USENIX Security '26 Artifact Appendix: ARTO: Efficient Execution Integrity Attestation for Real-Time Operation of Cyber-Physical Systems

Ruizhe Zhao[1], Cong Sun[1*], Zongzhen Li[1], Tiantian Wang[2], and Yunbo Wang[1]
[1]State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University
[2]Khoury College of Computer Sciences, Northeastern University, San Jose, CA, USA
[1]{ruizhezhao@stu.xidian.edu.cn, suncong@xidian.edu.cn}, [2]wang.tianti@northeastern.edu

A Artifact Appendix

A.1 Abstract

We present ARTO, a control-flow and data-flow protection approach that integrates control-flow attestation to enforce execution integrity for the real-time operation of cyber-physical systems, especially autopilot systems. In this artifact, we demonstrate ARTO's functionality by presenting its complete pipeline, including both offline and online phases, for the critical operation `update_xy_controller` of the ArduCopter autopilot. In the offline phase, we present 1) the prover construction with ARTO's compiler toolchain, 2) the offline measurement generation with ARTO's simulated executor, and 3) the prover-side database embedding. In the online phase, we present 4) the prover-side local control-flow verification and equivalence-class-based data-flow protection at runtime, 5) the prover-side runtime evidence collection for the protected operations, and 6) the verifier's control-flow integrity verification based on the offline control-flow evidence.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is designed to be conducted on an isolated virtual machine provided by the authors, which will not pose a risk to any user. To avoid the risk of misuse of the control-flow hijacking and data-only attacks described in this work, we decided not to include the attacks in the artifact. This artifact evaluation will be conducted on normal executions and on debugger-based attack injections of the protected autopilot system. Moreover, ARTO's public GitHub repository does not contain any attack that could lead to similar misuse. Besides, this artifact does not collect user data or process personal information; therefore, there is no privacy leak or misuse of user data.

*Corresponding author.

A.2.2 How to access

The source code of ARTO is available at https://github.com/zz-fz-john/ARTO-Project. For convenience, we recommend that this artifact evaluation use the pre-configured virtual machine image released at https://doi.org/10.5281/zenodo.20355901.

A.2.3 Hardware dependencies

The experiments of this artifact do not require any specialized hardware. Our artifact has been tested on a Linux workstation equipped with the Intel Xeon CPU and 64 GB of RAM. Therefore, we recommend at least 64 GB of RAM and 90 GB of storage for the artifact evaluation. In a typical application scenario, an ARTO-protected ArduCopter should run on a Raspberry Pi 3. However, for the purpose of demonstrating only ARTO's functionality, in this artifact evaluation, we use QEMU in our VM as the autopilot's execution environment. More instructions for running the ARTO-protected ArduCopter on a Raspberry Pi 3 are provided in the GitHub repository.

A.2.4 Software dependencies

Our virtual machine for artifact evaluation requires Oracle VirtualBox (v7.2.6), which has been tested with 32 GB of guest memory allocated. In the VM image, the environment for ARTO, as well as ARTO's key components, has been properly deployed, including

- Ubuntu 20.04 LTS guest operating system,
- QEMU 8.2.10 for emulating ArduCopter's execution,
- The compiler toolchain customized on LLVM 16.0.0,
- The simulated executor for offline measurement generation,
- The rewriter for embedding the hash database into the prover binary,
- The remote verifier for control-flow attestation.

We use `gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf` for ARM cross-compilation. The VM also provides the source code of the target autopilot (ArduPilot 4.3.0).

A.2.5 Benchmarks

None.

A.3 Set-up

Importing and launching the ARTO VM. Then, the starting point for this artifact evaluation is to use ARTO's key components to build the ARTO-protected ArduCopter (prover program) and to enforce runtime control-flow and data-flow protection, as well as control-flow attestation. In this case, the reviewers can skip all the steps in Section A.3.1.

Users who intend to deploy ARTO from the very beginning can start with a clean Ubuntu system and follow the instructions in Section A.3.1 or in ARTO's GitHub repository for the complete deployment procedure.

