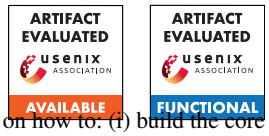


# A Artefact Appendix

## A.1 Abstract

Pistis artefact is a set of source files and scripts that can be evaluated partially on a standard Linux environment (local evaluation), and partially with the support of a MSP430F5529LP micro controller unit (MCU). However, we provide the reviewers with an SSH access to a VM connected to one of such board (remote evaluation). The VM is shared between the reviewers and does not allow multiple users to interact with the MCU at the same time. The reviewers are thus asked for their collaboration in sharing such VM instance. For the local evaluation, the reviewers will be asked to compile the core of Pistis and use the available scripts to compile the user-applications. For the remote evaluation, using the VM, they will be asked to check Pistis at runtime, debugging its execution using a GUI-based IDE.

## A.2 artefact check-list (meta-information)

- **Program:** TI MSP430 Benchmark, custom test bench
- **Compilation:** msp430-gcc-9.2.0.50, included, public
- **Transformations:** python-script
- **Run-time environment:** Linux, non-root
- **Hardware:** x86_64 Machine, (optional) MSP430F5529LP
- **Output:** console, graphical, interactive
- **Experiments:** Python scripts, bash scripts, CodeComposerStudio (CCS), debugging
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 2 hours
- **Publicly available (explicitly provide evolving version reference)?:** `https://github.com/MicheleGrisafi/PISTIS_AE`
- **Code licenses (if publicly available)?:** The 3-Clause BSD License
- **Archived (explicitly provide DOI or stable reference)?:** `https://github.com/MicheleGrisafi/PISTIS_AE/releases/tag/Artefact.v1`

## A.3 Description

Pistis is a Trusted Execution Environment (TEE) developed for MSP4305529 Micro Controller Units (MCUs). Being a fully software based TEE, its features are many spanning a complex software structure. Although we provide the entirety of Pistis, with all of its modules and test applications (as presented in the papers), we only provide instructions on how to evaluate part of it. This is due to the complex and time-demanding nature of a complete evaluation, which would also require a MSP4305529LP MCU.

In particular, we provide instructions on how to: (i) build the core of Pistis, (ii) use the custom toolchain to compile applications, (iii) check the run-time verification process, (iv) check the run-time memory protection. While some of these operations can be performed with a local environment (based on Ubuntu 20.04), others require SSH access to a shared VM connected to a MSP4305529LP board. This access must be shared between any reviewer who cannot interact concurrently with the board. The local evaluation will only require to run a few CLI commands, while the remote evaluation will require the reviewers to operate on an Eclipse-based IDE and its debugger. To facilitate the operations, we provide a few video tutorials.

### A.3.1 How to access

The artefact can be downloaded from the official artefact github repository: `https://github.com/MicheleGrisafi/PISTIS_AE`. It is worth noting that this repository is based on the official repository (link in the official paper). Given the strict hardware requirements for the evaluation of this artefact, we prepared a virtual machine (with ssh access) connected to a single MCU, the one we used in our experiments. Although this setting poses some limitations to the reviewers, it can be used to have a deeper inspection on how Pistis functions.

In order to access the VM we set up, the following command should be executed on a local graphical-based Linux environment: *no more available*[1]. This will establish a two-hops ssh connection to the private VM passing through a public VPS. The passwords for the two hops, which will be required upon each connection, are the following: `Pistis1940` for the pistisAE user (first hop) and `pistis` for the pistis user (the VM). The `-Y` option enables the forwarding of the graphical environment, thus allowing the reviewers to visualise on their own machine any GUI lunched on the remote machine.

### A.3.2 Hardware dependencies

In order to compile our binaries, an x86_64 Linux based machine should be used. Optionally, a MSP430F5529LP MCU can be used to perform locally also the remote evaluation. Nevertheless, this artefact provides the reviewer with a single shared VM instance connected to one of such MCUs. This will help the reviewers to execute code on the board. As a consequence, we proceed to describe two environment: a local environment (**local**) for the MCU-independent tests, and an MCU-connected environment (**remote**) for all the other tests.

### A.3.3 Software dependencies

To propose a baseline for the artefact evaluation, we base our experiments on a machine running Ubuntu Desktop 20.04. On the local environment, the following packages are required:

- make, python3, git
- *any code editor*

Notable is that some of them might already be included with standard Linux-based OSs.

[1]The maintenance of a remote environment is expensive. We ask any interested reader in either following the video tutorials or to follow the official instructions on the official github repository.

On the remote environment, the following software is required:

- libc6-i386 python2.7-dev libtinfo5 libusb-dev libgconf-2-4 python3-pip
- *any code editor*
- Code Composer Studio IDE (CCS)

The listed elements are however already installed on the remote machine. Still, we provide the setup instructions for the reviewers to set up their own remote environment in case they had an available MCU.

