. Therefore obtaining masked hardware from masked software using HLS is beneficial. We have shown that existing HLS actually does not guarantee the PSCA security of the generated RTL. To address

this shortcoming, we have developed MaskedHLS for generating 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 latencies as compared to 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. Our MaskedHLS generates RTL that uses minimum latency and registers to achieve PSCA security. In future, we plan to integrate randomness optimization strategies in our MaskedHLS tool.

REFERENCES

- [1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *AICC*. Springer, 1999, pp. 388–397.
- [2] H. Groß, S. Mangard, and T. Korak, "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order," *Cryptology ePrint Archive*, 2016.
- [3] G. Cassiers, B. Grégoire, I. Levi, and F.-X. Standaert, "Hardware private circuits: From trivial composition to full verification," *IEEE TC*, vol. 70, no. 10, pp. 1677–1690, 2020.
- [4] D. Knichel and A. Moradi, "Composable gadgets with reused fresh masks – first-order probing-secure hardware circuits with only 6 fresh masks," *Cryptology ePrint Archive*, 2023.
- [5] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of aes," in *SAC*. Springer, 2004, pp. 69–83.
- [6] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *CHES*. Springer, 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 Transactions on CHES*, pp. 256–292, 2019.
- [8] G. Cassiers and F.-X. Standaert, "Trivially and efficiently composing masked gadgets with probe isolating non-interference," *IEEE TIFS*, 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 *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 *2022 23rd ISQED*, 2022, pp. 1–1.
- [11] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, vol. 12, no. 4, pp. 133–136, 2020.
- [12] L. Zhang, D. Mu, W. Hu, Y. Tai, J. Blackstone, and R. Kastner, "Memory-based high-level synthesis optimizations security exploration on the power side-channel," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2124–2137, 2019.
- [13] S. C. Konigsmark, D. Chen, and M. D. Wong, "High-level synthesis for side-channel defense," in *IEEE 28th ASAP*. IEEE, 2017, pp. 37–44.
- [14] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 FPL*, pp. 1–4.

- [15] N. Pundir, S. Aftabjahani, R. Cammarota, M. Tehranipoor, and F. Farahmandi, “Analyzing security vulnerabilities induced by high-level synthesis,” *ACM JETC*, vol. 18, no. 3, pp. 1–22, 2022.
- [16] M. Rivain and E. Prouff, “Provably secure higher-order masking of aes,” in *CHES*. Springer, 2010, pp. 413–427.
- [17] B. Bilgin, “Threshold implementations: as countermeasure against higher-order differential power analysis,” 2015.
- [18] N. Svetla *et al.*, “Threshold implementations against side-channel attacks and glitches,” in *ICICS*. Springer, 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 *CHES*. Springer, 2011, pp. 63–78.
- [20] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, “Higher-order threshold implementations,” in *ASIACRYPT*. Springer, 2014, pp. 326–343.
- [21] G. Cassiers, B. Grégoire, I. Levi, and F.-X. Standaert, “Hardware private circuits: From trivial composition to full verification,” *IEEE TC*, vol. 70, no. 10, pp. 1677–1690, 2020.
- [22] K. K. Parhi, *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.
- [23] AMD, “Vivado HLS,” <https://www.xilinx.com/products/design-tools/vivado.html>, 2022.
- [24] 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 TECS*, vol. 22, no. 5, pp. 1–17, 2023.
- [25] A. Raghunathan, S. Dey, and N. K. Jha, “Register transfer level power optimization with emphasis on glitch analysis and reduction,” *IEEE TCAD*, vol. 18, no. 8, pp. 1114–1131, 1999.
- [26] E. Bendersky, “Pycparser c parser and ast generator written in python,” 2012.
- [27] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, “Present: An ultra-lightweight block cipher,” in *CHES 2007*. Springer, 2007, pp. 450–466.
- [28] D. Canright, “A very compact s-box for aes,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2005, pp. 441–455.
- [29] B. G. Tobias *et al.*, “Test vector leakage assessment (tvla) methodology in practice,” in *ICMC*, vol. 1001. sn, 2013, p. 13.