USENIX Security '24 Artifact Appendix: Opportunistic Data Flow Integrity for Real-time Cyber-physical Systems Using Worst Case Execution Time Reservation

Yujie Wang, Ao Li, Jinwen Wang, Sanjoy Baruah, Ning Zhang
Washington University in St. Louis

A Artifact Appendix

A.1 Abstract

OP-DFI is a security primitive designed to deploy Data-flow Integrity (DFI) into real-time systems with minimal impact on Worst-case Execution Time (WCET). OP-DFI leverages system reservation to enforce data flow integrity within the software, addressing the challenge of slack estimation and runtime policy swapping to opportunistically utilize extra time within the system. Our artifact includes the program analysis and software instrumentation tools, complete with end-to-end examples for demonstration. To utilize OP-DFI, a user needs to input the source code of the software being tested along with the timing analysis result of the hardware platform derived from WCET analysis tools. The expected outputs include the estimated slack and the security protection provided. To streamline the artifact evaluation (AE) process, we provide a pre-configured virtual machine (VM) with all necessary dependencies installed. For added convenience, remote VM access is available through TeamViewer.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Conducting the AE for OP-DFI does not raise any security, privacy, or ethical concerns. All programs and data are securely stored within a VM that is hosted on a remote server, and the AE activities are carried out inside this VM. This setup ensures that the AE process is isolated and avoids interaction with the reviewer's personal or sensitive code/data.

A.2.2 How to access

We have made the source code publicly available on GitHub: **Source code.** https://github.com/WUSTL-CSPL/OP-DFI/tree/00edcc646099516f9014bbae67f4050ea1b793e0

A.2.3 Hardware dependencies

To assess our artifacts, please ensure that your host system maintains a stable network connection to access the remote VM via TeamViewer. Our VM is equipped with an Intel i5-8400 4-Core Processor and 22GB of RAM. For an easy demonstration, the final ARM program is executed in QEMU within the VM.

A.2.4 Software dependencies

OP-DFI is developed on Ubuntu 22.04.3 LTS primarily using:

- KLEE: https://github.com/klee/klee , commit fc83f06b17221bf5ef20e30d9da1ccff927beb17).
- SVF: https://github.com/SVF-tools/SVF , commit 06920202d216e003efcac1469fc78b12904cd2c6).
- LLVM: https://github.com/llvm/llvm-project , commit 75e33f71c2dae584b13a7d1186ae0a038ba98838).

The main slack estimator tool was developed based on KLEE. The data-flow analysis was developed based on SVF. The code runtime switches and software instrumentation were developed based on LLVM. All other helper scripts were written in Python and Shell Script.

A.2.5 Benchmarks

None.

A.3 Set-up

Note: Reviewers can skip this section. We provide a fully configured remote VM to simplify the artifact evaluation process, accessible through TeamViewer. As a result, the only setup required is installing TeamViewer; no additional test environment is needed. This section is provided solely to document our VM setup process.

A.3.1 Installation (Optional)

Please do not re-install the environment on the given VM. Please setup environment only on a fresh VM.
Directory Setup: The running script assumes the VM has the following file directory.

```
$ git clone \
https://github.com/WUSTL-CSPL/OP-DFI ~/
$ mv ~/OP-DFI ~/opdfi
$ mkdir ~/toolchain && cd ~/toolchain
$ mkdir klee && mkdir SVF \
&& mkdir llvm-project
```

Then, clone the corresponding commit version of KLEE, SVF, and LLVM listed in Section A.2.4 under directory "~/toolchain". Then, apply patches located in "~/opdfi/KLEE_patch", "~/opdfi/SVF_patch", "~/opdfi/LLVM_patch" to "~/toolchain/klee", "~/toolchain/SVF", and "~/toolchain/llvm-project" respectively.

```
$ cd ~/toolchain/klee
$ git apply ~/opdfi/KLEE_patch/*
$ cd ~/toolchain/SVF
$ git apply ~/opdfi/SVF_patch/*
$ cd ~/toolchain/llvm-project
$ git apply ~/opdfi/LLVM_patch/*
```

Finally. compile them following guidance of the official repository of *klee* (https://github.com/klee/klee), *SVF* (https://github.com/SVF-tools/SVF) and *LLVM* (https://github.com/llvm/llvm-project) respectively. To ease the compilation process, the major steps for building them can be found in the directory "~/opdfi/build_scripts/", with the following scripts: `build_klee.sh`, `build_svf.sh`, and `build_llvm.sh`.

