CPA-Daemon: Mitigating Tool Restarts for Java-Based Verifiers

Dirk Beyer, Thomas Lemberger, and Henrik Wachowitz

LMU Munich, Munich, Germany

Abstract. We present CPA-Daemon, a microservice for continuous software verification of C code. CPA-Daemon provides full access to the verifier CPAchecker, but adds a clear network interface based on gRPC that abstracts from three different modes of execution: (1) running CPAchecker in a separate JVM, (2) running CPAchecker as a native executable compiled with GraalVM, and (3) running CPAchecker in a shared, continuously-running JVM. The last two are novel execution modes that greatly improve the response time of verification in different verification scenarios and enable the seamless integration of CPAchecker as an engine in other verification tooling. Our comparative evaluation shows that CPA-Daemon reduces the response time on small verification tasks down to 17 %, and that it can reduce the response time of existing cooperative verification techniques down to 70 %. While our implementation focuses on CPAchecker, the conceptual ideas are of general nature and can serve as a solution for other verification tools that face similar JVM-specific issues. CPA-Daemon is open source and available at https://gitlab.com/sosy-lab/software/cpa-daemon.

1 Introduction

We present the tool CPA-Daemon (Fig. 1), which is a microservice for continuous software verification with CPAchecker [1]. Our goal is to encourage a shift of perspective in the verification community: Verification tools should no longer be seen as monolithic systems, but as components in a larger verification ecosystem. The well-established microservice architecture suggests itself to this new paradigm: verifiers can still be developed independently, in different programming languages and with different dependencies. But through a clear network interface and existing microservice infrastructure, they can be seamlessly and efficiently integrated into larger verification tooling.

CPAchecker is a successful formal verifier for C programs, written in Java. It is used as a component in multiple other verification approaches [2–11]. Across these, we repeatedly observe two issues: First, CPAchecker is a command-line tool with no API (like most verifiers). This makes its integration in frameworks unnecessarily difficult. Second, small or "simple" verification tasks suffer from the large overhead introduced by CPAchecker's JVM start-up time. Historically, fast-to-solve verification tasks are considered uninteresting in research. This changes with incremental and cooperative verification approaches. They often divide one

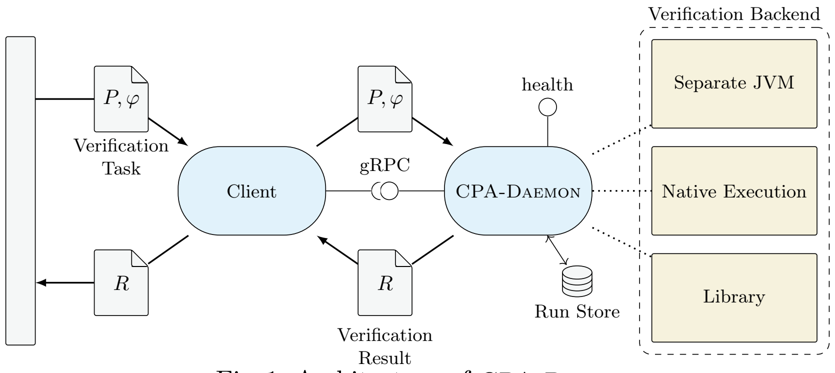


Fig. 1: Architecture of CPA-Daemon

large verification task into multiple smaller tasks—either iteratively [3, 5, 12–14] or as a a static decomposition [8, 15–19]. While this reduces the verification work per task, it multiplies the negative effect of a slow tool start-up. The easy solution would be to simply not use Java-based verifiers like CPAchecker. But this eliminates 7 out of 15 C verifiers that participated in SV-COMP 2024 [20], including the three highest-ranked tools. CPA-Daemon addresses both issues: (1) CPA-Daemon provides a clear network interface based on gRPC [21]. This enables a fast integration of CPAchecker in other frameworks. (2) CPA-Daemon provides different modes of operation to address different use cases, including fast-response scenarios.

Figure 1 shows the overall architecture of CPA-Daemon. CPA-Daemon decouples the user interface from the verification backend: it allows both local and distributed use through a unified gRPC interface. Clients control verification runs through that interface, and CPA-Daemon reports verification results through it. A health interface is available to check the status of CPA-Daemon, and a run store persists verification runs and results. CPA-Daemon provides three backends:

(1) **Separate JVM**: Run the original CPAchecker in a separate, short-lived JVM for each verification run.
(2) **Native Execution**: Run a native executable of CPAchecker that was precompiled with GraalVM [22].
(3) **Library**: Load and run CPAchecker's code in the continuously-running JVM of CPA-Daemon, similar to a software daemon.

