ACFA: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation

Adam Caulfield, *Rochester Institute of Technology;* Norrathep Rattanavipanon, *Prince of Songkla University, Phuket Campus;* Ivan De Oliveira Nunes, *Rochester Institute of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/caulfield

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

USENIX'23 Artifact Appendix: *ACFA*: Secure Runtime Auditing & Guaranteed Device Healing via Active Control Flow Attestation

Adam Caulfield
Rochester Institute of Technology

Norrathep Rattanavipanon
Prince of Songkla University, Phuket Campus

Ivan De Oliveira Nunes
Rochester Institute of Technology

A Artifact Appendix

A.1 Abstract

The artifact of *ACFA* is a hybrid (hardware/software) architecture to enable secure auditing of vulnerability sources and guaranteed remediation when compromise is detected on a remotely deployed MCU. *ACFA* prototype is written in C and Verilog. It is designed alongside an open-source TI MSP430 (openMSP430) and evaluated on a Basys3 FPGA. The artifact includes Python scripts to execute an end-to-end active *CFA* protocol between a remotely deployed MCU Prover ($\mathcal{P}$rv) equipped with *ACFA* and a Verifier ($\mathcal{V}$rf) who manages the MCU and verifies reports from $\mathcal{P}$rv. This appendix aims to assist evaluators in verifying the following *ACFA* major claims: low hardware cost of the hybrid *CFA* design, the ability to audit periodic runtime reports containing fixed-size control flow logs ($\mathcal{CF}_{Log}$), and the ability to execute a guaranteed remediation action as soon as a compromise is detected.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Commit tree `9cf6550` of the *ACFA* Github Repository.

A.2.3 Hardware dependencies

The Basys3 Artix-7 FPGA development board is required.

A.2.4 Software dependencies

The current version was evaluated using the 64-bit Ubuntu 18.04 OS. Xilinx Vivado Toolset 2021.1 or higher is required for synthesizing Verilog files and generating a bitstream for the Artix-7 FPGA. *ACFA* build scripts install Ubuntu packages dependencies in **Part 1** of Sec. A.3.1. A minimum Python version of 3.6.9 is required, and Python dependencies are specified in **Part 3** of Sec. A.3.1.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Part 1: Download *ACFA* source code and Ubuntu libraries:

1. Clone the *ACFA* Github Repository
2. `cd` into the directory `scripts`
3. Run `sudo make install`

Part 2: Download and install Xilinx Vivado

1. Visit Xilinx Vivado Download page.
2. Select the latest version of Vivado that supports Ubuntu.
3. Download and follow directions in the installer.

Part 3: Install `pyserial` using:

- `sudo apt install python3-serial`.

. Verify Python required packages from standard distribution:

- `time, hmac, hashlib, argparse, pickle, dataclasses, os, collections`.

Part 4: Create *ACFA* project in Vivado

- Follow the instructions from `README.md` in the *ACFA* Github Repository to *Create ACFA project in Vivado.*

A.3.2 Basic Test

A simple functionality test includes running a Vivado behavioral simulation of a basic application, an Ultrasonic Sensor, on *ACFA* equipped MCU.

1. Open `openMSP430_defines.v` to set *ACFA* configurations. For the basic test, everything will be simulated in Vivado. Therefore, the flag `ACFA_EQUIPPED` should be set. However, the flag `ACFA_HW_ONLY` should not be set for any simulation. Therefore, "comment-out" this flag by adding "//" to the start of line 58.

