USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

TRex: Practical Type Reconstruction for Binary Code

Jay Bosamiya, *Microsoft Research;*
Maverick Woo and Bryan Parno, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity25/presentation/bosamiya

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: TRex: Practical Type Reconstruction for Binary Code

Jay Bosamiya[1], Maverick Woo[2], and Bryan Parno[2]

[1]*Microsoft Research*
[2]*Carnegie Mellon University*

A Artifact Appendix

A.1 Abstract

In the paper, we present TRex, a tool that performs automated deductive type reconstruction, using a new perspective that accounts for the inherent impossibility of recovering lost source types. Compared with Ghidra, a state-of-the-art decompiler used by practitioners, TRex shows a noticeable improvement in the quality of output types on 124 of 125 binaries.

This artifact contains the source code for TRex, scripts for producing reproducible benchmark binaries, as well as the scripts needed to run the evaluation presented in §5 of the main paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact poses no security, privacy, or ethical risks during execution. The code does not perform any destructive actions, nor does it disable any existing security mechanisms.

A.2.2 How to access

This artifact includes (i) the source code for TRex, (ii) reproducible scripts for the benchmarks, and (iii) evaluation framework to reproduce the results in §5 of the paper.

The artifact is archived at https://doi.org/10.5281/zenodo.15611994. Within this artifact, the TRex tool itself sits within `trex/`, the benchmark scripts sit within `trex-usenix25/benchmarks/`, and the rest of the files in `trex-usenix25/` are the evaluation scripts/framework.

The existence of two top-level folders reflects the fact that we have two GitHub repositories: https://github.com/secure-foundations/trex for the tool itself, and https://github.com/secure-foundations/trex-usenix25 for the evaluation and benchmarks. The artifact contains a symlink from `trex-usenix25/trex` to `../trex` to establish the directory structure expected by our evaluation framework.

Details of software requirements, instructions, and more can be found in the top-level READMEs in each of the two top-level directories, but are also listed below.

A.2.3 Hardware dependencies

No special hardware is needed for running TRex, or indeed most of the evaluation, and a regular commodity x86-64 system suffices. While TRex itself works on ARM-based devices too, building the benchmark binaries requires an x86-64 machine. For future reference, we note that we ran our evaluation on a server with an Intel i9-10980XE (with 36 logical cores) and 256 GiB of system memory. We stress that the evaluation can be run using lower hardware specifications at the cost of a longer run time.

For replicating §5.3 in particular (i.e., evaluation against the Machine Learning-based prior work called ReSym), we recommend using a system with sufficiently powerful GPU (with CUDA support). The ReSym paper itself uses four A100 GPUs, thus we too used a server with four A100 GPUs.

Note that our evaluation scripts support running the GPU-intensive part of the computation on another server separate from the rest of the evaluation, if that is more convenient; details can be found in the relevant README.

A.2.4 Software dependencies

The evaluation has a small number of software dependencies that we list in the relevant READMEs. In short, we require an installation of Rust, Just, Python, uv, rename, and Ghidra. We have tested the evaluation locally on both Ubuntu 22.04 and 24.04, but recommend using the provided Dockerfile to ensure that all versions are set up correctly, and that paths and environment variables are correctly initialized.

A.2.5 Benchmarks

Benchmark binaries for the evaluation consist of COREUTILS and SPEC. For the former, no special access other than internet is needed, and the reproducible scripts will automatically download and build the binaries after confirming a

cryptographic checksum. For the latter, a license to SPEC CPU® 2006 is required to obtain the necessary source archive. Our reproducible scripts will confirm its cryptographic checksum and then use it for compiling the benchmark binaries.

