Research Internship
Computing Science - Cyber Security

Radboud University

Guidelines and takeaways for setting up an Ibex Core on the Arty A7 FPGA board

Author:
Stefan Popa
1027672

First supervisor/assessor:
Asst. Prof. Ileana Buhan
ileana.buhan@ru.nl

Second assessor:
Konstantina Miteloudi
kmiteloudi@cs.ru.nl

April 12, 2023

Abstract

Microcontrollers are used in many of the devices that we make use of on a daily basis. An important part in the design of such microcontrollers is to ensure they are secure, both at the hardware and software level. One key step is to detect which hardware modules are leaking information. There are two main methods for leak detection, **Test Vector Leakage Assessment** or TVLA, and **Side-channel Vulnerability Factor** or SVF. It has already been shown that not only these two methods give different results, but the choice of encryption algorithm also influences which hardware modules leak information. To this extent, originally we wanted to build upon previous work in this area and further explore the reasons for these differences. However, due to various issues that we have encountered, we decided on creating an up-to-date, step-by-step guideline for setting up the tools required for such an analysis to be performed.

Contents

Chapter 1

Introduction

Most of the cryptographic algorithms that are being used in practice are protected against logical vulnerabilities, as they go through various stages of testing before they are being standardized. However, the same algorithms can be vulnerable to implementation attacks. Running a cryptographic implementation on a device, which is also called a System on Chip (SoC), can leak information via physical channels. These include timing information, power consumption or electromagnetic radiation [18]. Exploiting such information is called a side-channel attack.

Various countermeasures can be used to protect against such attacks, such as masking. These countermeasures are not only error-prone and expensive, but also algorithm, and sometimes platform, specific. Moreover, a countermeasure for a certain attack could make other type of attacks easier. To this extent, Arsath et al. [2] suggest that in order to secure the execution of encryption algorithms against side-channel attacks, one can harden the platform on which these algorithms will run. In this way, we secure the platform itself and thus any algorithm running on this platform will implicitly be secure as well.

In order for such an approach to work, a *leakage detection method* is necessary. The most common leakage detection method is the Test Vector Leakage Assessment framework, or TVLA. However, Arsath et al. [2] make use of a different method called the Side-Channel Vulnerability Factor, or SVF. It is however been shown that the results can differ between the two methods, not only in the amount of leakage that is being detected, but also in identifying which modules cause said leakage [1, 19], however no concrete reasons for these differences have been found.

Our original concept was to continue the work that has been done in this area and look into possible explanations for why the above differences between

TVLA and SVF occur. However, the setup required for such an analysis turned out to be more difficult than expected, mainly due to lack of documentation for the required tools. Therefore, our scope changed to create a detailed guideline on how to create a proper setup for this experiment.

To this extent, in Section 2 we offer step-by-step instructions on how the process of finding the proper tools works as well as how do we install and use these tools. Next, in Section 3 we offer a short discussion about the issues we have encountered that caused us to reduce the scope of this project, as well as a few takeaways.

Chapter 2

Setup Guidelines

2.1 FPGA Board

The first step into creating a setup for analyzing side channel attacks is to decide which FPGA board to use for the experiments. In this case, the two boards at our disposal are a SAKURA-G board and an Arty A7 35T board.

2.1.1 SAKURA-G

The SAKURA-G FPGA board is "designed for research and development on hardware security, such as Side-Channel Attacks (SCA), Fault Injection Attacks (FIA), Physical Unclonable Functions (PUF), and dynamic reconfiguration" [7].

The main advantage of working with this board is that it has been used for various projects within the SCA lab team, and thus potential issues have already been encountered and documented.
However, due to the core that we are going to use (Section 2.2), working with this board is made slightly harder. Newer Xilinx tools do not support the FPGA chips used by SAKURA-G, and thus to set it up, Xilinx ISE 14.7 WebPack is necessary. ISE is a discontinued software tool from Xilinx for synthesis and analysis of HDL designs. Moreover, the core we have chosen is written is System Verilog, which ISE does not support. This means we would need to find other workarounds or convert the System Verilog code to Verilog code.

2.1.2 Arty A7 35T

The Arty A7 35T is a development platform making use of the Artix-7 Field Programmable Gate Array (FPGA). Moreover, "Arty is not bound to a single set of processing peripherals: One moment it is a communication powerhouse of UARTs, SPIs, IICs, and an Ethernet MAC, and the next it is a meticulous timekeeper with a dozen 32-bit timers" [20].

The main advantage of the Arty A7 is that it is a very common board used in FPGA programming, and thus a lot of examples are available, including for our chosen core (Section 2.2). Moreover, the board is better supported by the necessary tools needed for this project.
A disadvantage of using the Arty A7 is that it has not been used extensively within the SCA lab, and that any traces taken with it will need to be aligned. Thus, while the setup is easier, using it may result in unknown issues.

In the end, due to the extra steps required to set up a SAKURA-G board with our core, we decided that the Arty A7 35T is the better, easier choice.

2.2 Core

Next, we had to decide on a core. Of course, this core had to implement the RISC-V specification, as well as be small enough to fit on a Arty A7 35T board. We have used a comprehensive list of RISC-V cores [15] and in the end, the choice had to be done between the NEORV32 Core and the Ibex Core. Amongst others, these cores had enough documentation about their implementation and have been tested on relatively new boards and software.

We decided the Ibex core would be the better fit, as it has more contributors, more support in case of issues and because multiple previous projects have used it successfully [8]. For our setup we have also used the very useful example of a demo system using the Ibex core [9].

2.3 Tools & Prerequisites

There are various tools and prerequisites necessary for creating this setup. These will be listed and briefly described below:

- **Ubuntu (v.22.04)**
- **Python (v.3.5 or greater)**
- **Xilinx Vivado (v.2022.1):** Xilinx Vivado is a design suite created by Xilinx for the purposes of synthesis and analysis of Hardware Description Language (HDL). It is the successor of Xilinx ISE Design Suite and includes more features, as well as more support for newer devices.
- **Xilinx Vivado - Linux Cable Drivers:** In order for the Xilinx Vivado software to be able to communicate with the board(s) we are

connecting to the computer, certain drivers are necessary. On Windows machines, there is an optional selection during the installation process. On Linux systems, root or sudo access is necessary to install these drivers, and thus the option has been removed from the installer. There is now a script that has to be run manually with root access after the installation has completed.

