

# Case-Study Documentation for “AVR Processors as a Platform for Language-Based Security”

Florian Dewald, Heiko Mantel, and Alexandra Weber

Computer Science Department, TU Darmstadt, Germany  
{dewald,mantel,weber}@mais.informatik.tu-darmstadt.de

The security type system presented has been implemented in a tool called Side-Channel Finder<sup>AVR</sup> (SCF<sup>AVR</sup>). This document explains its usage and how to carry out our performed case-study.

## A Installation

To run SCF<sup>AVR</sup>, Python 3 with setuptools is required.

Running `apt-get install python3 python3-setuptools` installs the required packages on a Debian Jessie system.

SCF<sup>AVR</sup> can then be installed by running `python3 setup.py install`. Running this command installs additional required python packages and SCF<sup>AVR</sup>.

The AVR toolchain allows compilation and preparation of libraries and programs. Running `apt-get install gcc-avr binutils-avr avr-libc avrdude make` installs the toolchain on a Debian Jessie system.

## B Preparing the Input Program

For running an analysis, SCF<sup>AVR</sup> requires a disassembled object file. We call this format objdump as it may be obtained by using the eponymous tool. An objdump contains all functions present in the object file, marked as assembly labels. Instructions and their arguments are also found directly in the objdump.

The process of creating an objdump file starts from code available in C. This code can be compiled as usual. In the simplest setting, a call like `avr-gcc -g -c -Os -o file.o file.c` is enough to compile it. Flags `g` and `c` are used to correctly assemble the file, but not to link it yet. This is required, as we need a library to work with. We suggest to use the size optimization `-Os`, as it generates easier code that is more likely in the scope of SCF<sup>AVR</sup>.

Executing this command gives an object file `file.o`. This file is passed further to `avr-objdump` which creates the objdump file.

The final objdump file can be created by using `avr-objdump -d -r file.o > file.dump`. Flag `d` tells the disassembler that it should actually disassemble the file. Flag `r` includes the dynamic reallocation table. This makes branch, call and jump targets visible in the objdump.

## C Creating a Configuration

SCF<sup>AVR</sup> takes configuration files in JSON format. A configuration file specifies the function that shall be analyzed, parser constraints and the security domains of parameters and return values of the function to be analyzed.

To explain the configuration file format, we assume a function

```
char func(char secret, char secret1, short public)
```

in file `file.dump`. Parameters `secret` and `secret1` should be of  $\mathcal{H}$  confidentiality, while `public` is of  $\mathcal{L}$  confidentiality. The return value is of  $\mathcal{L}$  confidentiality.

```
{
  "file": "file.dump",
  "starting_function": "func",
  "timing_sensitive": true,
  "include_functions": ["func"],
  "parameters": [{
    "size": 1,
    "confidential": true
  }, {
    "size": 1,
    "confidential": true
  }, {
    "size": 2,
    "confidential": false
  }],
  "memory": true,
  "result": {
    "size": 1,
    "confidential": true
  }
}
```

Figure 1: Example configuration file.

The corresponding configuration file is presented in Figure 1. The `file` directive specifies the path to the objdump file that contains the assembler code that shall be analyzed. The function that shall be analyzed is given by the `starting_function` directive. By setting the `timing_sensitive` directive, the timing sensitivity of the analysis can be turned on or off. When it is deactivated, a normal flow analysis is performed. If a file contains more functions that might not be needed in the analysis, they can be excluded from getting parsed. This can be done by only specifying required functions in `include_functions`. This directive is optional. If it is not specified, then all functions in the objdump will be

included. Confidentiality of parameters can be set in the `parameters` directive. For each parameter in the function signature a `size` and `confidential` directive is required. The `size` is the number of bytes the input parameter requires. For standard types, this can be read of the AVR GCC manual [1]. Setting the `confidential` directive to `true` makes the parameter confidential. Note, that the ordering of parameters has to match the order as specified in the function signature. The same is done for the result in the `result` directive. Note, that setting `confidential` to `true` here sets the output argument to  $\mathcal{L}$ , such that no information from  $\mathcal{H}$  parameters is allowed to flow to the output. As parameters could be passed via memory as well, setting `memory` to `true` signals that there is some confidential data inside the memory.