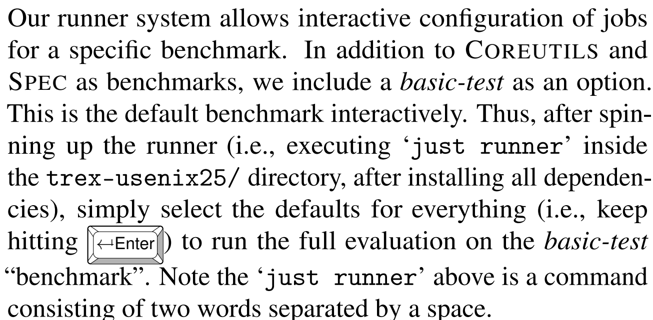
A.3 Set-up

A.3.1 Installation

Within `trex-usenix25/.docker`, simply run '`make`' (without quotes). This uses `podman` to build an image, spin up a container, and drop you into a shell with all dependencies installed. The artifact itself will be at `/trex-usenix25/`; please ensure that the main TRex code is cloned or symlinked correctly at `/trex-usenix25/trex/` (running '`just trex`' within the `/trex-usenix25/` directory should correctly clone this if you don't have the folder).

For the benchmark binaries (outside the aforementioned container), run `make` within the directory of the benchmark of your choice, within `trex-usenix25/benchmarks`; this should automatically build all the benchmarks and place them into the relevant `evalfiles` folder.

A.3.2 Basic Test

Our runner system allows interactive configuration of jobs for a specific benchmark. In addition to COREUTILS and SPEC as benchmarks, we include a *basic-test* as an option. This is the default benchmark interactively. Thus, after spinning up the runner (i.e., executing '`just runner`' inside the `trex-usenix25/` directory, after installing all dependencies), simply select the defaults for everything (i.e., keep hitting  (↵Enter) to run the full evaluation on the *basic-test* "benchmark". Note the '`just runner`' above is a command consisting of two words separated by a space.

This should finish in a few minutes, even if running only on a single core (although we recommend using multiple cores so that the various compilation steps are not slowing things down too much). Each runner sub-task (starting with "Confirm basic pre requisites" and finishing with "Summarize all metrics") will automatically report back with a success or failure. All jobs should show up with a checkmark indicating success. If any jobs fail, the runner gives you commands to run with more details and figure out what to fix.

The basic test runs all the same jobs as the full evaluation workflow using a couple of binaries instead of the complete benchmark. If the basic test succeeds, then the workflow on the complete benchmarks is expected to succeed.

A.4 Evaluation workflow

A.4.1 Major Claims

- **(C1):** TRex qualitatively improves upon the decompilers used by practitioners (namely, Ghidra, Binary Ninja, and Hex-Rays / IDA Pro). This is demonstrated by Experiment E1, described in §5.1 of the paper.
- **(C2):** TRex quantitatively outperforms Ghidra on real-world benchmarks (namely, COREUTILS and SPEC). This is demonstrated by Experiment E2, described in §5.2 of the paper.
- **(C3):** TRex outperforms state-of-the-art Type Prediction (i.e., machine-learning based technique). This is demonstrated by Experiment E3, described in §5.3 of the paper.

A.4.2 Experiments

(E1): Qualitative comparison [<10 human-minutes + <10 compute-minutes].

Preparation: Make sure that the basic test described in Appendix A.3.2 is successful.

Execution: A quick approximation is to simply look at the `trex-usenix25/benchmarks/basic-test/evalfiles` directory after running the basic test described in Appendix A.3.2—the *easy* and *hard* linked list binaries and the corresponding results are named `test-linked-list-slot1.*` and `test-linked-list-slot2.*` respectively. Specifically, looking at the `*.trex-clike` and `*.trex-st` files are similar to Figure 5 of the paper. However, for presentation purposes in the paper, we use mildly different CLI flags that improve readability and understanding. Thus, for a more direct mapping to the paper, we recommend running (within the `trex-usenix25/trex/trex/` directory) '`cargo run -- from-ghidra tests/test-linked-list-slot2.{lifted,vars} -Zdisable-type-rounding -Zdisable-signed-integer-preference`'. Run a similar command but with `slot1` rather than `slot2` to obtain the "easier" variant described in the text. To obtain the decompiler outputs to compare against, simply load the relevant binary `*.ndbg-bin` into the decompiler to see the produced output types. To run the ablation described at the end of §5.1 of the paper, add the `-Zdisable-aggregate-type-analysis` argument to the `cargo run` command above. To see the expected warning (in the ablation for slot 1), add `-dd` to the command.

