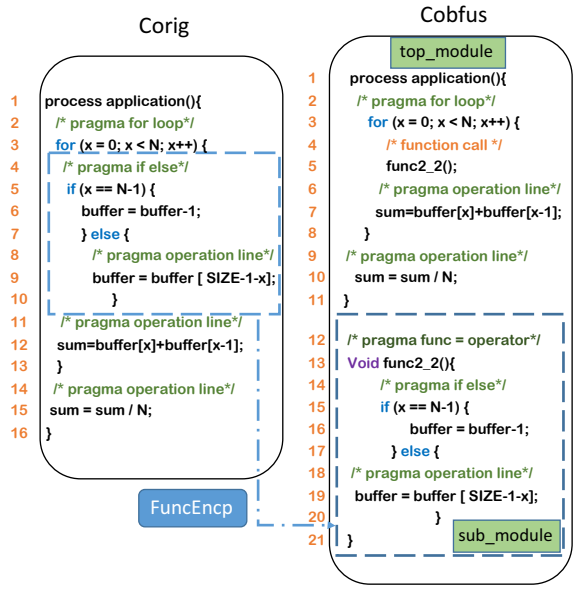

Figure 3: Example of function encapsulation.

new C function need to be declared as global variations in order to maintain the functionality. We call this step function encapsulation (FuncEncp).

Step 2: Individual Obfuscation

Since the functional operator option is used to encapsulate the portion of the behavioral description to be mapped to the eFPGA, as shown in Fig. 2, once the selected portion is automatically encapsulated into a function and the functional operator pragma specified, two separate files are generated when the behavioral description is parsed. One file with the main functionality of the application (application.c or *top_module*) and another with the part to be obfuscated (obfus.c or *sub_module*). These files can be in turn synthesized separately. The HLS proceeds with *top_module* (application.c) using the ASIC technology library ($HLSTechLib_{ASIC}$) and the encapsulated portion (obfus.c) using the eFPGA technology library ($HLSTechLib_{eFPGA}$), both generated in the library pre-characterization stage.

The result is two RTL descriptions; one for the top module (application.v) and one for the encapsulated module (obfus.v), and a report file with the quality of results (QoR_{HLS}) indicating the area of each module and latency. Our method also computes the security cost function S_i (Eq. 1) of this configuration. This allows our method to determine the overhead of mapping specific portions of code to the eFPGA quickly as well as measuring the added security.

Experimental results have shown that the QoR_{HLS} is good enough to guide our explorer in finding the best mappings, but that the actual area and delay information reported is often not too accurate (on average, we have observed differences of 20-30%). Thus, as shown in Fig. 2, the flow is extended to allow a full logic synthesis for the RTL code generated after HLS. The logic synthesis output is a gate netlist for the two modules and a new more accurate report (QoR_{LS}). This, allows us to more accurately characterize the obfuscated architecture.

Our proposed method iteratively considers each line of C code as the obfuscated portion and re-synthesizes it. If the area overhead

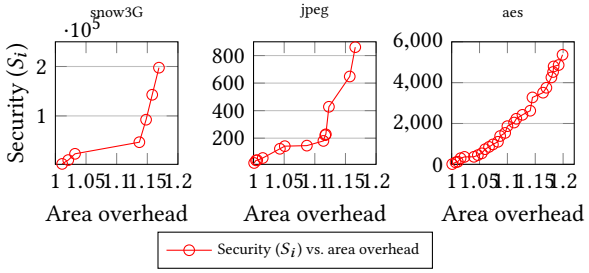

Figure 4: Area overhead vs. security cost function (Eq. 1) for pareto-optimal designs.

Table 1: Area overhead vs. security metric (TTB) for both brute-force and SAT-based attack [hours]

Benchmark	area overhead [%]	$\text{TTB}_{bfa}[hrs]$	$\text{TTB}_{sba}[hrs]$
interp	4	2^{65}	2^{56}
decim	7.8	2^{32}	2^{23}
jpeg	5.1	2^{1594}	2^{134}
kasumi	4.8	2^{1187}	2^{169}
snow3G	3.2	2^{201}	2^{119}
aes	4.7	2^{321}	2^{70}

exceeds a specified value (lines 13-15 and lines 25-29 in Algorithm 2), then this line will not be considered for inclusion in the obfuscated portion.