Environment Setup: After installation, setup the corresponding environmental variables by copying following command into "~/.bashrc".

```
export LLVM_DIR=~/toolchain/llvm-project/\
build
export KLEE_DIR=~/toolchain/klee/build
export SVF_DIR=~/toolchain/SVF
export KLEE_INC=$KLEE_DIR/../include
export PATH="$LLVM_DIR/bin:\
$KLEE_DIR/bin:$PATH"
export opdfi=/home/opdfi/opdfi
```

A.3.2 Basic Test

The successful setup of a KLEE environment can be verified with the following commands:

```
$ klee --version
```

To check the installation of SVF, execute:

```
$ $SVF_DIR/Release-build/bin/wpa --version
```

To check the installation of LLVM, execute:

```
$ clang -v
```

If the final outputs do not indicate any errors, it can be concluded that these dependencies are installed properly.

Table 1: Code structure.

Major Claim	Functionality	Code	Results
C1	slack constraint collection and processing	klee: lib/* slack_estimation/*	~/opdfi/test/ compile_results/*
	code version generation	llvm: code_version_generation/* SVF: dependency_analysis/*	
C2	runtime slack estimation	slack_estimation/*	~/opdfi/test/ runtime_results/*
	runtime code switching	code_switch/*	

```
1: if(len>MAX_DATAFRAME_LEN){//WCEP constraint, len>96
2:     len=MAX_DATAFRAME_LEN;
3:     handle_error_input();  }//a long time operation that can cause WCET
compile_results/slack_constraints/9.single_constraint: (bvslt (_ bv96 32)..)
compile_results/slack_estimate_policy: 0% { 0:0 1:0 2:1 3:0 9:1 }
```

Figure 1: Slack constraint example.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): At compile time, OP-DFI can offline collect and prepare path constraints for later runtime slack estimation. In addition, OP-DFI can generate multiple code versions that provide different levels of security protection for different slack scenarios.

(C2): At runtime, OP-DFI can runtime compute slack and perform runtime switching to different code version to provide different level of protection under the computed slack.

A.4.2 Experiments

The implementation code for each major claim is shown in Table 1, with details illustrated as follows.

(E1): *[Offline Slack Constraint Collection and Code Version Generation] [1 human-minutes + 2 compute-minutes]:*

How to: Conduct symbolic execution on program source code. The symbolic execution tool, modified based on KLEE, can automatically collect and process path constraints.

Then, the system conducts data-flow analysis on the program source code. The code version generation process first duplicates the original code and then conduct security instrumentation to deploy different DFI techniques on the various code versions.

Preparation: The provided VM is fully configured (see set-up details in Section A.3), with all compilation, program analysis, and security instrumentation toolchains set up. Thus, there are no additional steps needed to prepare for this experiment. The sampled program is located in ~/opdfi/src_test/crsf.cpp. For easy demonstra-

tion, QEMU is used to emulate the actual hardware, where the WCET analysis tool cannot be applied. Therefore, the timing result for each execution path is simulated as the instruction count on the path. We provide a script to compile OP-DFI to demonstrate the offline constraint collection and code version generation process.

Execution: To run these compilation, analysis and instrumentation processes, use the following command:

```
$ cd ~/opdfi/test
$ source ../compile.sh
```

The illustrations of these commands are as follows. The offline compilation can be conducted using the "compile.sh" script, which runs the *klee* executable on the sample program to generate path constraints. Then, the processed constraints are handled by "prepare_formula.py" to select the constraints to determine the relationship between the constraints and different levels of slack. In addition, the DFI protection level for different code versions is also determined as a part of the security policy. Based on the security policy, the program code is first duplicated into multiple code versions using the LLVM passes located in "$LLVM_DIR/../llvm/lib/Transforms/code_switch_llvmpasses/code_version_generation/". Next, each generated code version is instrumented with different levels of DFI protection using the SVF scripts located in "$SVF_DIR/tools/OPDFI/dependency_analysis/". Finally, slack estimator and code switch units are inserted as reference monitors.

