Towards a Timing Attack Aware High-level Synthesis of Integrated Circuits

Steffen Peter and Tony Givargis
Center for Embedded Systems and Cyber-Physical Systems (CECS)
University of California, Irvine
Email: st.peter@uci.edu

***Abstract*—Variabilities in the execution time of integrated circuits are frequently exploited as a side channel attack to expose secret information of deployed systems. Standard countermeasures analyze and change the explicit timing behavior in lower level hardware description languages, but their application is time consuming and error-prone. In this paper we investigate the integration of timing attack resilience into the high-level synthesis (HLS). HLS translates programs expressed in higher level programming languages, such as C, seamlessly to synthesizable hardware. We use timing annotations of basic blocks in C to add scheduling constraints that in the synthesis process balance the execution time of security-related execution branches. We integrate our approach to the scheduling of the open source LegUp HLS tool and apply the proposed method for the asymmetric cryptography algorithms RSA and ECC. The results proof the resistance against timing attacks, with a negligible overhead in synthesis efforts, area, and run-time.**

I. Introduction

Timing attacks are side-channel attacks that extract secret data from a deployed system by exploiting the fact that the processing time of the system varies for different data inputs [1]. Even though timing attacks are well-known, still recent system implementations are vulnerable, even for crypto implementations such as RSA [2] and Elliptic Curve Cryptography (ECC) [3]. Timing attacks might extract secret keys but can also threaten intellectual property and implementation details of integrated circuits, since the attacks may disclose algorithm details and key components. While timing attacks are known for both software and hardware, in this paper we focus on vulnerabilities of integrated circuits. The aim of our work is to guarantee that variabilities of the data input do not result in a changed timing, specifically that the number of clock cycles does not leak sensitive information.

A variety of countermeasures against timing attacks exist, both at higher level algorithm design and low-level implementation. Examples for low level approaches include path and timing obfuscation methods [4] or manual balancing of the HDL code. The application of such approaches is error prone and complicated since it contains many manual design steps. On a higher level, timing invariance has been addressed at the algorithm design, so that operations in different execution paths are balanced. Examples are the Lopez-Dahab ECC algorithm [5], or the aimed addition of dummy operations [6]. The problem is that implementations of balanced algorithms still might leak timing information on lower levels, because synthesis tools attempt to optimize register transfers and usage patterns of functional units.

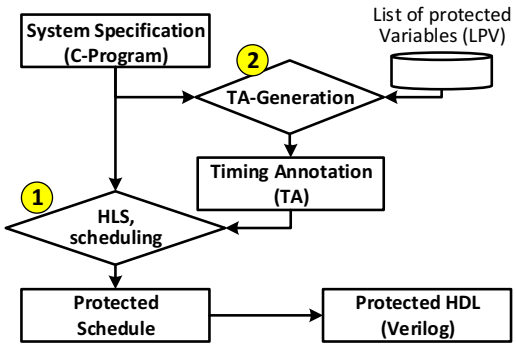

Fig. 1. Design flow for secure HLS: A protected HDL implementation is generated from a system specification, using timing annotations (TAs) to balance the execution time of sensitive paths. The paper addresses (1) the addition of scheduling constraints, and (2) TA-generation.

In this paper we study how timing attacks can be addressed in state-of-the-art high-level synthesis (HLS) tool chains. HLS translates a higher level description (e.g., in C or System C) of a system to implementable hardware, solving allocation of processing elements, binding and scheduling. HLS has found applications in a range of industrial and academic tools [7], [8]. These HLS tools improve the design practice of ICs, but they emphasize performance over security. Timing is only supported to determine the upper bound or the worst case execution time.

We propose the design flow shown in Figure 1. The primary input of the HLS flow is the system specification as C code, while the output is the secured implementation model in a hardware description language (HDL). Using a list of protected variables (LPV), a Timing Annotation (TA) for the C-code is generated. The TA provides constraints for the HLS and scheduling to balance the execution paths of the system. The result of the scheduling is an operation schedule in which all data and control dependencies originating from the LPV are time invariant.

