Designing Constant-Timed Accelerators using High-Level Synthesis: A Case Study of ECG Biometric Authentication

James Kuban and Tosiron Adegbija
Department of Electrical & Computer Engineering
University of Arizona, Tucson, USA
kubandj@arizona.edu; tosiron@arizona.edu

***Abstract*—High-level synthesis (HLS) is an increasingly popular approach for rapidly designing complex, high-performance, and energy-efficient application-specific accelerators, enabling a shorter time to market and increased productivity. This paper explores a workflow for generating constant-timed accelerators using generic HLS tools in order to eliminate the timing-based side-channel vulnerabilities intrinsic to accelerators generated using state-of-the-art HLS. Since security, performance, and energy are often conflicting design objectives, we also explore ways to mitigate the design overhead. We demonstrate the workflow using a case study of an ECG biometric authentication system, which exemplifies a real-world system with significant timing side-channel vulnerabilities. Results show that the workflow successfully generates constant-timed accelerators and enables designers to use generic HLS tools, thereby minimizing any negative impacts on the design process.**

I. Introduction and Motivation

High-Level Synthesis enables a rapid generation of hardware by raising the design abstraction from low-level hardware description languages (HDL), like Verilog and VHDL, to high-level languages like C or C++. An HLS tool (e.g., Vitis HLS, SmartHLS) can then be used to generate the HDL implementation for synthesis. Given its productivity gains, HLS is increasingly being used in numerous application domains, including security-critical domains, such as biometric authentication, cryptography, healthcare, etc. This raises an important issue: in addition to performance, energy, and area constraints, security constraints must now be considered in the HLS design process [1]. Ideally, designers should be able to use state-of-the-art HLS tools with which they are already familiar in order to maintain design productivity.

This paper explores changes that can be made to the high-level code to mitigate the side-channel vulnerabilities of HLS-generated accelerators. *Side-channel attacks* rely on monitoring physical signatures/leakages (e.g., timing or power leakage) to deduce the computations occurring or private information in the accelerator. For instance, timing attacks [2]—the focus of this paper—rely on timing variances in the accelerator when executing different portions of a workload or with different input data. When an attacker obtains timing patterns, the side-channel information can be analyzed to infer private information [2]. This vulnerability can be mitigated by designing *constant-timed systems* [3].

There has been some prior work to mitigate the side-channel vulnerability of accelerators. For example, Bayrak et al. [4] focused on obfuscating computation times by introducing jitter on the clock line. While this method can increase the complexity of attacks, an attacker can overcome it by increasing the number of power traces used. Furthermore, this approach drastically increases the area and power consumption, and increases the complexity of the design process. Alternatively, Peter et al. [3] proposed an approach to analyze the behavior of high-level code and use the information to instruct the HLS tool's scheduler to add idle cycles to balance the branches in the program. However, this approach is tightly coupled with the LegUp HLS tool on which it is implemented.

Given that HLS primarily aims to increase design productivity, our goal is to maintain—or minimally impact—the benefits of HLS by focusing on the high-level code being synthesized as opposed to the RTL. To enable continuity of productivity, it is important that designers can use HLS tools that they already know. At the same time, designers must be equipped with a straightforward repeatable process for modifying their code to mitigate the vulnerability of the generated accelerators to timing side-channel attacks.

This paper explores a workflow for generating constant-timed hardware accelerators with HLS, using industry-standard HLS tools like Vitis HLS and SmartHLS. The workflow focuses on identifying and modifying portions of the high-level code to maintain constant timing across different computations or input data. We demonstrate the proposed workflow using an ECG biometric authentication (EBA) algorithm and show that the approach achieves constant timing across different functions and data inputs. The approach also reduces the power consumption and resource usage compared to the state-of-the-art HLS approach, while trading off latency.

II. Background on ECG Biometric Authentication

ECG biometric authentication (EBA) is an increasingly popular approach for biometric user authentication based on physiological signals representing the electrical activity of the human heart. The ECG (electrocardiogram) signals—the heart's electrical activity—contain lots of useful information that are easy to obtain using readily available sensors, uniquely identifiable, and permanent. EBA is an excellent case study for

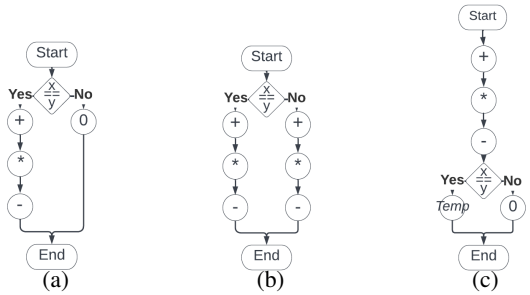

Fig. 1: Dataflow graphs (DFG) illustrating (a) an unbalanced branch, (b) a naively balanced branch, and (c) a branch balanced using our workflow

our work because: 1) biometric systems are a prime target for timing side-channel attacks [5]. As such, HLS design approaches are urgently needed to mitigate the vulnerability of EBA accelerators and 2) the EBA system is an important real-world complex system that presents a variety of challenges that must be overcome to achieve constant timing. The system consists of distinct functions with disparate timing behaviors between its functions and with different user inputs.