2. Now we are ready to synthesize openmsp430 with *ACFA* hardware. On the left menu of the PROJECT MANAGER, click "Run Synthesis", and select execution parameters (e.g., number of CPUs used for synthesis) according to your PC's capabilities. This step takes 2-10 minutes.
3. If synthesis succeeds, a window to "Run Implementation" will appear. Do not "Run Implementation" for the basic test, and close this prompt window.
4. In Vivado, click "Add Sources" (Alt-A), then select "Add or create simulation sources", click "Add Files", and select everything inside `openmsp430/simulation`.
5. Open the `tb_openMSP430_fpga.v` file and find lines 193-202. These lines open `*.cflog` files to simulate the transmission of $\mathcal{CF}_{Log}$ slices for the basic test. Therefore in lines 193-202, replace `<LOGS_FULL_PATH>` with the full file path of the `logs` subdirectory of the *ACFA* directory.
6. Now, navigate to the "Sources" window in Vivado. Search for `tb_openMSP430_fpga`, and in the "Simulation Sources" tab, right-click `tb_openMSP430_fpga.v` and set its file type as the top module.
7. Go back to the Vivado window, and in the "Flow Navigator" tab (on the left-most part of Vivado's window), click "Run Simulation," then "Run Behavioral Simulation."
8. On the newly opened simulation window, select 8ms as the time for the simulation to run. Then press "Shift+F2" to run.
9. The simulation waveform will show two *ACFA* triggers occur during the execution due to the device boot and the program ending. In the `logs` sub-directory of the *ACFA* directory, two `*.cflog` files were generated. If two `*.cflog` files are generated and match the contents of `logs/expected_cflogs_basic_test/`, the basic test has completed successfully.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) Hardware Cost: *ACFA* incurs an additional hardware cost of 275 Look-up Tables (LUTs) and 202 Flip-Flop registers (FFs). This is proven by the experiment (E1). The results of this experiment reflect the results illustrated in Fig. 11 and discussed in Sec. 6.1 of our paper.
(C2) Secure Auditing via Active *CFA*: *ACFA* generates a series of control flow logs of a maximum size which allow for a continuous and active control flow attestation protocol. This is proven by the experiment (E2) and demonstrates the offline and online phases described in Sec. 6.2 of our paper when $\mathcal{P}$rv is not compromised.
(C3) Compromise Detection & Guaranteed Healing: When a control-flow violation has been detected by $\mathcal{V}$rf, *ACFA* immediately executes a remediation action. This is proven by the experiment (E3) and demonstrates the effect of the remediation phase described in Sec. 6.2 of our paper when $\mathcal{P}$rv is compromised.

A.4.2 Experiments

(E1): *Verifying (C1) [10-15 minutes]*

This experiment determines additional LUTs and FFs required by *ACFA* hardware on top of the openMSP430 verilog project in order to estimate the hardware cost. To account for the cost of *ACFA* and all interconnections between *ACFA* and the openMSP430, we determine the cost by taking the difference between the cost of openMSP430+*ACFA* and openMSP430 alone.
[Preparation:] To prepare for this experiment, open *ACFA* Vivado project, as the Vivado toolset will be used to synthesize the Verilog design files into hardware. In addition, open the file `openMSP430_defines.v`. For both of these experiments, the flag `ACFA_HW_ONLY` will always be enabled (ensure no "//" precedes the `` `define ACFA_HW_ONLY `` on line 58). This ensures that only *ACFA* hardware is measured, and additional registers to simulate/emulate memory, which is not part of *ACFA* hardware, are not included in the measurement of hardware cost.
[Execution:] The first phase is to determine the cost of *ACFA* + openMSP430. Ensure `ACFA_EQUIPPED` is enabled (no "//" preceeds the `` `define ACFA_EQUIPPED `` on line 54 of `openMSP430_defines.v`). On the left menu of the PROJECT MANAGER, click "Run Synthesis" as performed in the *Setup*. After Synthesis completes, scroll down to "Utilization" in the "Project Summary" window. Press "Post-Synthesis" and "Table" to see a table of the hardware cost utilized by the synthesized Verilog files. The "Utilization Column" shows the total count of each resource in the "Resource" column. Therefore, this table will show the total LUT (row 1, column 2) and FF (row 3, column 2) required for *ACFA* + openMSP430. Take note of these values (referred to as $LUT_{ACFA+MSP430}$ and $FF_{ACFA+MSP430}$, respectively) since they are required to determine the final result. To get the cost of openMSP430 alone, open `openMSP430_defines.v`) and disable `ACFA_EQUIPPED` (add "//" to the beginning of line 54). Next, save all changes and rerun Synthesis. After it completes, check the LUT and FF utilization using the previous steps. This time, the listed LUT and FF specify the cost of openMSP430 without *ACFA*. Note these values (referred to as LUT_{MSP430} and FF_{MSP430}, respectively) since they are required to determine the final result.
[Results:] The cost of *ACFA* is determined by taking the difference between the cost of *ACFA* + openMSP430 and the cost of openMSP430 alone. Below are the expected results.
Look-Up Tables (LUTs):

- $LUT_{ACFA+MSP430} = 12373$

- $LUT_{MSP430} = 12098$
- $LUT_{ACFA} = LUT_{ACFA+MSP430} - LUT_{MSP430}$
- $LUT_{ACFA} = 12373 - 12098 = 275$

Flip-Flop Registers:

- $FF_{ACFA+MSP430} = 1844$
- $FF_{MSP430} = 1642$
- $FF_{ACFA} = FF_{ACFA+MSP430} - FF_{MSP430}$
- $FF_{ACFA} = 1844 - 1642 = 202$

(E2): *Verifying (C2): [45-75 minutes]*

This experiment demonstrates *ACFA* ability to provide secure auditing of periodic reports through enabling active *CFA*. In this experiment, *ACFA* executes a simple program that receives a remote user's password input and compares it with an expected password. After receiving a correct password, the Prover ($\mathcal{P}$rv) records six readings from an ultrasonic sensor. The Verifier ($\mathcal{V}$rf) has configured *ACFA* to have a maximum $\mathcal{CF}_{Log}$ size of 256B, and the timeout period is set as the maximum value to effectively deactivate triggers due to a timeout. This experiment demonstrates *ACFA* ability to halt execution and a series of fine-grained and timely reports. In addition, this experiment demonstrates the effectiveness of the end-to-end demo's offline and online phases.

