Smooth Passage with the Guards: Second-Order Hardware Masking of the AES with Low Randomness and Low Latency

Artifact for TCHES 2024, Issue 1, Submission #48

Our paper proposes a COTG-based second-order masked AES design which is based on an optimized S-box. In the original paper, Section 3 focuses on our optimized second-order DOM AES S-box (Figure 1), and compares it to two other designs (Table 1). We give the RTL code and testbench for all three designs in Section 2, explain how to formally verify it with Coco [Gig+21] in Section 4 and provide scripts for reproducing area numbers in Section 6. Sections 4 and 5 of the original paper describe our novel COTG-based second-order masked AES architecture including a Trivium RNG to supply the required randomness (Figure 4), and compare it to two other less optimal designs (Table 3c). Therefore, the artifact includes the RTL code and testbench for both the new and reference designs (Section 3), as well as instructions to produce the area numbers from Table 3c (Section 6). We describe how to formally verify our new design with Coco in Section 4, according to Section 6.2 in the paper, and explain how to perform the experimental verification on an FPGA board in Section 5.

The included artifacts can be summarized as follows:

- [**Section 2**] **Masked AES S-boxes**
  - Refers to Section 3, Figure 1, Table 1 in paper
  - Includes RTL code and testbenches
  - Optimized second-order DOM AES S-box (using the new types of multipliers and 78 bits of fresh randomness)
  - Insecure second-order DOM AES S-box [GMK16] (using original DOM-*dep* multipliers and 84 bits of fresh randomness)
  - Fixed second-order DOM AES S-box (using fixed DOM-*dep* multipliers and 104 bits of fresh randomness)
- [**Section 3**] **Masked AES architectures**
  - Refers to Section 4, Section 5, Figure 4, Table 3c in paper
  - Includes RTL code and testbenches
  - Optimized DOM-AES with COTG (using 1 Trivium instance and 3 200 random bits per encryption)
  - Optimized DOM-AES without COTG (using 7.5 Trivium instances and 15 600 random bits per encryption)
  - DOM-AES with fixed DOM-*dep* without COTG (using 10 Trivium instances and 20 800 random bits per encryption)
- [**Section 4**] **Formal verification**
  - Refers to Section 6.2 in the paper
  - Includes Coco source code and verification scripts

  - Formal verification of fixed DOM-*dep* multiplier and masked AES S-box

- [Section 5] **Experimental Verification**
  - Refers to Section 6.3 in the paper
  - Includes description of evaluation setup, the script to record power traces, the Vivado project, the bitstream file, and a sample trace set
  - Subject of the experimental evaluation is the Optimized DOM-AES with COTG using 1 Trivium instance and 3 200 random bits per encryption

- [Section 6] **Generation of area numbers**
  - Refers to Table 1, Table 3a, Table 3c in the paper
  - Includes information about proprietary synthesis flow used in the paper
  - Includes instructions to use alternative Open Source synthesis flow using Yosys and NanGate OCL

- `supplementary.pdf`: The supplementary PDF which was originally attached to our submission.

1 Preliminaries

The artifacts are stored in `artifact.zip`. Extract the directory. In the following, we will refer to this location as `$ARTIFACT_HOME`.

All experiments are executed on a PC running Ubuntu 22.04.3 LTS, except for the experimental verification (Section 5), where we use Ubuntu 22.04.2 LTS. Although these are the recommended platforms to run the evaluations, it might also work for others.

Some experiments require Verilator. We recommend to use 5.014 (built from source using Github commit e6b0bdd4d), but the experiments could also work with other versions/commits. When building from source is not possible, a list of pre-built packages for various distributions can be found here. Alternatively, one could use the docker container. In case of using pre-built images, adaptions to the Makefiles are necessary whenever `$VERILATOR_HOME_DIR` is used. The Makefile expects that `$VERILATOR_HOME_DIR/bin/verilator` points to the Verilator executable, which needs to be adapted accordingly.

2 Masked AES S-boxes

For the three different S-boxes mentioned in Table 1 in the paper, we give the SystemVerilog code and a Verilator testbench in the directory `sboxes`.

2.1 Directory Structure

- `sboxes/rtl`: SystemVerilog source code
  - `sboxes/rtl/sbox_opt`: SystemVerilog code of our optimized second-order DOM AES S-box using the new types of multipliers and 78 bits of fresh randomness. It follows the structure of Figure 1 in the paper.

  - `sboxes/rtl/sbox_insecure`: SystemVerilog code of the second-order DOM AES S-box proposed by [GMK16] using the original DOM-*dep* multipliers which we show to be insecure in the presence of glitches, and 84 bits of fresh randomness. It follows the structure of Figure A3 in the supplementary material.
  - `sboxes/rtl/sbox_fixed`: SystemVerilog code of the fixed second-order DOM AES S-box using the fixed DOM-*dep* multipliers with 104 bits of fresh randomness. It follows the structure of Figure A3 in the supplementary material. The implementation of the fixed DOM-*dep* multiplier is given in `sboxes/rtl/sbox_fixed/aes_dom_dep_mul_gf2pn.sv`

- `sboxes/tb/tb_sbox.cpp`: Verilator testbench written in C++. Since all three S-box implementations have the same input and output signals, a single testbench can be used to simulate all of them. It also produces a VCD trace file (`sboxes/build/aes_sbox_dom.vcd`).
- `sboxes/build`: directory used to store build artifacts such as the Verilator model and the VCD trace file
- `sboxes/Makefile`: Makefile used to build the Verilator model. The supported targets are `sbox_opt`, `sbox_insecure`, `sbox_fixed` to simulate the S-boxes, and `clean` to remove build artifacts.