In summary, the EBA algorithm contains four main functions that must be performed on input signals after data acquisition: *filtering, segmentation, feature extraction,* and *matching*. Here, we briefly describe these functions, but direct readers to [6] for the algorithm's low-level details.

Filtering: Filtering processes the input signal to remove noise and enhance the quality of the biometric traits. Noise can occur from a variety of factors such as muscle movement, electromagnetic interference (EMI), or improper probe contact. These signal spikes can cause false R-peak detection—the maximum amplitude in the electrocardiogram—and must be filtered before further processing.

Segmentation: Segmentation splits the ECG signal into its unique *P, QRS,* and *T* waveforms. Given that these waveforms typically repeat throughout the signal, segmentation also identifies and eliminates redundancies to reduce template size and simplify the matching process.

Feature extraction: Feature extraction uses the segmentation output to find wavelets in the original signal to parameterize the individual characteristics of each user. This function generates each user's ECG fingerprint using the amplitude and temporal locations of the waveforms' fiducial points (P, QRS, and T), which are recorded as a fraction of the wavelet size and averaged over the dataset.

Matching: The final step compares the averaged features to a set of stored user profiles using a simple Euclidean distance algorithm. The comparison—whether or not the new user's biometric traits match the stored template—determines the new input user's authorization to access the system.

III. Workflow for HLS of Constant-Timed Accelerators

The constant timing approach is well-known for mitigating the vulnerability of processing systems to side-channel attacks [6], [7], [3]. In general, the primary source of timing inconsistency in HLS-generated accelerators is the presence of *unbalanced branches* in the high-level program. Unbalanced branches are data-dependent branches with varying numbers and complexity of computations such that the branch path affects how long it takes to execute the program. Fig. 1 depicts an arbitrary dataflow graph (DFG) to illustrate the security-relevant kinds of branches that might occur in a program. Fig. 1a illustrates an unbalanced branch whose timing depends on the value of x: the code takes much longer to execute if $x == y$ than if x != y. *Balanced branches*, on the other hand, are those that have an equivalent number and complexity of computations regardless of the branch direction. Balanced branches enable constant timing regardless of the data inputs. Fig. 1b illustrates a naively balanced branch with the computations on the longer branch path of Fig. 1a replicated on the shorter path; dummy/idle operations can be performed on the no path to maintain the result's consistency.

Our approach aims to minimize the overhead (especially the power and resource overhead) of balanced branches by reducing the redundancy. The branch balancing achieved using our workflow is illustrated in Fig. 1c. Given the resource constraints of the target system, our approach trades off execution latency for minimizing the resource usage, while also limiting the latency overhead. In what follows, we describe the proposed workflow and suggested high-level code modifications for generating constant-timed accelerators.

A. Overview of the workflow

Fig. 2 depicts an overview of the workflow for modifying high-level programs to generate constant-timed accelerators. Given an input program (in this case, the EBA algorithm), the program is broken down into its component functions (see Section II). The granularity depends on the complexity of the program and can also affect the tractability of the design process. The EBA algorithm can be cleanly broken down into different functions based on the different operations being performed. Some other programs might not have the functions as clearly demarcated and would require some designer effort to break them down into their functions. Each function's timing is then analyzed to determine the timing variations. This can be achieved using simulations of the high-level code, via RTL behavioral simulations, or synthesis for different inputs. The goal is to ensure that timing variations do not exceed an *attack threshold*—the time required for an attacker to gather the necessary traces for an attack. For the EBA system, we used a threshold of 500µs, based on prior work [8], [6]. Once timing information has been obtained, if unbalanced branches are identified, the program is modified to achieve *intra-function constant timing*. Then, *inter-function constant timing* can be achieved by reducing the number of functions, optimizing the longest function, and elongating shorter functions, if necessary.

