USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

KINTSUGI: Secure Hotpatching for Code-Shadowing Real-Time Embedded Systems

Philipp Mackensen, *Ruhr University Bochum;* Christian Niesler, *University of Duisburg-Essen;* Roberto Blanco, *Eindhoven University of Technology/MPI-SP;* Lucas Davi, *University of Duisburg-Essen;* Veelasha Moonsamy, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity25/presentation/mackensen

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: Kintsugi: Secure Hotpatching for Code-Shadowing Real-Time Embedded Systems

Philipp Mackensen[*], Christian Niesler[†], Roberto Blanco[‡], Lucas Davi[†], Veelasha Moonsamy[*]
[*]Ruhr University Bochum, [†]University of Duisburg-Essen, [‡]Eindhoven University of Technology/MPI-SP

A Artifact Appendix

A.1 Abstract

This artifact appendix provides the source code for the prototype implementation of KINTSUGI, all source codes for the experiments described in the paper, and a Dockerfile that enables the creation of an image containing all the prerequisites necessary for successfully evaluating KINTSUGI. For each performance experiment, we provide the measurement files used as source for the tables and plots in the paper's evaluation. Additionally, for each of the 10 real-world CVEs and the security experiments, we provide logs reflecting the expected outputs. We made our artifact openly accessible and focused on an easy-to-evaluate solution. Once the Docker image is built, one can run each experiment using the provided scripts.

A.2 Description & Requirements

To run all artifacts, you must have an nRF52840-DK board, which has been used to test all experiments. All experiments were conducted (building/flashing/evaluating) on a host machine running Kubuntu 22.04.4 LTS x86/x64, featuring a 12th Gen Intel i7-1206P (16) processor. We provide a docker file that contains all the necessary commands to successfully build an environment that allows to build, flash, debug and evaluate KINTSUGI.

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, there are no security risks associated with running our artifacts on the host machine of any evaluators. However, we must note that the Docker is run with `--privileged` due to the need to flash the device's firmware.

A.2.2 How to access

The artifacts are publicly available at https://doi.org/10.5281/zenodo.15592036.

A.2.3 Hardware dependencies

Nordic *NRF52840-DK* development board: Required to run the firmware and reproduce the core results.

Host machine (Linux): Used to build and flash firmware, interface with the *NRF52840-DK*, collect results and output tables / plots. A standard desktop or laptop with $\approx 8GB$ of RAM and $\approx 64GB$ of storage, with USB and internet access is sufficient.

A.2.4 Software dependencies

We recommend using Kubuntu 22.04.4 LTS, as all experiments have been successfully tested on this distribution. We have also verified that the experiments work on a virtual machine running Ubuntu 22.04 LTS. Additionally, it is necessary to have bash installed (which comes standard with Linux) and Docker, which will handle the installation of all other prerequisites.

A.2.5 Benchmarks

None.

A.3 Set-up

We provide a Dockerfile that allows to build a docker image in which everything to perform our experimental evaluation is gets installed.

A.3.1 Installation

To begin, install the Docker engine on Linux: https://docs.docker.com/engine/install/. Next, download the KINTSUGI artifact, extract it, and run the `build.sh` script to set up the Docker image. Building the Docker image will take approximately 30 minutes and it will occupy approximately $28GB$ of storage.

A.3.2 Basic Test

To verify that Docker can successfully build every experiment, please run the `verify.sh` script. This script will check that

all experiments are built correctly; if the Docker image is missing, it will build it as well. For each experiment, the script will output whether the build was successful or not. It takes approximately 8 min to verify everything.

Note, that this test does not require the existence of the hardware as this is a pure software test.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): KINTSUGI introduces minimal performance overhead across the hotpatching process. Validation and scheduling have consistent measurement times, independent of hotpatch sizes. This was proven by experiment (**E1**) and explained in *Section 6.1*, "Manager" (Table 2), for hotpatch sizes based on `RapidPatch` and `AutoPatch`. An extended version is provided in *Appendix D* (*Table 5*), and hotpatch sizes from KINTSUGI are discussed in *Section 6.4*, with results in *Appendix E* (*Table 6*).

(C2): The performance overhead introduced by KINTSUGI into the context switch of an RTOS is minimal. This is proven by experiment (**E2**) and described in *Section 6.1*, "Guard & Applicator" (*Table 3*).

(C3): Successively processing and applying hotpatches scales linearly with the number of hotpatches, as demonstrated with up to 64 hotpatches. This was proven by experiment (**E3**) and is described in *Section 6.2*, with the results shown in *Figure 5*.

(C4): The memory overhead introduced by KINTSUGI grows predictably with the maximum hotpatch size and slot count. The overhead ranges from an average of $3.5KB$ to $42KB$. This is proven by experiment (**E4**) and described in *Section 6.3*, with the shown in *Figure 6*.

(C5): KINTSUGI is capable of hotpatch real-world vulnerabilities as we demonstrated with 10 CVEs. This is proven by experiment (**E5**) and is described in *Section 6.4*. Resulting hotpatch sizes are shown in *Table 4*.