2.2 Simulation

In order to simulate a specific S-box design with Verilator, the following steps are necessary:

1. Install the necessary software:
   - Verilator (5.014 2023-08-06 rev v5.014-35-ge6b0bdd4d). Build it from source by cloning the github repository. Follow the build instructions there. Use commit e6b0bdd4d. Additional information can be found in Section 1.
   - gcc/g++ 11.4.0
   - *(Optional)* GtkWave for viewing the resulting VCD trace file
2. Navigate to the correct directory:
   ```
   cd $ARTIFACT_HOME/sboxes
   ```
3. Set the environment variable `$VERILATOR_HOME_DIR` such that it points to the Verilator installation directory. For example, if you cloned the Verilator repository in `$HOME`, it should be set to `$HOME/verilator`, e.g.:
   ```
   export VERILATOR_HOME_DIR=$HOME/verilator
   ```
4. Select a design, either `sbox_opt`, `sbox_insecure` or `sbox_fixed`, and build the Verilator model, e.g.:
   ```
   make sbox_opt
   ```
5. Start the simulation, which does 10 000 evaluations of the masked S-box and checks if the result is correct. The simulation trace file can be found in `sboxes/build/aes_sbox_dom.vcd`.
   ```
   ./build/aes_sbox_dom
   ```

2.3 Description of ports

The top module of the respective S-box design can be found in `aes_sbox_dom.sv` in the respective subdirectory. It computes $y = \text{sbox}(x)$ using the three shares of x s.t. $x = x_0 \oplus x_1 \oplus x_2$, and outputs the three shares of y s.t. $y = y_0 \oplus y_1 \oplus y_2$. In Table 1 we give a preciser description of all input ports. In Figure 1 we give an example of the timing of input and output signals.

Port name	Direction	Width	Description
`clk_i`	input	1	Primary clock
`rst_ni`	input	1	Reset signal, active = 0, inactive = 1
`en_i`	input	1	Enable signal, must be high during S-box computation
`finished_o`	output	1	High for exactly one cycle after S-box computation has finished
`data_i`	input	8	x_0 (share 0 of x)
`mask0_i`	input	8	x_1 (share 1 of x)
`mask1_i`	input	8	x_2 (share 2 of x)
`prd_i`	input	`sbox_opt`: 78 `sbox_insecure`: 84 `sbox_fixed`: 104	Fresh randomness consumed by the design
`data_o`	output	8	y_0 (share 0 of y)
`mask0_o`	output	8	y_1 (share 1 of y)
`mask1_o`	output	8	y_2 (share 2 of y)

Table 1: Description of ports for S-box designs

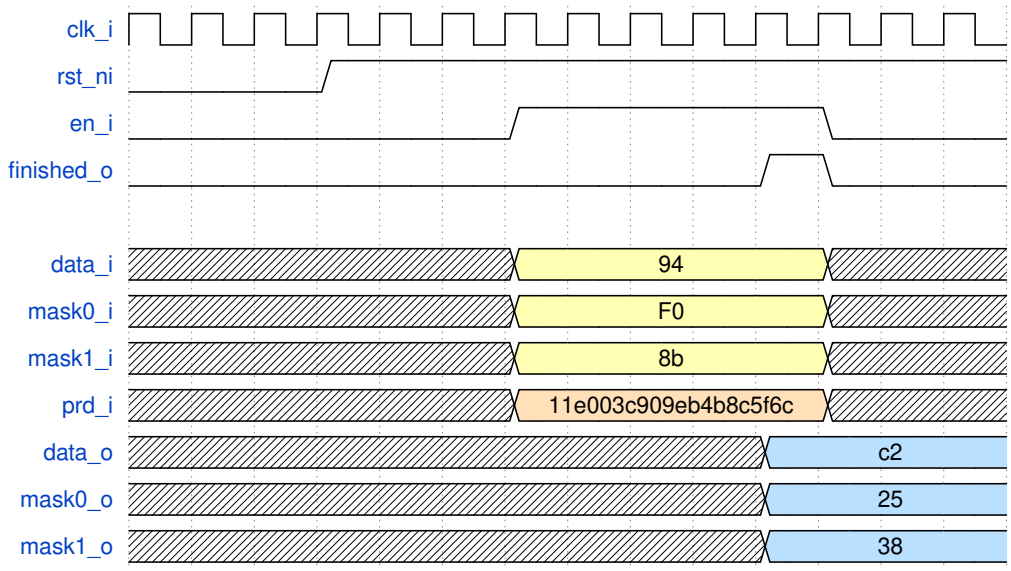


Figure 1: Timing of signals in `sbox_opt` for S-box input $x = ef = 94 \oplus f0 \oplus 8b$, computing the output $y = df = c2 \oplus 25 \oplus 38$

3 Masked AES architectures

For the three different AES architectures mentioned in Table 3c in the paper, we give the SystemVerilog code and a Verilator testbench in the directory `aes_architectures`.

3.1 Directory Structure

- `aes_architectures/rtl`: SystemVerilog source code
  - `aes_architectures/rtl/opt`: SystemVerilog code of our optimized AES implementation using COTG connected to a Trivium RNG. The general architecture is sketched in Figure 4 in the paper. It uses 3 200 random bits per encryption.
  - `aes_architectures/rtl/opt_nocotg`: SystemVerilog code of our optimized AES implementation without using COTG. It is connected to 7.5 Trivium instances. It also follows the architecture sketched in Figure 4, with the difference that the Trivium RNG must deliver 480 bits of randomness per cycle. Therefore, the code base is the same as for the `opt` architecture, except for `top.sv` and `aes_top.sv`
  - `aes_architectures/rtl/fixed`: SystemVerilog code of a DOM AES implementation without using COTG, using the fixed version of DOM-*dep* multipliers, i.e., the `sbox_fixed` described in the previous chapter. It needs to be connected to 10 Trivium instances since it requires 640 bits of fresh randomness per cycle.
- `aes_architectures/tb/tb_sbox.cpp`: Verilator testbench written in C++. Since all three AES architectures have the same input and output ports, a single testbench can be used to simulate all of them. It also produces a VCD trace file (`aes_architectures/build/aes.vcd`)
- `aes_architectures/build`: directory used to store build artifacts such as the Verilator model and the VCD trace file
- `aes_architectures/Makefile`: Makefile used to build the Verilator model. The supported targets are `opt`, `opt_nocotg`, `fixed` to simulate the AES architecture, and `clean` to remove build artifacts.