- **RV32IMC GCC Toolchain (v.20220524-1):** The programs we will put on the FPGA board are written in C, and thus we need a way to convert the C code into assembly code which will then again be converted to machine code. For this, we need a toolchain that does this job for us, which is the RISC-V C and C++ cross-compiler.
- **Yosys (v.0.9, Synthesis - Method 1):** Yosys is a framework for RTL synthesis tools. It has vast Verilog support and provides a basic set of synthesis algorithms for various applications.
- **OpenSTA (v.2.2, Synthesis - Method 1):** This is a gate level static timing verifier. It can be used to verify the timing of a design. A TCL command interpreter is used to read a design, specify the timing constraints and output the timing reports.
- **sv2v (Synthesis - Method 1):** This is an open-source tool which transforms System Verilog code into Verilog code. Such a conversion is necessary for Yosys, which does not support System Verilog code.
- **Nangate45 (Synthesis - Method 1):** The Nangate Open Cell Library is a generic open-source, standard-cell library provided for the purposes of research, testing, and exploring EDA flows.
- **srecord (v.1.65, Synthesis - Method 2):** The SRecord package is a collection of powerful tools for manipulating EPROM load files, and is written in C++.
- **fusesoc (v.2.0, Synthesis - Method 2):** FuseSoC is a package manager and a set of build tools for HDL code.
- **cmake (v.3.22.1, Synthesis - Method 2):** CMake is an open-source tool created for easier build automation, packaging and installation of other software.
- **openocd (v.0.11.0, Synthesis - Method 2):** The Open On-Chip Debugger allows for debugging and programming of embedded devices.
- **screen (v.4.09, Synthesis - Method 2):** GNU Screen is a terminal multiplexer which allows the user to create a session and then open virtual terminals inside that session. This tool will be used to observe the output of the Arty A7 35T board.

2.4 Setup General Tools

2.4.1 Xilinx Vivado

1. Download the `Xilinx Unified Installer` for the correct operating system [21]. For the purposes of this project we have used version `Vivado ML Edition - 2022.1`.

2. In order to download the installer, we will need to create an account. We can use our institutional account for this.

3. The installer will be downloaded in the `Downloads` folder. However, before we proceed to start the installation process, we need the following packages, otherwise the installer will get stuck.

   ```
   sudo apt install libtinfo5
   sudo apt install libncurses5
   sudo apt install libtinfo-dev
   sudo apt install python3-pip
   sudo apt install openjdk-11-jre-headless
   ```

4. Right-click on the installer and select `Properties`. Select the `Permissions` tab and we ensure that `Allow executing file as program` is checked. Afterwards, we open the installer by using the following command:

   ```
   sudo ./Xilinx_Unified_2022.1_0420_0327_Lin64.bin
   ```

5. The installer will open and we press `Next`. We will now need to add our login details to proceed.

6. Press `Next`, and after selecting `Vivado` for the preferred version, press `Next` again.

7. Now, select `Vivado ML Standard` and press `Next`.

8. At this screen, leave all settings at their default options.
   Please Note: For Windows Systems, we can also install the required cable drivers at this stage. For Linux systems, please refer to 2.4.2.

9. Select `I Agree` for all four boxes and press `Next`.

10. We can now select the location on our system where we want `Xilinx Vivado` to be installed. We can leave the other settings at their default options. Press `Next` and then press `Install`.

11. The installation might take some time, so ensure that a stable internet connection is available. The overall installation process also takes around 160 GB of space so we need to make sure enough storage is available on our system. After the installation is complete, we can ignore the version updates notifications and any possible warnings.

12. We will now need to activate our licence for the product. We can do so by first opening `Xilinx Vivado`.
    Please Note: On Linux Systems, the following two commands are necessary:

    ```
    source /tools/Xilinx/Vivado/2022.1/settings64.sh
    vivado
    ```

    It also might be required to execute these commands every time the system is restarted.

13. After the software opens, select `Help -> Manage Licence` from the toolbar. Now, select `Obtain License`, leave the default setting and select `Connect Now`.

14. We will again have to use our credentials for this. After we log in, we leave all the default settings and press `Next`. Now, we select `Vivado Design Suite HL, WebPACP 2015 and Earlier License` and press `Generate Node-Locked License`.

15. A pop-up will appear. Press `Next` twice. The license should have been generated and can now be downloaded (or sent to the email address we have used for creating the account). It will be saved as `Xilinx.lic`.

16. We open `Xilinx Vivado` again, select `Help -> Manage Licence`, and then select `Load License`. Browse for the file we have downloaded in the previous step. The product should now be activated.

2.4.2 Xilinx Vivado - Linux Cable Drivers

1. To install the Cable Drivers on a Linux system, we need the following commands:

    ```
    cd ${vivado_install_dir}/data/xicom/cable_drivers/
    lin64/install_script/install_drivers/

    sudo ./install_drivers
    ```

2. Follow the steps in the installer and everything should be successfully installed. The cable drivers will now allow for the FPGA board to be recognized and communicate with our system.

3. More information can be found on the Xilinx Docs [22].

2.4.3 RV32IMC GCC Toolchain

1. We first need to download the `RV32IMC GCC Toolchain` to our machine [11]. We need to scroll down to version `20220524-1` and donwload `lowrisc-toolchain-gcc-rv32imcb-20220524-1.tar.xz`

2. After we extract the archive, we can move the folder and its contents to the desktop. As the this toolchain is pre-built, we can use it as is.

2.5 Synthesis

The next step into a setup is to figure out whether the core is small enough to fit on the chosen board. To that extent, we had two methods to choose from. The first method makes use of the the open-source synthesis flow offered by Ibex core itself. The second method makes use of the example demo system included in the Ibex Demo System repository, which also initializes the memory contents.

