USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EKC: A Portable and Extensible Kernel Compartment for De-Privileging Commodity OS

Jiaqin Yan, *Shanghai Jiao Tong University, Southern University of Science and Technology;* Qiujiang Chen, Shuai Zhou, and Yuke Peng, *Southern University of Science and Technology;* Guoxing Chen, *Shanghai Jiao Tong University;* Yinqian Zhang, *Southern University of Science and Technology*

https://www.usenix.org/conference/usenixsecurity25/presentation/yan-jiaqin

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: EKC: A Portable and Extensible Kernel Compartment for De-Privileging Commodity OS

Jiaqin Yan[*†], Qiujiang Chen[†], Shuai Zhou[†], Yuke Peng[†], Guoxing Chen[*], Yinqian Zhang[†]
[*]Shanghai Jiao Tong University, [†]Southern University of Science and Technology
yan2364728692@gmail.com, {12012211, zhous2021, pengyk}@mail.sustech.edu.cn,
guoxingchen@sjtu.edu.cn, yinqianz@acm.org

A Artifact Appendix

A.1 Abstract

The artifact includes the following components:

- `RustEKC_src.zip`: Contains the complete source code and documentation for RustEKC, consistent with the open-source repository. It enables users to examine the EKC's internal design, modify the implementation, and recompile it for integration into custom applications.
- `payload_src.zip`: Provides the source code of the supported OS kernels—rCore, FreeRTOS, and TinyLinux—demonstrating how each integrates with EKC. Users can freely inspect, modify, and rebuild these kernels as needed.
- `RustEKC_artifact.zip`: Offers pre-compiled binaries for execution on the QEMU platform, allowing users to evaluate RustEKC without setting up a full build environment. All binaries are compiled from the sources in `RustEKC_src.zip` and `payload_src.zip`, and are intended for basic testing.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The execution of this artifact does not pose any risks to the evaluator's system security, data privacy, or ethical considerations. Once the environment is set up and installed, the artifact can be executed and tested without requiring Internet access.

A.2.2 How to access

This submission complies with the Open Science guidelines of USENIX Security '25. All relevant artifacts have been publicly released on `Zenodo`:

doi.org/10.5281/zenodo.15534623

Additionally, the latest working copy of the artifact for is maintained in an open-source repository. For artifact evaluation, a stable release is available at:

github.com/EmbeddedKC/RustEKC/releases/tag/AEC

Future updates and bug fixes will be published through this repository.

A.2.3 Hardware dependencies

The artifact supports both simulation and hardware-based testing.

Simulation: An x86 host machine is sufficient.

Hardware (board) testing: An x86 host is required, along with one or more of the following development boards:

- Raspberry Pi 4B (for the AArch64 instruction set)
- Allwinner Nezha D1-H (for the RISC-V instruction set)

In addition, UART and USB cables are needed for FEL flashing and serial debugging.

A.2.4 Software dependencies

The following software tools are required:

- `QEMU` 6.2.0 — for emulated testing.
- `xfel` — for deploying EKC on the Allwinner D1 board.
- Pi4 flashing utility — for Raspberry Pi 4B support.
- `PuTTY` — for serial port debugging.

The following toolchains are required:

- `Rust` nightly-2023-06-25
- `gcc-arm-none-linux-gnueabi` 2021-07
- `gcc-riscv64gc-unknown-none-elf` 10.2.1
- `gcc-aarch64-none-elf` 4.3.2

A.2.5 Benchmarks

All test examples and benchmarks are included in the artifact.

A.3 Set-up

A.3.1 Installation

To run the artifact in an emulator, QEMU must be installed. The following shell script provides a quick installation method on Ubuntu 22.04:

```
sudo apt update
sudo apt install -y build-essential \
  make git pkg-config libglib2.0-dev \
  zlib1g-dev libpixman-1-dev \
  ninja-build python3 python3-pip \
  libncurses5-dev libspice-server-dev \
  libncursesw5-dev libspice-protocol-dev \
  libcap-dev libvirglrenderer-dev \
  libsdl2-dev libgtk-3-dev libattr1-dev

git clone https://gitlab.com/qemu-project/
    qemu.git
cd qemu
git checkout v6.2.0

./configure --target-list=arm-softmmu,\
aarch64-softmmu,riscv64-softmmu

make -j$(nproc)
sudo make install
```

To compile the artifact, you must install Rust and the required cross-compilation toolchains. Follow these steps:

Install `Rust nightly-2023-06-25`[1].

Install `gcc-arm-none-linux-gnueabi` 4.3.2[2].

Install `riscv64gc-unknown-none-elf-gcc` 10.2.0[3].

Install `aarch64-none-elf-gcc` 2021-07[4].

Install `xfel`[5] to run EKC on the Allwinner D1H board.

A custom SD card flashing tool for Raspberry Pi 4B is included in the artifact package.