3.2 Simulation

In order to simulate a specific AES design with Verilator, the following steps are necessary:

1. Install the necessary software:
   - Verilator (5.014 2023-08-06 rev v5.014-35-ge6b0bdd4d). Build it from source by cloning the github repository. Follow the build instructions there. Use commit e6b0bdd4d. Additional information can be found in Section 1.
   - gcc/g++ 11.4.0
   - *(Optional)* GtkWave for viewing the resulting VCD trace file
2. Navigate to the correct directory:

```
cd $ARTIFACT_HOME/aes_architectures
```

3. Set the environment variable `$VERILATOR_HOME_DIR` such that it points to the Verilator installation directory. For example, if you cloned the Verilator repository in `$HOME`, it should be set to `$HOME/verilator`, e.g.:

```
export VERILATOR_HOME_DIR=$HOME/verilator
```

4. Select a design, either `opt`, `opt_nocotg` or `fixed`, and build the Verilator model, e.g.:

```
make opt
```

5. Start the simulation, which first initializes the Trivium RNG(s) with a random key and IV, and then does 1000 AES encryption with a random key and plaintext. The ciphertext is compared with the output of the *tinyAES128* implementation. The simulation trace file can be found in `aes_architectures/build/aes.vcd`.

```
./build/aes
```

3.3 State machine and block diagram

The organization of Verilog modules to build our optimized AES architecture using COTG is shown in Figure 4. The encryption is controlled by two state machines. The outer state machine, as shown in Figure 2, allows to either reseed/initialize the RNG, or start the AES encryption. The inner state machine, as shown in Figure 3, is used for the AES encryption itself.

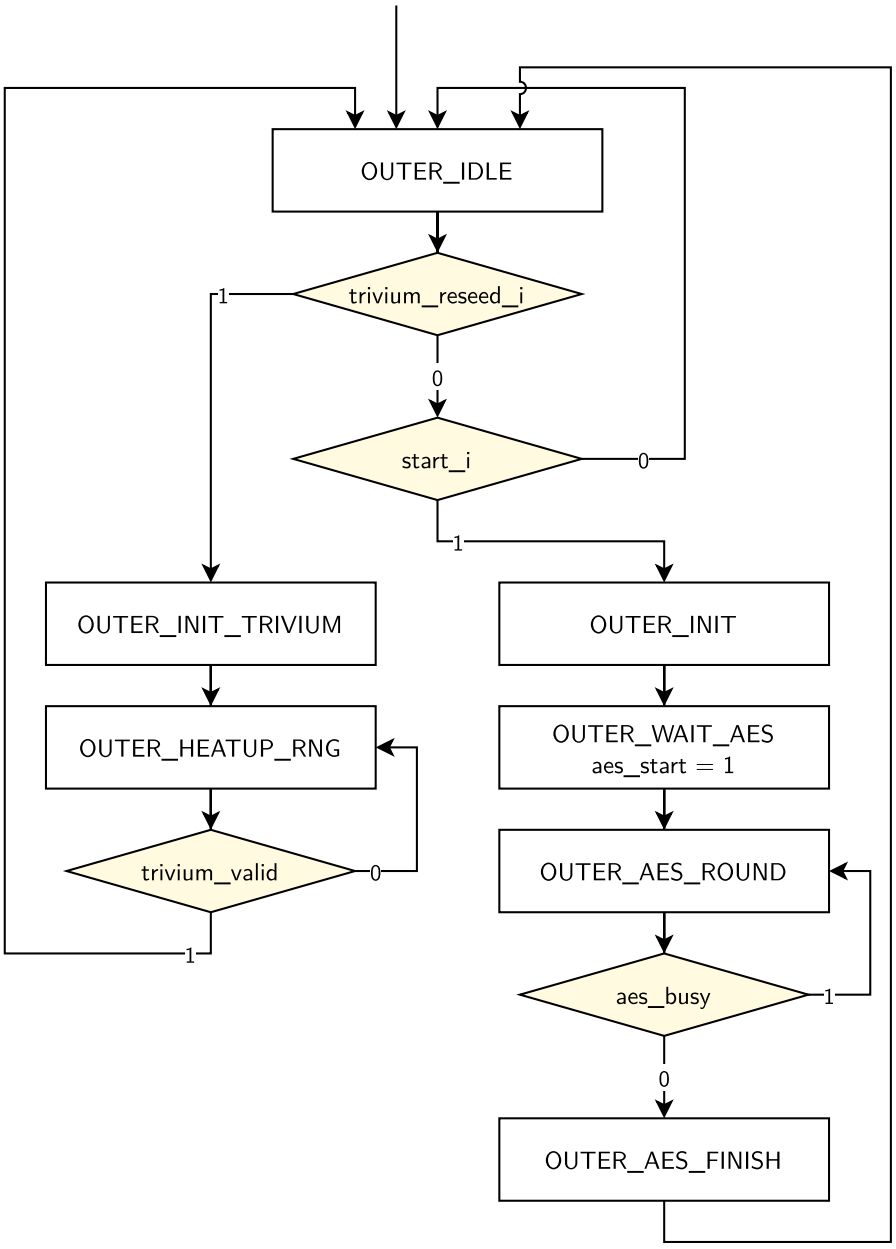


Figure 2: State machine controlling the RNG reseeding and AES encryption in `top`. By setting `trivium_reseed_i` high, the RNG can be reseeded. Reseeding is over if `trivium_valid` is high. By setting `start_i` high, an AES encryption is started. In state `OUTER_WAIT_AES`, the control signal `aes_start` which is connected to port `start_i` of the `aes` module is set high. The encryption is finished if (`aes_busy` turns from high to low).