(C6): KINTSUGI is able to prevent tampering attacks from an adversary before, during and after the hotpatching process. This is proven by experiment (**E6**) and is described in *Section 8*.

A.4.2 Experiments

We provide evaluation scripts in the directory `evaluation_scripts` which are interactive and include descriptions of their working and how to correctly use them. If a script automatically produces outputs (i.e., the performance experiments) they will be stored in the folder `measurement_results` under their corresponding subdirectories. We clearly mark experiments that do *not* automatically produce outputs. Those experiments require access to the UART of the board through, e.g., `screen`, `minicom` or `puTTY` with a baudrate of 115200.

(E1): [`micro-benchmarks.sh`] [$\approx$ 6 computer-hours]
Execute the *Micro-Benchmarking* experiments of the *Manager* to prove (**C1**), measuring how each component inside of the *Manager* takes to execute.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.
Results: Upon execution, the user will be asked to select the hotpatch sizes to measure. They can choose between *RapidPatch & AutoPatch* (Section 6.1) or *Kintsugi* (Section 6.4). Measuring a single configuration takes approximately 100 seconds, hence why the script will stay at "*Reading UART output*" for a while. All results will be stored in `micro-benchmarks/manager` with raw measurements being stored in `measurements` and the resulting tables in `output`. The measurements in the table should be as close as possible to those claimed in (**C1**).

(E2): [`context-switch.sh`] [$\approx$ 2 computer-minutes]
Execute the *Micro-Benchmarking* experiments of the *Guard & Applicator* focusing on the *Context-Switch* of the RTOS to prove (**C2**), demonstrating how much time KINTSUGI adds to the context-switch of the RTOS.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.
Results: All results will be stored in `micro-benchmarks/context-switch` with raw measurements being stored in `measurements` and the resulting tables in `output`. The measurements in the table should be as close as possible to those claimed in (**C2**).

(E3): [`scalability.sh`] [$\approx$ 4 computer-hours]
Execute the *Hotpatching Scalability* experiments to prove (**C3**), measuring KINTSUGI's performance when applying multiple consecutive hotpatches, demonstrating that it grows linearly in the number of hotpatches.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.
Results: All results will be stored in `scalability` with raw measurements being stored in `measurements` and the resulting plot will be stored in `output`. The plot should show linear growth in the dimension of number of hotpatches and be as close as possible to the results claimed in (**C3**).

(E4): [`resource-utilization.sh`] [$\approx$ 3 computer-hours]
Execute the *Resource Utilization / Memory Overhead* experiments to prove (**C4**), measuring the memory overhead introduced by KINTSUGI under different parameter configurations of hotpatch counts and sizes.
Automatic Output: *Yes.*
Execution: The measurement does not require user interaction.

Results: All results will be stored in `resource-utilization` with raw measurements being stored in `measurements` and the resulting plot will be stored in `output`. The plot should show an increase in memory overhead in both dimensions and be as close as possible to the results claimed in (**C4**).

(E5): [`realworld-cves.sh`] [≈ 30 minutes total]
Perform the *Real-World Hotpatching* experiments to prove (**C5**), demonstrating that KINTSUGI is capable in hotpatching real-world CVEs.
Automatic Output: *No.*
Preparation: Connect to the device's UART as described above. Typically the access will be over `ttyACM0` or `ttyACM1`.
Execution: Upon executing the script the user will be asked to input a number between 1 and 10 to decide which CVE to execute on the board.
Results: The results differ for each CVE experiment. We provide details about expected outputs in `experiments/realworld_cves/README.md`. This file also reflects all the outputs that we have obtained during our experimental evaluation.

(E6): [`security.sh`] [≈ 10 minutes total]
Perform the *Security* experiments to prove (**C6**), demonstrating that KINTSUGI is resistant against adversaries *before*, *during* and *after* the hotpatching process.
Automatic Output: *No.*
Preparation: Connect to the device's UART as described above. Typically the access will be over `ttyACM0` or `ttyACM1`.
Execution: Upon executing the script the user will be asked to input a number between 1 and 3 to decide which of the three security experiments to run.
Results: The results differ for each security experiment. We provide details about expected outputs in `experiments/security/README.md`. This file also reflects all the outputs that we have obtained during our experimental evaluation.

A.5 Notes on Reusability

We provide additional details on how to properly integrate the tool into the *Zephyr* and *FreeRTOS* RTOSes in the `example_rtos_integration` folder. We provide the necessary files that must be adapted to the respective RTOS and instructions on how to do so manually. In our Docker, we require patches to modify the corresponding Git repositories. Therefore, the git-patches for the RTOSes can potentially be used immediately.

Integrating KINTSUGI into an RTOS should be straightforward. To do this, create a task for the *Manager* and modify the context switch to check if the *previous* or *next* task is the *Manager*. Additionally, call the *Guard & Applicator* before restoring the execution context of the next task.

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/.