2.5.1 Synthesis Prerequisites

First, we need to install some prerequisites. These can differ per operating system. Since we are using `Ubuntu 22.04`, we have to use the following:

```
sudo apt-get install autoconf automake autotools-dev \
curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk \
build-essential bison flex texinfo gperf libtool \
patchutils bc zlib1g-dev libexpat-dev ninja-build
```

2.5.2 Synthesis - Method 1

sv2v

1. We first need to clone the `sv2v` repository to our machine [16]. This can be done with:

   ```
   git clone https://github.com/zachjs/sv2v.git
   ```

2. Next, we need to install the `Haskell Stack` [17] before we can compile the `sv2v` library. This can be done with:

   ```
   sudo curl -sSL https://get.haskellstack.org/ | sh
   ```

3. We can now compile the `sv2v` library using:

   ```
   cd sv2v
   sudo make
   ```

4. Finally, we need to add the correct path to `PATH` variable. **Please Note:** It might be necessary to repeat this step each time the system is restarted:

```
export PATH=/home/spopa/.local/bin:$PATH
```

Yosys

1. We first need to clone the `Yosys` repository to our machine [23]. This can be done with:

```
git clone https://github.com/YosysHQ/yosys.git
```

2. Now we need to install some prerequisites. These can differ per operating system. Since we are using `Ubuntu 22.04`, we have to use the following:

```
sudo apt-get install build-essential clang bison flex \
libreadline-dev gawk tcl-dev libffi-dev git \
graphviz xdot pkg-config python3 libboost-system-dev \
libboost-python-dev libboost-filesystem-dev zlib1g-dev
```

   For figuring out the prerequisites for other systems, please refer to the GitHub repository.

3. To build Yosys simply type `make` in this directory.

```
sudo make
sudo make install
```

OpenSTA

1. We first need to clone the `OpenSTA` repository to our machine [14]. This can be done with:

```
git clone https://github.com/The-OpenROAD-Project/OpenSTA.git
```

2. Before we can compile the program, we need to install `cmake`. This can be done using the following command:

```
sudo apt install cmake
```

3. Next, to build the OpenSTA library and excutable we use the following commands:

```
cd OpenSTA
mkdir build
cd build
cmake ..
make
```

Nangate45 Standard Cell Library

1. The `Nangate45 Standard Cell Library` can be found on the GitHub repository [13].
2. The correct file can be found under
   `lib -> NangateOpenCellLibrary_typical`
3. The file can be downloaded anywhere on the system, as its absolute path will be used for the synthesis process.

Execution

1. In the cloned Ibex GitHub repository, we go to `syn`.
2. The synthesis flow is configured via environment variables. The shell script file is used to set the environment variables for the flow and any changes made should be placed there. To that extent, an example `syn_setup.example.sh` is included, which needs to be renamed `syn_setup.sh`. The values in it also need to be set appropriately for the flow to work.
3. The enivronment variable that we need to modify are as follows:
   - `LR_SYNTH_CELL_LIBRARY_PATH`: The absolute path to the Nangate45 standard cell library.
   - `LR_SYNTH_CELL_LIBRARY_NAME`: The name of the standard cell library. Currently `nangate` is the only supported value.

   These values need to be uncommented in the shell script.
4. On our system, we also had to make slight modifications in the shell script `syn_yosys.sh`. Whenever `sv2v` is used, the absolute path to where the library was installed needs to be used, otherwise the command will not be recognized.
5. Once these changes are completed, we can run the synthesis flow by calling `syn_yosys.sh`. This can be done using the following command from the `syn` folder:

   ```
   ./syn_yosys.sh
   ```
6. All outputs are placed under the `syn/syn_out` directory with the prefix `ibex_` with the current date/time forming the rest of the name. The synthesis report can be found under `syn/syn_out/ibex_date/reports`, in the file called `area.rpt`. More information about the different output files given by this synthesis flow can be found on the repository [10].

Unfortunately, the synthesis report was very high-level about the resources being used. Therefore, the next step was to use `Xilinx Vivado` for the synthesis step.

2.5.3 Synthesis - Method 2

We can make use of the in-built synthesis tool used by `Xilinx Vivado`, as the report that is being generated is much more detailed about the resources that are being used by the Ibex core. We can set up such a synthesis in two ways: **manually**, which does not initialize the memory contents, or **automatically**, using the scripts offered by the Ibex Demo System, which will also generate the memory contents.

Manual Setup - Memory Uninitialized

1. Open `Xilinx Vivado`, as explained in Section 2.4.1.
2. Select `Create Project`, and a pop-up will appear. Press `Next`.
3. We can now type the name of our project, and leave the other options on their default settings. Press `Next`.
4. Select `RTL Project` and `Do not specify sources`, and press `Next`.
5. The `Constraints` list will now appear. In our case, we do not need any, and thus we can simply press `Next` again.
6. Now we will have to select the part `XC7A35TICSG324-1L`, as this is the FPGA chip used in our board. Now we press `Next`, and then `Finish`.
7. The project will now be created and opened in `Xilinx Vivado`. In the `Sources` panel, we will be able to see all the source files we will be adding.
8. For ease of use, we have generated a TCL file that `Xilinx Vivado` can use to easily add all the necessary files to the project. This file can be used by pressing `Tools -> Run TCL Script` from the top toolbar.
   We have also uploaded a complete project on our repository, in case the TCL file does not work. Both of these can be found on the GitHub repository we have created [12].
9. Now that all files have been added to our project, we can run the synthesis tool. We first make sure that `ibex_top.sv` is listed as the top module in the `Project Summary` panel. If that is the case, we simply press on `Run Synthesis` from the `Flow Navigator` on the left-hand side.

10. If any pop-ups or warnings show up, we can safely ignore them and start the synthesis. Once the synthesis is finished, a pop-up will appear, and we can press `Cancel`. To see the generated report, we can select the `Reports` tab, which we can find in the bottom panel. The file that we need is the first file that appears under `Synthesis/Synth Design`.