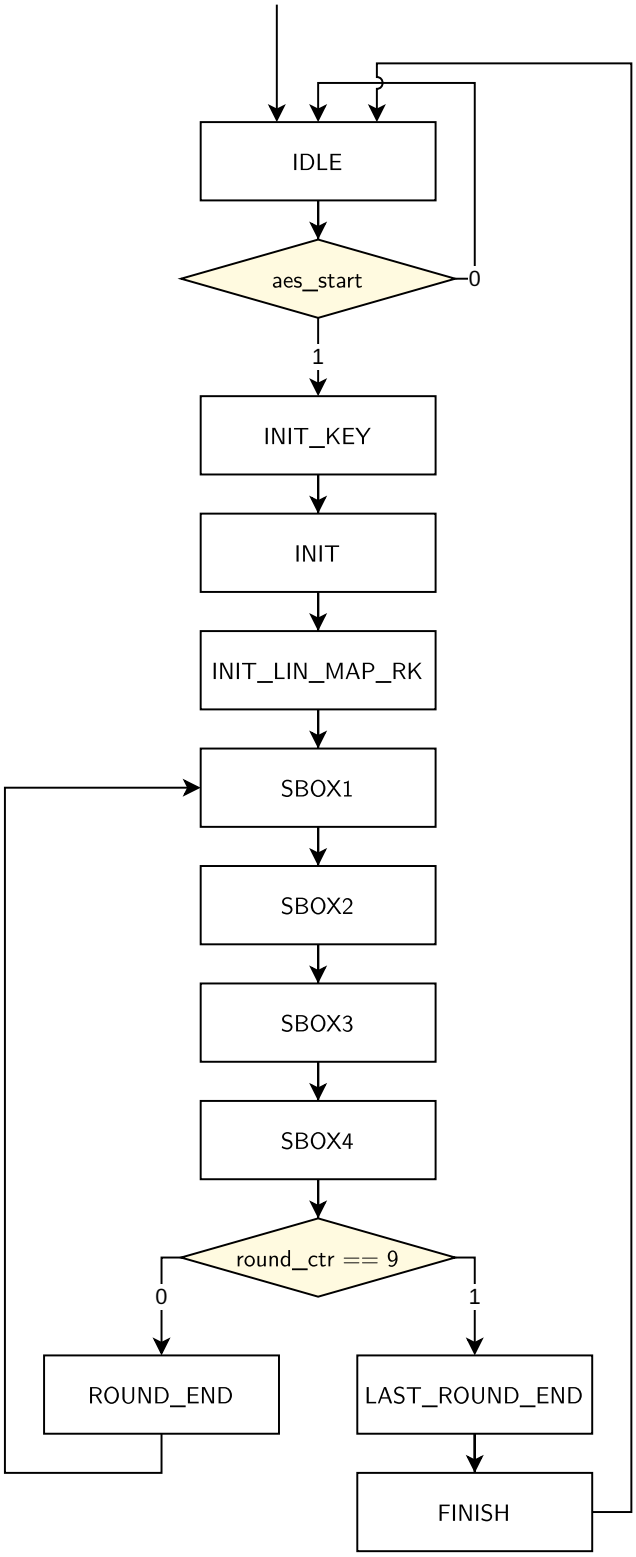


Figure 3: State machine controlling the AES encryption. The encryption starts if the input signal `aes_start` is high. The output signal, `aes_busy`, is high whenever the encryption is not in state `IDLE`. After some initialization and a dedicated state to let the key schedule start a cycle earlier (cf. our paper), every round consists of four S-box states and `ROUND_END` to compute the linear operaitons.

