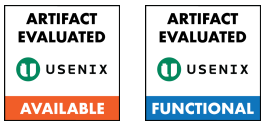

USENIX Security '26 Artifact Appendix: Khost: KVM-based Near Native MCU Firmware Rehosting

Chunlin Wang, Yicheng Yang, Yuan Zhang, Haoyu Xiao, Yifan Zhang, Jiarun Dai
Fudan University, China

A Artifact Appendix

A.1 Abstract

Khost is a KVM-based near native rehosting framework for MCU firmware. This artifact describes the process of deploying and evaluating Khost on the ARMv8-A platform.

A.2 Description & Requirements

Khost is developed and evaluated on Ubuntu 22.04 using two ARM-based platforms: (1) a Huawei KunPeng server (TaiShan 2280) with dual 32-core Cortex-A72 processors clocked at up to 2.4 GHz and 256 GB DDR4 RAM, and (2) a Raspberry Pi 4B with a quad-core Cortex-A72 processor clocked at up to 1.8 GHz, up to 8 GB LPDDR4 RAM, and 64 GB flash storage. To reproduce our results, we recommend using the same hardware and system configuration.

A.2.1 Security, privacy, and ethical concerns

On the one hand, Khost relies on KVM hardware features and requires specific kernel capabilities. If these features are not properly configured or used, it may lead to instability or even crashes on the host system. On the other hand, new bugs may be discovered during the evaluation of this artifact. We would appreciate responsible disclosure of any such issues to the maintainers.

A.2.2 How to access

Our research adheres to the principles of open science. All artifacts, including datasets, scripts, the source code of our tool, and patches to existing tools for ARM platform support, are publicly available at a stable reference: https://doi.org/10.5281/zenodo.17976459 . In addition, any subsequent updates, including new features or bug fixes, are maintained at https://github.com/seclab-fudan/Khost.

A.2.3 Hardware dependencies

Khost is a KVM-based MCU firmware rehosting framework that requires specific hardware features for reproducibility. First, it relies on ARM platforms that support the ARM KVM extension. Second, it leverages the backward compatibility of the ARMv8-A instruction set architecture, which ensures that most MCU instructions can be executed correctly. Finally, sufficient computational resources are required to reproduce our experiments smoothly, specifically at least 8 GB of RAM and 64 GB of storage.

Currently, Khost has been successfully evaluated on both a Huawei KunPeng server (TaiShan 2280) and a Raspberry Pi 4B development board (8 GB RAM and 64 GB flash storage). Since the experiments reported in this paper were primarily conducted on the KunPeng server, which is located within an internal network and is not publicly accessible due to privacy constraints, we recommend using a Raspberry Pi 4B for artifact evaluation. Although the results obtained on the Raspberry Pi 4B do not match the performance achieved on the Kunpeng server, they are sufficient to reproduce the experiments and demonstrate the key outcomes of our work.

A.2.4 Software dependencies

Khost is developed and evaluated on Ubuntu 22.04 in our experiments. Due to a Linux kernel bug that prevents vCPU state restoration in AArch32 system mode, the kernel must be updated to version 5.15.0.1070 or later to include the corresponding fix. In addition, Khost relies on several third-party tools and libraries, including Capstone, Keystone, and Fuzzware. To deploy Khost, the required dependencies should be installed by following the instructions in Section A.3.

A.2.5 Benchmarks

We evaluate Khost on three different datasets. (1) `CoreMark-PRO`, an industry-standard benchmark designed to measure system performance on complex computational workloads. (2) `Simbench`, a comprehensive microbenchmark suite for evaluating system-level performance of full-system emulators. (3) `Real-world Firmware Dataset`, a newly constructed dataset derived from firmware used in prior work. The prebuilt benchmark cases and firmware binaries are available at experiments.

A.3 Set-up

This section describes the process of deploying Khost on an ARM platform, using the Raspberry Pi 4B as a reference. In total, the environment setup and subsequent installation take approximately two hours.

Memory and Swap Requirements: Khost is memory-intensive, so 8 GB of RAM is recommended. If the available physical memory is insufficient, swap space must be configured to prevent the program from being terminated by the Linux Out-Of-Memory (OOM) killer, although this will degrade performance due to increased memory access latency. The following commands can be used to configure swap space.

```
sudo swapoff /swapfile
sudo fallocate -l 8G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

KVM Permissions: Access to /dev/kvm is restricted to users in the KVM group. If you encounter the error permission denied accessing /dev/kvm, add your user to the kvm group using `sudo usermod -aG kvm $USER`. To take effect, you must fully log out and log back in after running this command. You can verify the change using `groups | grep kvm`. If /dev/kvm is still inaccessible after re-login, ensure that your kernel has KVM support enabled. **Alternatively**, if the system is accessed by logging in as the root user, the above KVM permission configuration steps are not required.

Patch Kernel Bug: Due to a bug in the Linux kernel that prevents vCPU restoration in AArch32 system mode, the kernel must be updated to version 5.15.0.1070 or later to incorporate the corresponding fix. More details can be found in the following upstream KVM patch.

On Raspberry Pi 4B, you can either run `sudo apt install -y linux-image-raspi && sudo reboot` or download the kernel from linux-kernel-for Raspberry and update manually.

The packages and third-party tools required by Khost can be installed using the following terminal commands.

Listing 1: Packages

