Nacpa: Native Checking with Parallel-Portfolio Analyses
(Competition Contribution)

Thomas Lemberger and Henrik Wachowitz

LMU Munich, Munich, Germany

Abstract. We present Nacpa, a meta-verifier based on parallel portfolio and native compilation of backend verifiers. Nacpa does not implement any software analyses itself, but uses the Java-based CPAchecker as off-the-shelf verification backend in different configurations; each called as a separate, external process. To avoid the overhead of starting the Java Virtual Machine multiple times and to improve the run time on fast-to-solve tasks, we created a natively compiled version of CPAchecker for Nacpa. Nacpa is a conceptually simple framework, yet proved to be competitive in SV-COMP 2025.

1 Verification Approach

Nacpa is a meta-verifier for C that implements an efficient parallel portfolio on the basis of an external verifier—currently CPAchecker [1, 2]—with two goals: (1) create a technologically simple parallel portfolio of different verifier configurations, and (2) achieve faster startup times than CPAchecker can achieve in its default distribution that is based on the Java Virtual Machine (JVM). Nacpa does not implement any new program analysis but delegates all analysis tasks to the external verifier.

Parallel Portfolio. SV-COMP [3] gives verifiers a time limit of 900 s *CPU time* per task. This provides verifiers with a strong incentive not to run analyses concurrently, but sequentially, so that CPU time is only used when beneficial. But verifiers employing a sequential portfolio of analyses—like CPAchecker before version 4.0—have a practical problem: The analyses are executed in a fixed sequence and the analysis that can successfully solve a verification task may be scheduled late in that sequence. This means that it can take a significant amount of time to solve an otherwise fast-to-solve verification task. While this approach is valid for SV-COMP, where it—in essence—only matters that a result is produced within the *CPU* time limit, it is not a good day-to-day experience for users and makes debugging difficult.

Nacpa addresses this issue by executing all analyses in parallel. Figure 1 shows the workflow of Nacpa. After receiving a program P and a specification φ, Nacpa first extracts features from the program under verification by calling CPAchecker

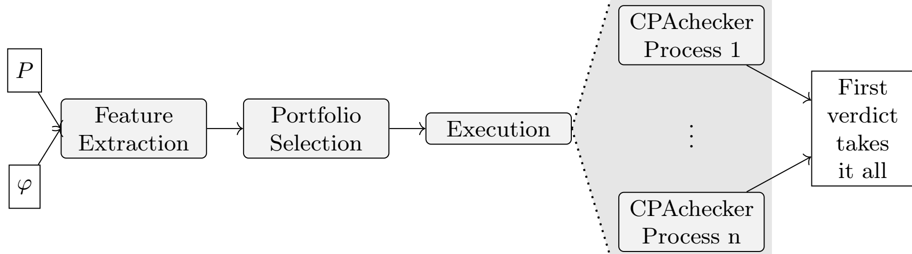

Fig. 1: Workflow of Nacpa

with a configuration [2, 4] that extracts and outputs these features. Based on the features, Nacpa selects one of multiple pre-defined analysis portfolios (each analysis portfolio is a set of configurations together with a wall-time limit). For each analysis in the selected portfolio, Nacpa launches a separate verifier process. The first process that finishes with a verdict of `TRUE` or `FALSE` wins, and Nacpa reports this verdict. If all processes finish with `UNKNOWN`, Nacpa returns `UNKNOWN`.

While Nacpa currently only calls CPAchecker and no other verifiers, we use CPAchecker as-is and call it as an external process. Because of this, we are certain that this external parallelization generalizes well to other verifiers.

Faster Startup Time. Many successful verifiers are implemented in Java [2,5–8]. But setting up the JVM, loading all the classes and starting the verifier can take several seconds [9]. Multiplying this overhead by the number of analyses in the portfolio can lead to a significant overhead. To improve on this, we compile CPAchecker to a native executable with Oracle GraalVM. This not only reduces the overhead of the parallel portfolio but also significantly improves the run time for fast-to-solve tasks, compared to the traditional CPAchecker.