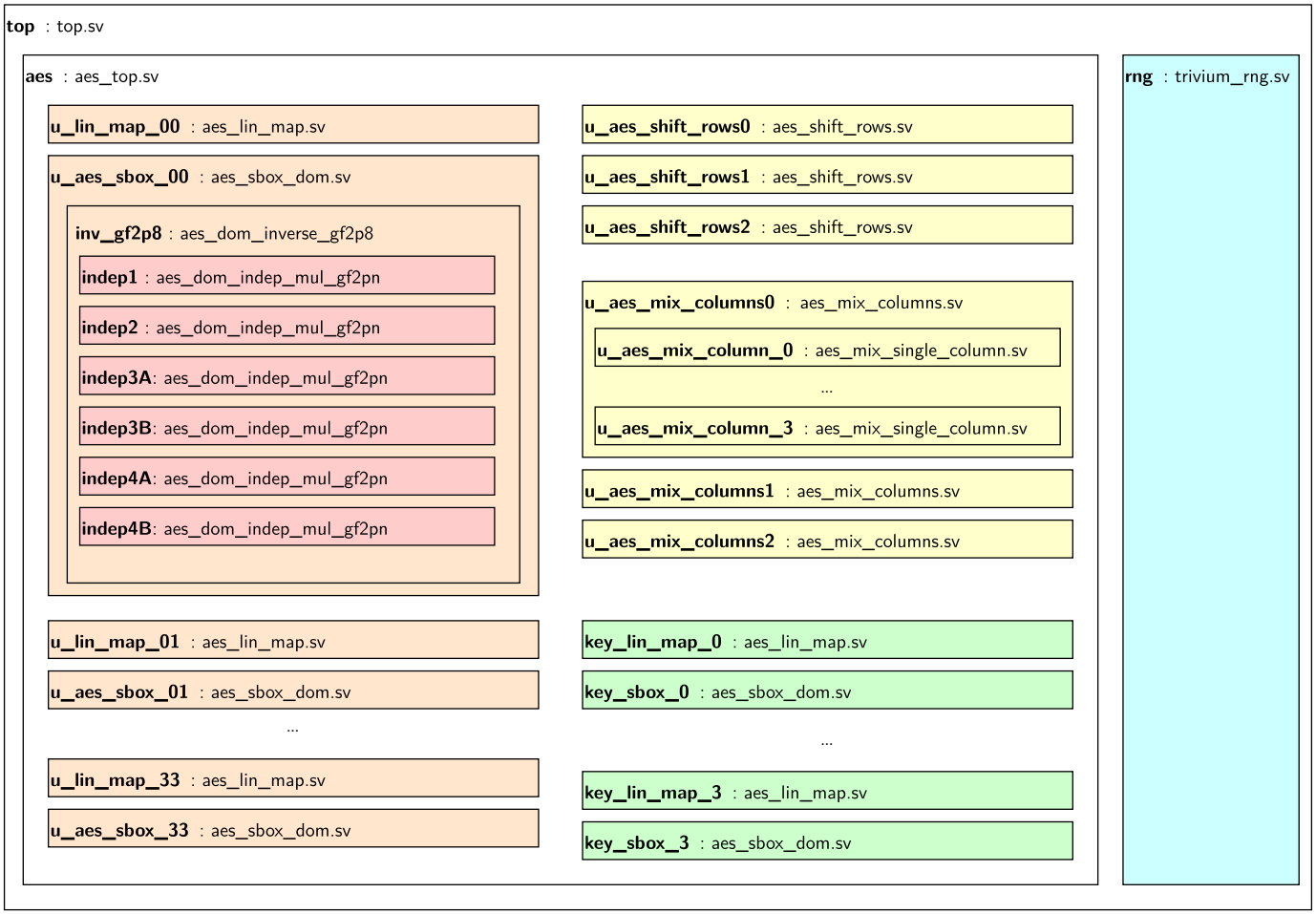


Figure 4: Organization of Verilog modules to build our optimized AES architecture. The top module `top` contains the AES core (`aes`), and the Trivium RNG (`trivium`, in blue). The AES core instantiates 16 S-box modules (`u_aes_sbox_`ij for $0 \leq i, j \leq 3$) and respective linear maps (`u_lin_map_`ij for $0 \leq i, j \leq 3$), sketched in orange. Every single S-box module instantiates 6 DOM-*indep* multipliers (`indep1`, `indep2`, `indep3A`, `indep3B`, `indep4A`, `indep4B` in red). Additionally, the AES core instantiates 4 S-box modules and linear maps for the key schedule (`key_lin_map_`i, `key_sbox_`i for $0 \leq i \leq 3$, shown in green). The linear parts of the AES cipher, consisting of `u_aes_shift_rows`i and `u_aes_mix_columns`i are instantiated once per share and marked in yellow.

3.4 Description of ports

The top module of the AES design can be found in `top.sv` in the respective subdirectory. It connects an RNG to the AES encryption core, which is second-order masked. The AES computes $y = \text{enc}_k(x)$ using three shares of the plaintext x s.t. $x = x_0 \oplus x_1 \oplus x_2$ and three shares of the key k s.t. $k = k_0 \oplus k_1 \oplus k_2$, and outputs the three shares of the ciphertext y s.t. $y = y_0 \oplus y_1 \oplus y_2$. In Table 2 we give a preciser description of all input ports. In Figure 5 we give an example of the timing of input and output signals.

Port name	Direction	Width	Description
`clk_i`	input	1	Primary clock
`rst_ni`	input	1	Reset signal, active = 0, inactive = 1
`start_i`	input	1	Start signal for AES encryption, must be high for at least one cycle to start AES encryption
`trivium_reseed_i`	input	1	Start signal for reseeding the RNG with the given key and IV, must be high for at least one cycle to start reseeding
`busy_o`	output	1	High if either AES encryption or Trivium reseeding is in progress (i.e., if the current state is anything but `OUTER_IDLE`)
`aes_plain_i`	input	3 x 128 bit	Shares of the plaintextt (x_0, x_1, x_2)
`aes_key_i`	input	3 x 128 bit	Shares of the key (k_0, k_1, k_2)
`aes_ct_o`	output	3 x 128 bit	Shares of the ciphertext (y_0, y_1, y_2)
`trivium_key_i`	input	80	Key used by Trivium RNG
`trivium_iv_i`	input	80	IV used by Trivium RNG

Table 2: Description of ports for AES designs

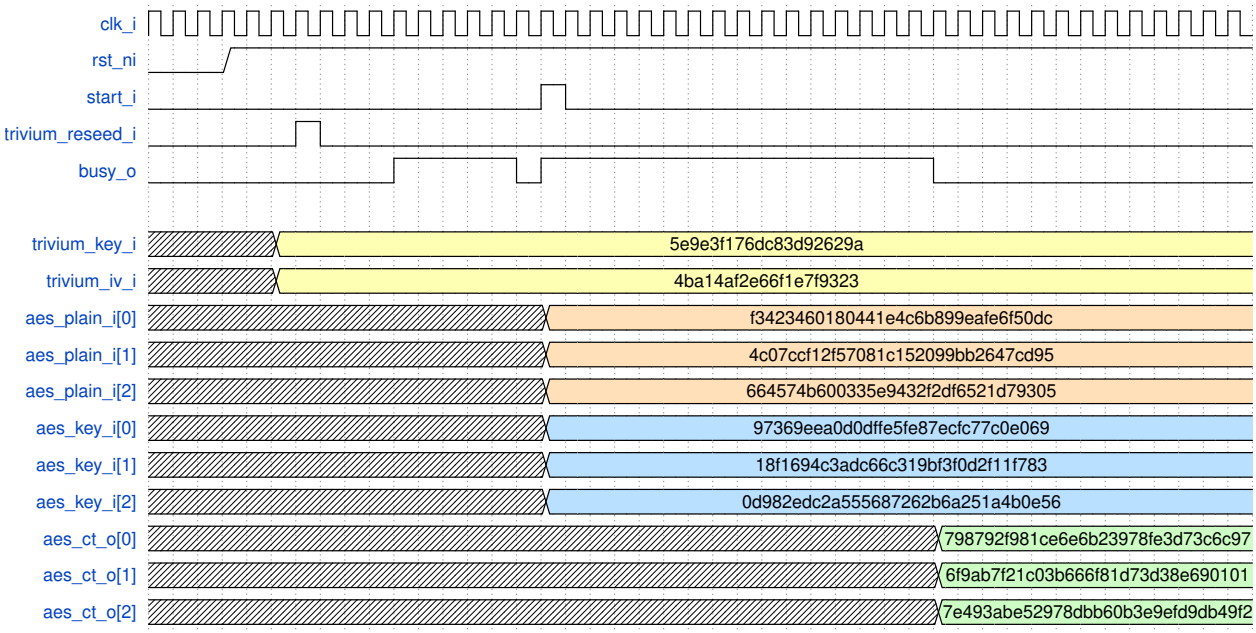


Figure 5: Timing of signals in the `opt` AES implementation. Note that in reality, the `busy_o` signal is higher for more cycles than depicted here (it was shortened to fit easier into the picture).

4 Formal verification

As described in Section 6.2 in the paper, we formally verify our design using Coco [Gig+21] . All the necessary files can be found in `formal_verification`.