```
sudo apt install curl vim cmake make bc git gcc g++ \
 llvm-dev libyaml-cpp-dev tmux ninja-build meson \
 libglib2.0-dev gcc-arm-none-eabi python3-pip \
 python3-virtualenv python3-virtualenvwrapper
```

Listing 2: Keystone

```
git clone https://github.com/keystone-engine/keystone.git
cd keystone && git checkout fb92f32391c6cced8
mkdir build && cd build && ../make-share.sh
sudo make install && sudo ldconfig && cd ../../
```

Listing 3: Capstone

```
git clone https://github.com/capstone-engine/capstone.git
cd capstone && git checkout 106f7d3b949f5dd8e6c
./cmake.sh && make -j && sudo make install && sudo
     ldconfig && cd ../
```

Listing 4: Cargo and Rustup

```
sudo apt install iproute2 cargo
sudo curl --proto '=https' --tlsv1.2 -sSf \
https://sh.rustup.rs | sh
```

Listing 5: Add the following code to ~/.cargo/config

```
[target.armv7-unknown-linux-gnueabihf]
  linker = "arm-linux-gnueabihf-gcc"
```

A.3.1 Installation

Navigate to the root directory of Khost and run the following commands to compile Khost with different fuzzing backends. Due to differences in their configurations, Khost must be recompiled when switching between fuzzing backends. In addition, in the LibAFL version, backend processes may persist even after the program is forcefully terminated. Therefore, before starting a new round of experiments on your platform, it is necessary to ensure that all running khost-libafl-fuzz backend processes have been terminated.

Listing 6: AFL++ Backend

```
./clean.sh && ./build.sh
```

Listing 7: LibAFL Backend

```
./clean.sh && cargo clean && cargo build --release
```

A.3.2 Basic Test

After the compilation completes, you can verify the build by running the following terminal. Both commands will display their respective help messages.

```
./build/runner/khost -h
./build/runner/khost-debug -h
./build/fuzzer/khost-afl-fuzz -h
```

A.4 Evaluation workflow

To evaluate Khost, we conduct three major experiments. You can reproduce them with the following commands (all operations are performed in the root directory of the Khost project).

Listing 8: Configure QEMU

```
wget https://download.qemu.org/qemu-8.2.10.tar.bz2
tar xvf qemu-8.2.10.tar.bz2
cp experiments/patches/qemu/stm32f405_soc.h \
     qemu-8.2.10/include/hw/arm/
cd qemu-8.2.10 && mkdir build && cd build
../configure --target-list="arm-softmmu" --disable-capstone
make -j$(($(nproc) - 1))
cd ../../
```

Listing 9: Performance Test on CoreMark-PRO[1]

```
./scripts/coremark_test_qemu.sh
./scripts/coremark_test_khost.py
./scripts/coremark_test_native.sh
```

[1]When executed, the script interactively prompts the user to choose whether to perform ablation experiments.

Listing 10: Performance Test on Simbench

```
./scripts/simbench_test_qemu.sh
./scripts/simbench_test_khost.py
```

Listing 11: Bypass the AFL++ Coredump Check

```
echo core | sudo tee /proc/sys/kernel/core_pattern
cd /sys/devices/system/cpu
echo performance | sudo tee cpu*/cpufreq/scaling_governor
cd -
```

Listing 12: Fuzzing Test with AFL++ Backend (Usage + Case)
Usage: ./scripts/fuzz_start.py firmware_id/all[2]

```
./scripts/fuzz_start.py 01_PLC
./scripts/fuzz_start.py all
```

Listing 13: Fuzzing Test with LibAFL Backend (Usage + Case)
Usage: ./scripts/libafl_fuzz_start.py firmware_id/all

```
./scripts/libafl_fuzz_start.py 01_PLC
./scripts/libafl_fuzz_start.py all
```

Listing 14: Coverage Calculation with AFL++ Backend (U. + C.)
Usage: ./scripts/fuzz_replay.py firmware_id

```
./scripts/fuzz_replay.py 01_PLC
```

Listing 15: Coverage Calculation with LibAFL Backend (U.+C.)
Usage: ./scripts/libafl_fuzz_replay.py firmware_id

```
./scripts/libafl_fuzz_replay.py 01_PLC
```

Listing 16: Replay a Test Case (Usage + Case)
Usage: ./build/runner/khost-debug firmware_config input_file

```
./build/runner/khost-debug \
experiments/fuzz/01_PLC/config.yml \
experiments/fuzz/01_PLC/crashes/id\:000074\,sig\:00\,src
    \:000414\,time\:9648045\,execs\:5625492\,op\:havoc\,
    rep\:13  \
-l debug -o all
```

Listing 17: Overhead of Coverage Collection