Finally, download and extract `RustEKC_src.zip` and `RustEKC_artifact.zip` from Zenodo.

A.3.2 Basic Test

Open the `RustEKC_artifact` directory to access a set of pre-compiled binary files that can be executed directly using QEMU. To run a demo, navigate to the corresponding subdirectory and run one of the following commands:

```
make INFO=1  % Verbose output
# or
make INFO=0  % Minimal output
```

This will produce QEMU output, either with full logging (`INFO=1`) or minimal output (`INFO=0`). Refer to the expected output described in the `Explanation.md` file within each demo folder. This basic test only requires QEMU to be installed on the host machine.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *EKC can be deployed across multiple OS kernels and instruction set architectures (ISAs). This is proven by the experiment (E1) and (E2) described in section 3.1 and Table 4.*

(C2): *EKC can protect user data across various application scenarios. This is proven by the experiment (E1) and (E2) described in section 3.4.*

(C3): *EKC has been successfully deployed on multiple development boards. This is proven by the experiment (E1) and (E2) described in section 3.1 and Table 4.*

A.4.2 Experiments

(E1): *[Compile EKC for different platforms] [1 human-hour]:*

How to: Modify the configuration in the source code and compile the binaries for execution in QEMU.

Preparation: Open `RustEKC_src` and edit the `Makefile`. Adjust the following parameters: `BOARD` specifies the target platform, selectable from `codes/arch`. The `TARGET` variable specifies the Rust compilation target. Typical values include:

- arm32-none-linux-gnueabi
- aarch64-unknown-none
- riscv64gc-unknown-none-elf

Execution: For each configuration, run `make env` followed by `make all` to compile EKC for the selected platform. Use `make run` to launch EKC under QEMU.

Results: EKC will initialize on the selected platform and terminate with a panic, as expected, due to the absence of a payload.

(E2): *[Run EKC with different payloads in QEMU] [1 human-hour]:*

Preparation: Compile each payload following the instructions in its source directory. In the `Makefile`, assign the compiled binary to the `PAYLOAD` variable. Configure `BOARD` and `TARGET` accordingly.

Execution: After updating the configuration, run `make env` and `make all`, then use `make run` to launch EKC with the selected payload on QEMU.

[1] https://www.rust-lang.org/tools/install
[2] https://ftp.gnu.org/gnu/gcc/gcc-4.3.2/
[3] https://github.com/riscv-collab/riscv-gnu-toolchain
[4] https://developer.arm.com/downloads/-/gnu-a/10-3-2021-07
[5] https://github.com/xboot/xfel

Results: EKC will execute with the specified payloads. The output should match the results observed in `RustEKC_artifact`.

(E3): *[Run EKC with payloads on development boards] [2 human-hours]:*
Preparation: Take Raspberry Pi 4B as an example. Recompile both EKC and the payload for the AArch64 Raspberry Pi platform to produce binaries. Use the provided flash tool at `tools/pi-fel-tool` to write the binaries to an SD card, then insert it into the board.
Execution: Connect the board to your computer via a serial cable and open a terminal using PuTTY. Power on the board to observe its output in the PuTTY console.
Results: The output should be consistent with the QEMU results observed in experiment (E2).

A.5 Notes on Reusability

You may deploy your own payload with EKC on a specific platform. To integrate EKC into a custom operating system kernel, follow these steps:

1. **Modify the linker script** (typically `linker.ld`) to set the entry point to the EKC jump address, which is defined in `config.rs`.

2. **Include the EKC API library**, available from the GitHub:

   - For Rust projects, add the `mmi` dependency to `Cargo.toml`.
   - For C/C++ projects, add `libekc/include` to the compiler's header search path, and link against the static library `libekc/build/libmmk_arch.a`.

3. **Invoke EKC API functions** during kernel initialization (usually in `start.S`) to configure memory permissions for the `.text`, `.data`, and `.bss` segments.

   By default, EKC grants full access only to the first memory page. Therefore, the memory permissions for these segments must be configured within the first page during initialization.

4. **Replace existing memory management module (if any).** If your OS contains a memory management system, decouple and remove it. All original function calls should be replaced with EKC API equivalents. This may involve non-trivial code refactoring.

   - If your OS lacks a memory management module (e.g., RTOS), this step can be skipped.
   - If your OS uses a decoupled architecture (e.g., microkernel), integration should be relatively straightforward.

5. **Integrate EKC with the trap handler.** If your OS defines a trap or interrupt handler, modify the instructions that set the interrupt vector register (e.g., `stvec` on RISC-V or `VBAR` on ARM) to route through the EKC API.

6. **Compile your kernel to obtain the binary image.** If compilation succeeds, the output binary can be placed at a designated location such as `payload/your_own_OS.img`.

7. **Run your kernel with EKC.** Update the `Makefile` by setting the `PAYLOAD` to the path of your kernel image. You can then launch the system with EKC and utilize its security features via the provided API.

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/usenixsec2025/.