Results: The produced types at stdout (both structural and C types) should be virtually identical to Figure 5 in the paper. Similarly, the alternate "easier" variant (i.e., `slot1`) produces the expected types, and the ablation study produces the types described in the paper. Note that there may be minor naming or presentational differences, such as the use of curly braces for better readability in the paper, or numbering of types in the automatically-generated

type names—for example, t31 in the paper might be α-converted to t29 if there are minor differences in internal numbering, due to (say) debug logs.

(E2): Quantitative comparison [<10 human-minutes + up-to-overnight compute time].

Preparation: Make sure that the basic test described in Appendix A.3.2 is successful. Also, *outside* the container, within `trex-usenix25/benchmarks/<benchmark>`, run '`make`' to ensure that all the `.../evalfiles/*.binar.xz` (i.e., benchmark binaries) are generated (the run time depends on the parallelism afforded by your machine; while we found it to run on the order of minutes, it would not hurt to brew a cup of coffee/tea at this time). These binaries are what will be used by the runner for the evaluation.

Execution: Within `trex-usenix25/`, run '`just runner`', and select the *<benchmark>* when interactively prompted. Use the defaults for all other prompts, unless you explicitly want to change something (for example, use fewer than max cores). Even the jobs picked should be left at the defaults (see Experiment E3 for when the currently-not-selected ones are activated). At this point, the entire evaluation should run autonomously, and depending on the amount of parallelism, might finish in under an hour, or might take overnight. Some jobs (e.g., Ghidra) are known to be flaky, so the runner will automatically re-attempt them up to 3 times if it detects they have failed. Re-executing the runner with same settings will also re-use cached results for any previously-successful tasks. Thus, if the runner needs to be stopped halfway for some reason and then restarted, not much progress is lost.

Results: All results produced by the runner can be found in the `trex-usenix25/benchmarks/<benchmark>` directory. The `std-metrics.csv`, `summary.tex`, and `eval-*.pdf` files should match the results reported in the paper, specifically, Tables 2 and 3, and Figures 7 and 8. For the ReSym row in the tables, please see Experiment E3.

(E3): Comparison to ML-based technique [<30 human-minutes + ~overnight compute time].

Preparation: Make sure to have successfully finished Experiment E2 first (the summarization scripts will overwrite previously-generated summary results; thus it might help to move the existing summary files to a separate directory for future reference, before moving on to this experiment). Then follow the instructions in `trex-usenix25/tools/evaluating_resym/README.md`, to set up a GPU-capable server with ReSym, and pass the correct environment variables.

Experiment: With the environment variables set up correctly, the runner will automatically recognize that it must evaluate ReSym too (i.e., the defaults automatically update such that the previously-disabled ReSym jobs get activated). The runner will also reuse any existing intermediate files from Experiment E2 whenever possible, so as to not waste time re-computing deterministic results.

Results: The generated summary files in the `.../<benchmark>` directory should match the results reported in the paper (Tables 2 and 3, and Figures 9 and 10), modulo randomization-induced noise due to the non-deterministic nature of the machine-learning based ReSym. Note: the "generous scoring" results are prefixed with `gen` in the produced results.

A.5 Notes on Reusability

The TRex tool should be independently reusable, even outside the evaluation. Instructions for this are provided in the TRex README. For customizing behaviors (e.g., to toggle the flags we describe in §3.2 of the paper), see TRex's `--help` output. Additionally, the benchmark scripts, and evaluation machinery should be usable for future research as-is. Introducing a new tool for future research to the evaluation scripts simply involves making sure that the tool can generate files of the `*.<toolname>-st` format, and adding a task to the runner to execute the tool.

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/usenixsec2025/.