DmaAuth: A Lightweight Pointer Integrity-based Secure Architecture to Defeat DMA Attacks

Xingkai Wang, Wenbo Shen, Yujie Bu, Jinmeng Zhou,
and Yajin Zhou, *Zhejiang University*

https://www.usenix.org/conference/usenixsecurity24/presentation/wang-xingkai

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

USENIX Security '24 Artifact Appendix: DmaAuth: A Lightweight Pointer Integrity-based Secure Architecture to Defeat DMA Attacks

Xingkai Wang, Wenbo Shen✉, Yujie Bu, Jinmeng Zhou, and Yajin Zhou

Zhejiang University

A Artifact Appendix

A.1 Abstract

To defend DMA attacks effectively with low overhead, we design and implement DmaAuth, a novel hardware-software co-design to enforce spatial and temporal protection for DMA buffers. It registers information for each DMA buffer and signs the DMA pointers when creating DMA buffer mappings, and authenticates the DMA using the signature and corresponding buffer information, including a unique identifier and bounds. Before designing DmaAuth, we performed a detailed Characterization. We implemented DmaAuth on RISC-V and ARM QEMU, as well as on Rocket Chip based RISC-V SoC on FPGA.

This artifact contains the QEMU setup and analyzing tool to perform characterization, ARM and RISC-V QEMU with DmaAuth emulation, and FPGA-based SoC with DmaAuth and IOMMU hardware, as well as Linux kernels with corresponding drivers for each of the platforms described in implementation.

A.2 Description & Requirements

The private key and ssh configuration is provided in the HotCRP submission.

A.2.1 Security, privacy, and ethical concerns

None. But the reviewer may want to use VPN to connect to the provided server to avoid any potential information leakage.

A.2.2 How to access

The DOI is `10.5281/zenodo.12216074`. It can be accessed via https://doi.org/10.5281/zenodo.12216074. All the required copy of the files are contained in `/home/reviewer/Artifacts` in the provided server. Reviewers may want to use the local version directly since the artifacts is quite large.

A.2.3 Hardware dependencies

The provided server is already connected to the hardware.

XCKU040 FPGA, FMC-to-PCIe adaptor card, SD card, PCIe device required to test the functionality of the SoC with DMAAUTH support.

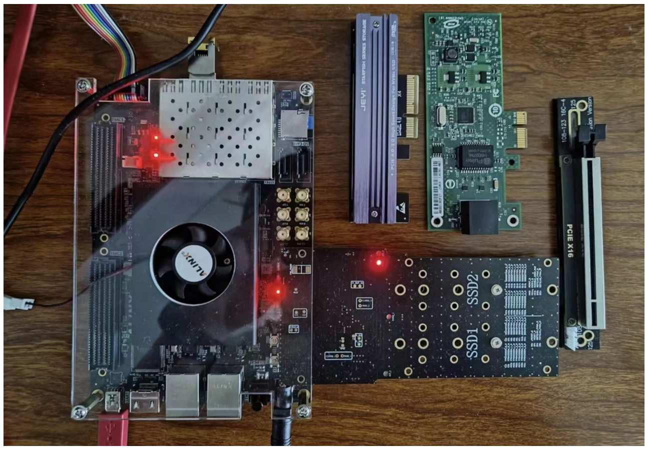

Figure 1: Hardware platform for the artifacts.

A.2.4 Software dependencies

The provided server is already capable of running the artifact. The environment is Ubuntu 22.04.4 LTS with the required package to build Linux kernel, and AMD Vivado 2022.02.

A.2.5 Benchmarks

None.

A.3 Set-up

The environment is already provided on the server. If using our provided server, skip the A.3.1 section.

For reviewers who want to try the artifact on their own machine, the following steps are required.

✉ Corresponding author.

1. Please install the dependencies required to build the Linux kernel. 2. Please install Vivado 2022.2 to build the FPGA bitstream. 3. Download and extract the artifacts from provided zenodo DOI.

A.3.1 Installation

If using the provided server, no extra steps are required.

When evaluating the artifacts on your local machine, about 300GB of disk space is required. And we recommend using Ubuntu 22.04, the same distribution as we provided.

Please install build dependencies for the kernel and emulator using command `sudo apt install build-essential libncurses-dev bison flex libssl-dev libelf-dev bc ninja-build libsdl2-dev libpixman-1-dev libslirp-dev gcc-aarch64-linux-gnu gcc-riscv64-linux-gnu autoconf python3-tqdm iperf3`.

Please download and install Vivado 2022.2 from https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html. In the installation options, please select Vitis to avoid potential problems.

To synthesis the FPGA bitstream, a enterprise license is required. But flashing the bitstream to the FPGA does not require a license. Please follow the official guide https://docs.amd.com/r/en-US/ug973-vivado-release-notes-install-license/Managing-Licenses-with-the-Vivado-License-Manager to install the license.

