



USENIX Security '24 Artifact Appendix: Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers

Marton Bognar, Cas Magnus, Frank Piessens, Jo Van Bulck

DistriNet, KU Leuven, 3001 Leuven, Belgium

A Artifact Appendix

A.1 Abstract

This artifact provides source code for the individual attack primitives and end-to-end attack scenarios that can be run on off-the-shelf TI MSP430 microcontrollers with Intellectual Property Encapsulation (IPE) support. We also provide source code to reproduce evaluation results for the software mitigation framework, as well as the openMSP430/Sancus-based hardware mitigation against controlled `call` corruption.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact demonstrates attacks on real-world TI MSP430 microcontrollers. There are no security risks for evaluators, as the only code executed on the host machine is compilation of the example projects using standard tools. All attack code runs locally on the specific device under test.

The attack code provided in this artifact is solely intended for reproduction of our results. Any uses of these results on real-world microcontrollers should be conducted responsibly.

A.2.2 How to access

The artifact files are accessible in the following repository:
<https://github.com/martonbognar/ipe-exposure/tree/usenix24-artifact>.

A.2.3 Hardware dependencies

Our repository contains proof-of-concept code for four TI development boards: MSP-EXP430FR5994, MSP-EXP430FR5969, EVM430-FR6047, MSP-EXP430FR6989.

A.2.4 Software dependencies

Compiling and running our artifact requires the following software, available on all major operating systems:

- Code Composer Studio (CCS) integrated development environment (IDE). Can be downloaded from the TI website (we used regular CCSTUDIO version 12.6.0): <https://www.ti.com/tool/CCSTUDIO#downloads>.
- Python 3 with `pycparser` (v2.21), `pycparserext` (v2021.1), and `pyelftools` (v0.29) libraries.
- For the Sancus experiment: `gcc-msp430`, `cmake`, `iverilog` (e.g., via standard Ubuntu packages).

A.2.5 Benchmarks

No external benchmarks were used for our evaluation.

A.3 Set-up

A.3.1 Installation

First, clone the artifact repository:

```
$ git clone --recurse-submodules \
--branch usenix24-artifact \
https://github.com/martonbognar/ipe-exposure
```

Then, install the software dependencies from above.

1. Download CCS via the link above and proceed with the installation, then install the necessary drivers:

```
$ wget https://dr-download.ti.com/software-
  ↳ development/ide-configuration-compiler-
  ↳ or-debugger/MD-J1VdearkvK/12.6.0/CCS12
  ↳ .6.0.00008_linux-x64.tar.gz
$ tar -xvzf CCS12.6.0.00008_linux-x64.tar.gz
$ cd CCS12.6.0.00008_linux-x64/
$ ./ccs_setup_12.6.0.00008.run # choose ~/ti
  ↳ as installation directory
$ cd ~/ti/ccs1260/ccs/install_scripts/
$ sudo ./install_drivers.sh
```

2. Install the required Python 3 dependencies:



```
$ cd ipe-exposure/05_framework/framework
$ pip install -r requirements.txt --no-deps
```

3. Install Sancus dependencies:




```
$ sudo apt install gcc-msp430 cmake iverilog
```

A.3.2 Basic Test


To test out the setup, we provide a simple “hello world” CCS project that can be run on the target MSP430 board with IPE as per the steps below (please refer to the repository’s top-level `README.md` for detailed screenshots and troubleshooting):

1. Launch the CCS IDE and create a new workspace in an empty directory when prompted on startup.
2. In CCS, choose *File* \triangleright *Open Projects from File System*. Now select the directory `00_helloworld` in the cloned `ipe-exposure` repository.
3. With the microcontroller connected to the system, start the debug session (F11,  \vee).
4. After successfully launching the debug session, resume the program (F8, ).
5. Expected output should now appear in the Console pane:

```
-----  
Reading secret from main: 1234 (IPE disabled)  
Reading secret from IPE : 1234
```

6. In order to activate IPE, the device needs a hard reset. For this, first pause the running debug session (Alt+F8, ), then select “Hard Reset” from the dropdown next to the Reset button ( \vee):
7. The microcontroller will now reboot with IPE enabled. After resuming the program (F8, ), you should see the following output in the Console pane:

```
-----  
Reading secret from main: 3fff (IPE enabled)  
Reading secret from IPE : 1234
```

8. The CCS debug session can now be terminated via the stop button (Ctrl+F2, ).

A.4 Evaluation workflow

A.4.1 Major Claims

- C1** The attack primitives from §3 directly or indirectly break confidentiality and integrity of IPE-protected memory, as summarized in Table 1 (cf. E1).
- C2** The covert channels from §3.4 enable deterministic leakage with performance as reported in Table 4 (cf. E2).
- C3** The three end-to-end attack scenarios from §4 can be reproduced, showing successful corruption or leakage of secrets from complete programs (cf. E3).
- C4** Buffering the program counter register as a hardware mitigation prevents similar attacks on openMSP430/Sancus, as explained in §3.1 (cf. E4).
- C5** Our software mitigation framework blocks all architectural attacks demonstrated in this paper (cf. E5).
- C6** The micro- and macrobenchmarks in §6.4 (Tables 5 and 7) describe the software framework’s overhead (cf. E6).

A.4.2 Experiments

E1: *[Attack primitives] [40 human-minutes]*: Reproduction of three architectural and three side-channel attack primitives by running minimal proof-of-concept programs, one per primitive, in standalone CCS projects.

Preparation: Launch CCS and open all relevant projects under the `01_attack_primitives` directory.