4.1 Directory Structure

- `formal_verification/coco`: Source code of the modified Coco verification tool used in the paper
- `formal_verification/*.fish`: Verification scripts
- `formal_verification/*.ys`, `formal_verification/*.cpp`, `formal_verification/rtl/`: additional files required for formal verification with Coco
- `formal_verification/build`: directory used to store build artifacts such as the Verilator model, the VCD trace file and the generated CNF formulas

4.2 Prerequisites

In order to perform formal verification for any of the following scenarios, the necessary software needs to be installed, which includes:

- fish shell (version 3.3.1). On Ubuntu, it should suffice to install it with:

  ```
  sudo apt install fish
  ```

  Running `which fish` afterwards should return `/usr/bin/fish`.
- Yosys (0.32+51 (git sha1 6405bbab1, gcc 11.4.0-1ubuntu1 22.04 -fPIC -Os)). Build it from source by cloning the github repository. Follow the build instructions there. Use commit 6405bbab1.
- Verilator (5.014 2023-08-06 rev v5.014-35-ge6b0bdd4d). Build it from source by cloning the github repository. Follow the build instructions there. Use commit e6b0bdd4d. Additional information can be found in Section 1.
- gcc/g++ 11.4.0
- Kissat 3.1.1. Build it from source by cloning the github repository. Follow the build instructions sthere. Use commit 71caafb.
- Python 3.10.12
- Python packages (can simply be installed with pip):
  - networkx 3.1
  - pickle 4.0
  - dataclasses
  - argparse 1.1
  - json 2.0.9
  - logging 0.5.1.2

- gmpy 1.17
- pysat 0.1.8.dev9

4.3 Formally verifying the fixed DOM-*dep* multiplier

In Section 6.2 in the paper, we describe that we formally verify with Coco that our proposed fix for the second-order DOM-*dep* multiplier is secure. The source code of the fixed multiplier is given in `sboxes/rtl/sbox_fixed/aes_dom_dep_mul_gf2pn.sv`.

To reproduce this, the following steps are necessary:

1. Navigate to the correct directory:

```
cd $ARTIFACT_HOME/formal_verification
```

2. Set the environment variables `$YOSYS_HOME_DIR` and `$VERILATOR_HOME_DIR` such that they point to the installation directories. For example:

```
export YOSYS_HOME_DIR=$HOME/yosys/
export VERILATOR_HOME_DIR=$HOME/verilator
```

3. *Parse* the design. This verification step transforms the Verilog design into a netlist using Yosys, and stores the netlist in graph format such that Coco can handle it. Note that in order to correctly represent the dependence of inputs for the multiplier, we build a wrapper file around the DOM-*dep* multiplier which effectively sets the input shares of A equal to the input shares of B.

```
./verify_dom_dep.fish parse
```

4. *Trace* the design. With Verilator, we now simulate the obtained netlist for a few cycle without input data, to get values for the control signals which are used during the verification.

```
./verify_dom_dep.fish trace
```

5. *Verify* the design. We now label the circuit inputs as shares, fresh randomness or unimportant (default) to indicate their meaning during the verification. Then, we start Coco to build the SAT formula, and check the formula with CaDiCal.

```
./verify_dom_dep.fish verify
```

   Verification takes a few seconds. The output should be:

```
---> Success. Design is secure.
---> Verify finished successfully.
```

6. As a sanity check, try to label the inputs `q0` and `q1` as unimportant (i.e. comment out lines 103 and 104 in `verify_dom_dep.fish`). The output should be:

```
---> Error. Design is NOT secure!
Leak: build/dbg-label-trace-0.txt
---> Verify finished successfully.
```

4.4 Formally verifying the S-box

In Section 6.2 in the paper, we describe that we formally verify with Coco that our optimized S-box design is secure. We take the source code of the S-box from `sboxes/rtl/sbox_opt`. Note that the complete verification takes at least about 1.5-2 days.

The following steps are necessary:

1. Navigate to the correct directory:

```
cd $ARTIFACT_HOME/formal_verification
```

2. Set the environment variables `$YOSYS_HOME_DIR` and `$VERILATOR_HOME_DIR` such that it points to the installation directories. For example:

```
export YOSYS_HOME_DIR=$HOME/yosys/
export VERILATOR_HOME_DIR=$HOME/verilator
```

3. *Parse* the design. For more details on this see Section 4.3. This time we do not need a wrapper top module.

```
./verify_sbox_opt.fish parse
```

4. *Trace* the design. For more details on this see Section 4.3.

```
./verify_sbox_opt.fish trace
```

5. *Verify* the design. For more details on this see Section 4.3. To verify the complete S-box, the verification needs to run for 11 cycles (because the design is idle the first few cycles). Hence, to verify everything until the data reaches the pipeline registers in stage 4 (cf. Figure 1 in the paper) it needs 10 cycles, everything until stage 3 needs 9 cycles, everything until stage 2 needs 8 cycles, and only the first stage needs 7 cycles. Building the SAT formulas is very fast, even for 11 cycles. The main bottleneck is solving the SAT formulas, which is why we rely on Kissat, a more efficient solver. Running the following command will print the SAT formulas in DIMACS CNF format and store them to `formal_verification/build/formulas`:

```
./verify_sbox_opt.fish verify
```

   Per default it also starts solving with CaDiCal, which can however simply be cancelled.

6. To solve the SAT formulas with Kissat, first set the environment variable `$KISSAT_HOME_DIR` to the installation directory, e.g.:

```
export KISSAT_HOME_DIR=$HOME/kissat
```

7. Start Kissat with the following command:

```
cat build/formulas/formulasecret0_.cnf build/formulas/allsame.cnf
 | $KISSAT_HOME_DIR/build/kissat --unsat
```

   There exists one formula per secret bit. To get a full security proof, all formulas need to be solved. A secret bit is not leaking if the SAT formula is UNSAT. In this case, the output of Kissat contains:

```
c ---- [ result ] ------------------------------------------------------------
c
s UNSATISFIABLE
c
```

8. *(Optional)* As already mentioned before, solving for 11 cycles will take at least about 1.5-2 days. Alternatively, S-box can be checked for less cycles, i.e., verifying only the until the first, second, third, ... etc pipeline stage. To do so, modify line 115 in `verify_sbox_opt.fish` such that the SAT formulas are built for less cycles. Solving these SAT formulas is then much faster, and it gives confidence for the first few pipeline stages, although of course not the complete S-box is verified.