RootFS are provided directly in the `emulator/images` directory. If you want to build the rootfs from scratch, please follow these steps:

1. Download buildroot from https://buildroot.org/download.html, and extract the archive.
2. Enter the buildroot directory, and run `make meunuconfig`. And change the following options:

   `Target options` → `Target Architecture` → `RISCV` or `AArch64 (little endian)`.

   `Filesystem images` → `ext2/3/4 root filesystem` → `ext2/3/4 variant (ext4)`.
3. Save the configuration and run `make` to build the rootfs.
4. Copy the generated `output/images/rootfs.ext4` to the `emulator/<arch>.img` directory.

A.3.2 Basic Test

Please follow the following steps to run the basic test.

1. Go to the artifacts directory.
2. Enter the `emulator` directory.
3. Run `make riscv` to start the RISC-V QEMU emulator with DMAAUTH support.
4. Enter `root` to login, a shell should start.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *We characterize the DMA buffer usage in Linux kernel and show that the DMA buffer is reused and shared among different devices. As depicted in Section 3 in the paper, in Table 1 and Figure 2. The functionality is tested in (E1).*

(C2): *We implement DMAAUTH on RISC-V and ARM QEMU, as discussed in Section 6 in our paper. The functionality is tested in (E2).*

(C3): *We implement DMAAUTH based on a RISC-V SoC on FPGA, as discussed in Section 6 in our paper. The functionality is tested in (E3).*

A.4.2 Experiments

(E1): *[Characterization] [10 human-minutes + 1 compute-hour + 30GB disk]: This runs the characterization of DMA behavior of different peripherals and corresponding result analysis.* More details are provided in the `Artifacts/character/readme.md`

Preparation: In the provided server, enter `Artifacts/character`. Or download the character.tar.gz from zenodo and extract it, then enter the `character` directory.

Execution: Run the following command.
1. `make clean`
2. `make qemu` to build the emulator.
3. `make linux` to build the Linux kernel.
4. `make analyze` to start automatic data collection and then analyze the collected data.

Results: A analysis of the collected DMA behavior will be displayed.

(E2): *[Emulator Functionality] [1 human-hour + 1 compute-hour + 20GB disk]: This tests the functionality of the DMAAUTH implemented on the QEMU emulators. More details are provided in the* `Artifacts/emulator/readme.md`

Preparation: In the provided server, enter `Artifacts/emulator`. Or download the emulator.tar.gz from zenodo and extract it, then enter the `emulator` directory.

Execution: Run the following commands.
1. `make clean`
2. `make arm`, then the ARM QEMU will start running. Then, enter `root` and hit enter, then you will get the shell. You can now `mount /dev/nvme0n1p1 .mount` and read and write the `.mount` directory to test if the NVMe disk works on PCIe bus, and afterwards, you can ping some websites to see if the NIC works.

3. `make riscv`, then the RISC-V QEMU will start running. Then, enter `root` and hit enter, then you will get the shell. You can now `mount /dev/nvme0n1p1 .mount` and read and write the `.mount` directory to test if the NVMe disk works on PCIe bus, and afterwards, you can ping some websites to see if the NIC works.

Results: During the execution, the DMA mapping, unmapping and authentication process will be displayed in the terminal.

(E3): *[FPGA Functionality] [1 human-hour + 8 compute-hour + 20GB disk]: This allows you to flash the baseline, iommu and DMAAUTH SoCs to FPGA board to see if the hardware design works correctly. More details are provided in the* `Artifacts/fpga/readme.md`

Preparation: In the provided server, enter `Artifacts/fpga`. Or download the fpga.tar.gz from zenodo and extract it, then enter the `fpga` directory.

Execution: Run the following commands.
1. `make clean`
2. `make bare-kernel && make bare-fpga` to build kernel and bitstream.
3. `make bare-flash` to flash the bare FPGA bitstream to the FPGA. Afterwards wait for less than 1 minutes until a menu pops up, allowing you to select the kernel. It should be something like this:

```
Scanning mmc 0:1...
Found /extlinux/extlinux.conf
Retrieving file: /extlinux/extlinux.conf
RISC-V Boot Options.
1:      auth
2:      iommu
3:      bare
Enter choice:
```

Enter the number of `bare` option, and hit enter.

```
Enter choice: 3
3:      bare
Retrieving file: /extlinux/bare-image
```

After the kernel starts, enter `root` to login and get the shell.
Run `rm log` then `/disk.sh`, the script finishs in about 30 minuts. Then you can `cat log` to see if the PCIe bus works for the NVMe disk.
Finally hit `Ctrl+]` to quit the terminal.
4. `make iommu-kernel && make iommu-fpga`
5. `make iommu-flash`, this is the same as the `bare-flash`, but select the `iommu` option in the menu. Then perform the same process as in the `bare-flash` process.
6. `make auth-kernel && make auth-fpga`
7. `make auth-flash`, this is the same as the `bare-flash`, but select the `auth` option in the menu. Then perform the same process as in the `bare-flash` process.

Results: The kernels and FPGA bitstreams will be generated and flashed to the FPGA. In the serial terminal, `/disk.sh` should run correctly, whose log can be captured by using `cat log`.

A.5 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/.