We implemented the approach in the open source HLS tool LegUp [8], using the LLVM compiler back-end and the SDC scheduler [9]. We tested the approach for a range of benchmark applications, including implementation for the cryptographic standards ECC and RSA. The strength of our approach is that the techniques work fully automated, without notable overhead in synthesis, area or worst-case performance.

978-1-5090-5142-7/16/$31.00 ©2016 IEEE

II. Preliminaries

As shown in Fig. 1, the HLS process synthesizes an implementation model in a hardware description language (HDL) from a system description in a high-level programming language. Specifically in this paper, we generate a Verilog program from a C program. To synthesize a system, the HLS needs to allocate the functional units (FUs), bind operations to FUs and schedule when the operations are executed.

The scheduler decisions are based on the control and data flow graph (CDFG). The CDFG contains basic blocks (BB), operations (Op), data dependencies ($Op \times Op$) and control flows ($BB \times BB$). A basic block contains a sequence of operations with data dependencies, but without loops and branches. Loops and branches are indicated as possible control paths between BB. The CDFG does not contain scheduling information, neither for the operations in a BB nor for the sequence in which BBs are executed. Figure 2 (A) shows a CDFG with four BBs and one condition at the end of BB1.

The scheduler assigns the operations to states. The state diagram can be expressed as a a finite state machine with data (FSMD), as shown in Fig. 2 (B). A state can execute more than one operation (S2 and S6), if data dependencies are resolved and no resource conflicts are present. An operation requires more than one state when it cannot be executed in one cycle. In Fig. 2, BB2.i2 needs three cycles (S6 to S8).

While the FSMD assigns operations to states, it does not present a global schedule, but several execution paths exist. In the example of Figure 2, the program requires 6 cycles if $cond = 1$, and 7 cycles if $cond = 0$. The aim of this paper is to balance the two possible execution paths, if, and only if $cond$ is based on data that needs to be protected. In that case, idle states will be inserted into the FSMD with the aim to balance the execution time of BBs. In the example of Figure 2 we could add one state between S4 and S5 to balance the execution time for $cond = 0$ and $cond = 1$. We will address this issue in the scheduler of the HLS.

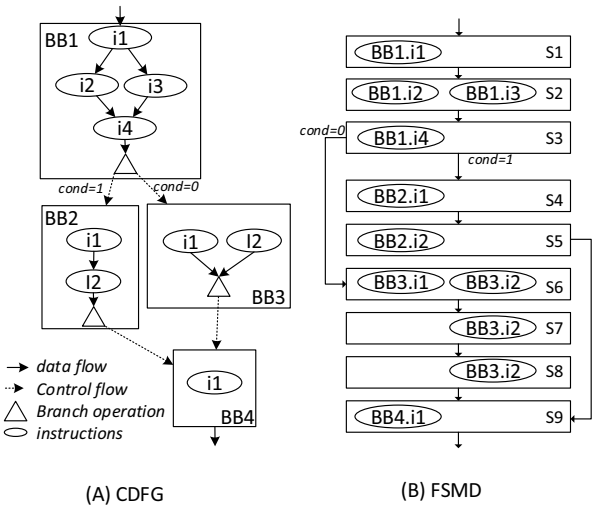

Fig. 2. An example for a (A) CDFG and a possible FSMD (B).

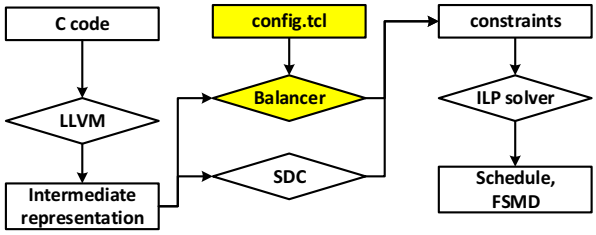

Fig. 3. Extract of the LegUp HLS scheduler tool chain. We added the *Balancer* module and annotations into the *config.tcl*

SDC scheduling: The underlying scheduler of our work is the SDC scheduler [9]. SDC expresses the scheduling constraints as a set of integer difference constraints. The set of constraints is solved by an integer linear programming (ILP) solver. The general format of the constraints is $\sum_i sv_i \leq b$, for scheduling variables sv_i, and the constraint b.

