

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SoK: On the Fragility of Memory Error Exploit Mitigations

Adriaan Jacobs, *DistriNet, KU Leuven;* Mahmoud Ammar, *Huawei Research, Germany;* Stijn Volckaert, *DistriNet, KU Leuven*

https://www.usenix.org/conference/usenixsecurity26/presentation/jacobs

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: SoK: On the Fragility of Memory Error Exploit Mitigations

Adriaan Jacobs, Mahmoud Ammar, and Stijn Volckaert

A Artifact Appendix

A.1 Abstract

This artifact contains the Python implementation of the graph-based framework described in Section 3 as well as visualized in Figre 2 in the paper. The artifact implements the memory corruption exploit graph in addition to declarative mappings of defenses to inhibited graph edges. It also provides scripts for automated fragility, interoperability, and portability tests. Additional scripts are also provided for reproducibility, including one that iterates over all mapped defenses and generates a table similar to Table 4 in the paper. The artifact is intended to support the Functional and Results Reproduced badges by allowing reviewers to reproduce the modeled defense mappings and the automated analyses reported in Sections 4, 5, and 6.

A.2 Description & Requirements

The artifact is fully implemented in Python and is straightforward to execute. It requires no compilation, execution of exploits, specific hardware, or custom benchmarks.

The main files of our implementation are:

- `ExploitGraph.py`: The exploit graph model.
- `Explorer.py`: Utility class that tracks inhibited knowledge and edges, and computes reachability queries over an ExploitGraph.
- `Defenses.py`: Contains the list of modeled defenses on the graph, as methods that apply inhibitions to an Explorer instance. It also serves as the command-line fragility/interoperability entry points.
- `generate_defense_table.py`: Iterates through all defenses defined in Defenses.py and evaluates their effectiveness. It outputs Table 4 from the paper.
- `cla_generation.py`: Performs the interoperability evaluation described in Section 6.1 of the paper, between minimal first-level defense sets and Safe Rust.
- `portability_test.py`: Tests the fragility and platform-independence of the modeled defenses by temporarily disabling foundational, assumed protections (like DEP or read-only code) and observing if new attack vectors open up. Described in Section 6.2 of the paper.

A.2.1 Security, privacy, and ethical concerns

The artifact performs abstract graph reachability analyses. No security or privacy risks as it does not generate exploits, run vulnerable programs, disable active security mechanisms, collect data, or access the external network during execution. The scripts write local output files such as `defense.svg`. Reviewers may delete these generated files after evaluation.

A.2.2 How to access

The artifact is archived on Zenodo at https://doi.org/10.5281/zenodo.20315524. The development repository is available at https://github.com/adriaanjacobs/sec26-sok-fragility.

A.2.3 Hardware dependencies

None. The artifact runs on a commodity laptop or desktop with any budget in terms of computing power or storage. The experiments should finish in a few seconds on a typical machine or workstation.

A.2.4 Software dependencies

The artifact was tested on Linux with Python 3.12.3 and the Python `graphviz` package version 0.20.2. The code uses modern Python type syntax, so Python 3.10 or newer is recommended.

A.2.5 Benchmarks

None. The experiments operate only on the graph model and defense mappings included in the artifact.

A.3 Set-up

A.3.1 Installation

Considering the Git repository as the main source of accessing the artifact:

```
git clone https://github.com/adriaanjacobs/
    sec26-sok-fragility.git
cd sec26-sok-fragility
python3 -m venv .venv
. .venv/bin/activate
python3 -m pip install graphviz
```

Listing 1: Artifact Setup Instructions

For the zenodo version, simply download and unzip the zenodo artifact instead, and then start from the `venv` step.

On Debian/Ubuntu systems, install the Graphviz renderer:

```
sudo apt-get update
sudo apt-get install graphviz
```

Listing 2: Desktop Graphic Renderer

A.3.2 Basic Test

Run a single fragility query:

`python3 Defenses.py fragility SoftBound_wo`

The expected results or output should be something similar to the below screenshot.

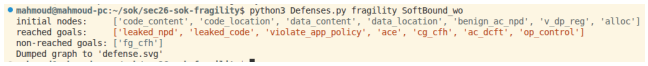


Figure 1: Expected Results when running an automated fragility test for SoftBound Write-only version.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The artifact implements the exploit graph and declarative defense-mapping interface used by the paper to map defenses onto inhibited edges and/or knowledge nodes. This supports the claims and descriptions in Section 3 and 4, and Appendix B. It is reproduced by experiment (E1).

(C2): The artifact automatically evaluates the fragility of mapped defenses and outputs results concerning which assumptions or goals hold and which ones are (partially) violated, as shown in Table 4. This is reproduced by experiment (E2), which mainly depends on experiment (E1).

(C3): The artifact supports and reproduces the interoperability analysis discussed in Section 6.1: Safe Rust fails to compose with several complete first-level defense combinations; It is only interoperable with the combinations with the design-level properties summarized in Table 5 (i.e, these four combinations eliminate all CLA paths when deployed with Safe Rust on the same platform). This is reproduced by experiment (E3).

(C4): The artifact supports and reproduces the portability analysis discussed in Section 6.2: defenses that rely on DEP or read-only code become fragile when such basic defenses are missing in target architectures. This is reproduced by experiment (E4).

A.4.2 Experiments

(E1): Graph modeling, Defense mapping, and a basic fragility test [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment is very similar to the basic test descried in Section A.3.2.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: No further preparation needed.
Execution: Run:
`python3 Defenses.py fragility <defense_name>`

`Defenses.py` includes 50+ mapped defenses. The reviewer can choose to run the automated fragility test of any of these defenses (e.g., `SoftBound`, `LowFat`, `SafeInit`, `DSR`, `DFI`, `SafeStack`, `Clang_IFCC`, etc.) or map new ones and test them.
Results: Figure 2 shows the expected results when running the automated fragility test for `SafeStack`, `Clang_IFCC`, and `CPI` respectively. The printed output shows both reachable and non-reachable attacker goals. Reviewers can also visualize blocked or reachable goals by rendering the resulted `.svg` files using `Graphviz` or any other suitable tool.