Automatic Setup - Memory Initialized

srecord

1. To install `srecord` on our `Ubuntu 22.04` system we need to use the following commands:

```
sudo add-apt-repository ppa:pmiller-opensource/ppa
sudo apt-get update
sudo apt-get install srecord
```

2. Some warnings might show up in the terminal, but we can safely ignore them. More information about `srecord` can be found on the website [3].

fusesoc

1. FuseSoC is provided as `fusesoc` Python package and installed through `pip`, the Python package manager. To this extent, it might be required to install this package manager first. On our `Ubuntu 22.04`, this can be done with the following command:

```
sudo apt install python3-pip
```

2. Now we can install `fusesoc` system-wide using the following command:

```
sudo pip3 install --upgrade fusesoc
```

3. To verify whether the package was properly installed on our system we can use:

```
fusesoc --version
```

4. More information about installing this package can be found on the website [5].

cmake

1. To install the `cmake` tool package on `Ubuntu 22.04` we need the following command:

```
sudo apt install cmake
```

openocd

1. To install the `openocd` tool package on `Ubuntu 22.04` we need the following command:

   ```
   sudo apt install openocd
   ```

2. More information about installing this package can be found on the website [5].

screen

1. To install the `screen` tool package on `Ubuntu 22.04` we need the following command:

   ```
   sudo apt install screen
   ```

2. More information about installing this package can be found on the website [4].

After we have installed the previous prerequisites, we can proceed to use the automatic setup offered by the Ibex Demo System.

Execution

1. We first need to clone the `Ibex Demo System` repository to our machine [9]. This can be done with:

   ```
   git clone https://github.com/lowRISC/ \
   ibex-demo-system.git
   ```

2. Before we can run this example, we need to install the Python virtual environment package and create a virtual environment. After we start the environment, we need to install the Python requirements found under the `ibex-demo-system` folder. This can be done using:

   ```
   sudo apt install python3.10-venv
   python3 -m venv .venv
   source .venv/bin/activate
   pip3 install -r python-requirements.txt
   ```

3. Now, we need to modify the `PATH` variable such that the previously installed tools can be used by the scripts. The required commands are:

   ```
   export PATH=/home/spopa/.local/bin:$PATH
   export PATH=/home/spopa/Desktop/ \
   lowrisc-toolchain-gcc-rv32imcb-20220524-1/bin:$PATH
   source /tools/Xilinx/Vivado/2022.1/settings64.sh
   ```

Please Note: It might be necessary to execute these commands every time the system/virtual environment is restarted.

4. Now we need to build the software that we will load into the FPGA board, after we load the synthesized Ibex processor. To do this, we need the following commands:

```
mkdir sw/build
pushd sw/build
cmake ../
make
popd
```

   These command should create a memory file for each of the available example programs. They can be found under the folder `sw/build` from the main `ibex-demo-system` folder.

5. The next step is to build the FPGA bitstream. From the main folder we run the command:

```
fusesoc --cores-root=. run --target=synth --setup \
--build lowrisc:ibex:demo_system
```

   This command should create a `Xilinx Vivado` project and a bitstream file. These can be found under the folder `build`, which is created under the main repository folder. We can also find the synthesis report under:

```
/Desktop/ibex-demo-system/build/ \
lowrisc_ibex_demo_system_0/ \
synth-vivado/lowrisc_ibex_demo_system_0.runs \
/synth_1/top_artya7_utilization_synth.rpt
```

2.6 Programming the FPGA

After executing the synthesis step for the automatic setup described in 2.5.3, the example code which programs the Arty A7 35T has been compiled as well as transformed into the bitstream necessary to program the FPGA board.

Before we proceed further, it is important to understand how an FPGA board can be programmed, and especially, how the memory contents of the Ibex core are initialized. Thus, the memory can be programmed as follows:

1. Hard code the contents by specifying a block RAM initialisation file. This means our bitstream contains the program to run.
2. We can add a JTAG core and use that to program the memory.

3. Hard code a software bootloader into the memory. This software reads the actual program from the UART, for example, and writes the received data to an offset in memory or a separate memory, and then jumps to that location.

In our case, the example uses option 3. Moreover, all the necessary files are available which allows us to load other kind of programs, such as an encryption algorithm, to the board as well. To upload the generated bitstream to the Arty A7 35T board, we need to proceed as follows:

1. We open `Xilinx Vivado` as explained in Section 2.4.1. Then, we select `Open Project` and browse to the file containing the project generated by the shell script, which is `lowrisc_ibex_demo_system_0.xpr` under `/build/lowrisc_ibex_demo_system_0/synth-vivado`.

2. Once the project has opened, we connect the board to our system using the USB cable and port. To check if the board has indeed connected successfully, we can open a terminal and type the following command:

   ```
   ls /dev
   ```

   This will list all the available `ttyUSB` devices. If `ttyUSB1` is displayed, then the board has connected successfully to our system.

3. Now, from the `Flow Navigator` on the left-hand side, we press on `Hardware Manager`. This should now open a new window, and we expand all the drop-down options.

4. We select `Program device` from the top green bar, and then `xc7a35t_0`. A pop-up will now appear where we select our `.bit` file in the `Bitstream file` box.

5. Finally, we press on `Program` and we wait on the bitstream to be loaded on the board.

6. If desired, we can also use the following command from the main repository folder:

   ```
   make -C ./build/lowrisc_ibex_demo_system_0/ \
   synth-vivado/ pgm
   ```

   However, on our system, the above does not always work properly.

2.7 Loading and Running Software on the FPGA

Now that we have finished setting up and programming the Arty A7 35T, we can proceed with loading and running software on the board. We will

first test the example `Hello World` program from the `Ibex Demo System Repository`. Afterwards we will explain how to load our own programs - in this case, an implementation of the AES encryption algorithm called `TinyAES` - into the board.

2.7.1 `Hello World` Program

1. Before we load the program onto the board, we need a way to verify the output that is being generated. To do so, we open a separate terminal and type the following command:

   ```
   sudo screen /dev/ttyUSB1 115200
   ```

   In order to close this window, press `Ctrl + A`, release these two keys and then press `D`.