Results: The outputs of the compilation are located under the directory "~/opdfi/test/compile_results", and consist of the final executable, the processed constraints for slack estimation, the generated code versions, and the security policy. The final executable is "./execute_me". The processed constraints for slack estimation are located in "./compile_results/slack_constraints". In this folder, each "{constID}.single_constraint" file is a constraint for a program branch in SMT formula format. The security policy between the constraint evaluation result and the slack level is located in "./compile_results/slack_estimate_policy". In this file, the first column is the amount of slack (represented as a percentage of WCET). The second column consists of the expected evaluation results for each constraint: {constraintID:evaluationResult}. This allows the system to eliminate a slack level at runtime if the runtime constraint evaluation result does not satisfy the expected result. An example is shown in Figure 1. In the sample code, the WCET should only occur when the branch "if(len>MAX_DATAFRAME_LEN)" is taken, where the branch constraint (with ID "9") is stored in the file "9.single_constraint". Then, in the security policy file, the evaluation result is "True/1" for the runtime evaluation of this constraint to be taken, as indicated in the last row of Figure 1. Therefore, during runtime, if its evaluation result is "False/0", the WCET will be eliminated, indicating that there is slack available for security protection.

The generated code versions are located in "./compile_results/code_versions". In this folder, each "code_ver.bc.v{versionID}.dfi" file is a code version with code duplicated and DFI checks deployed on it.

To see the duplicated code, use the following commands.

```
$ cd ~/opdfi/test/compile_results
$ cd code_versions
$ llvm-nm code_ver.bc.v3.dfi \
 | grep _version_opdfi
$ llvm-nm code_ver.bc.v1.dfi \
 | grep _version_opdfi
```

The last two commands will show that each function's code is duplicated with the postfix "_{versionID}_version_opdfi".

To see the different amounts of DFI checks, use the following commands.

```
$ cd ~/opdfi/test/compile_results
$ cd code_versions
$ llvm-dis ./*
$ grep opdfi_dfi \
code_ver.bc.v3.dfi.ll | wc
$ grep opdfi_dfi \
code_ver.bc.v1.dfi.ll | wc
```

The last two commands will approximately show the amount of DFI checks deployed in two different code versions (i.e., v3 and v1). This will show the number of DFI checks that differ from each other. The protection coverage for each code version is located in "./compile_results/version_info", where the first column is the code version ID, and the second is the protection coverage.

(E2): *[Runtime Slack Calculation and Code Switching] [1 human-minutes + 2 compute-minutes]:*

How to: With the slack estimation and code-switching reference monitors inserted into the program code, and the offline-generated security policy, the slack estimation reference monitor evaluates the constraints using inputs to obtain an estimated slack, and the code-switching reference monitor then switches to the corresponding code version according to the estimated slack.

Preparation: The provided VM is fully configured (see set-up details in Section A.3), with a QEMU emulator set up for executing on the AArch64 platform. Thus, there is no additional step to prepare for this experiment. The compilation step generates a final executable, which we can run to demonstrate the runtime slack calculation and code-switching process.

Execution: The experiments can be conducted by run-

ning the final executable "execute_me" located in the "compile_results/" folder. To run these experiments, use the following command to run the executable with QEMU emulation:

```
$ cd ~/opdfi/test
$ source ../compile.sh
$ qemu-aarch64 \
-L /usr/aarch64-linux-gnu/ \
./compile_results/execute_me
```

Results: The executable will run multiple iterations with the test inputs stored in "test_inputs.txt". During execution, the number of iterations, slack, and DFI protection coverage will be displayed. The runtime execution logs are stored in "runtime_result.txt" under the "runtime_results/" directory, where the first column is the estimated slack (in the form of a percentage of WCET), and the second column is the protection coverage. As the results show, the protection level depends on the amount of available slack, where usually more slack indicates higher protection coverage. Moreover, since the program executes a loop, which depends on the input value, this correlation is reflected between the values in the "test_inputs.txt" file and the runtime logs.

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/usenixsec2024/.