Figure 2: Expected Results when running an automated fragility test for SafeStack, Clang-IFCC, and CPI respectively.

(E2): Regeneration of the Defense Table (Table 4) [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment mainly generates a table similar to Table 4 in the paper, covering all mapped defenses. The expected table to be generated by the reviewers should cover defenses shown in Table 4 as well as in Table 6.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: The below commands/definition must be added to the latex environment (e.g., in the preamble file) in order to have the nice visualization inside certain

cells of the generated table.

```
\definecolor{darkgreen}{rgb}{0.0, 0.5, 0.0}
\definecolor{babyblueeyes}{rgb}{0.63, 0.79,
↪ 0.95}
\newcommand{\cmark}{\textcolor{darkgreen}{\larg
↪ e\ding{51}}}
\newcommand{\cmarkbad}{\textcolor{red}{\normals
↪ ize\ding{51}}}
\newcommand{\xmark}{\textcolor{red}{\ding{55}}}
\newcommand{\xmarkgood}{\textcolor{darkgreen}{\
↪ ding{55}}}
\newcommand{\nmarkr}{\textcolor{red}{\ding{108}
↪ }}
\newcommand{\nmarkg}{\textcolor{darkgreen}{\din
↪ g{108}}}
\newcommand{\nmarko}{\textcolor{orange}{\ding{1
↪ 08}}}

\newcommand{\numcircleptrinvalid}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backgray,
    ↪ text=textgray, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backpurple1}{HTML}{76608A}
\definecolor{textpurple1}{HTML}{FFFFFF}

\newcommand{\numcirclewriteabuse}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backpurple1,
    ↪ text=textpurple1, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backblue}{HTML}{DAE8FC}
\definecolor{textblue}{HTML}{000000}

\newcommand{\numcirclereadabuse}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backblue,
    ↪ text=textblue, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backorange}{HTML}{FFE6CC}
\definecolor{textorange}{HTML}{000000}

\newcommand{\numcircleleakage}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backorange,
    ↪ text=textorange, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backblack}{HTML}{000000}
\definecolor{textblack}{HTML}{FFFFFF}

\newcommand{\numcircleexploitdispatch}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backblack,
    ↪ text=textblack, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backpink}{HTML}{F8CECC}
\definecolor{textpink}{HTML}{4D4D4D}

\newcommand{\numcircleexploitexecute}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backpink,
    ↪ text=textpink, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}
```

Execution: Run:

```
python3 generate_defense_table.py > def_tbl.tex
```

Results: A `.tex` file will be generated containing a table similar to Table 4 in the paper covering all mapped defenses.

(E3): Cross-language Interoperability Analysis [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment is very similar to the basic test descried in Section A.3.2.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: No further preparation needed.
Execution: Run:

```
python3 Defenses.py interop --set1 Rust
   --set2 FineIBT intelSHSTK

python3 cla_generation.py > cla_results.txt
```

To run the interoperability test for two selected sets of defenses, the reviewer should execute the first command shown above. In this example, we use the same defense sets as in the paper: Safe Rust as the first set, and FineIBT with Intel Shadow Stack as the second set. To evaluate interoperability between Safe Rust and all modeled first-level defenses, as reported in the paper, and to obtain

results similar to Table 5, the reviewer should execute the second command shown above.
Results: Figure 3 shows the expected results when running the automated interoperability test for `Safe Rust` vs `FineIBT` and `Intel Shadow Stack`, confirming that `Safe Rust` is not interoperable with such Integrity-of-Use defenses. Reviewers can also visualize blocked or reachable goals by rendering the resulted `.svg` files using `Graphviz` or any other suitable tool. We do not show results for the second command line, but expected ones should be in the form of records telling whether `Safe Rust` is interoperable with the selected set of first-level defenses. Out of 55 records, only 4 records should block all CLA paths when deployed with Rust as shown in Table 5 in the paper.



Figure 3: Expected Results when running an automated Interoperability test for Safe Rust vs FineIBT & Intel Shadow stack.

(E4): Portability Analysis [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment is very similar to the basic test descried in Section A.3.2.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: No further preparation needed.
Execution: Run:
`python3 portability_test.py`

Executing the above command runs a reachability analysis query for all mapped defenses under the assumption that these defenses are deployed on platforms lacking W⊕X protection. The obtained results are then compared against the default deployment settings, and the script prints the defenses for which new attack goals become reachable due to the absence of W⊕X.
Results: Figure 4 shows the defenses identified by the portability test. The results indicate that these defenses are not portable to platforms that lack W⊕X protection, since W⊕X is a core assumption in their threat models.

A.5 Notes on Reusability

The artifact is designed to be extended by adding new defense mechanisms to `Defenses.py`. To be mapped, a defense should receive an `ExploitGraph` and an `Explorer`, then marks the graph edges or knowledge nodes it inhibits. Optional annotations such as `@citation`,



Figure 4: Evaluating the Portability of mapped defenses when deploying them in modern architectures, e.g., Nvidia GPUs, that lack W⊕X protection.

`@prettyname`, `@defense_strategy`, `@defense_category`, and `@threatmodel` are consumed by the table-generation and analysis scripts. After adding a defense, reviewers or future users can run `generate_defense_table.py`, `Defenses.py fragility`, and `Defenses.py interop` to evaluate it under different assumptions or deployment scenarios as explained in earlier sections.

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/usenixsec2026/.