Both Native Execution and Library are novel execution modes of CPAchecker that did not exist before. While Native Execution can also be executed stand-alone from a command line, Library is only available through CPA-Daemon.

To evaluate the three backends, we use the largest available benchmark set for the verification of C code, sv-benchmarks [23]. In this process, we show that CPA-Daemon significantly improves the response time on fast-to-solve verification tasks (significantly reduced to 17 % of the original response time).

In addition, we show that this faster response time *per small task* also benefits iterative verification techniques on more complex verification tasks. We do this on the example of component-based CEGAR (C-CEGAR) [3]. C-CEGAR is a verification technique that iteratively calls verifier components as executables.

If we replace these verifier executions by CPA-Daemon, it reduces the response time of C-CEGAR to 70 % (compared to previous work).

Contributions. This work provides the following contributions:

- CPA-Daemon introduces two new flavors of CPAchecker: Native Execution and Library. Both enable a near-instantaneous runtime of CPAchecker, a requirement for the integration in many modern software-developer workflows.
- CPA-Daemon's microservice architecture provides a programming-language-agnostic interface to CPAchecker. We provide an exemplary command-line client written in Python.
- We perform a thorough experimental evaluation on sv-benchmarks that shows significant performance gains through CPA-Daemon.
- We demonstrate how to turn a software verifier into a microservice. This is a requirement to lifting verifiers into the cloud. The presented concepts are not specific to CPAchecker and should transfer to other Java-based verifiers.

Related Work. This work shares the goals of decomposition, loose coupling, cooperation, and modular architectures with the approaches to apply the paradigms of *verification by transformation* [24] and *cooperative verification* [25].

There is research on scaling verification through incremental techniques [5,26–30], and distributed computing [18,31–36]. While CPA-Daemon is a microservice and this enables work distribution, we do not use distributed computing in this paper. Instead, we focus on another aspect of scalability that is relevant for Java-based verifiers: reduction of the overhead of tool restarts.

The program-analysis framework Ultimate [37] has decomposed verification frontends and backends into separate components that can be plugged together with the Eclipse Plug-in Development Environment (PDE). Four verifiers are compositions of these components: Ultimate Automizer [38,39], Ultimate Kojak [40], Ultimate Taipan [41], and Ultimate GemCutter [42]. CPA-Daemon decomposes the user interface and verification backend into separate components and makes it configurable through the modern open-source framework Quarkus [43].

CoVeriTeam Service [44] is a web service that allows users to run verification tools. Users specify options and the download location for a tool through a YAML-based configuration [45,46], which makes the service flexible in regards to the tools it can run. However, the main objective of CoVeriTeam Service is to enable users to use, integrate, or try out existing tools and tool combinations, without having to install them locally. It places a low priority on performance and scalability.

Unite [47] is also a framework for exposing verifiers to the web. Unite serves as an adapter that transforms a verifier into a web service that is compliant with the Open Services for Lifecycle Collaboration (OSLC) standard. OSLC-compliant clients can then directly communicate with the verification service. Unite executes each verification run as a fresh process. We imagine CPA-Daemon in collaboration with this framework: Integrating CPA-Daemon as a backend verifier in Unite would allow users to benefit from the performance improvements of CPA-Daemon without changing their OSLC-compliant clients.

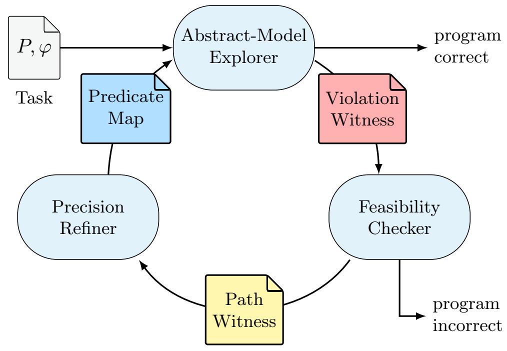


Fig. 2: Verification approach of C-CEGAR (adopted from [3])

Both CoVeriTeam Service and Unite make arbitrary verifiers available via the web. They do not try to improve the performance of the verifiers themselves but are merely interfaces bridging web requests and verifiers. In contrast, CPA-Daemon only focuses on CPAchecker with the goal to improve its verification performance. CPA-Daemon also exposes a network API of CPAchecker, but the interface is tailored to CPAchecker.