Step 3: Merging of Results

Our proposed method continues by merging the results obtained in step 1, whereby each result is due to the obfuscation of a single line of C code. In this step, multiple lines of C code are considered for possible concurrent obfuscation. To achieve this, the proposed method grabs each valid single obfuscated line, in turn, and iteratively seeks to merge additional subsequent obfuscated lines of C code. Once the area constraint would be violated, merging ceases and the merging process starts over with the next valid single line of obfuscated C code (lines 19-30 in Algorithm 2). Note that each single valid line of obfuscated C code is trial merged with subsequent lines of obfuscated C code until the area constraints are violated. This means that there may be considerable overlap, in terms of lines of obfuscated C code, between the various mergings. However, this design space will be significantly pruned, retaining only the Pareto-optimal points, after considering the area overhead, delay overhead and security of each merging.

6 EXPERIMENTAL RESULTS

Different computationally intensive applications, amiable to hardware acceleration, were selected in order to test our proposed method. For this purpose, six SystemC benchmarks from the freely available Synthesizable SystemC Benchmark suite S2CBench [12] were used. In particular, they are : 3-stage *interpolation* filter, 5-stage *decimation* filter, *jpeg* encoder, *snow3G* stream cipher, *kasumi* block cipher used in mobile communications and *aes*.

The HLS tool used is CyberWorkBench from NEC [6] and the target technology is Globalfoundries 65nm. The HLS target frequency is fixed in all cases to 200MHz and the logic synthesis tool used is Synopsys's Design Compiler. The experiments are conducted

Table 2: FSM increase for configurations shown in Table 1

Benchmark	obfuscated FSMs	original FSMs	increased FSMs
interp	5	4	1
decim	9	8	1
jpeg	48	46	2
kasumi	4	3	1
snow3G	4	3	1
aes	10	9	1

on an Intel i7-6700 3.50GHZ CPU and 16 GB memory, running CentOS 7.0. It should be noted that in order to get accurate results, the results presented are those reported after logic synthesis.

Fig. 4 shows the Pareto-optimal trade-off curve design space exploration results for benchmarks *snow3G*, *jpeg* and *aes* as an example of area overhead versus security cost function. The y-axis represents the security cost function (Eq. 1) when different portions of the behavioral description are mapped onto the eFPGA. The area overhead was restricted to 20 percent more than the original circuit size. It shows, as expected, a relationship between the size of the circuit mapped to the eFPGA block and the security. The larger the portions of the circuit mapped to the eFPGA block, the more secure the circuit becomes.

Table 1 lists area overhead versus the TTB security metric for both brute-force and SAT-based attacks for the six benchmarks used. We select the obfuscated designs (D_{obfus}) with the area overhead fixed at around 5% from the Pareto-optimal configurations as an example to show the TTB in hours. TTB is obtained from Eq. 2 and Eq. 3 in Section 4 where bitstream size ($bitstream_i$), logic cell number ($cell_i$) and $latency_i$ are obtained from QoR of the logic synthesis step. In each case, for all practical purposes, it would not be possible for an attacker to reverse engineer the design.

One additional dimension that Fig. 4 does not cover is the performance overhead introduced by mapping a portion of the design to the eFPGA. In order to simplify the analysis, Table 2 summarizes the increase in FSMs for configurations shown in Table 1. In all cases, the target operating frequency of 200MHz could be met, as we provided the HLS tools with the detailed technology libraries of the ASIC and eFPGA, and hence the scheduling phase of the HLS process can insert more or less logic circuits in a single step so that the 5ns maximum delay (1/200MHz) is not violated. The performance degradation is thus observed as the number of FSMs required to produce a new output, as the extra delay introduced from the eFPGA block can force more scheduling steps. Table 2 shows the FSM increase of the six benchmarks. Only one additional state is needed after obfuscation for 5 of the 6 benchmarks.