2 Software Architecture

Nacpa 1.0 leverages CPAchecker (revision b24a0863) as verification backend. We use Oracle GraalVM 22.0.1 to compile this revision of CPAchecker to a native executable. The native executable is bundled with the necessary third-party Libraries, e.g., MathSAT 5, and the necessary configurations for Nacpa.

Nacpa itself is written in Go because it has parallelization directives built into the language and compiles to a statically linked executable with low overhead.

Portfolio Selection. Nacpa uses seven hard-coded portfolios. The configurations of these portfolios are taken from CPAchecker and are semantically equivalent to the configurations used by CPAchecker in SV-COMP 2025, with two exceptions: (1) we discard almost all time limits (see below) and (2) if the program contains recursive function calls or concurrency features (`pthread_create`), we add configurations for recursive and concurrent programs to the selected portfolio. This is different from CPAchecker's SV-COMP submission, which starts with analyses that do not support recursion or concurrency, and only switches to supporting

configurations when the original analyses fail because one of these features is encountered. Compared to CPAchecker, Nacpa's approach to recursion and concurrency is simpler, but solves the same number of tasks.

Managing the Portfolio. Nacpa starts a separate process for each analysis. Nacpa then simultaneously waits on all processes to finish. Whenever a process finishes, Nacpa parses the produced console output for the reported verdict. If the run did not crash and the verdict is `TRUE` or `FALSE`, Nacpa terminates all remaining processes and reports this verdict to the user. Otherwise, Nacpa continues to wait on the remaining processes.

Resource Limits. To enforce time limits on the individual analysis runs, we rely on Go's internal process management. Nacpa runs most analyses without any time limit. There are two exceptions: data-flow analysis and symbolic execution are limited to only a few seconds of runtime each (5 s and 10 s, respectively), because these analyses only help on fast-to-solve tasks. Nacpa does not enforce any memory limits on the individual analysis runs. As soon as one analysis runs into the SV-COMP memory limit, Nacpa dies.

Native Compilation. Compiling a large project like CPAchecker to a native binary poses several challenges [9]. The biggest challenge for Nacpa is the extensive use of Java reflection in CPAchecker: When GraalVM builds the native binary, it only includes classes that are reachable from the program entry. It is not able to derive classes that are reached through code reflection, and these will miss during run time. To avoid this we need to tell GraalVM about them when starting the compilation process. We contribute a build script that collects this reflection information from exemplary CPAchecker runs.

3 Strengths and Weaknesses

Regarding its analysis, Nacpa fully depends on CPAchecker [2] and shares all strengths and weaknesses. The parallel portfolio and native compilation introduce some additional strengths and weaknesses.

Strengths. The parallel portfolio of Nacpa is conceptually simple, with the implementation consisting of less than 800 lines of source code. Nacpa is also conceptually independent of CPAchecker. It implements a parallel portfolio that is independent of CPAchecker's internal parallel portfolio. Other verifiers can be used in the backend by adding the commandline to call to Nacpa and adjusting the verdict parsing for the new output.

Specific to CPAchecker, Nacpa significantly speeds up the analysis for fast-to-solve verification tasks (similar to the speed up CPA-Daemon provides with its native backend [9]).

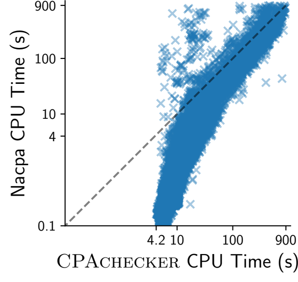

Fig. 2: Comparison of the CPU-time seconds spent by CPAchecker and Nacpa for each solved task

Figure 2 shows the CPU time seconds that CPAchecker (x-axis) and Nacpa (y-axis) require for each verification task in SV-COMP 2025 [3]. The plot only shows data for tasks that both CPAchecker and Nacpa solved correctly. Each data point below the diagonal represents a task for that Nacpa is faster than CPAchecker, and each data point above the diagonal represents a task for that CPAchecker is faster than Nacpa. The plot shows that Nacpa is significantly faster than CPAchecker for a large number of tasks. Nacpa's fastest correct verification run (0.11 s CPU time) is a magnitude faster than CPAchecker's (4.2 s CPU time).