2 Background

CPAchecker [1] is a successful and versatile framework for software verification. It is a large software project that is written in Java, has been in development since 2007, has over 390 000 lines of code, has very high activity with a large number of users and developers, and would require more than 106 years of reimplementation effort for a single software engineer [48].

Counterexample-guided abstraction refinement [49] (CEGAR) is an important approach in model checking [50]. It derives a program abstraction that is as precise as necessary to prove the correctness of a program, yet as coarse as possible to keep the verification effort low. Component-based CEGAR (C-CEGAR, Fig. 2) is the decomposition of CEGAR into individual components: Separate, off-the-shelf tools implement the three steps of the CEGAR algorithm. The Abstract-Model Explorer always verifies the full program-under-verification, but with a changing precision. It starts with a coarse precision that becomes increasingly precise in later iterations. If the Abstract-Model Explorer reports no violation, C-CEGAR stops and the program is assumed to be correct. If the Abstract-Model Explorer reports a violation, the Feasibility Checker checks a subset of program executions that may lead to that violation. If one of them actually leads to the violation, the program is assumed to be incorrect. If none leads to a violation, the Precision Refiner extracts a new precision from that subset of program executions and gives it to the abstract model explorer for a new verification run. In previous work [3], the most effective configuration of C-CEGAR used three instances of CPAchecker, each instance configured differently to fulfill the corresponding C-CEGAR component. While the most effective, this configuration led to a median run-time increase of 280 %.

3 CPA-Daemon Architecture

We implement CPA-Daemon in Quarkus, a Java EE framework for web services. Quarkus enables users to easily configure server-related parameters through environment variables and command-line arguments, e.g., how many verification requests to handle in parallel and the verification backend to use. CPA-Daemon comprises about 5000 source lines of Java code and has an estimated development effort of 7 person-months.

The communication with clients happens through gRPC, an open-source request-response protocol that allows stateful network communication through remote-procedure calls in code. A gRPC interface is defined through a set of message types (requests and responses) and a set of methods that can be called by clients. From this interface, users can auto-generate client libraries for various programming languages.

As an example, CPA-Daemon has a method `StartRun` that clients can use to schedule a new verification run. Listing 1 shows three gRPC messages that are used with this method. It receives a `StartRunRequest` and returns a `RunResponse`. A `StartRunRequest` consists of the program to verify, the CPAchecker config, the program specification to check, and some additional options. A `RunResponse` can either contain an error or an embedded message of type `Run`. A Run contains a unique `name` and a `status` that signals whether the Run is pending, running, or finished. If the run is finished, it also contains a result verdict, one or more result messages that give more information about the verdict, and run details (e.g., produced output files). If the run is not finished, these values are `null`. The full definition of gRPC methods and messages is available in the CPA-Daemon repository.

Example Client. We provide an examplary Python client that allows users to send verification tasks to CPA-Daemon through the command line. It illustrates a possible sequence of communication between clients and CPA-Daemon, from starting a verification run to signaling that its information can be dismissed. Figure 3 shows the procedure that the client follows to do this. Each arrow from left (Client) to right (CPA-Daemon) is labelled with the gRPC method that the client calls at the CPA-Daemon instance, and the sent request messages. Each arrow from right to left is labelled with the gRPC response message that the CPA-Daemon instance answers.

Separate-JVM Backend. The verification backend Separate JVM executes each verification run as a new, independent CPAchecker process in its own JVM. With this backend, CPA-Daemon provides a microservice to CPAchecker without any changes in the verification behavior. This backend also provides the baseline for the other two verification backends.

Running CPAchecker in a separate, new JVM on each verification request has the following benefits: (a) Compared to Native Execution, the JVM may apply just-in-time (JIT) optimizations to the code of CPAchecker. These optimizations can be more powerful than compile-time optimizations because they are based on the current runtime information. (b) Compared to Library, the memory management is simple because each process has its own memory and a separate garbage collector. Once a process closes, the operating system automatically frees all of its memory.

```
message StartRunRequest {
  string programCode = 1;
  string config = 2;
  string specification = 3;
  /*..snip other options..*/
}
/*..snip other request messages..*/

message RunResponse {
  oneof result {
    string error = 1;
    Run run = 2;
  }
}
message Run {
  string name = 1;
  RunStatus status = 2;
  Verdict resultVerdict = 3;
  repeated string resultMessage = 4;
  RunDetails details = 5;
}
```