## A.4 Installation

**Local Environment** We leave the installation of a valid Ubuntu 20.04 desktop distribution to the reviewers. This is trivial due to the multitude of available tutorials online. Given that our fresh and minimal installation of Ubuntu comes with some packages already installed, we only perform the steps in Listing 1. These steps install the required packages, fetch the github repository and create an alias for a tool required in the later evaluation.

```
1 $ sudo apt install git make
2 $ cd ~/Documents/ && git clone https://github.com/
      MicheleGrisafi/PISTIS_AE.git
3 $ echo 'alias mspdump="~/Documents/PISTIS_AE/
      toolchain/compiler/msp430-gcc-9.2.0.50_linux64
      /bin/msp430-elf-objdump"' >> ~/.bashrc
4 $ source ~/.bashrc
```

Listing 1: Steps to install the required packages on the local environment

As already mentioned, we provide a "plug-n-play" VM to be used as remote environment by any reviewers. Still, for the sake of transparency and in the eventuality that the reviewer wanted to set up their own remote environment, in Listing 2 we provide the steps used to set it up.

```
1 $ sudo apt install libc6-i386 python2.7-dev
      libtinfo5 libusb-dev libgconf-2-4 python3-pip
2 $ pip3 install pyserial
3 $ cd ~/Downloads && wget https://dr-download.ti.
      com/software-development/
      ide-configuration-compiler-or-debugger/
      MD-J1VdearkvK/11.2.0.00007/CCS11.2.0.00007
      _linux-x64.tar.gz
4 $ tar -xvf CCS11.2.0.00007_linux-x64.tar.gz
5 $ ./CCS11.2.0.00007_linux-x64/ccs_setup_11
      .2.0.00007.run
6 $ echo 'PATH="/home/pistis/ti/ccs1120/ccs/eclipse:
      $PATH"' >> ~/.bashrc
7 $ source ~/.bashrc
```

Listing 2: Steps to set up the remote environment.

These steps download the required packages, fetch the CCS binary from the official TI website, extract it and install it. Finally, they add the installation directory to the Linux PATH. During the CCS installation, we should select the minimal installation and only select the "MSP430 ultra-low power MCUs" option. Afterward, we have to install the MSP430 toolchain from the official repository or link CCS with our version. We will proceed with the first option. In CCS, we should go to `Help/CCS App Center`, select the checkbox under MSP430 GCC and click `Install Software`. After the installation we can restart CCS.

## A.5 Experiment workflow

This artefact evaluation contains two types of experiments. The first mainly allow the reviewers to evaluate the custom toolchain, the second allow them to inspect the run-time behaviour of Pistis.

Please be aware that for the second set of experiments, an SSH connection to our remote environment is required. Furthermore, we remind the reviewers that it is a single shared instance connected to a single MCU. As a consequence, they cannot operate simultaneously on the remote environment. We leave the organisational task to the reviewers.

**Disclaimer:** The debugging experience with Code Composer Studio, the official debugger for the MSP430 MCU, can be non-ideal and lead to inconsistent results. There might be executions with weird behaviours. This can be attributed to the state of the MCU and some internal CCS/debugger issues. The reader is kindly asked to follow the instructions as close as possible, sending us (michele.grisafi@unitn.it) an email in case of any issues. Finally, the interaction with the remote environment might not be ideal (slow and not quite responsive). We leave some video tutorials performed on the exact same setup on how to perform the experiments. If the reviewers cannot manage to operate on the remote environment, they are more than encouraged to check our result in such videos.

## A.6 Evaluation and expected results

**Code inspection** The `README.md` file in the github repository contains the repository folder structure, with a description of the various content. The repository only contains the source files, which can be inspected at the reviewers discretion. The reviewer might inspect the various module composing the TCM, inside the `TCM/` folder, and the toolchain scripts, inside the `toolchain/` folder. The inspection of the source files can be skipped in favour of the following steps and evaluations. Still, given that Pistis is a complex software, we encourage the reviewer to inspect as much of it as possible. We authors remain available for any question on this matter.