[Preparation:] Add "//" to the start of line 59, remove any "//" from the start of line 54 in `demo_prv/main.c`, and save changes. Then, open a terminal window and `cd` into `scripts`. Run `make demo` to compile the software. After this, open `openmsp430_defines.v` and make sure `ACFA_EQUIPPED` is enabled and `ACFA_HW_ONLY` is disabled. Save all changes, then run Synthesis as performed in the Basic Test. Once Synthesis completes, select "Run Implementation." This process takes 30mins-1hour. After Implementation completes, select "Generate Bitstream," which will take 1-2mins. $\mathcal{P}$rv will execute on the FPGA using the bitstream that was just generated. $\mathcal{V}$rf will execute using a Python script during offline and online phases. During the online phase, $\mathcal{V}$rf and $\mathcal{P}$rv connect through a USB-UART interface. Connect the Basys3 FPGA to the machine using the USB cable included with the board. Then, determine which serial port the device is connected to (using `dmesg` command or some other means). After determining the serial port, update lines 16-17 in `demo_vrf/vrf_online.py` to reflect the correct port. The openMSP430 design shares a port between the GPIO and UART, and the GPIO port is selected by default. Therefore to select (and enable) the UART, turn on the physical switch `SW1` on the Basys3 FPGA board.

[Execution]: First complete the offline phase of $\mathcal{V}$rf. During this phase, $\mathcal{V}$rf computes the control flow graph (CFG) of the application software and computes an HMAC over the expected application binary. This is completed by the Python script `vrf_offline.py`. Open a terminal and execute this script by running `python3 vrf_offline.py`, and the binary objects from the offline phase are seen in `demo_vrf/objs`. Next, start the online phase of $\mathcal{V}$rf by running `python3 vrf_online.py`. $\mathcal{V}$rf will start running and wait for a report. Next, in Vivado, click "Open Hardware Manager" and then click "Auto-Connect". The FPGA should now be displayed on the hardware manager menu. Right-click the FPGA and select "Program Device ."After this, the *ACFA*-equipped MCU is programmed onto the FPGA, and $\mathcal{P}$rv will start running.

[Results]: During the online phase, $\mathcal{V}$rf receives reports from $\mathcal{P}$rv which contain a $\mathcal{CF}_{Log}$. The directory `/logs` will be populated with four $\mathcal{CF}_{Log}$ slices during the execution of the online phase. During each iteration of the active *CFA* protocol, $\mathcal{V}$rf will authenticate and verify CFlog slices by comparing them to the CFG and maintaining a shadow stack. In `demo_vrf/objs`, a binary object of the shadow stack is stored and modified during the online phase. *ACFA* generates and sends each report because of an *ACFA* trigger; `0.cflog` and `3.cflog` are generated due to the boot/end of program trigger; `1.cflog` and `2.cflog` are generated due to $\mathcal{CF}_{Log}$ reaching maximum size. The seven segment display of the FPGA board will show the current instruction address: the end of program (`0xe24a`).

(E3): *Verifying (C3) [45-75 minutes]*

This experiment executes the same example application as (E2). However, a buffer overflow is purposefully introduced in this experiment to cause a control flow attack. Therefore, this experiment shows that after a control flow violation has occurred, *ACFA* guarantees the remediation mechanism executes immediately.

[Preparation:] First add "//" to the start of line 54, remove any "//" from the start of line 59 in `demo_prv/main.c`, and save changes. This is the reverse of the first step in (E2) and enables the buffer overflow. After this, follow all remaining steps in the *Preparation* of (E2).

[Execution]: Follow the same steps of *Execution* of (E3).

[Results]: $\mathcal{V}$rf detects the buffer-overflow during the first intermediate report (`1.cflog`). Because of this, $\mathcal{V}$rf sends a command to execute the healing mechanism: shut down $\mathcal{P}$rv. On MSP430, this is achieved by setting a bit in the status register. After doing so, $\mathcal{P}$rv does not continue executing and pauses in TCB. This demonstrates that the compromised $\mathcal{P}$rv could not continue executing due to *ACFA*. In addition, because *ACFA* triggers sent $\mathcal{V}$rf an intermediate report, $\mathcal{V}$rf could find the vulnerability before the adversary could complete their attack. The seven segment display shows that the software is contained at TCB-Heal (`0xa606`).

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.