B. Intra-function constant timing

Intra-function constant timing ensures that the timings within functions remain constant regardless of the input data. To this end, the number and types of computations in all

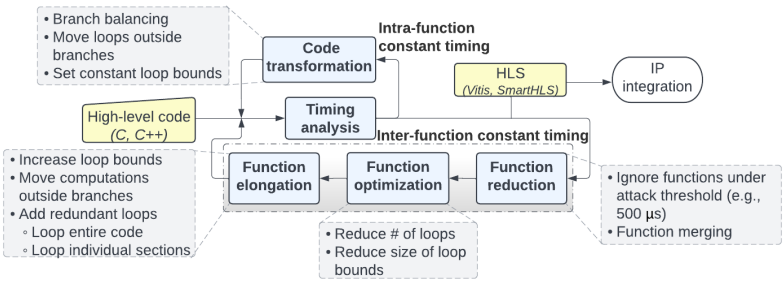

Fig. 2: Overview of the constant timing workflow. The workflow provides a systematic approach and suggestions for modifying the high-level code to enable constant timing, while mitigating the attendant overheads.

branch paths are first determined and the shorter branch path is taken as the lower limit of the branch's timing. Thereafter, the excess computations are moved before the branch and assigned to dummy variables. The computations in the branches can then be replaced by an assignment of those dummy variables to their final variables.

C. Inter-function constant timing

Inter-function constant timing ensures that the timings across different functions remain constant. This mitigates the vulnerability in an attacker being able to infer what operations are being performed at any given time, especially in systems, like the EBA system, where the timing of the current function can provide clues about the operations being performed [5].

One approach to achieving inter-function constant timing is to elongate the shorter functions in the program to match the longest function. This approach was followed in prior work [6] using architecture changes (e.g., slower clocks) to lengthen shorter functions to match the longest functions. From the HLS perspective where using different clock frequencies might not be an option, computations can be taken out of all the branches and the bounds of each loop increased. When there is still a substantial difference in latency, loops can be added around computationally complex segments of the code to artificially inflate the latency (at the potential expense of energy). Similarly, loops can be placed around shorter functions to make their latency equal to that of the longest function.

D. Strategies for mitigating the latency overhead

An important challenge that arises with inter-function constant timing is the increase in latency since the shorter functions are elongated to match the longest. As a result, the longest function's latency will be multiplied by the total number of functions. At a high level, this overhead can be mitigated by either reducing the latency of the longest function or by reducing the number of functions.

The easiest approach for mitigating the latency overhead is *function merging*, which involves reducing the number of functions by combining them. Functions can be merged when the output of one of the functions is only required by one other function. As such, the code can be written to consolidate both functions into one. For example, the output of the filtering function in the EBA algorithm is only required by the segmentation function, and the filtering does not need to finish before segmentation can begin. Thus, the signal can be pipelined to overlap filtering and segmentation, thereby reducing the overall latency while consolidating the functions. The latency overhead can be further mitigated by exploring additional optimizations (e.g., reducing loop bounds, reducing the number of loops, removal of nested loops, etc.).

IV. Experimental Results

This section reports the results of employing the proposed workflow to design a constant-timed accelerator for the EBA system. We evaluate the approach with respect to the timing variance of the original synthesized code—the original algorithm is implemented in C—compared to that of the balanced code using the workflow described herein.

To compare our work with the state-of-the-art, we implemented the EBA algorithm using the current HLS approach [9] and analyzed the timing across different functions and with 8 different users (the input data) from the Physionet Fantasia dataset [10]. We used Xilinx Vitis HLS to synthesize the C program to Verilog and used Xilinx Vivado to program the FPGA. We implemented the EBA system on a Nexys4_DDR FPGA at a 100MHz clock frequency.

A. Intra- and inter-function constant timing

Overall, our workflow successfully generated similarly constant-timed accelerators to prior work [3], [11] with the notable distinction that our approach can be used with any state-of-the-art HLS tool.

From analyzing the EBA algorithm, we found that the segmentation and feature extraction steps had the largest timing variations (far surpassing the attack threshold). We also observed that the process may need to be iterated upon. For example, after modifying the program for inter-function constant timing, the intra-function timing should be checked to ensure that the behavior has not changed. Our workflow substantially reduced the timing variations in the different functions. For instance, the segmentation function's timing variance reduced by 1770x, from 39,473.79 µs to 22.3 µs.

Furthermore, to mitigate the latency overhead, we also used function merging (Section III-D) for the filtering and segmentation functions. We found that filtering takes significantly less time than segmentation and the outputs of filtering are only utilized by segmentation, making them great candidates for merging. This approach reduced the latency overhead by 33.5% compared to an unmerged system (additional figures and analysis omitted for brevity).