9. *(Optional)* As a sanity check, try to label label the fresh randomness `prd_i` as unimportant (i.e. comment out line 103). Even for 11 cycles and solving with CaDiCal it quickly (2-3 seconds) points out leakage.

5 Experimental verification

As described in Section 6.3 in the paper, we experimentally verify our design on an FPGA board. All the necessary files can be found in `experimental_verification`.

5.1 Directory Structure

- `experimental_verification/cotg_vivado`: Vivado project including constraint files, a Verilog wrapper to communicate with the CW microcontroller and our `opt` AES architecture.
- `experimental_verification/Cw305_AES_cotg.py`: Script to obtain power traces
- `experimental_verification/requirements.txt`: Python packages which need to be installed to run the measurement script
- `experimental_verification/cw305_opt.bit`: Bitstream file containing our optimized AES implementation
- `experimental_verification/cw305_defines.v`: Used by measurement script
- `experimental_verification/data`: Used to store traces and t-test results
  - `experimental_verification/data/traces_11282023_160108.npy`: Example set of power traces when measuring our `opt` design.
  - `experimental_verification/data/traces_11282023_160108.npy`: Example trigger signal.

5.2 Evaluation devices

Our evaluation setup consists of the following devices:

- The FPGA evaluation board: NAE-CW305-04-7A100-0.10-X
- An external power supply: R&S HM7042-5
- An oscilloscope: PicoScope 6404C
- A PC running Ubuntu 22.04.2 LTS
- An SMA cable for recording the power trace
- A cable to connect to the external power supply
- A probe for synchronizing the clock signal: TA133 (10:1, 500 MHz)
- A probe for recording the Trigger: TA375 (1:1, 500 MHz)
- An USB-A cable for connecting the board to the PC

5.3 Evaluation setup

We show the finished setup in Figure 6a, Figure 6b and Figure 6c. In order to reproduce this, follow these steps:

1. Connect X4 with the SMA cable to Channel A of the PicoScope (see 1 in Figure 6a and Figure 6b).
2. Connect TP1 with the 1:1 probe to Channel B of the PicoScope (see 2 in Figure 6a and Figure 6b).
3. Connect P4 with the 10:1 probe to the AUX in connector of the Picoscope (see 3 in Figure 6a and Figure 6c).
4. Connect the board to the exernal power supply (see 4 in Figure 6a and Figure 6b). The external supply should be set to output 1 V. The switch located right below the positive banana jack input must be set to "external".
5. Connect the board to the PC via USB (see 5 in Figure 6a).

5.4 Recording power traces

In order to record a power trace, the following steps are necessary:

1. Install the necessary software:
   - ChipWhisperer framework. Follow the build instructions here. Use commit a1fa53bf59 from the github repository.
   - PicoScope 7 Software and Drivers. Follow the instructions here. This should include the picosdk-python-wrappers.
   - Further required Python libraries and their respective versions are listed in `requirements.txt`. This includes SCALib 0.5.5, numpy 1.24.3, and pycryptodome 3.18.0. They can be installed using the command:

     ```
     pip install -r requirements. txt
     ```

   - Vivado 2021.1
2. Navigate to the correct directory:

   ```
   cd $ARTIFACT_HOME/experimental_verification
   ```

3. Start Vivado and import project:

   ```
   vivado -source project.tcl
   ```

   The project should be created without errors. It includes all constraints file, source files and a wrapper to communicate with the CW microcontroller.
4. Generate the bitstream by pressing the "Generate bitstream" button. Alternatively, we provide a pre-generated bitstream file (`cw305_opt.bit`).
5. Adapt the measurement script `Cw305_AES_cotg.py`.
   - Fill in the correct name/path of the bitstream file in line 38.

- Recording is done by capturing N traces per block. The number of blocks is M. Both parameters can be set in lines 565 and 566.
- *(Optional)* The design operates on a clock frequency of 1.5625 MHz. If you want to run the measurement on another frequency, adapt the `FpgaClkFreq` in line 67.
- *(Optional)* If you want to use other channels of the PicoScope, or do not want to use the external clock, do the modifications in lines 369-379.
- *(Optional)* The PicoScope sampling rate is set to 6.25 Ms/s (timebase 3 = sample interval 1.6 ns). It can be changed in line 396.
- *(Optional)* One trace consists of 22600 samples. This can be changed in lines 573 and 575.