In Table 3, the second column is the number of lines of C code in the benchmarks. The third column is the number of pragmas, or candidate code lines that could be obfuscated. The fourth column is the number of lines of code implemented in the eFPGA, for the results in Table 1. The last column shows the running time of our proposed design space exploration method for the results in Table 1, where in this case the run time of the HLS process accounts for 25% of the total runtime while the logic synthesis accounts for the remaining 75%. In all cases, the TTB is so long that for all practical purposes the designs are secure and the area overhead is around 5%. In most cases the latency was increased by only one FSM, while the computation time are modest.

Table 3: Selective extraction algorithm runtime [seconds] for the results shown in Table 1

Benchmark	# of code lines	pragma#	Obf.# of code lines	runtime
interp	155	28	22	18
decim	433	34	13	22
jpeg	482	72	62	51
kasumi	314	28	4	25
snow3G	458	73	6	26
aes	892	59	10	64

7 CONCLUSIONS

In this work we have addressed the hardware security problem using an algorithm and architecture to map selective portions of a behavioral description for HLS to an eFPGA. In this manner, only the end-user has access to the full functionality of the chip. Using six benchmark circuits, we showed that the TTB is so long (at least 8 million hours) that for all practical purposes the designs are secure, while incurring area overheads of around 5%. Further, latencies were only slightly increased, while the computation times are under one minute.

REFERENCES

[1] Achronix. 2018. Speedcore eFPGA. (2018).
[2] Yousra Alkabani et al. 2007. Active Hardware Metering for Intellectual Property Protection and Security. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS'07)*. Article 20, 16 pages.
[3] R. S. Chakraborty and S. Bhunia. 2009. HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection. *IEEE TCAD* 28, 10 (Oct 2009), 1493–1502.
[4] Y. Lao et al. 2015. Obfuscating DSP Circuits via High-Level Transformations. *IEEE TVLSI* 23, 5 (May 2015), 819–830.
[5] Bao Liu et al. 2014. Embedded Reconfigurable Logic for ASIC Design Obfuscation Against Supply Chain Attacks. In *DATE*. 243:1–243:6.
[6] NEC. 2015. CyberWorkBench v.5.2. (2015).
[7] Quicklogic. 2018. ArticPro. (2018).
[8] J. Rajendran et al. 2013. Is split manufacturing secure?. In *2013 DATE*. 1259–1264.
[9] J. Rajendran et al. 2013. Security Analysis of Integrated Circuit Camouflaging. In *SIGSAC Conference on Computer & Communications Security (CCS '13)*. 709–720.
[10] J. Rajendran et al. 2015. Fault Analysis-Based Logic Encryption. *IEEE Trans. Comput.* 64, 2 (Feb 2015), 410–424.
[11] Jarrod A. Roy et al. 2008. EPIC: Ending Piracy of Integrated Circuits. In *DATE*. 1069–1074.
[12] Carrion Benjamin Schafer et al. 2014. S2CBench:Synthesizable SystemC Benchmark Suite. *IEEE Embedded Systems Letters* 6, 3 (2014), 53–56.
[13] Mustafa M. Shihab et al. 2019. Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.
[14] S.Liu et al. 2013. Achieving Energy Efficiency Through Runtime Partial Reconfiguration on Reconfigurable Systems. *ACM Trans. Embed. Comput. Syst.* 12, 3 (2013), 72:1–72:21.
[15] Jingxiang Tian et al. 2017. A field programmable transistor array featuring single-cycle partial/full dynamic reconfiguration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1336–1341.
[16] T. Winograd et al. 2016. Hybrid STT-CMOS designs for reverse-engineering prevention. In *DAC*. 1–6.
[17] M. Yasin et al. 2016. On Improving the Security of Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 9 (Sept 2016), 1411–1424.