## D Running SCF<sup>AVR</sup>

For running SCF<sup>AVR</sup>, Python 3 and its `setuptools` are required. SCF<sup>AVR</sup> can then be installed using `python3 setup.py install` inside the folder containing the setup file. This will install SCF<sup>AVR</sup> as a command line utility as well as the required graph library NetworkX [2].

The analysis can then be started by running `scfavr configuration.json`, where `configuration.json` is the path to the configuration file that shall be used. The output is in JSON format, just like the input configuration.

```
{
    "execution_point": null,
    "result_code": 0,
    "result": "SUCCESS",
    "unique_ret": "True"
}
```

Figure 2: Example output.

Code	Result	Explanation
0	SUCCESS	The analysis has found no security issues.
1	INFORMATION_LEAK	Parameters marked as $\mathcal{H}$ are leaked to a result set to $\mathcal{L}$ .
2	TIMING_LEAK	A branch depending on $\mathcal{H}$ data has different region execution times.
3	LOOP_ON_SECRET_DATA	A loop whose execution count is depending on $\mathcal{H}$ data has been found.

Table 1: Possible result values.

A possible output is shown in Figure 2. The `result` and `result_code` directives present the analysis result. In the example, the analysis has found no security issues. Possible values for both directives are shown in Table 1. If the analysis finds security issues, the execution point where this issue occurred is given in `execution_point`. Finally, `unique_ret` verifies whether the assumption of a unique return instruction is satisfied by the program.

## E Running our Case-Study

We have performed the case study using version 20140813 of  $\mu\text{NaCl}$ . For compiling, we have used `avr-gcc 4.8.1`. We have modified the provided `Makefile` inside `avrnacl_small` slightly. In order to get the full assembly code, we have removed the compiler flag `--mcall-prologues`. This flag uses combined functions to get accordance to calling conventions. It stores all used registers on the stack and restores them after function execution. Combining them into such a function saves space. We require the full assembly code, thus we have removed this flag.

The compiled `libnacl.a` file from the small package has then been passed to `avr-objdump` as described in Section B for obtaining the corresponding `objdump libnacl.dump`, on which our tool operates.

We provide suitable configuration files and the compiled library inside the corresponding folder. The case-study can be run by executing the prepared `run-case-study.sh` script.

To reproduce the case study for `verify_leaky` starting from the source, run (while in the top-level folder):

1. Run `avr-gcc -g -c -Os -o case-study/verify_leaky.o case-study/verify_leaky.c`
2. Run `avr-objdump -d -r case-study/verify_leaky.o > case-study/verify_leaky.dump`
3. Run `scfavr case-study/verify_leaky.json`

To reproduce the results for  $\mu\text{NaCl}$  from the source:

1. Download the  $\mu\text{NaCl}$  source code (Version 20140813) from <http://munacl.cryptojedi.org/data/avrnacl-20140813.tar.bz2>
2. Change the file `avrnacl-20140813/avrnacl_small/Makefile` by removing the flag `--mcall-prologues`.
3. Run `make` inside `avrnacl-20140813/avrnacl_small/`
4. Run `avr-objdump -d -r avrnacl-20140813/avrnacl_small/obj/libnacl.a > libnacl.dump`
5. Replace the file `libnacl.dump` on the top-level folder `case-study` by the new file.
6. Run `scfavr case-study/crypto_stream_salsa20.json`
7. Run `scfavr case-study/crypto_stream_xsalsa20.json`
8. Run `scfavr case-study/crypto_onetimeauth_poly1305.json`
9. Run `scfavr case-study/crypto_verify_16.json`
10. Run `scfavr case-study/crypto_verify_32.json`

## References

1. Foundation, F.S.: GCC Wiki. <https://gcc.gnu.org/wiki/avr-gcc>, accessed on 29.03.2016
2. Hagberg, A.A., Schult, D.S., Swart, P.J.: Exploring Network Structure, Dynamics, and Function using NetworkX. In: Proceedings of the 7th Python in Science Conference (2008)