```
./scripts/count_overhead_cov.py
```

A.4.1 Major Claims

In our paper, we make the following four major claims.

(C1): Compared to QEMU, Khost reduces the overhead of handling complex computational tasks in CoreMark-PRO by 90.0% to 95.5% on the Kunpeng server, and even on a Raspberry Pi 4B, it reduces overhead by more than 80%.

(C2): Khost reduces the overhead on system-level tasks of Simbench by up to 55% compared to QEMU on the Kunpeng server, except for the exception test cases, and even on a Raspberry Pi 4B, it reduces overhead by more than 22.5%.

[2]The parameter `firmware_id` specifies a single firmware case (e.g., `01_PLC`), while `all` indicates that all 12 firmware instances are fuzzed simultaneously. Please ensure that your platform has sufficient CPU cores and memory resources when using `all`.

(C3): On the Kunpeng server, Khost achieves up to 197.5× higher throughput and improves basic block coverage by up to 6x compared to the existing fuzzing tool (HALucinator).

(C4): Khost is enable to uncover new bugs.

(C5): The overhead caused by coverage collection is approximately 4.71%.

A.4.2 Experiments

To demonstrate our major claims and open-science contributions, we provide the following experiments (approximately nine days of runtime with sufficient hardware resources):

(E1): [Performance Test with CoreMark-PRO] *[10 human-minutes + 7 compute-hours (three rounds) + 64GB disk]*: Evaluate the performance of Khost in handling complex computational tasks, with expected results close to those described in Claim C1 in Section A.4.1. As discussed in our paper, disabling the Auxiliary Page Table (APT) causes all memory regions to be marked as Device-nGnRnE, which severely degrades rehosting performance and requires more than 5 compute-hours to complete the tests. Without the ablation experiments, all tests can be completed within one compute-hour.
Steps: First, deploy Khost on the target platform according to Section A.3. Then, build QEMU following the commands in Listing 8. Next, start performance tests using the commands shown in Listing 9. Finally, the result of each test case will be printed as a log entry, where QEMU reports the metric as `Time Interval` and Khost reports it as `passing time`. More details and examples can be found at: performace-test-on-coremark-pro

(E2): [Performance Test with Simbench] *[10 human-minutes + 210 compute-hours (three rounds) + 64GB disk]*: Evaluate the performance of Khost in handling system-level tasks, with expected results close to those described in Claim C2 in Section A.4.1.
Steps: After completing the configuration in E1, performance tests on Simbench can be directly launched using the commands shown in Listing 10. As discussed in our paper, disabling the APT or moving rehosting components outside KVM severely degrades rehosting performance, requiring more than 208 compute-hours to complete the tests. Without ablation experiments, all tests can be completed within 3 compute-hours.

(E3-1): [Fuzzing Test with AFL++ Backend] *[5 human-minutes + 120 compute-hours (five rounds with all 12 firmware tested simultaneously) or 1440 compute-hours (five rounds with firmware tested one by one) + 64GB disk]*: Evaluate the throughput and basic block coverage of Khost during fuzzing with the AFL++ backend, with expected results close to those described in Claim C3 in Section A.4.1.
Steps: First, deploy Khost on the target platform accord-

ing to Section A.3. Then, you can conduct fuzzing tests with AFL++ backend using the commands shown in Listing 12. Finally, the results of fuzzing can be calculated with commands in Listing 14.

(E3-2): [Fuzzing Test with LibAFL Backend] *[5 human-minutes + 120 compute-hours (five rounds with all 12 firmware tested simultaneously) or 1440 compute-hours (five rounds with firmware tested one by one) + 64GB disk]*: Evaluate the throughput and basic block coverage of Khost during fuzzing with the LibAFL backend, with expected results close to those described in Claim C3 in Section A.4.1.
Steps: After E3-1, you can start fuzzing tests with LibAFL backend using the commands shown in Listing 13. Finally, the results of fuzzing can be calculated with commands in Listing 15.

(E4): [Uncover Bugs] *[5 human-minutes + 24 compute-hours + 64GB disk]*: Evaluate the ability of Khost in finding bugs, with expected results close to those described in Claim C4 in Section A.4.1.
Steps: You can start fuzzing the firmware to find bugs. Alternatively, to facilitate evaluation, we also upload the crash cases to the repository. The seeds that trigger crashes can be found in the crashes directory under the echo firmware subdirectory, for example, `experiments/fuzz/01_PLC/crashes`. Any crash case can be replayed using the commands shown in Listing 16.

(E5): [Overhead of Coverage Collection] *[5 human-minutes + 3.5 compute-hours (3 rounds) + 64GB disk]*: Evaluate the overhead of rewrite-based coverage collection in Khost, with expected results close to those described in Claim C5 in Section A.4.1.
Steps: After deploying Khost, run the test by executing the commands shown in Listing 17.

A.5 Notes on Reusability

This document primarily describes the workflow for using Khost. Detailed instructions on deploying Fuzzware and HALucinator are available at Run Fuzzware on ARM64 and Run HALucinator on ARM64, respectively.

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/usenixsec2026/`.