Listing 1: Data structures for communication

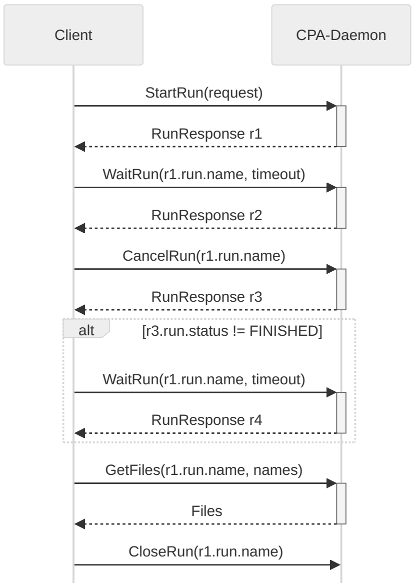


Fig. 3: Sequence diagram of the Python client's communication with CPA-Daemon

But it also has the following drawbacks: (a) Optimization information is not shared between different verification runs. (b) For each verification run, a new JVM process has to be started. Launching a new JVM also includes loading all relevant CPAchecker classes, which can take several seconds.

Native-Execution Backend. Native compilation of program code is an established practice to reduce an application's startup time and memory overhead. GraalVM Native Image [22] enables native compilation for Java: it is a free toolchain that compiles Java bytecode to native machine code. We use GraalVM to compile CPAchecker to a native executable and give that to the Native Execution backend, which executes each verification run as a new, independent process that runs this native CPAchecker executable.

This has the following benefits: (a) The native executable starts faster than the JVM process, as no class-loading and no code interpretation is required. (b) Memory management does not induce additional effort (see Separate JVM).

But with the current GraalVM version, it has the following drawbacks: (a) JIT optimizations are not possible. (b) There are fewer garbage collectors available than for the JVM. (c) Not all Java features are supported yet. In our case, the last drawback is negligible: Only the SVG image export of CPAchecker misses support in the native CPAchecker executable.

There are also two generic challenges in the native compilation of CPAchecker: First, GraalVM Native Image requires a list of all Java classes that may be accessed through reflection during runtime. If the native CPAchecker executable tries to access a class through reflection that we failed to provide to GraalVM

Native Image, the execution crashes. Unfortunately, CPAchecker heavily relies on reflection to initialize its concrete verification configuration at startup. The classes accessed through reflection depend on the concrete CPAchecker configuration, the program specification, and whether recursion or concurrency occurs in the program under verification. To collect all relevant Java classes, we run CPAchecker on a pre-selected list of verification tasks and include a tracer that records all classes accessed through reflection. We selected the vrification tasks such that they cover various configurations of CPAchecker, all program specifications that are used in SV-COMP [20], and programs with and without recursion and concurrency.

Library Backend. Like CPAchecker, CPA-Daemon is written in Java. This enables it to directly import and use CPAchecker's Java classes at runtime. With the Library backend, CPA-Daemon manually translates a given verification task into the corresponding CPAchecker configuration and then runs the verification through CPAchecker's public Java methods. This happens, for all runs, in the same JVM that CPA-Daemon runs in—just in a separate thread per run, to allow for multiple in parallel.

Running all verification runs in a single JVM has the following advantages: (a) Similar to the Separate JVM backend, the JVM may apply just-in-time (JIT) optimizations to the code of CPAchecker. (b) Unlike the Separate JVM backend, this optimization information is potentially shared between different verification runs. (c) The relevant CPAchecker classes need only be loaded once for the first verification run, and are then kept in memory. (d) CPA-Daemon can have more fine-grained control over the behavior of CPAchecker. Currently, CPA-Daemon uses this control to manage the point in time at which CPAchecker exports statistics and produces output files.

However, importing CPAchecker as library also has one drawback: Running all verification runs in a single JVM adds complexity to memory management, as the garbage collector must handle the memory of all runs simultaneously. This increases the risk of heap-space exhaustion.

Health Interface. CPA-Daemon provides a REST interface that allows to query its health status. This interface is compatible with modern microservice tooling like Kubernetes. We implement a simple health heuristic: CPA-Daemon can be configured to signal that it requires a restart when a fixed number of verification runs have finished. We also provide a watchdog that monitors the health status of CPA-Daemon and restarts it on demand.

Run Store. CPA-Daemon keeps all information about runs in a run store. This includes a run's outputs on the console, produced output files, and current status.

4 Evaluation

We answer the following research questions with an experimental evaluation:

RQ 1 Consistency: Does CPA-Daemon produce results that are consistent with CPAchecker?

RQ 2 Response Time: Have the CPA-Daemon backends Native Execution and Library a smaller response time compared to Separate JVM?