A.3.1 Installation

Download the repository. We recommend installing ARTO in the user's home directory to avoid configuration errors. In the following instructions, we assume the repository is located at `$HOME/ARTO`.

```
$ sudo apt install -y git curl
$ cd $HOME
$ git clone https://github.com/zz-fz-john/ARTO-Project ARTO
$ cd ARTO
```

Dependency Installation. ARTO depends on several external libraries, toolchains, and Python packages. To simplify the installation process, we provide requirement files that summarize the required dependencies. Reviewers can install these dependencies using the following commands:

```
$ cd $HOME/ARTO
$ ./build_requrement.sh
```

Build Compile Toolchain. To simplify reproduction, we provide an automated script that installs and builds ARTO's compilation toolchain. The script prepares the necessary dependencies, including `gold`, `binutils`, LLVM, and SVF. Users can run the script from the ARTO's root directory:

```
$ cd $HOME/ARTO
$ chmod +x build_llvm.sh
$ ./build_llvm.sh
$ chmod +x build_svf.sh
$ ./build_svf.sh
```

A.3.2 Basic Test

The installation of the ARTO-customized LLVM can be confirmed by invoking `llvm-config` from the LLVM build directory.

```
$ cd \$HOME/ARTO
$ ./llvm-16.0/llvm-project-16.0.0/build/bin/llvm-config \
    --version
```

A successful installation should return `16.0.0`.

A.4 Evaluation workflow

A.4.1 Major Claims

In the following claims, C1 declares ARTO's offline functionality, while C2 and C3 describe the online functionality.

(C1): Offline functionality. ARTO's compiler toolchain can automatically analyze and instrument the protected operations in real-time cyber-physical systems to construct the prover program. ARTO's simulated executor can generate the offline measurements. ARTO's rewriter can embed the hash database into the final prover binary.

(C2): Normal execution and attestation. ARTO can collect compact runtime evidence of the protected operations at runtime. ARTO's remote verifier can verify the prover's execution integrity based on the control-flow offline measurement collected by the simulated executor.

(C3): Attack detection and diagnosis. ARTO can perform local control-flow verification, equivalence-class-based data-flow protection, and remote control-flow violation diagnosis at runtime.

A.4.2 Experiments

(E1): Offline functionality [15 compute-minutes + 5GB disk]:

How to: Run the following commands.

```
$ cd $HOME/ARTO
$ ./compile_0_step.txt
$ ./compile_1_step.txt
$ ./compile_2_step.txt
$ ./compile_3_step.txt
$ ./compile_4_step.txt
```

These steps insert the local verify engine, use may-alias analysis to group variables into equivalence classes, and instrument their def- and use-sites with the definition-table update or checking operation to enforce execution integrity. This procedure also identifies control-flow-related conditional branches within the critical operation.

```
$ ./compile_5_step.txt
```

`compile_5_step` invokes ARTO's LLVM backend pass to insert control-flow measurement and SFI-related code at the assembly level.

```
$ ./compile_6_step.txt
```

`compile_6_step` launches the simulated executor to generate the hash database and hash-trace map.

```
$ ./compile_7_step.txt
```

`compile_7_step` rewrites the ArduCopter binary to embed the hash database into its reserved section.

Results: The resulting binary is located at `$HOME/ARTO/ardupilot/build/SITL_arm_linux_gnueabihf/arducopter_ARTO`. To confirm that ARTO has inserted the local verify engine and data-flow protection functionality, reviewers can inspect the assembly code of ArduCopter:

```
$ arm-linux-gnueabihf-objdump -s -d \
```

```
arducopter_ARTO > arducopter_ARTO.S
```

In the assembly, calls to `checkpoint` serve as local control-flow integrity verifications. The calls to `def_check` and `use_check` implement the runtime data-flow protection for security-critical definitions and uses. The presence of these calls confirms that ARTO's control-flow and data-flow protection mechanisms have been successfully inserted into the target ArduCopter binary.

