Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage

Yufei Du, *UNC Chapel Hill and University of Rochester;* Zhuojia Shen, Komail Dharsee, and Jie Zhou, *University of Rochester;* Robert J. Walls, *Worcester Polytechnic Institute;* John Criswell, *University of Rochester*

https://www.usenix.org/conference/usenixsecurity22/presentation/du

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

A Artifact Appendix

A.1 Abstract

Our artifacts include the source code of all components of our Kage implementation, including the LLVM-based [36] compiler, the FreeRTOS-based [6] embedded OS, the microbenchmarks, the macrobenchmark, the binary code scanner, the corresponding libraries, and our scripts to find stitchable gadgets. Our hardware requirements include a host Linux machine and an STM32L475 Discovery board [43]. Our software requirements include Linux, a C/C++ compiler (e.g., Clang, gcc) and associated tools for compiling Clang and LLVM, the OpenSTM32 System Workbench IDE, Python 3, and the pyelftools library. We provide automated evaluation scripts to generate the performance results, code size results, and most of the security evaluation results included in the paper. Specifically, the performance results produced by these artifacts correspond to the results found in Tables 2, 3, 4, 5, 6, and 7 of the paper. Due to minor bug fixes and code structure adjustments, the artifact results will vary slightly from the results presented in the paper, but the key results and the main claims of the paper remain valid.

A.2 Artifact check-list (meta-information)

- **Program: CoreMark [28] (included), Microbenchmarks (included).**
- **Compilation: Our LLVM-based compiler.**
- **Transformations: Our compiler passes (shadow stack, store hardening, and CFI).**
- **Run-time environment: Fedora 35.**
- **Hardware: STM32L475 Discovery board.**
- **Metrics: CoreMark: Iter/s; microbenchmark: cycles; code size: bytes; security: gadgets.**
- **Output: Serial output containing the numerical results.**
- **Experiments: Execute the automated evaluation scripts.**
- **How much disk space required (approximately)?: 5GB.**
- **How much time is needed to prepare workflow (approximately)?: Two hours.**
- **How much time is needed to complete experiments (approximately)?: 20 minutes.**
- **Publicly available?: Yes.**
- **Code licenses: Kage, LLVM compiler, CoreMark: Apache License 2.0; Newlib: GNU General Public License 2; AWS FreeRTOS: MIT License.**
- **Archived?: https://github.com/URSec/Kage commit #195d489**

A.3 Description

A.3.1 How to access

The source code of Kage is publicly available as a GitHub repository: https://github.com/URSec/Kage.

A.3.2 Hardware dependencies

An STM32L475 Discovery board is required. Other STM32 development boards may work but are untested. A Linux x86 host machine is also required in order to build and flash the benchmarks to the board and to read the experimental results.

A.3.3 Software dependencies

We require the host machine to run a Linux distribution. We evaluated Kage using a host machine running the rolling release of Arch Linux, updated in June 2021. We have also tested Kage on Fedora 35.

Our build script uses the manufacturer-provided IDE to build the binaries. Therefore, we require the OpenSTM32 System Workbench IDE to be installed on the host machine. The IDE is publicly available at https://www.openstm32.org/HomePage. Note that users are required to register for a free web account to download the IDE suite.

Our binary code scanner requires Python 3 and the `pyelftools` library.

Finally, our automated evaluation script requires Python 3, the `colorama` Python library, and the `pyserial` Python library.

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

N/A

A.4 Installation

A.4.1 Setting Up Kage on a Local Machine

We provide a detailed guide to install the dependencies and to set up Kage in the `readme.md` document of our GitHub repository.[5] As discussed in Section A.3.2, we require an STM32L475 Discovery board to run the compiled ARMv7-M binaries.

A.5 Experiment workflow

We provide a detailed guide to run the experiments in the `readme.md` document of our GitHub repository.

[5] https://github.com/URSec/Kage

A.6 Evaluation and expected results

A.6.1 Key Results in the Paper

There are three main claims in our paper. First, Kage incurs only minor performance overhead in the macrobenchmark, CoreMark [28], even though some of its components show a more significant overhead in microbenchmarks. Second, Kage incurs acceptable code size overhead. Third, Kage eliminates stitchable code-reuse gadgets.

For the first claim, the key result is that Kage incurs 5.2% mean performance overhead compared to the baseline FreeRTOS [6] in CoreMark. Table 3 in the paper lists the detailed CoreMark results. For the performance overhead of Kage's components, Table 5 and Table 6 in the paper list the microbenchmark results.

For the second claim, the key result is that Kage incurs 49.8% code size overhead compared to the baseline FreeRTOS and 14.2% code size overhead compared to FreeRTOS with MPU enabled, when comparing the CoreMark binaries that use three threads. Table 4 in the paper lists the detailed code size results.

For the third claim, the key result is that, for the CoreMark binaries that use three threads, Kage significantly reduces the number of reachable code-reuse gadgets and eliminates stitchable gadgets. Table 7 in the paper lists the detailed code-reuse gadget results for the security evaluation.

A.6.2 Reproducing the Results

As Section A.5 states, we provide a detailed guide to run the automated scripts in the `readme.md` document of our repository. This document includes detailed steps to build our toolchain, generate the performance and code size results, and generate the security evaluation results.

Because we discovered and fixed additional minor bugs in our workflow after we submitted the paper, and because we adjusted the source code to enable automated evaluation, the artifact results will include minor differences from the original results included in the paper. For the microbenchmarks, the results may include variations up to 25 cycles. For the performance evaluation of CoreMark, the results may include variation up to 0.05 Iter/s. For the code size evaluation of CoreMark, the code size of the untrusted code includes a difference of 16 bytes. Finally, for the security evaluation, the number of reachable gadgets includes a difference of one gadget. These differences do not significantly impact the key results and claims of the paper.

We note that our automated evaluation scripts produce a larger set of performance metrics than the set we included in the paper. For example, Table 6 in the paper shows the microbenchmark results for FreeRTOS, FreeRTOS with MPU enabled, and Kage. Our evaluation script, `run-benchmarks.py`, also shows the microbenchmark results for Kage's OS mechanisms. Similarly, for code size, the paper only includes the results of the CoreMark binaries that use three threads while the script also shows the code size results for the microbenchmark binaries as well as other binaries of CoreMark that use one or two threads.

Finally, while our scripts generate most of the results automatically, our security evaluation script, `run-gadget.py`, cannot automatically generate the number of stitchable gadgets because the process requires manual inspection. Section 6.2 of our paper explains how we analyze the reachable gadgets to determine if they are stitchable.

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.