The most important variables we consider in this paper are the start ($sv_{beg}(i)$) and the end ($sv_{end}(i)$) states of operations $i \in Op$. The difference constraints allow to express control and data dependencies, such as

$$sv_{beg}(BB1.i2) - sv_{end}(BB1.i1) \geq 1,$$

latencies, such as

$$sv_{end}(BB3.i2) - sv_{beg}(BB3.i2) \geq 3,$$

as well as resource and timing constraints. The typical optimization criterion of the SDC scheduler is to minimize the total execution time, i.e. minimize the sum of the timings of the scheduled operations: $\min \sum_{v \in V_{op}} sv_{beg}(v)$. The result is the FSMD in which the scheduling constraints are satisfied and all operations are assigned to states.

III. Balancing Execution Paths

In this section we add new scheduling constraints and variables to balance the execution time of different paths in the FSMD. We extend the tool chain of the LegUp open source HLS tool and its SDC scheduler. As highlighted in Fig. 3, which outlines the LegUp tool chain, we add a Path *Balancer* module parallel to the SDC module. Like the SDC module, the *Balancer* uses the intermediate code representation of the C program, generated by LLVM, and adds scheduling constraints that are solved by an ILP solver to generate a preferable schedule. In this section we discuss the *Balancer* and the required annotations in the C-code and the external configuration file `config.tcl` that helps to parametrize the synthesis process.

A. Scheduling Variables for BBs

The SDC scheduler performs the scheduling for each BB separately, so that timings for operations within a BB are fixed. However, BBs might be executed earlier or later depending on the run-time branch decisions. To balance executions paths in this environment, it is our aim to instruct the scheduler to add idle states into shorter BBs, so that all possible execution paths that follow a security-related branch require an equal amount of clock cycles. To balance the execution time of

BBs, variables for the relative start and end time of BBs are required. Therefore we add new scheduling variables:

- $sv_{beg}(BB_i)$ is the relative start time of basic block i, which is bound to the earliest start state $sv_{beg}(op)$ of any operation $op \in BB_i$
- $sv_{end}(BB_i)$ is the relative end time of basic block i, which is bound to the latest end state $sv_{end}(op)$ of any operation $op \in BB_i$
- $latency(BB_i)$ is the total execution time of basic block i, which is difference between end and start time of BB_i: $latency(BB_i) = sv_{end}(BB_i) - sv_{beg}(BB_i)$.

The three new variables facilitate a range of options to constrain the execution time. For instance, by constraining $latency(BB_i)$ to a certain value, we enforce the relative difference of start and end state of the BB, which in turn constrains the assigned state for the first and the last operation. Therefore, the scheduler has to insert idle states, if $latency$ exceeds the number of actual operations.

B. Static BB Latency Constraints

Assumed the identifiers of the BBs and the required states of each BB are known, the latency of the BBs can simply be constrained by defining a fixed $latency$ variable. Practically that can be achieved by adding an annotation in the `config.tcl` configuration file. `config.tcl` is evaluated in the ***Balancer***, so that the parameter

```
set_parameter Cycles_BB_i n
```

defines $latency(BB_i) = n$, i.e., the latency of BB_i is constrained to n.

One advantage of the direct BB annotations is that it does not require changes of the C code. However, the identification of BB-id in practice is non-trivial as it requires insight into enumerations of the intermediate LLVM representation.

C. Automatic Path Balancing

Now we discuss approaches that do not need that detailed low-level knowledge. Instead we balance execution paths that follow a branch automatically. First, we consider paths consisting a single BB only, then we describe an algorithm to identify and annotate paths with nested sub-branches.