6. Run the measurement script:

```
python3 Cw305_AES_cotg.py --univ-ttest
```

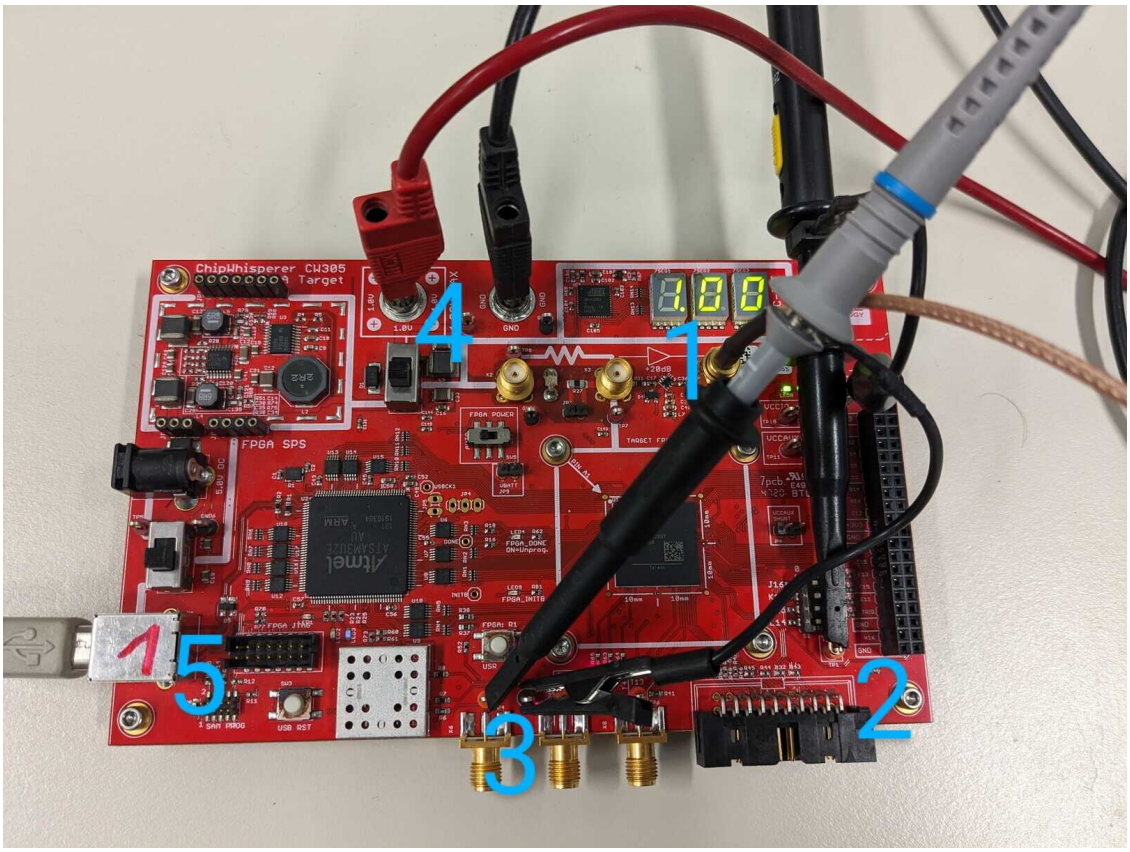
The script first programs the FPGA, performs some functionality tests (i.e. sending a random plaintext and key to the FPGA, reading back the encryption result and comparing it with an AES implementation in Python), configures the PicoScope and then starts the measurements. N traces are recorded, and the t-test results are updated. After `plot_delta` have been recorded, the intermediate t-test results are stored in `data/`. The traces are not stored. To store the traces and trigger signal, run:

```
python3 Cw305_AES_cotg.py --univ-ttest --store-traces --with-trigger
```

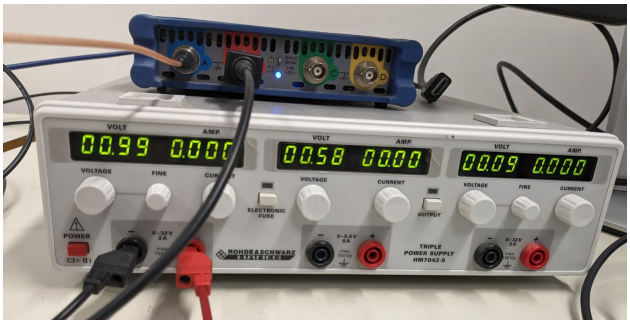
7. Plot a sample power trace and trigger signal:

```
python3 plot_traces.py --traces data/traces_.xxx.npy
    --trigger data/trigger_.xxx.npy
```

We provide an example trace and trigger signal in the `data/` directory.



(a) FPGA evaluation board



(b) Oscilloscope and external power supply

(c) Oscilloscope from the back

Figure 6: Measurement setup

6 Generation of area numbers

For synthesizing our designs and obtaining area numbers, we used a proprietary toolchain which is not freely available. This toolchain consisted of:

- Cadence Genus Synthesis Solution 19.11-s087_1
- Standard cell library: UMK65LSCLLMVBBR_B (based on UMC's 65nm Low-K Low Leakage process)
- To compute the area in kGE, we use the area of one NAND gate with two inputs (ND2), which is 1.44 μm^2.

Using this toolchain to synthesize the designs given in Section 2 and Section 3 will give the area numbers as shown in Table 1, Table 3a and Table 3c in the paper.

6.1 Alternative open-source synthesis flow

We provide an alternative synthesis flow based on Yosys and the NanGate Open Cell Library. Although it does not allow to reproduce the exact area numbers from the paper, the results are similar. In the directory `area` we provide Yosys synthesis scripts. In order to obtain area numbers for a specific design, the following steps are necessary:

- Download/install the prerequisites:
  - Yosys (0.32+51 (git sha1 6405bbab1, gcc 11.4.0-1ubuntu1 22.04 -fPIC -Os)). Build it from source by cloning the github repository. Follow the build instructions there. Use commit 6405bbab1.
  - Download the NanGate OCL liberty file here. This is revision 1.0 (Thu 10 Feb 2011, 18:11:20). Copy it to the `area` directory.
  - *(Optional - only if you want to obtain area numbers for the complete AES design)* sv2v (v0.0.11-17-g764a11a). Build it from source by cloning the github repository. Follow the build instructions there. Use commit 764a11af7f86.
- Navigate to the correct directory:

```
cd $ARTIFACT_HOME/area
```

- Set the environment variable `$YOSYS_HOME_DIR` such that it points to the Yosyxs installation directory. For example, if you cloned the Yosys repository in `$HOME`, it should be set to `$HOME/yosys`, e.g.:

```
export YOSYS_HOME_DIR=$HOME/yosys
```

- If you want to obtain area numbers for the complete AES design, we need to first convert our SystemVerilog AES design to Verilog because Yosys only has limited SystemVerilog support. For that, set the environment variable `$SV2V_HOME_DIR`. For example, if you cloned the sv2v repository in `$HOME`, it should be set to `$HOME/sv2v/bin`, e.g.:

```
export SV2V_HOME_DIR=$HOME/sv2v
```

- Start the synthesis process with using the Makefile. For example, to synthesize our optimized S-box design, run:

```
make area_sbox_opt
```

There are multiple targets available, including:

- `area_sbox_opt`, `area_sbox_fixed`: building S-boxes from Table 1
- `area_aes_opt`, `area_aes_opt_nocotg`, `area_aes_fixed`: building AES architectures from Table 3b, converts it to Verilog first

- Observe the output of the command, which somewhere in the last lines says:

```
Chip area for module '\aes_sbox_dom': 3471.034000
```

This is the area given in μm^2.

- Convert the computed area to kGE. The area of one `NAND2_X1` gate is $0.798\,\mu\mathrm{m}^2$. For example, the area of the `sbox_opt` design would then be approximately 4.3 kGE.

References

[Gig+21] Barbara Gigerl et al. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 2016, p. 3.