RQ 3 Application to Cooperative Verification: Can cooperative verification approaches benefit from CPA-Daemon, compared to off-the-shelf CPAchecker?

Tool Versions. We use verification tasks from sv-benchmarks [51] in the version used for SV-COMP 2023 [52]. We use CPA-Daemon release 1.0 [53] (branch `main`), which includes CPAchecker [1] revision 44879 (branch `refactor-run-method`) as a sub-module. This branch of CPAchecker includes memory-leak fixes for use in Library. We use this version for both standalone CPAchecker and as the basis for the different CPA-Daemon backends. We use Oracle GraalVM 21+35.1 to run CPA-Daemon. To run C-CEGAR, we use CoVeriTeam [54] revision 722b8d4. For reliable resource measurement, we use BenchExec [55] revision 6e7f9c5. Response time is the elapsed time while the run was executed (called 'wall time' in BenchExec).

Experiment Design. For RQ 1 and RQ 2, we run CPAchecker as a standalone tool and CPA-Daemon with its three different backends, each with the CPAchecker configuration of SV-COMP 2023. We use the Python client that is distributed with CPA-Daemon. We configure CPA-Daemon and the accompanying watchdog to perform a server restart after every 100th verification run. The server restart is implicitly measured in the response time when a client tries to run a verification task while CPA-Daemon is restarting. We use all verification tasks with the reachability-safety property *unreach-call* that are easy to solve for CPAchecker. We define 'easy to solve' as all tasks that CPAchecker can solve correctly in less than 60 s of response time; this accounts to 1 416 tasks. While we are interested in the response time and BenchExec can measure that well, it *limits* the run time using a CPU time limit. So we give both CPAchecker and CPA-Daemon a generous maximum time limit of 300 s of CPU time per task.

For RQ 3, we compare C-CEGAR in its original configuration [3] with CPAchecker in all three components, and C-CEGAR with the Python client for CPA-Daemon in all three components. For each verification run, a single new local instance of CPA-Daemon is started. All three Python clients connect to this instance. We use all 8 872 available verification tasks with the reachability-safety property *unreach-call*. Each full C-CEGAR run gets a maximum memory limit of 15 GB and a CPU-time limit of 900 s per task. These resource limits are the same that SV-COMP uses. The start of CPA-Daemon at the beginning of each verification run is part of the resource measurements.

All experiments are carried out on machines equipped with one CPU (Intel Xeon E3-1230 v5, 3.4 GHz, 8 processing units) and 33 GB of RAM, running Ubuntu 22.04.3 LTS with 64 bit and GNU/Linux 5.15.0. All runs have access to all 8 processing units and at most 10 GB Java heap memory.

RQ 1 Consistency. We compare the overall verification results of all four considered approaches: CPAchecker as a standalone application, as well as CPA-Daemon with the three backends Separate JVM, Native Execution, and Library. Table 1 shows that CPA-Daemon with backends Separate JVM and Library provides results that are consistent with CPAchecker: Separate JVM and Library can solve two and one tasks more than CPAchecker, respectively. Backend

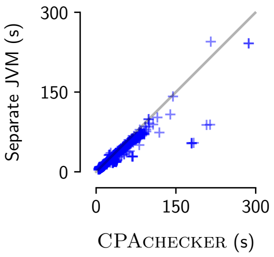


Fig. 4: Response time

Result	CPAchecker	CPA-Daemon Separate JVM	CPA-Daemon Native	CPA-Daemon Library
Correct	1 411	1 413	1 390	1 412
Wrong	0	0	0	0
Unknown	5	3	26	4

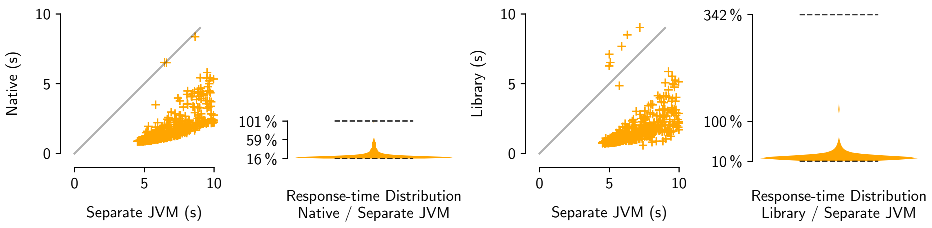
Table 1: Overall verification results

Native Execution can solve 21 tasks less than CPAchecker, due to a high number of non-optimized equals-comparison invocations in the explicit-state analysis. CPAchecker, Separate JVM, and Library perform a JIT-optimization for these comparisons and can solve them very quickly. The response-time comparison of CPAchecker with Separate JVM in Fig. 4 shows that CPA-Daemon does not introduce significant overhead. The median response time per verification task of CPA-Daemon is only 89 % compared to the response time of CPAchecker.

