ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

USENIX Security '24 Artifact Appendix: ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture

Supraja Sridhara Andrin Bertschi Benedict Schlüter Mark Kuhne Fabio Aliberti Shweta Shinde
ETH Zurich

A Artifact Appendix

ACAI enables Arm CCA protected computation to securely connect and use accelerators. To demonstrate the feasibility and overheads of ACAI we evaluate its design on 2 accelerators: GPU and FPGA. We prototype ACAI on Arm's public simulator that supports CCA. We perform our experiments using a standard set of benchmarks from the Rodinia test suite for the GPU and custom-coded examples for the FPGA.

A.1 Abstract

This artifact appendix provides source code and build environments to download, compile, and run ACAI. We prototype our implementation on a publicly available simulation software, called Fixed Virtual Platform (FVP). To implement ACAI we change the TF-A, RMM, and Linux kernels. Our artifact includes different toolchains cross-compiled for Arm platforms, Linux kernels, and root file systems along with our benchmarks. These components are required to launch a realm VM on the FVP. The FVP does not provide interfaces to connect to PCIe devices. To evaluate ACAI, we build an FVP escape mechanism. To enable the FVP escape and communication with PCIe accelerators, our artifact relies on host kernel patches. To reproduce the results of our evaluation, our artifact requires specific accelerators which we are unable to provide access to. Therefore, we only aim for Artifacts Available and Artifacts Functional badges. We provide a minimal working example that doesn't require specialized accelerators and demonstrates ACAI's functionality. We have automated the build process of all software components in a Docker container. Further, the build can also be reproduced by continuous integration.

A.2 Description & Requirements

Software Requirements. We provide scripts that automate the build process for the artifact. We require an x86-64 Linux host machine with a running instance of X11, Docker, make, bash, and Distrobox[1]. Distrobox is a set of bash wrapper scripts to ease the use of Docker. X11 is required to run the simulation software and to show the terminal output on the screen. We have automated the build process using a self-hosted GitHub runner. The runner executes and replicates the precise steps needed to build ACAI. Throughout the artifact evaluation period, this self-hosted runner remains accessible and serves as a resource to solve build-related challenges that may arise. While any Linux distribution can be used to launch Docker, the GitHub runner operates on Ubuntu 20.04 which is the version we recommend.

[1]Distrobox: https://github.com/89luca89/distrobox

Hardware Requirements. The build process for this artifact is resource-intensive and can be time-consuming. Our scripts build several Linux kernels and cross-compile root file systems. We recommend using a powerful machine with several cores. The build requires $\approx$ 100 GB of available storage. Our self-hosted Github runner which uses an Intel Xeon Gold 6346 clocked at 3.1GHz with 62 GB of RAM and requires $\approx$ 3 hours to build the artifact.

A.2.1 Security, privacy, and ethical concerns

Our artifact operates within a simulated Aarch64 environment on x86, made possible by the pre-built FVP provided by Arm. The entire process of building and launching ACAI, alongside the FVP, is encapsulated within a Docker container. To streamline the setup process, we simplify access by mapping the user's home directory into the container. Additionally, Docker requires access to X11, which is required to launch the FVP simulation software from within the Docker container.

The artifact does not involve destructive actions. Beyond the Docker permissions mentioned above, it does not compromise or disable security mechanisms.

A.2.2 How to access

We host the artifact on GitHub in a multi-repository, where the main repository downloads dependencies with git submodules. The submodules are also hosted on GitHub. A GitHub runner provides continuous integration (CI) by building the artifact on a new commit.

Github Organization https://github.com/sectrs-acai/

Main Repository https://github.com/sectrs-acai/acai/tree/490966daf6f3be8798db2de99e2ecdce4deccd0e

GitHub Runner https://github.com/sectrs-acai/acai/actions/workflows/build-acai.yml

A.2.3 Hardware dependencies

We evaluate ACAI with two PCIe-based accelerators. The FVP does not provide a functional interface to connect to PCIe accelerators. We implement an escape mechanism to allow realm VMs in the FVP to communicate with functional accelerators on the underlying x86 host machine. The accelerators we used are:

(1) Nvidia GeForce GTX 460 SE GPU

(2) Xilinx Virtex Ultrascale+ VCU118 FPGA

Interacting with these accelerators requires a custom host kernel. While we provide and build these software components, we do not provide the hardware to evaluate their benchmarks. Therefore, we aim for the available and functional badges and do not impose other hardware dependencies apart from a commodity x86 host machine to build and run a minimal working example.

A.2.4 Software dependencies

See A.2 for the list of software dependencies.

A.2.5 Benchmarks

For GPU benchmarks, we require an additional data set which we download in a script[2]. We use the Rodinia dataset v3.1[3]. Benchmark files are available in the following repositories.

GPU Benchmarks https://github.com/sectrs-acai/acai-rodinia

For FPGA benchmark, we hand-code benchmarks using existing algorithms from existing from Vitis Libraries and Vitis HLS.

FPGA Bitstreams https://github.com/sectrs-acai/acai-fpga-bench

FPGA Runs Scripts https://github.com/sectrs-acai/acai/tree/trusted-periph/master/src/benchmarking/fpga/scripts

[2] https://github.com/sectrs-acai/acai-rodinia/blob/trusted-periph/master/download-assets.sh

[3] http://www.cs.virginia.edu/~skadron/lava/Rodinia/Packages/rodinia_3.1.tar.bz2

A.3 Set-up

A.3.1 Installation

To build the artifact, follow the tutorial hosted in the main GitHub repository at `/doc/artifact-evaluation.md`.