2. The `Ibex Demo System` contains a script that allows us to load any program into the FPGA, not just the examples that are provided. To load the `Hello World` program, we need the following:

   ```
   ./util/load_demo_system.sh run \
   ./sw/build/demo/hello_world/demo
   ```

3. If nothing went wrong, then we should be able to see the LED lights on the board constantly changing colors, as well seeing output being generated and displayed in the terminal from step 1.

4. In case of any errors or further explanation, please refer to the `Ibex Demo System` GitHub repository [9].

2.7.2 `TinyAES` Program

1. We first need to clone the `TinyAES` repository to our machine [6]. This can be done with:

   ```
   git clone https://github.com/kokke/tiny-AES-c.git
   ```

2. Then, we have to copy this to the `Ibex Demo System` folder, under `sw/demo/tiny-AES-c`. We also have to move `test.c` from `test_package` to the main `tiny-AES-c` folder, as well as delete `test_package`.

3. The files that require our attention are `crt0.s`, `link.ld`, `test.c` and various `cmake` files. `crt0.s` and `link.ld` do not need to be changed, however they both need a `main()` function in the C code files in order to work.

4. Now, we will need to modify various `cmake` files in order to tell the scripts to also compile the `TinyAES` code using the `RV32IMC GCC` compiler. Moreover, we need to include some libraries and functions in `test.c` that are required for running the code on the Arty A7 35T board. The contents of the file can be found in the appendix A and B or in our repository [12].

5. Before we load this new program on the Arty A7 35T, it might be necessary to press the `PROG` button on the board and remove the built software from the main `Ibex Demo System` folder using:

   ```
   rm -r sw/build
   ```

   Then, we follow step 4 of section 2.5.3 as well as program the FPGA again using the steps in section 2.6.

6. Again, before we load the program onto the board, we need a way to verify the output that is being generated. To do so, we open a separate terminal and type the following command:

   ```
   sudo screen /dev/ttyUSB1 115200
   ```

   In order to close this window, press `Ctrl + A`, release these two keys and then press `D`.

7. To load the `TinyAES` into the FPGA we have to use the following command:

   ```
   ./util/load_demo_system.sh run \
   ./sw/build/demo/tiny-AES-c/tiny_aes
   ```

8. Now we should be able to say the output generated by the AES algorithm. It might be the case that due to the I/O process being slower than the actual algorithm, the output might not appear be fully displayed, even if we press the `RST` button on the board. However, we can see that `SUCCESS` is being shown and thus the algorithm executes correctly.

This completes the setup guidelines. It is possible to use the guidelines as a model for other cores and/or boards, but it is important to note that changing the core and/or board requires specific changes in how certain tools and programs are set up or initialized.

Chapter 3

Discussion

In this section we will discuss a few attention points. Firstly, the reason why we experienced difficulties with the setup was two-fold. The initial plan was to make use of a core from the SHAKTI family, as this was successfully used in a previous project. However, the official documentation was almost non-existent and the steps offered in the mentioned project were not up-to-date, as we encountered issues that we could not fix in time. Therefore, we decided to look for other cores that implemented the RISC-V implementation. The lack of proper documentation was a problem for many cores that we have found. Moreover, even though Ibex has an extensive documentation by itself, the examples they offer sometime require changes that are not mentioned, or the tools that are being used caused errors that took a long time to fix.

Related to this issue is also the fact that the Arty A7 35T board is not optimized for collecting traces, as they would require to be aligned before any analysis can be applied. Therefore, we were planning to use the SAKURA-G board, as the traces collected on this board would not require such an alignment. However, the SAKURA-G board is older, and thus also only supported by outdated software which does not recognize System Verilog, which the Ibex core is written in. That means a conversion between System Verilog and Verilog would be required, which might result in its own errors and issues.

Chapter 4

Conclusion

In this internship, we wanted to look at the differences between the two main methods of leakage detection, Test Vector Leakage Assessment or TVLA, and Side-Channel Vulnerability Factor or SVF. To this extent, we had to create a setup that allowed us to collect power traces generated by the execution of an encryption algorithm on an FPGA board, which would have then been analyzed in order compare the results given by the two leakage detection methods. Unfortunately, due to the difficulties we encountered, we had to reduce the scope to compiling a guideline for creating a setup that allows for such experiments to be conducted. Therefore, we have written step-by-step instructions on how to setup an Ibex core on the Arty A7 35T board, as well as how to program this board to run an encryption algorithm. We concluded our work with a short discussion about the different takeaways to remember about our setup process.

Bibliography

[1] Anonymous. Plan your defense: The impact of leakage detection methods for hardware modules. 2021.

[2] Muhammad Arsath K F, Vinod Ganesan, Rahul Bodduna, and Chester Rebeiro. Param: A microprocessor hardened for power side-channel attack resistance. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 23–34, 2020.

[3] Scott Finneran. SRecord. `https://srecord.sourceforge.net/`.

[4] GNU. Screen. `https://www.gnu.org/software/screen/manual/screen.html`.

[5] Olof Kindgren. FuseSoC. `https://fusesoc.readthedocs.io/en/stable/user/installation.html`.

[6] kokke. TinyAES. `https://github.com/kokke/tiny-AES-c`.

[7] Satoh Lab. SAKURA. `https://satoh.cs.uec.ac.jp/SAKURA/hardware/SAKURA-G.html`.

[8] lowRISC. Ibex Core. `https://github.com/lowRISC/ibex`.

[9] lowRISC. Ibex Demo System. `https://github.com/lowRISC/ibex-demo-system`.

[10] lowRISC. Ibex Synthesis Flow. `https://github.com/lowRISC/ibex/tree/master/syn`.

[11] lowRISC. RV32IMC GCC Toolchain. `https://github.com/lowRISC/lowrisc-toolchains/releases`.

[12] Stefan Popa. Internship Repository. `https://github.com/Archangel2153/research-internship`.

[13] The OpenROAD Project. Nangate 45. `https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/tree/master/flow/platforms/nangate45`.

[14] The OpenROAD Project. OpenSTA. https://github.com/The-OpenROAD-Project/OpenSTA.

[15] RISC-V. RISC-V Cores List. https://github.com/riscvarchive/riscv-cores-list.