**Compiling the Trusted Computing Base** The Trusted Computing Base (TCM) is the core of Pistis, containing both its basic functionalities and some Trusted Applications, i.e., additional features. To compile Pistis into a deployable we can follow the commands in Listing 3. Step (4) allow the inspection of the disassembly of the binary, while step (5) allow the inspection of code sections of the ELF file. We leave this data, which describes the TCM, for the more experienced and curios reviewers. Any interaction with the deployable file is more than welcome.

```
$ cd ~/Documents/PISTIS_AE/TCM
$ make clean && make
$ mspdump -D deployable.out > /tmp/dump.txt
$ cat /tmp/dump.txt
$ readelf -S deployable.out
```

Listing 3: Steps for the TCM compilation

**A functioning toolchain** One of the claim in the paper is a modified toolchain that transparently instruments the untrusted application code. This allow the new application image to be executed on a Pistis-enabled device. We ask the reviewers to perform a few steps to evaluate our toolchain (note that the reviewer is free to use the following indications as mere guidelines and perform his/her own tests). In particular, we will compile and inspect the instrumentation for a single application: XorCypher. Next, we describe the required steps for this evaluation. Each step is linked to the commands in Listing 4 (the number of the line will be included in parenthesis, e.g., (1)).

- Move to the `UpdateApplication` folder inside the repository (1) and clean it (2) to make sure no other residual file is present (traces of old compilations).
- Copy the source files of the XorCypher application in the `src` sub directory (3).
- Compile the application without using the modified toolchain (4). If the compilation was successful, the following message should pop out in the console: "*Metadata added –> created file deployable.out*". This informs us that our custom binary was created with the addition of some metadata (only required for the transmission of the binary).
- We can use mspdump[2] to retrieve the content of the binary (5). Mspdump is a disassembler for MSP430 binaries. Since mspdump can only read valid ELF files, and that our binary is a custom optimised format, we use mspdump on the original binary: `appWithNoMetadata.out`.
- Inspect the dump of the binary (6), where illegal instructions can be found. For instance, the reviewers can check for the presence of the `reta` instruction, which is not compatible with Pistis (i.e., it is an unsafe instruction). Alternatively, the assembly file in `UpdateApplication/asm/cryptoXor.s`[3] can be inspected (9).
- (Optional) Compare the size of the two binaries (7): the `deployable.out` and the `appWithNoMetadata.out`. It can be seen how our binary is considerably smaller, as the paper claims.
- Re-compile the application using our modified toolchain (8).
- Inspect the file as before (5)(6) and observe how there is no trace of `reta` instructions anymore: they have been virtualised.
- To have a better look at the instrumentation, open the assembly file `UpdateApplication/asm` which contains the new instrumented assembly code (9). The instrumentation will be contained within comments, e.g. starting from `";Old instruction: RET"` to the `";End safe sequence"`. Furthermore, observe the CFI NOP Slides inserted after each `CALL` statement. The reviewer is welcome to deeply inspect such assembly files.

```
$ cd ~/Documents/PISTIS_AE/UpdateApplication
$ make clean && rm src/* -rf && mkdir src
$ cp ../TestApps/XorCypher/xorCypher.c src/
$ make USE_NEW_LIB=0 VERIFY=0
$ mspdump -D appWithNoMetadata.out > /tmp/
    dump.txt
$ cat /tmp/dump.txt
$ stat -c%s deployable.out appWithNoMetadata.
    out
$ make clean && make libraries && make
$ cat asm/xorCypher.s
```

Listing 4: Steps for the toolchain evaluation

For the second part of this evaluation, we will show how Pistis toolchain rejects applications having illegal instructions, i.e., instructions trying to explicitly violate the access control policy enforced by Pistis. To demonstrate this, we crafted one such application containing a single illegal instruction: `BR #0x3400`. Such instruction is indeed trying to jump to the `0x3400` address which is in RAM, thus illegal[4]. Listing 5 show the required steps.

```
$ cd ~/Documents/PISTIS_AE/UpdateApplication
$ make clean && rm src/* -rf && mkdir src
$ cp ../TestApps/Malicious/rejectmalicious.c
    src/
$ make
```

Listing 5: Steps for the toolchain evaluation

If everything functioned properly, the compilation at the last step should output an error *"The compiled application*

[2]The alias was created during the installation phase

[3]Note that this assembly file does not contain the stdlib code.

[4]We recall that Pistis enforces a non-executable RAM.

*has some unsafe code segments. Stop"*. This is because the toolchain found an illegal instruction.[5]

The reviewers are more than welcome to perform any variation of this test. For instance, they could try and compile the other applications or craft an application of their own. To do so, the only step that needs to be adapted is (3), where the reviewers should copy the source files of their liking.

**Runtime verification in action** One of the key features of Pistis is the ability to inspect any deployed binary at runtime, performing a verification. This step ensures that the binary has indeed been compiled with our custom toolchain, ultimately ensuring the presence of the instrumentation. To evaluate the runtime verification of the untrusted code by Pistis, we will use the debugging features of CCS. Given the necessity of a MCU, we provide the reviewers with an VM instance connected to a MSP430F5529LP board. Since CCS is a GUI-based application, we provide a video-tutorial on the various steps required for this evaluation: `https://youtu.be/tpEBLgRCVAU` . This should help the reviewer in performing the same evaluation. Nevertheless, we provide some guidelines on what we will need to do.