The steps listed in the tutorial include:

(a) Download all submodules and build dependencies,

(b) Build and enter the docker container,

(c) Build the artifact, and

(d) Launch the FVP and run a minimal working example.

The build is automated and only requires the execution of a few scripts.

A.3.2 Minimal Working Example

Follow the steps in `/doc/artifact-evaluation.md#minimal-working-example` to run a minimal working example. The example launches a realm VM on the FVP and uses ACAI to delegate realm private memory to a PCIe test engine.

A.4 Evaluation workflow

We list all our claims and the experiments we performed to support them. For the Artifacts functional badge, we provide a minimal working example in Section A.4.2. For completeness, we list all the experiments we performed in Section A.4.3.

A.4.1 Major Claims

(C1): ACAI maintains compatibility with existing accelerator applications, runtimes, and drivers.

(C2): The bounce buffer design executes more instructions than ACAI (e.g., 26.8$\times$ more for GPU benchmarks)

(C3): ACAI adds a minimal overhead for normal world operation (e.g., 3.8% for GPU benchmarks)

A.4.2 Feasible experiments without special hardware

(E1): *Minimal Working Example* [9 human minutes + 20 compute minutes]: The minimal working example, demonstrates ACAI on an Arm FVP with CCA support and uses a test engine in the FVP to emulate device accesses. This example first sets up all components that ACAI requires to verify functionality: (a) 2 GPTs at boot, (b) SMMU configurations in the monitor, (c) new interface to the realm to delegate memory to be device accessible. Once the realm boots, the example invokes `rsi_delegate_prot_mem` to correctly configure the GPTs and the SMMU. Then, it uses the test engine on the FVP to access the delegated memory.

How to: Please follow the build and run instructions in A.3.1, A.3.2.

Results: The example first demonstrates that the test engine fails to access the memory. After ACAI has delegated the memory, the access succeeds.

A.4.3 Experiments with special hardware

We demonstrate ACAI with GPU and FPGA accelerators. The subsequent benchmarks use the same datasets but differ in their implementation variant.

Preparation: Boot a realm VM with 1 GB of RAM, *niceness -20*, and pin it to the third core. In the hypervisor, reserve core 3 from being used by the scheduler. Use the PCIe bypass mechanism to exchange memory mappings between the x86 host machine and the FVP.

GPU Benchmarks: In the realm VM, load the ACAI-Helper kernel module and install the GPU kernel driver[4].

FPGA Benchmarks: In the realm VM, load the ACAI-Helper kernel module and install the FPGA kernel driver.

Execution: Run the corresponding benchmark with the setup script. The script configures the number of iterations for the benchmark.

Results: Find a capture file on the underlying host machine in `/tmp/arm*` with the number of executed instructions, context, and security domain switches.

Source code of TFA, RMM, and Linux kernel for different variants:

ACAI *in (A_e) and (B_v):*
branch `trusted-periph/unmodified`

ACAI *in (A_p) and (A_v):*
branch `trusted-periph/master`

Implementation variants for GPU [5]:

(I1): ACAI *in encryption mode (A_e):*
Run `cuda_enc/setup.sh run` in a realm VM.

(I2): ACAI *in protection mode (A_p):*
Run `cuda/setup.sh run` in a realm VM.

(I3): ACAI *changes on Normal World (A_v):*
Run `cuda_ns/setup.sh run` in normal world.

(I4): *Normal world with unmodified software (B_v):*
Run `cuda_ns/setup.sh run` in normal world.

Implementation variants for FPGA[6]:

(I1): ACAI *in encryption mode (A_e):*
Run `setup.sh{xdma_enc|svd_32_enc|svd_48_enc|{matmul_enc}` in a realm VM.

(I2): ACAI *in protection mode (A_p):*
Run `setup_devmem.sh{xdma|svd_32|svd_48|matmul}` in a realm VM.

(I3): ACAI *changes on Normal World (A_v):*
Run `setup.sh{xdma|svd_32|svd_48|matmul}` in normal world.

(I4): *Normal world with unmodified software (B_v):*
Run `setup.sh{xdma|svd_32|svd_48|matmul}` in normal world.

Experiments:

(E2): Compare results of **I1** and **I2** for claim **C2**. All implementation variants execute the benchmarks with unmodified application, runtime and driver code. This validates claim **C1**.

(E3): Compare results of **I3** and **I4** for claim **C3**.

A.5 Notes on Reusability

- We implement a reusable setup to build TFA, RMM, Linux kernels, and Aarch64 root file systems based on Buildroot. The setup can be used for further development on the platform.
- We implement a PCIe escape mechanism that can be used for accelerators other than an FPGA and GPU. To enable computation in a realm VM to escape the FVP, we implement stub drivers in the realm VM which use a page fault-based mechanism. To control these page faults and identify pages allocated to the devices, we modify the memory allocation routines on the host for the FVP process. The page-fault mechanism coupled with the memory allocation routines can be reused to implement the escape mechanism for other PCIe devices.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

[4] Run experiments: `https://github.com/sectrs-acai/acai/blob/trusted-periph/master/doc/run_experiments.md`

[5] Script directory `/src/gpu_driver/rodinia-bench/` in `https://github.com/sectrs-acai/acai`

[6] Script directory `/src/benchmarking/fpga/scripts` in `https://github.com/sectrs-acai/acai`