1) Balancing single BB-Paths: Execution time variability are generally caused by conditional branches such as

```
if cond then BBx else BBy,
```

so that either BB_x or BB_y are executed, depending on condition `cond`. Considering that binary structure, we can constrain the latency of the two BBs with the scheduling constraint $latency(BB_x) - latency(BB_y) = 0$.
The constraint enforces the faster of the two BBs to extend its latency and add idle cycles, because the BB with the higher latency cannot reduce its latency without violating other scheduling constraints. In case of the example in Fig. 2, we would enforce that the latency of the two BBs following the conditional statement in BB1 are equivalent, that is, $latency(BB_2) - latency(BB_3) = 0$. Since BB3 requires 3 cycles and BB2 needs 2 cycles, this constraint would force BB2 to add one idle state.

The BBs of the two paths do not need to be addressed directly, when we label and identify the conditional statement in the C code, and balance the following two BBs.

2) Balancing Paths with Sub-Branches: For systems containing nested branches in the paths that need to be secured, a manual annotation is not trivial, since we need to ensure the equivalent latencies of all possible execution paths. Therefore, starting from the annotated security-related branch statement, we need to

1) identify the earliest point of reconvergence,
2) identify all adjacent BBs between the secure branch and the point of reconvergence, and
3) tie the schedule for all pairs of adjacent BBs: i.e. $sv_{beg}(BB_{drain}) - sv_{end}(BB_{source}) = 1$.

The requirements can be implemented, applying the BB graph structure from the LLVM intermediate representation. We achieve the timing invariance by extending the schedules of each separate BB, so that all feasible execution paths result in the same number states.

IV. Evaluation

We implemented the presented techniques, i.e. the support for the timing annotation, the balanced scheduling, and the verification for the LegUp HLS tool chain and applied it to two generic benchmark applications and two cryptographic systems. We compare systems that were synthesized with and without the path balancing. The evaluated systems are:

- the *Max* that computes the maximum of two 32 bit numbers,
- the SRA algorithm to compute an approximation for $\sqrt{a+b}$, while the assumed secret is b,
- the 2048-bit modular exponentiation ($x^e \ mod \ p$) of the RSA crypto algorithm, with the secret exponent e, and
- the 233-bit (kP) point multiplication of the ECC cryptographic algorithm [5], with the secret factor k.

For ECC and RSA we used pre-synthesized units for the algebraic operations in the finite field, which means that the complex multipliers were not synthesized in the HLS process.

Each system was synthesized with three settings:

1) No annotations (NOA), i.e., the original design,
2) Static balancing (STA), as discussed in Sec. III-B, and
3) Automatic path balancing (APB), discussed in Sec. III-C.

The run time was measured empirically by applying various random test inputs, including data inputs with all bits set to 0 and all bits set to 1. We provide the data for the technology-dependent properties as percentage in relation to the original design. The systems were synthesized for an ALTERA Cyclone V FPGA, simulated with Modelsim 15. The area is based on the reported FPGA utilization, the power consumption is estimated using the Quartus PowerPlay tool. The synthesis time includes the required time for the HLS but does not include time for hardware synthesis and mapping. The experiments were conducted on a i7 PC with 16GB RAM.

In Table I we see that the execution time varies for all of the original designs. That means that all tested original designs contain timing side-channels that might be exploitable.

TABLE I
SYNTHESIS, EXECUTION TIME, AND VERIFICATION RESULTS.

Design case		synth time [sec]	execution time [cycles]	area [%]	longest path to original design	power
Max	NOA	4	10-11	-	-	-
	STA	4	11	=0.0	=0.0	-0.5
	APB	4	11	=0.0	=0.0	-0.5
SRA	NOA	4	23-26	-	-	-
	STA	4	26	=0.0	=0.0	-0.2
	APB	4	26	=0.0	=0.0	-0.2
ECC	NOA	6	12952-13416	-	-	-
	STA	6	13414	+0.0	=0.0	-0.0
	APB	7	13414	+0.0	=0.0	-0.0
RSA	NOA	8	0.1-1.6mio	-	-	-
	STA	9	1,598,356	+0.2	=0.0	-5.2
	APB	9	1,594,260	+0.2	=0.0	-5.2