[16] Zachary Snow. sv2v. https://github.com/zachjs/sv2v.

[17] Stack. Haskell Stack. https://docs.haskellstack.org/en/stable/.

[18] François-Xavier Standaert. *Introduction to Side-Channel Attacks*, pages 27–42. 12 2010.

[19] Niels van Drueten. Examining a leakage identification proposal for a hardened microprocessor, 2022.

[20] AMD Xilinx. Arty A7-35T. https://www.xilinx.com/products/boards-and-kits/1-elhaap.html.

[21] AMD Xilinx. Xilinx Vivado. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2022-1.html.

[22] AMD Xilinx. Xilinx Vivado Docs. https://docs.xilinx.com/r/en-US/ug973-vivado-release-notes-install-license/Install-Cable-Drivers.

[23] YosysHQ. Yosys. https://github.com/YosysHQ/yosys.

Appendix A

CMake Files

ibex-demo-system/sw/demo/CMakeLists.txt

```
add_subdirectory(hello_world)
add_subdirectory(lcd_st7735)
add_subdirectory(tiny-AES-c)
```

ibex-demo-system/sw/demo/tiny-AES-c/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)

add_prog(tiny_aes "test.c;aes.c")

target_include_directories(tiny_aes
    INTERFACE
        ${CMAKE_CURRENT_LIST_DIR}
)
```

Appendix B

C File

ibex-demo-system/sw/demo/tiny-AES-c/test.c

```c
#include "demo_system.h"
#include "timer.h"
#include "gpio.h"
#include "pwm.h"
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

// Enable ECB, CTR and CBC mode. Note this can be done before including
 ↪  aes.h or at compile-time.
// E.g. with GCC by using the -D flag: gcc -c aes.c -DCBC=0 -DCTR=1 -DECB=1
#define CBC 1
#define CTR 1
#define ECB 1

#include "aes.h"


#define USE_GPIO_SHIFT_REG 0

void test_uart_irq_handler(void) __attribute__((interrupt));

void test_uart_irq_handler(void) {
  int uart_in_char;

  while ((uart_in_char = uart_in(DEFAULT_UART)) != -1) {
    uart_out(DEFAULT_UART, uart_in_char);
    uart_out(DEFAULT_UART, '\r');
    uart_out(DEFAULT_UART, '\n');
  }
}
```

```
// static void phex(uint8_t* str);
static int test_encrypt_cbc(void);
static int test_decrypt_cbc(void);
static int test_encrypt_ctr(void);
static int test_decrypt_ctr(void);
static int test_encrypt_ecb(void);
static int test_decrypt_ecb(void);
static void test_encrypt_ecb_verbose(void);

int main(void)
{
    int exit;

    install_exception_handler(UART_IRQ_NUM, &test_uart_irq_handler);
    uart_enable_rx_int();
    set_outputs(GPIO_OUT, 0x0);
    set_global_interrupt_enable(0);

#if defined(AES256)
    puts("\nTesting AES256");
    putchar('\n');
    putchar('\n');
#elif defined(AES192)
    puts("\nTesting AES192");
    putchar('\n');
    putchar('\n');
#elif defined(AES128)
    puts("\nTesting AES128");
    putchar('\n');
    putchar('\n');
#else
    puts("You need to specify a symbol between AES128, AES192 or AES256.
 ↪  Exiting");
    return 0;
#endif

    exit = test_encrypt_cbc() + test_decrypt_cbc() +
        test_encrypt_ctr() + test_decrypt_ctr() +
        test_decrypt_ecb() + test_encrypt_ecb();
    test_encrypt_ecb_verbose();

    return exit;
}


/*
// prints string as hex
static void phex(uint8_t* str)
{

#if defined(AES256)
    uint8_t len = 32;
#elif defined(AES192)
    uint8_t len = 24;
```

```
#elif defined(AES128)
    uint8_t len = 16;
#endif

    unsigned char i;
    for (i = 0; i < len; ++i)
        printf("%.2x", str[i]);
    printf("\n");
}*/

static void test_encrypt_ecb_verbose(void)
{
    // Example of more verbose verification

    uint8_t i;

    // 128bit key
    uint8_t key[16] =        { (uint8_t) 0x2b, (uint8_t) 0x7e, (uint8_t)
 ↪  0x15, (uint8_t) 0x16, (uint8_t) 0x28, (uint8_t) 0xae, (uint8_t) 0xd2,
 ↪  (uint8_t) 0xa6, (uint8_t) 0xab, (uint8_t) 0xf7, (uint8_t) 0x15,
 ↪  (uint8_t) 0x88, (uint8_t) 0x09, (uint8_t) 0xcf, (uint8_t) 0x4f,
 ↪  (uint8_t) 0x3c };
    // 512bit text
    uint8_t plain_text[64] = { (uint8_t) 0x6b, (uint8_t) 0xc1, (uint8_t)
 ↪  0xbe, (uint8_t) 0xe2, (uint8_t) 0x2e, (uint8_t) 0x40, (uint8_t) 0x9f,
 ↪  (uint8_t) 0x96, (uint8_t) 0xe9, (uint8_t) 0x3d, (uint8_t) 0x7e,
 ↪  (uint8_t) 0x11, (uint8_t) 0x73, (uint8_t) 0x93, (uint8_t) 0x17,
 ↪  (uint8_t) 0x2a,
                               (uint8_t) 0xae, (uint8_t) 0x2d, (uint8_t)
 ↪  0x8a, (uint8_t) 0x57, (uint8_t) 0x1e, (uint8_t) 0x03, (uint8_t) 0xac,
 ↪  (uint8_t) 0x9c, (uint8_t) 0x9e, (uint8_t) 0xb7, (uint8_t) 0x6f,
 ↪  (uint8_t) 0xac, (uint8_t) 0x45, (uint8_t) 0xaf, (uint8_t) 0x8e,
 ↪  (uint8_t) 0x51,
                               (uint8_t) 0x30, (uint8_t) 0xc8, (uint8_t)
 ↪  0x1c, (uint8_t) 0x46, (uint8_t) 0xa3, (uint8_t) 0x5c, (uint8_t) 0xe4,
 ↪  (uint8_t) 0x11, (uint8_t) 0xe5, (uint8_t) 0xfb, (uint8_t) 0xc1,
 ↪  (uint8_t) 0x19, (uint8_t) 0x1a, (uint8_t) 0x0a, (uint8_t) 0x52,
 ↪  (uint8_t) 0xef,
                               (uint8_t) 0xf6, (uint8_t) 0x9f, (uint8_t)
 ↪  0x24, (uint8_t) 0x45, (uint8_t) 0xdf, (uint8_t) 0x4f, (uint8_t) 0x9b,
 ↪  (uint8_t) 0x17, (uint8_t) 0xad, (uint8_t) 0x2b, (uint8_t) 0x41,
 ↪  (uint8_t) 0x7b, (uint8_t) 0xe6, (uint8_t) 0x6c, (uint8_t) 0x37,
 ↪  (uint8_t) 0x10 };