Execution: For every project individually, analogous to §A.3.2: launch the debug session, trigger a hard reset (to activate IPE), then run the code.

Results: Refer to the README of each project for the expected output, which should match the console. These projects demonstrate the effectiveness of the attack primitives and, thus, validate claim C1.

E2: *[Covert channels] [20 human-minutes]*: Reproduction of the three covert channel setups.

Preparation: Launch CCS and open all projects under the `02_covert_channel` directory.

Execution: For every project individually, analogous to §A.3.2: launch the debug session, trigger a hard reset (to activate IPE), then run the code.

Results: Refer to the README of each project for the expected output, which should match the console and the numbers in the second column of Table 4. These projects demonstrate the presence of the covert channels and their measured performance, validating claim C2.

E3: *[End-to-end attacks] [20 human-minutes]*: Reproduction of end-to-end attacks.

Preparation: Launch CCS and open all relevant projects under the `03_end_to_end_attacks` directory.

Execution: For every project individually, analogous to §A.3.2: launch the debug session, trigger a hard reset (to activate IPE), then run the code. Note: for the `init_struct_overwrite` exploit, two successive hard resets are required (see the corresponding README).

Results: Refer to the README of each project for the expected output, which should match the console. These projects demonstrate the effectiveness of the attacks and, thus, validate claim C3.

E4: *[Sancus defense] [10 human-minutes]*: Reproduction of the existing hardware mitigation preventing controlled `call` corruption on Sancus. The cycle-accurate openMSP430 Verilog simulation shows that the attack fails on the original Sancus, but succeeds after deliberately omitting the program counter buffering.

Preparation: Make sure the `sancus-core` git submodule is initialized (execute `git submodule init; git submodule update` if needed).

Execution: Run the `run-sancus-eval.sh` script in the `04_sancus_exploit/sancus-exploit/` directory. This will perform all necessary steps for this experiment.

Results: The script will first run a controlled `call` corruption attack against the upstream version of Sancus.

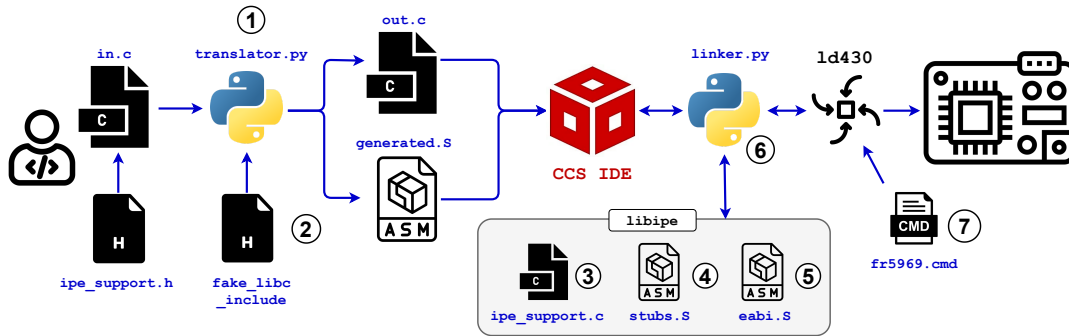


Figure 1: Overview of the general workflow for the software mitigation framework.

This attack should result in a memory violation error, without overwriting the secret value. Next, the script will apply a minimal patch that removes the buffered program counter. After running the same attack again, no memory violation will occur, and the secret value will be overwritten, validating claim C4.

E5: [Framework security] [40 human-minutes]: Demonstration of the mitigation framework’s security by recompiling and running a vulnerable example project.

Preparation: Launch CCS and open the `demo_all` project under the `05_framework/security_eval/` directory. Now execute `run.sh` in that same directory to apply the framework (cf. Figure 1) on the vulnerable application and generate a new `demo_all_mitigated` project. Open this new project in CCS as well.

Execution: Run both the vulnerable and the mitigated projects and examine the `fail_code` and `public` variables using the CCS debugger, as shown in the screenshots of the README file. Repeat this three times, for all values of the `attack` global variable in `main.c`.

Results: The values will show that while in the unprotected version all attacks successfully change and leak values from the IPE region, applying our framework disables these attacks, validating claim C5.

E6: [Benchmarks] [40 human-minutes]: Reproduction of the micro- and macrobenchmarks by measuring the timing of projects secured by the framework.

Preparation: Follow the steps in the README in the `06_benchmarks` directory to simultaneously debug a timer and a benchmark project on two connected boards.

Execution: Always start a new debug session of the `timer` project and resume its execution first before launching and resuming the benchmarked program in a separate CCS instance. After successfully collecting 100 measurements, the `timer` project prints the collected numbers in a comma-separated value format to the console, which can be copied verbatim into a `.csv` file. Collect microbenchmark measurements for a `software-bor/bor_timing_*/` project, depending on the evaluation target. Next, collect macrobenchmark

measurements for both the `hmac/base_attestation` and `hmac/translated_attestation` projects.

Results: Use the `measurements/calculator.py` script to compute the mean and standard deviation for the collected `.csv` files. These values should be similar to those reported in the paper (small deviations are expected), showing the limited overhead of our defense and validating claim C6.

A.5 Notes on Reusability

Based on the `05_framework/security-eval` and `06_benchmarks/hmac` examples, our software mitigation framework (cf. Figure 1) could be applied to other programs, collecting additional evidence for its overhead and effectiveness. In future work, the framework could also be improved further to support a wider range of programs.

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/usenixsec2024/>.