The execution time for the annotated designs is invariant for all tested designs. For the RSA case the execution time for the static annotation is slightly larger than the one for the automatic balancing. This is caused by the fact, that without complete knowledge of the basic blocks and their operations, a manual annotation leads to wrong, unsatisfiable, or non-optimal schedules, which is a known limitation of manual annotation approaches and motivated our work towards the automatic balancing. It should be noted that with manual fine tuning the static approach could result in the same figures as the automated annotations.

Overall we see that the average execution time of the balanced designs is clearly higher than the original designs, while the worst case execution times are similar to the original designs. That observation is not surprising since the fundamental idea of our work is the addition of idle states to balance execution paths. Therefore the additional run-time is a trade-off for security, that is already know for low-level countermeasures such as [6].

Performance and Overhead: In general the measured overheads of synthesis time, area, longest path, and power consumption are negligible. The synthesis time increases by up to 10%. The increase is expected and is caused by processing time in the added *Balancer* module, as well as for additional constraint processing in the ILP solver. We expect that at tighter integration of the *Balancer* and the SDC scheduler could reduce the overhead further.

The longest path, is not affected by the path balancing for any of the tested designs. This result was expected because the longest combinatorial path typically is part of the data path which should not be affected by the added complexity of the control path.

The added states in the control path cause the small increase of area for the synthesized design. Overall the area is only affected marginally, which also means that, at least for our test cases, the idle-cycle-imposed reduction of resource utilization did not lead to any reduction of functional units or registers.

The power consumption is affected marginally, showing a small reduction of average power consumption, most notable for the RSA design. This behavior is expected since idle states naturally reduce the power consumption of a system.

V. CONCLUSIONS

High-Level Synthesis (HLS) is an important technique to improve the quality and productivity of designing integrated circuits (ICs). In this paper we have shown that HLS is also suitable to facilitate resistance of an IC against timing attacks, without the need for the designer to address the timing manually in a low-level hardware description language. We have shown how a system specification in C-code can be translated to an HDL description that provides time invariance on all security-related paths of the design.

Key contribution for the time invariance is the generation of new scheduling constraints, which reflect annotations regarding the timing of security-related branches and execution paths. Integrated in an HLS tool chain the scheduling constrains are considered in the schedule generation and result in time-balanced control paths. One strength of the presented approach is the compatibility to established HLS optimizations since we solely add scheduling constraints without interfering with other synthesis steps.

The paper outlined a practical implementation of the framework in the open source LegUp HLS tool chain. The practical evaluation for a range of benchmark applications as well as an Elliptic Curve Cryptography (ECC) and RSA implementation could demonstrate the practicability as well as a negligible overhead for synthesis time, as well as area overhead, longest path, and power consumption of the synthesized designs.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under NSF grant numbers 1563652 and 1136146.

REFERENCES

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in CryptologyCRYPTO96*, 1996, pp. 104–113.

[2] B. Mao, W. Hu, A. Althoff, J. Matai, J. Oberg, D. Mu, T. Sherwood, and R. Kastner, "Quantifying timing-based information flow in cryptographic hardware," in *International Conference on Computer Aided Design (ICCAD)*, 2015.

[3] X. Fan, S. Peter, and M. Krstic, "Gals design of ecc against side-channel attacks - a comparative study," in *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2014.

[4] R. S. Chakraborty and S. Bhunia, "Harpoon: an obfuscation-based soc design methodology for hardware protection," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1493–1502, 2009.

[5] J. López and R. Dahab, "High-speed software multiplication in f_2m," in *INDOCRYPT*, 2000, pp. 203–212. [Online]. Available: citeseer.csail.mit.edu/lopez00highspeed.html

[6] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[7] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of todays high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[9] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd annual Design Automation Conference (DAC)*, 2006, pp. 433–438.