(E2): Normal execution, evidence collection, and remote verification [10 compute-minutes]:

How to: Execute a takeoff mission of the ARTO-protected ArduCopter in QEMU. During execution, the instrumented code automatically records the runtime control-flow events of the critical operation `update_xy_controller`. Conduct the following steps:

1. On the host VM, enter the QEMU environment, configure the prover, and start QEMU:
   ```
   $ cd ~/qemu-emlid-system
   $ ./config.txt
   $ ./start.sh
   ```
   Log in to the guest system in QEMU with the username `pi` and the password `raspberry`.
2. Run the instrumented ArduCopter and execute the takeoff mission. In the QEMU guest system, launch ArduCopter:
   ```
   $ ./arducopter_ARTO -S --model + --speedup 1 \
       --defaults ./copter.parm -I0
   ```
   Open another terminal in the host VM and start MAVProxy:
   ```
   $ mavproxy.py --master tcp:127.0.0.1:5760 \
       --out 127.0.0.1:14550 --console --map
   ```
   Wait until the message `AP: EKF3 IMU0 is using GPS` appears in the MAVProxy console. Then, execute the following commands in the MAVProxy console:
   ```
   STABILIZE> mode guided
   GUIDED> arm throttle
   GUIDED> takeoff 15
   ```
   Allow ArduCopter to run for approximately 30 seconds after takeoff. Then, press `Ctrl+C` in the QEMU terminal to terminate ArduCopter and complete the online evidence-collection process.
3. Run the remote verifier in the host VM.
   ```
   $ cd $HOME/ARTO/ardupilot
   $ scp -P 10022 \
       pi@127.0.0.1:/home/pi/branch_trace.txt \
       $HOME/ARTO/ardupilot
   $ scp -P 10022 \
       pi@127.0.0.1:/home/pi/hash_value.txt \
       $HOME/ARTO/ardupilot
   $ ./remote_verify.sh
   ```

Results: Upon the completion of the remote verifier's verification, the host VM terminal displays `All segments verification succeeded!`.

(E3): Local attack detection against simulated data manipulation attack injected by debugger [5 compute-minutes]:

How to: We use a GDB script to modify the runtime state of ARTO-protected ArduCopter, thereby simulating a data-manipulation attack. Specifically, we modify the runtime value of the critical variable `curr_pos` immediately before the use-site checking function is invoked. Run the following commands to launch ArduCopter in QEMU:

```
$ gdb --args ./arducopter_ARTO  -S --model + \
    --speedup 1 --defaults ./copter.parm -I0
(gdb) source modify_r5.gdb
(gdb) r
```

Then, follow the commands of **(E2)** in the host VM to start MAVProxy and control the drone to take off.

Results: The execution of ArduCopter will then be terminated by ARTO's data-flow protection. Reviewers will observe ArduCopter's termination in the QEMU terminal, together with the error message `use_check data flow error!`.

(E4): Local attack detection and remote diagnosis against simulated control-flow hijacking attack injected by debugger [5 compute-minutes]:

How to: We use a GDB script to hijack ArduCopter's runtime control flow. Specifically, in `Update_xy_controller`, we modify the return target of the last call to function `xy()` and redirect the return to another code location. First, run the commands in QEMU:

```
$ gdb --args ./arducopter_ARTO  -S --model + \
    --speedup 1 --defaults ./copter.parm -I0
(gdb) source redirect_return.gdb
(gdb) r
```

Then, follow the **(E2)** commands in the host VM to start MAVProxy and control the drone to take off. Reviewers will observe ArduCopter's termination in the QEMU terminal, demonstrating ARTO's local CFI enforcement can effectively identify the attack and stop the vulnerable execution. Furthermore, we run the remote verifier at the host VM to validate the collected execution evidence.

```
$ cd $HOME/ARTO/ardupilot
$ scp -P \
    10022 pi@127.0.0.1:/home/pi/branch_trace.txt \
    $HOME/ARTO/ardupilot
$ scp -P 10022 pi@127.0.0.1:/home/pi/hash_value.txt \
    $HOME/ARTO/ardupilot
$ scp -P 10022 \
    pi@127.0.0.1:/home/pi/target_buffer.txt \
    $HOME/ARTO/ardupilot
$ ./remote_verify_attack.sh
```

Results: The remote verifier reconstructs the execution path from the collected evidence and identifies the point at which the control-flow hijacking occurred.

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/usenixsec2026/.