    // print text to encrypt, key and IV
    puts("ECB encrypt verbose:");
    putchar('\n');
    putchar('\n');
    /*
    printf("plain text:\n");
    for (i = (uint8_t) 0; i < (uint8_t) 4; ++i)
    {
        phex(plain_text + i * (uint8_t) 16);
    }
```

```
    printf("\n");

    printf("key:\n");
    phex(key);
    printf("\n");

    // print the resulting cipher as 4 x 16 byte strings
    printf("ciphertext:\n");
    */

    struct AES_ctx ctx;
    AES_init_ctx(&ctx, key);

    for (i = 0; i < 4; ++i)
    {
      AES_ECB_encrypt(&ctx, plain_text + (i * 16));
      //phex(plain_text + (i * 16));
    }
    //printf("\n");
}


static int test_encrypt_ecb(void)
{
#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b,
 ↪  0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                      0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d,
 ↪  0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t out[] = { 0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c, 0x06,
 ↪  0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8 };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8,
 ↪  0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
                      0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t out[] = { 0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f, 0xf7,
 ↪  0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc };
#elif defined(AES128)
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab,
 ↪  0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t out[] = { 0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60, 0xa8,
 ↪  0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97 };
#endif

    uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
 ↪  0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a };
    struct AES_ctx ctx;

    AES_init_ctx(&ctx, key);
    AES_ECB_encrypt(&ctx, in);

    puts("ECB encrypt: ");

    if (0 == memcmp((char*) out, (char*) in, 16)) {
```

```
        puts("SUCCESS!");
        putchar('\n');
        return(0);
    } else {
        puts("FAILURE!");
        putchar('\n');
        return(1);
    }
}

static int test_decrypt_cbc(void)
{

#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b,
 ↪  0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                      0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d,
 ↪  0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t in[]  = { 0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba, 0x77,
 ↪  0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
                      0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d, 0x67,
 ↪  0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d,
                      0x39, 0xf2, 0x33, 0x69, 0xa9, 0xd9, 0xba, 0xcf, 0xa5,
 ↪  0x30, 0xe2, 0x63, 0x04, 0x23, 0x14, 0x61,
                      0xb2, 0xeb, 0x05, 0xe2, 0xc3, 0x9b, 0xe9, 0xfc, 0xda,
 ↪  0x6c, 0x19, 0x07, 0x8c, 0x6a, 0x9d, 0x1b };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8,
 ↪  0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52,
 ↪  0x2c, 0x6b, 0x7b };
    uint8_t in[]  = { 0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d, 0x71,
 ↪  0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
                      0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4, 0xe5,
 ↪  0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a,
                      0x57, 0x1b, 0x24, 0x20, 0x12, 0xfb, 0x7a, 0xe0, 0x7f,
 ↪  0xa9, 0xba, 0xac, 0x3d, 0xf1, 0x02, 0xe0,
                      0x08, 0xb0, 0xe2, 0x79, 0x88, 0x59, 0x88, 0x81, 0xd9,
 ↪  0x20, 0xa9, 0xe6, 0x4f, 0x56, 0x15, 0xcd };
#elif defined(AES128)
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab,
 ↪  0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t in[]  = { 0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46, 0xce,
 ↪  0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
                      0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee, 0x95,
 ↪  0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2,
                      0x73, 0xbe, 0xd6, 0xb8, 0xe3, 0xc1, 0x74, 0x3b, 0x71,
 ↪  0x16, 0xe6, 0x9e, 0x22, 0x22, 0x95, 0x16,
                      0x3f, 0xf1, 0xca, 0xa1, 0x68, 0x1f, 0xac, 0x09, 0x12,
 ↪  0x0e, 0xca, 0x30, 0x75, 0x86, 0xe1, 0xa7 };
#endif
    uint8_t iv[]  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 ↪  0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
    uint8_t out[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
 ↪  0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
```

```
                    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e,
 ↪  0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                    0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5,
 ↪  0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                    0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad,
 ↪  0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 };
//  uint8_t buffer[64];
    struct AES_ctx ctx;

    AES_init_ctx_iv(&ctx, key, iv);
    AES_CBC_decrypt_buffer(&ctx, in, 64);

    puts("CBC decrypt: ");

    if (0 == memcmp((char*) out, (char*) in, 64)) {
        puts("SUCCESS!");
        putchar('\n');
        return(0);
    } else {
        puts("FAILURE!");
        putchar('\n');
        return(1);
    }
}

static int test_encrypt_cbc(void)
{
#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b,
 ↪  0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d,
 ↪  0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t out[] = { 0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba, 0x77,
 ↪  0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
                    0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d, 0x67,
 ↪  0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d,
                    0x39, 0xf2, 0x33, 0x69, 0xa9, 0xd9, 0xba, 0xcf, 0xa5,
 ↪  0x30, 0xe2, 0x63, 0x04, 0x23, 0x14, 0x61,
                    0xb2, 0xeb, 0x05, 0xe2, 0xc3, 0x9b, 0xe9, 0xfc, 0xda,
 ↪  0x6c, 0x19, 0x07, 0x8c, 0x6a, 0x9d, 0x1b };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8,
 ↪  0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5, 0x62, 0xf8, 0xea, 0xd2, 0x52,
 ↪  0x2c, 0x6b, 0x7b };
    uint8_t out[] = { 0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d, 0x71,
 ↪  0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
                    0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4, 0xe5,
 ↪  0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a,
                    0x57, 0x1b, 0x24, 0x20, 0x12, 0xfb, 0x7a, 0xe0, 0x7f,
 ↪  0xa9, 0xba, 0xac, 0x3d, 0xf1, 0x02, 0xe0,
                    0x08, 0xb0, 0xe2, 0x79, 0x88, 0x59, 0x88, 0x81, 0xd9,
 ↪  0x20, 0xa9, 0xe6, 0x4f, 0x56, 0x15, 0xcd };
#elif defined(AES128)
```