On a small number of tasks, Nacpa's different configuration of parallel-portfolio strategies leads to some better verification results than CPAchecker: Nacpa solves 120 tasks in SV-COMP 2025 that CPAchecker can not solve. This accounts for 0.5 % of all tasks that Nacpa solved correctly.

Weaknesses. Because Nacpa splits its strategies into separate processes, the exchange of information is more difficult than in a multi-threaded approach like CPAchecker's internal parallel portfolio. For example, each analysis run that Nacpa starts parses the program on its own, while CPAchecker's internal parallel portfolio only parses the program once for all analyses. But we observe that this redundant program parsing has no significant negative effect in SV-COMP.

In contrast, the native compilation of CPAchecker sometimes has strong disadvantages: While it starts significantly faster than traditional CPAchecker, it does not provide a just-in-time compiler. For some tasks, this leads to worse performance than the JVM provides (cf. our experiments with CPA-Daemon [9]). We can see outliers above the diagonal in Fig. 2. For these tasks, Nacpa is significantly slower than CPAchecker. This, together with the different configuration of parallel-portfolio strategies, leads to some worse verification results [3] than CPAchecker: CPAchecker solves 446 tasks in SV-COMP 2025 that Nacpa can not solve. This accounts for 2 % of all tasks that CPAchecker solved correctly.

4 Setup and Configuration

Nacpa 1.0, the version used for SV-COMP 2025, is shipped as a statically linked binary and with a CPAchecker version natively compiled for Ubuntu 24.04 on x86.

Installation. The Nacpa 1.0 distribution is available on Zenodo [10]. The easiest way to install Nacpa is through fm-weck [11]. The following command installs Nacpa into a new directory called `nacpa/`:

```
pipx run fm-weck install -d nacpa/ nacpa:1.0
```

Use. To run Nacpa on program `prog.c` with program property `spec.prp` and data-model `ILP32`, execute from the directory where Nacpa is installed:

```
./bin/nacpa --spec spec.prp --data-model ILP32 prog.c
```

Nacpa requires the program-under-verification to be preprocessed. Nacpa supports the data-models `ILP32` and `LP64` and all SV-COMP properties.

Project Information. Nacpa participates in all categories of SV-COMP. It is maintained by Henrik Wachowitz and Thomas Lemberger at LMU Munich.

Data-Availability Statement. The source code of Nacpa is available at https://gitlab.com/sosy-lab/software/nacpa and the version used in SV-COMP 2025 is archived at Zenodo [10]. Nacpa is licensed under Apache-2.0.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

References

1. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
2. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
3. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
4. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features: A simple but effective approach. In: Proc. ISoLA. pp. 144–159. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11, https://www.sosy-lab.org/research/pub/2018-ISoLA.Strategy_Selection_for_Software_Verification_Based_on_Boolean_Features.pdf
5. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
6. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Taipan 2023 (competition contribution). In: Proc. TACAS (2). pp. 582–587. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_40
7. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: Ultimate GemCutter and the axes of generalization (competition contribution). In: Proc. TACAS (2). pp. 479–483. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_35
8. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
9. Beyer, D., Lemberger, T., Wachowitz, H.: CPA-Daemon: Mitigating tool restarts for Java-based verifiers. In: Proc. ATVA. Springer (2024)
10. Lemberger, T., Wachowitz, H.: Nacpa release 1.0. Zenodo (2024). https://doi.org/10.5281/zenodo.14203473
11. Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3