In this evaluation we will use the malicious application shown in listing 6. The application contains two lines of assembly: a `MOV` operation loading an address (pointing to RAM) into a CPU register, and a `BR` instruction jumping to that address (via the register). This application is malicious since it tries to jump to an address in RAM, thus violating the memory protection imposed by Pistis.

```
1  __asm("MOV #0x3400, R9");
2  __asm("BR R9");
```

Listing 6: Malicious application that tries to jump in RAM with a dynamic BR instruction.

For the evaluation we perform the following steps (also shown in the video tutorial):

- Compile the application using the `make VERIFY=0` command, which invokes a non-modified version of the compiler. This will produce an application binary without any instrumentation.
- Start a debugging session of Pistis using CCS. In this session we set a few breakpoints in some sensitive points in the code. Specifically, we want to break the execution when we reach either one of the following: verification passed, verification failed.
- Start the execution of Pistis, which will proceed with its RemoteUpdate feature and wait for an incoming image on the serial communication.
- Deploy the new application binary (without instrumentation) through our python deploy script.
- Observe how the second breakpoint is triggered: the verification fails and the application is not lunched.

This tutorial hence shows how Pistis bounds applications to our instrumentation, i.e., to using our toolchain. Pistis will only accept binaries which have indeed been compiled with our toolchain.

In the next tutorial, we will show how Pistis run-time memory protection protects the MCU from the malicious activity of an application compiled with our toolchain.

**Memory protection in action** To evaluate the memory protection offered by Pistis on the MCU we will use the debugging features of Code Composer Studio (CCS). These will enable a run-time debugging of the MCU, allowing us to check the operations of Pistis at run-time. For this purpose, we deploy the same application of listing 6. However, contrarily to previous experiment, we will instrument the malicious application with our custom toolchain[6]. This will allow it to be deployed, pass the verification and then be executed. However, since the unsafe instruction (the jump to a register) is indeed an illegal operation, this will be caught at run-time and the application will be stopped. Given that an interaction with a GUI is necessary for this step, we provide a video tutorial: `https://youtu.be/OhhJiyQC0bk`. Nevertheless, we report the main steps that we are going to do:

- Compile the application using the `make` command, which invokes a modified version of the compiler. This will produce an application binary with Pistis instrumentation.
- Start a debugging session of Pistis using CCS. In this session we set a few breakpoints in some sensitive points in the code. Specifically, we want to break the execution when we reach either one of the following: verification passed, verification failed, virtual safe BR function invoked. The latter is the function that checks all unsafe BR instructions in the code (which have been replaced by a call to this virtual safe function by our toolchain).
- Start the execution of Pistis, which will proceed with its RemoteUpdate feature and wait for an incoming image on the serial communication.
- Deploy the new application binary using our custom python deployer.
- Observe how the first breakpoint is triggered: the verification succeed and the application is lunched.
- Observe how the third breakpoint is reached: the application BR instruction is correctly virtualised.

[5] Note that illegal instructions are rejected right away, while unsafe instructions are virtualised. The latter cannot be rejected right away because their outcome depends on the run-time state of the MCU.

[6] Notably, the application is not rejected by our toolchain, but its unsafe instructions are instead virtualised.

- Observe how the safe BR function performs some security checks on the original instruction and stops the execution of the application, given that the original jump is illegal.

## A.7 Experiment customization

The reviewers are encouraged to perform any experiment of their liking on the local environment. For instance, they could choose to execute or compile different applications, or even craft their own. However, given the scarce resources of the remote environment, we kindly ask them no to deviate from the provided instructions. Any modification could impede the work of the other reviewer. Moreover, the remote environment is provided with root access, thus allowing the reviewers to completely compromise it if they operate outside of our guidelines.

## A.8 Notes

This artefact evaluation covers a few of the main functionalities of Pistis, showing its potential. Pistis is almost fully implemented (a few bugs and minor tweaks still to be addressed) and it has been fully tested and evaluated (as documented in the paper). However, the full evaluation is a cumbersome and time-demanding process requiring several technical ability. Furthermore, setting up a tutorial on how to properly test all of its features is even a more challenging task (especially considering the remote nature of the majority of the tests). For these reasons, this artefact only presents some of the possible tests that could be performed on the executable. The creation of complete tutorials is a future work.

Nevertheless, the repository contains a README.md file that describes in details some additional steps required to use Pistis. Note that this artefact document summarises only some of these steps, providing some techniques to evaluate it without owning the proper hardware.

## A.9 Version

Based on the LaTeX template for artefact Evaluation V20220119.