CPA-Daemon is consistent with CPAchecker. Under certain conditions, backend Native Execution shows worse results than the other backends.

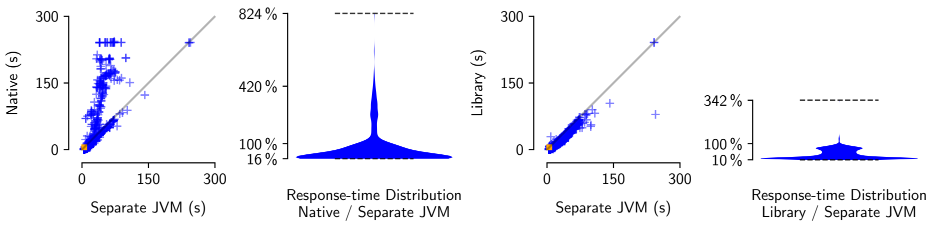
RQ 2 Response Time. Figures 6a and 6c compare the response times (in seconds) of CPA-Daemon with backend Separate JVM (on the x-axis) and CPA-Daemon with backend Native Execution or Library (on the y-axis), respectively. We consider only verification tasks that both backends can solve in 10 s or less. This highlights fast-response scenarios, the focus of CPA-Daemon. Points below the diagonal show that Native Execution (resp. Library) responds faster than Separate JVM. The violin plots in Fig. 6b and Fig. 6d summarize the effect that the backends have on the response time compared to backend Separate JVM, and how this effect distributes across tasks. A value below 100 % means that CPA-Daemon with backend Native Execution (or Library) is faster than Separate JVM. A value above 100 % means that CPA-Daemon with backend Native Execution (or Library) is slower than Separate JVM. It is visible that for most of the tasks, both Native Execution and Library are significantly faster than Separate JVM. In the worst case, Native Execution takes about the same time as Separate JVM. For Library, there is a small number of tasks for which it is slower than Separate JVM. We attribute this to the more complex memory management with Library. The median response time of Library is only 17 % of the response time of Separate JVM, and in 95 % of the cases, Library requires less than 59 % the time that Separate JVM takes.

To evaluate scalability, Figs. 6a and 6c compare the response times of backends over all considered verification tasks. This shows that CPA-Daemon with backend Native Execution is still significantly faster than Separate JVM for most tasks (median: 27 %). But there are also tasks for which Native Execution is visibly slower than Separate JVM. This can be attributed to the difference between native execution and interpretation by the JVM with JIT optimiza-



(a) Response time (b) Violin plot (c) Response time (d) Violin plot

Fig. 5: Response time (in s) per verification task of backends Native Execution and Library in relation to backend Separate JVM (tasks with response time $\leq 10\,$s)



(a) Response time (b) Violin plot (c) Response time (d) Violin plot

Fig. 6: Response time (in s) per verification task of backends Native Execution and Library in relation to backend Separate JVM (all tasks)

tions. In contrast, we observe that Library stays competitive to Separate JVM over all tasks: The response times of both backends converge, but Library is faster than Separate JVM in almost all cases.

In fast-response scenarios, both Native Execution and Library are significantly faster than Separate JVM for the majority of verification tasks. For longer-running tasks, Native Execution does not scale well, but Library scales comparable to Separate JVM and is a bit faster for most tasks.

RQ 3 C-CEGAR. For this research question, we use all 8 872 ReachSafety verification tasks of SV-COMP and a time limit of 900 s. Figure 7 compares the response times (in seconds) of C-CEGAR in its original configuration with CPAchecker, and C-CEGAR that instead connects to CPA-Daemon with backend Native Execution and backend Library. On the x-axis, the quantile plot shows the n$^{\text{th}}$ fastest correct result for each approach. On the y-axis, the quantile plot shows the response time that this n$^{\text{th}}$ fastest correct result took. For example, the plot shows at x-value 500 that C-CEGAR with Native Execution produces its 500 fastest correct results in 3 s or less per result, while C-CEGAR with CPAchecker and C-CEGAR with Library require for their 500 fastest correct results up to 5 s per result. In fact, C-CEGAR with CPAchecker and C-CEGAR with Library only start producing results at about 3 s (y-value at the start of their plots). Looking at the overall results, C-CEGAR with Native Execution and C-CEGAR with Library are both significantly faster than C-CEGAR with