```
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab,
↪   0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t out[] = { 0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46, 0xce,
↪   0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
                      0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee, 0x95,
↪   0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2,
                      0x73, 0xbe, 0xd6, 0xb8, 0xe3, 0xc1, 0x74, 0x3b, 0x71,
↪   0x16, 0xe6, 0x9e, 0x22, 0x22, 0x95, 0x16,
                      0x3f, 0xf1, 0xca, 0xa1, 0x68, 0x1f, 0xac, 0x09, 0x12,
↪   0x0e, 0xca, 0x30, 0x75, 0x86, 0xe1, 0xa7 };
#endif
    uint8_t iv[]  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
↪   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
    uint8_t in[]  = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
↪   0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                      0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e,
↪   0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                      0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5,
↪   0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                      0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad,
↪   0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 };
    struct AES_ctx ctx;

    AES_init_ctx_iv(&ctx, key, iv);
    AES_CBC_encrypt_buffer(&ctx, in, 64);

    puts("CBC encrypt: ");

    if (0 == memcmp((char*) out, (char*) in, 64)) {
        puts("SUCCESS!");
        putchar('\n');
        return(0);
    } else {
        puts("FAILURE!");
        putchar('\n');
        return(1);
    }
}

static int test_xcrypt_ctr(const char* xcrypt);
static int test_encrypt_ctr(void)
{
    return test_xcrypt_ctr("encrypt");
}

static int test_decrypt_ctr(void)
{
    return test_xcrypt_ctr("decrypt");
}


static int test_xcrypt_ctr(const char* xcrypt)
{
#if defined(AES256)
```

```
    uint8_t key[32] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
↪   0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                        0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
↪   0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t in[64]  = { 0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
↪   0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
                        0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
↪   0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5,
                        0x2b, 0x09, 0x30, 0xda, 0xa2, 0x3d, 0xe9, 0x4c,
↪   0xe8, 0x70, 0x17, 0xba, 0x2d, 0x84, 0x98, 0x8d,
                        0xdf, 0xc9, 0xc5, 0x8d, 0xb6, 0x7a, 0xad, 0xa6,
↪   0x13, 0xc2, 0xdd, 0x08, 0x45, 0x79, 0x41, 0xa6 };
#elif defined(AES192)
    uint8_t key[24] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
↪   0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
                        0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t in[64]  = { 0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
↪   0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
                        0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
↪   0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94,
                        0x1e, 0x36, 0xb2, 0x6b, 0xd1, 0xeb, 0xc6, 0x70,
↪   0xd1, 0xbd, 0x1d, 0x66, 0x56, 0x20, 0xab, 0xf7,
                        0x4f, 0x78, 0xa7, 0xf6, 0xd2, 0x98, 0x09, 0x58,
↪   0x5a, 0x97, 0xda, 0xec, 0x58, 0xc6, 0xb0, 0x50 };
#elif defined(AES128)
    uint8_t key[16] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
↪   0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t in[64]  = { 0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
↪   0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
                        0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
↪   0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff,
                        0x5a, 0xe4, 0xdf, 0x3e, 0xdb, 0xd5, 0xd3, 0x5e,
↪   0x5b, 0x4f, 0x09, 0x02, 0x0d, 0xb0, 0x3e, 0xab,
                        0x1e, 0x03, 0x1d, 0xda, 0x2f, 0xbe, 0x03, 0xd1,
↪   0x79, 0x21, 0x70, 0xa0, 0xf3, 0x00, 0x9c, 0xee };
#endif
    uint8_t iv[16]  = { 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
↪   0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff };
    uint8_t out[64] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
↪   0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                        0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
↪   0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                        0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
↪   0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                        0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
↪   0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10 };
    struct AES_ctx ctx;

    AES_init_ctx_iv(&ctx, key, iv);
    AES_CTR_xcrypt_buffer(&ctx, in, 64);

    //printf("CTR %s: ", xcrypt);
    puts("CTR\n");
```

```
    if (0 == memcmp((char *) out, (char *) in, 64)) {
        puts("SUCCESS!");
        putchar('\n');
        return(0);
    } else {
        puts("FAILURE!");
        putchar('\n');
        return(1);
    }
}


static int test_decrypt_ecb(void)
{
#if defined(AES256)
    uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe, 0x2b,
 ↪  0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
                      0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d,
 ↪  0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4 };
    uint8_t in[]  = { 0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c, 0x06,
 ↪  0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8 };
#elif defined(AES192)
    uint8_t key[] = { 0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52, 0xc8,
 ↪  0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
                      0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b };
    uint8_t in[]  = { 0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f, 0xf7,
 ↪  0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc };
#elif defined(AES128)
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab,
 ↪  0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
    uint8_t in[]  = { 0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60, 0xa8,
 ↪  0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97 };
#endif

    uint8_t out[]   = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
 ↪  0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a };
    struct AES_ctx ctx;

    AES_init_ctx(&ctx, key);
    AES_ECB_decrypt(&ctx, in);

    puts("ECB decrypt: ");

    if (0 == memcmp((char*) out, (char*) in, 16)) {
        puts("SUCCESS!");
        putchar('\n');
        return(0);
    } else {
        puts("FAILURE!");
        putchar('\n');
        return(1);
    }
}
```