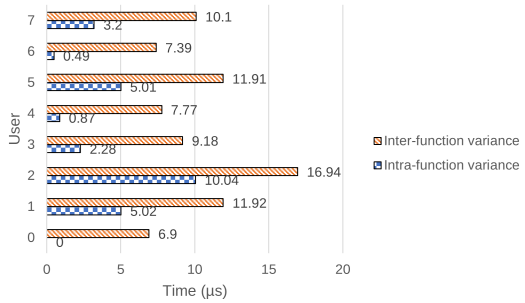

Fig. 3: Post-modification intra- and inter-function variance for the different users. Inter-function variance is with respect to each function's timing for individual users and intra-function variance is compared to user 0 which had the lowest latency.

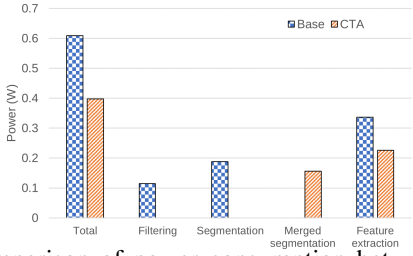

Fig. 4: Comparison of power consumption between the base accelerator and the constant-timed accelerator (CTA). The matching function is not shown since it was unchanged.

Fig. 3 depicts the post-modification maximum intra- and inter-function timing variances across the different users. Our workflow reduced the maximum variance from 9322.94 µs to 16.94 µs, falling well below the attack threshold. Note that the constant timing was achieved at the expense of latency, which increased from 0.104s to 1.1s—a 10.6x increase. However, this latency tradeoff is in line with prior works [6], [7] that have shown that achieving constant timing systems usually involves a latency tradeoff. Importantly, no functional timing violation occurs as a result of our approach.

B. Power consumption and resource usage

Unlike latency, the constant-timed accelerator *reduced* the power and resource usage compared to the base accelerator (using the state-of-the-art HLS). Fig. 4 compares the power consumption of the constant-timed accelerator (CTA) to the base accelerator. The figure depicts the overall and function-specific power comparisons. Overall, our workflow decreased the power by 34.6% (from 0.609 to 0.398 W). The merged segmentation function (consolidating the filtering and segmentation functions into a single unit within the accelerator) reduced the power by 45.8% compared to the sum of the individual functions. Even though the modified segmentation function increased the number of calculations being performed, the function merging process enabled a power reduction as a result of the reduction in resource utilization.

Table I summarizes the CTA's resource usage compared to the base accelerator. On average for the different kinds of resources, the CTA reduced the resource usage by 27%. The CTA achieved the highest reduction in the block RAM (BRAM) utilization, by 73%, due to the merging of filtering and segmentation, which reduced the need to store/transfer data between functions.

TABLE I: Resource utilization

Resource	Base	CTA	% improvement
LUT	39351	32950	16%
LUTRAM	919	587	36%
FF	53600	43830	18%
BRAM	105.5	29	73%
DSP	96	51	47%
IO	35	35	0%
BUFG	1	1	0%

V. Conclusion

In this paper, we explored a workflow for modifying a high-level program to enable the synthesis of constant-timed accelerators. The goal is to enable designers to use generic HLS tools with which they are familiar and mitigate the overheads of designing constant-timed accelerators. We evaluated the proposed workflow using an ECG biometric authentication accelerator. The results show that the proposed approach successfully enables constant timing between different algorithm functions and input data. The proposed approach also reduces the power consumption and resource usage compared to the state-of-the-art, at the expense of latency overhead. Future work involves exploring ways to mitigate the latency overhead, exploring the generalizability of the workflow to a wider variety of applications, and extending the workflow to account for power side-channel attacks.

References

[1] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni, "Securing hardware accelerators: A new challenge for high-level synthesis," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 77–80, 2018.

[2] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 355–371.

[3] S. Peter and T. Givargis, "Towards a timing attack aware high-level synthesis of integrated circuits," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 452–455.

[4] A. G. Bayrak, N. Velickovic, F. Regazzoni, D. Novo, P. Brisk, and P. Ienne, "An eda-friendly protection scheme against side-channel attacks," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 410–415.

[5] J. Galbally, "A new foe in biometrics: A narrative review of side-channel attacks," *Computers Security*, vol. 96, p. 101902, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820301784

[6] R. Cordeiro, D. Gajaria, A. Limaye, T. Adegbija, N. Karimian, and F. Tehranipoor, "Ecg-based authentication using timing-aware domain-specific architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3373–3384, 2020.

[7] Z. B. Aweke and T. Austin, "Øzone: Efficient execution with zero timing leakage for modern microarchitectures," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1123–1128.

[8] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732.

[9] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[10] I. Silva and G. B. Moody, "An open-source toolbox for analysing and processing physionet databases in matlab and octave," *Journal of open research software*, vol. 2, no. 1, 2014.

[11] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang, "High-level synthesis with timing-sensitive information flow enforcement," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.