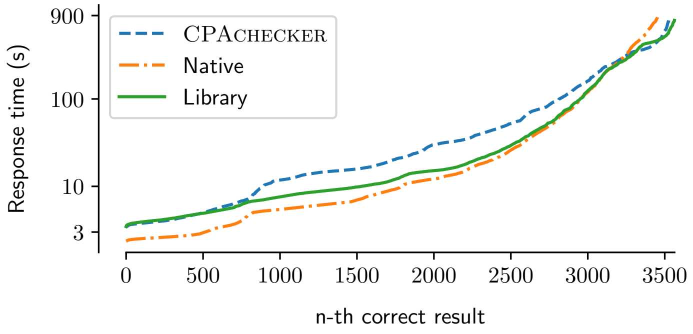


Fig. 7: Response time of C-CEGAR with: original CPAchecker, CPA-Daemon with backend Native Execution, and CPA-Daemon with backend Library; logarithmic scale on y-axis

the original CPAchecker: their y-values are significantly smaller for x-values 800 to 3 000. C-CEGAR with Library solves the most tasks (right end of plot). It solves 35 tasks more than C-CEGAR with the original CPAchecker, and reduces the median response time down to 70 %.

Figure 7 also visualizes the generic difference between the CPA-Daemon backends Native Execution and Library: Native Execution starts out faster than Library for fast-to-solve tasks, because it does not have to load Java classes for the first run (remember that C-CEGAR performs three verifier calls for each complete CEGAR iteration). But Library scales better for longer-running tasks, and the initial overhead is amortized if CPA-Daemon is called more often.

The cooperative verification approach C-CEGAR benefits from CPA-Daemon: The application of CPA-Daemon with Library reduces its median response time to 70 %.

5 Conclusion

CPA-Daemon is a microservice for continuous software verification. It uses two systematic approaches to reduce the response time of CPAchecker: native compilation and daemonization via use as a library. Our evaluation shows that CPA-Daemon can reduce the median response time of CPAchecker to 26 % and to 17 %, respectively, for simple verification tasks. This speed up and architecture finally enables the integration of Java-based software verifiers into continuous software-development workflows. Perhaps even more importantly, we have shown how to convert a standalone verifier into a microservice with a simple interface. We expect these approaches to transfer to other Java-based verification tools.

Data-Availability Statement. CPA-Daemon release 1.0 is freely available [53]. The tools that we used for our experiments and all data are available as reproduction artifact [56] and at: `https://www.sosy-lab.org/research/cpa-daemon/`

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and the LMU PostDoc Support Fund.

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. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Proc. FASE. pp. 49–70. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_3
3. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing software verification into off-the-shelf components: An application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064
4. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
5. Beyer, D., Jakobs, M.C., Lemberger, T.: Difference verification with conditions. In: Proc. SEFM. pp. 133–154. LNCS 12310, Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_8
6. 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
7. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141. IEEE (2019). https://doi.org/10.1109/ASE.2019.00121
8. Haltermann, J., Jakobs, M.C., Richter, C., Wehrheim, H.: Parallel program analysis via range splitting. In: Proc. FASE. pp. 195–219 (2023). https://doi.org/10.1007/978-3-031-30826-0_11
9. Haltermann, J., Wehrheim, H.: CoVEGI: Cooperative verification via externally generated invariants. In: Proc. FASE. pp. 108–129. LNCS 12649 (2021). https://doi.org/10.1007/978-3-030-71500-7_6
10. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
11. Novikov, E., Zakharov, I.S.: Towards automated static verification of GNU C programs. In: Proc. PSI. pp. 402–416. LNCS 10742, Springer (2017). https://doi.org/10.1007/978-3-319-74313-4_30
12. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
13. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11
14. Yin, B., Chen, L., Liu, J., and P. Cousot, J.: Verifying numerical programs via iterative abstract testing. In: Proc. SAS. pp. 247–267. LNCS 11822, Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_13
15. Pauck, F., Wehrheim, H.: Together strong: Cooperative Android App analysis. In: Proc. ESEC/FSE. pp. 374–384. ACM (2019). https://doi.org/10.1145/3338906.3338915
16. Yang, G., Do, Q.C.D., Wen, J.: Distributed assertion checking using symbolic execution. ACM SIGSOFT Softw. Eng. Notes **40**(6), 1–5 (2015). https://doi.org/10.1145/2830719.2830729

17. Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Proc. TACAS (2). pp. 249–265. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_15
18. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPoPP. pp. 202–216. ACM (2020). https://doi.org/10.1145/3332466.3374529
19. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: Proc. ASE. pp. 753–764. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115686
20. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
21. Google: gRPC. https://grpc.io/, accessed: 2024-08-26
22. Oracle: GraalVM. https://www.graalvm.org/, accessed: 2024-08-26
23. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, accessed: 2024-08-26
24. Beyer, D., Lee, N.Z.: The transformation game: Joining forces for verification. In: Proc. Festschrift Katoen 60th Birthday. Springer (2024), https://www.sosy-lab.org/research/pub/2024-Katoen60.The_Transformation_Game_Joining_Forces_for_Verification.pdf
25. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
26. Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS. pp. 207–213. LNCS 10206 (2017). https://doi.org/10.1007/978-3-662-54580-5_12
27. Asadi, S., Blicha, M., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N.: SMT-based verification of program changes through summary repair. Formal Methods Syst. Des. **60**(3), 350–380 (2022). https://doi.org/10.1007/s10703-023-00423-0
28. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
29. Seidl, H., Erhard, J., Vogler, R.: Incremental abstract interpretation. In: From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement. pp. 132–148. LNCS 12065, Springer (2020). https://doi.org/10.1007/978-3-030-41103-9_5
30. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013). https://doi.org/10.1145/2491411.2491429
31. Barnat, J., Havlícek, J., Rockai, P.: Distributed LTL model checking with hash compaction. ENTCS **296**, 79–93 (2013). https://doi.org/10.1016/j.entcs.2013.07.006
32. Yang, G., Qiu, R., Khurshid, S., Pasareanu, C.S., Wen, J.: A synergistic approach to improving symbolic execution using test ranges. Innov. Syst. Softw. Eng. **15**(3-4), 325–342 (2019). https://doi.org/10.1007/s11334-019-00331-9
33. Brain, M., Schanda, F.: A lightweight technique for distributed and incremental program verification. In: Proc. VSTTE. pp. 114–129. LNCS 7152, Springer (2012). https://doi.org/10.1007/978-3-642-27705-4_10

34. Lopes, N.P., Rybalchenko, A.: Distributed and predictable software model checking. In: Proc. VMCAI. pp. 340–355. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_24
35. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE. pp. 634–644. ACM (2018). https://doi.org/10.1145/3238147.3238195
36. Beyer, D., Kettl, M., Lemberger, T.: Decomposing software verification using distributed summary synthesis. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660766
37. Dietsch, D., Klumpp, D., Schüssele, F., Heizmann, M.: Ultimate repository. https://github.com/ultimate-pa/ultimate, accessed: 2024-08-26
38. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
39. Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate Automizer 2023 (competition contribution). In: Proc. TACAS (2). pp. 577–581. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_39
40. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: Ultimate Kojak with memory safety checks (competition contribution). In: Proc. TACAS. pp. 458–460. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44
41. 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
42. 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
43. RedHat: Quarkus. https://quarkus.io/, accessed: 2024-08-26
44. Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTeam Service: Verification as a service. In: Proc. ICSE, companion. pp. 21–25. IEEE (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00017
45. Beyer, D.: Conservation and accessibility of tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024), https://www.sosy-lab.org/research/pub/2024-Podelski65.Conservation_and_Accessibility_of_Tools_for_Formal_Methods.pdf
46. Beyer, D.: Tools for formal methods. https://fm-tools.sosy-lab.org/, accessed: 2024-08-26
47. Vašíček, O., Fiedor, J., Kratochvíla, T., Bohuslav, K., Smrčka, A., Vojnar, T.: Unite: An Adapter for Transforming Analysis Tools to Web Services via OSLC. In: Proc. ESEC/FSE. ACM (2022). https://doi.org/10.1145/3540250.3558939
48. Synposis: Black duck open hub: CPAchecker. https://openhub.net/p/cpachecker, accessed: 2024-08-26
49. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
50. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT (1999), https://www.worldcat.org/isbn/978-0-2620-3270-4
51. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2023 and Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627783

52. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
53. Beyer, D., Lemberger, T., Wachowitz, H.: CPA-Daemon release 1.0. Zenodo (2024). https://doi.org/10.5281/zenodo.13373907
54. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
55. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
56. Beyer, D., Lemberger, T., Wachowitz, H.: Reproduction package for ATVA 2024 submission 'CPA-Daemon: Mitigating tool restarts for Java-based verifiers'. Zenodo (2024). https://doi.org/10.5281/zenodo.11147333