Bubaak-SpLit: Split what you cannot verify (Competition contribution)

Marek Chalupa[1(✉)] * and Cedric Richter[2]

[1] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
mchalupa@ist.ac.at
[2] Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
cedric.richter@uol.de

Abstract. Bubaak-SpLit is a tool for dynamically splitting verification tasks into parts that can then be analyzed in parallel. It is built on top of Bubaak, a tool designed for running combinations of verifiers in parallel. In contrast to Bubaak, that directly invokes verifiers on the inputs, Bubaak-SpLit first starts by splitting the input program into multiple modified versions called *program splits*. During the splitting process, Bubaak-SpLit utilizes a *weak* verifier (in our case symbolic execution with a short timelimit) to analyze each generated program split. If the weak verifier fails on a program split, we split this program split again and start the verification process again on the generated program splits. We run the splitting process until a predefined number of *hard-to-verify* program splits is generated or a splitting limit is reached. During the main verification phase, we run a combination of Bubaak-Lee and Slowbeast in parallel on the remaining unsolved parts of the verification task.

1 Verification approach

Bubaak [7] is a program analysis tool that runs multiple verifiers at the same time, and uses ideas from runtime monitoring and enforcement [5,10] to mediate the communication of useful information between the verifiers, such as invariants or already explored parts of the program. As of this year, the verifiers can be executed in an arbitrary combination of sequential and parallel portfolio, fully dynamically based on the information learned during the verification process.

With Bubaak-SpLit, we explore *program splitting* [12,13] as a way to improve the scalability of the verification process. The main idea behind program splitting is to split a given program P into multiple subprograms $P_1, \ldots, P_n$ which then can be analyzed in parallel. As a result, Bubaak-SpLit can verify multiple subprograms with multiple verifier instances at the same time.

* Jury member

© The Author(s) 2024
B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 353–358, 2024.
https://doi.org/10.1007/978-3-031-57256-2_20

```c
1  int main(void) {
2    int x = nondet();
3    if( x >= 1000 ) abort();
4
5    if( x <= 10 ){
6      hard_to_verify_1(x);
7    } else {
8      hard_to_verify_2(x);
9    }
10 }
```

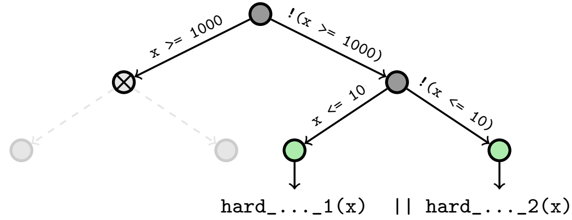

Fig. 1. Overview over the verification process of Bubaak-SpLit for the given example. Bubaak-SpLit splits program that are too hard to be verified by a weak verifier (gray nodes), stops for easy-to-verify nodes (crossed out nodes) and it proceeds until n hard-to-verify splits are found (green nodes).

Control-flow Splitting. Bubaak-SpLit adopts *control-flow splitting*[3] [13] for splitting programs into subprograms. Control-flow splitting splits a program P at the first branching point B creating two subprograms P^+ and P^-. P^+ and P^- each represent the program P when assuming that the branching condition at B is evaluated to `true` or `false` respectively. For example, Figure 2 depicts P^+ and P^- when splitting the program in Figure 1 at the first branching point in Line 3. Syntactically splitting a program might result in suboptimal splits [12] where one part of the split is *easy-to-verify* and the other remains *hard-to-verify*. To mitigate the problem of suboptimal splits, Bubaak-SpLit implements a dynamic splitting strategy: (1) we first check if the given program (or split) is hard-to-verify by running a weak verifier, (2) if it is hard-to-verify we split the program and repeat the process on the generated splits, (3) if it is not hard-to-verify we record the result of the weak verifier and continue with the other splits (if any). We continue this process until a fixed number of hard-to-verify splits is generated or a splitting limit is reached. If the problem is solved during the splitting process, we report the results of the weak verifiers.

Figure 1 provides an example of the splitting process. After splitting two times, Bubaak-SpLit identifies two hard-to-verify splits which are then verified by two verifiers in parallel in the main verification phase. Existing static splitting strategies for C programs [12] might stop after the first split, resulting in a suboptimal split (with little to no benefits for the verification process).

Verification technology. Bubaak-SpLit in SV-COMP 2024 utilizes verifiers based on *forward* and *backward symbolic execution*.

(Forward) symbolic execution (SE) [14] systematically explores program's executions from the initial location. Backward symbolic execution (BSE) [8] explores executions that reach a given (error) location and it does so by analyzing the program backwards from the locations. We employ a variant of BSE with

[3] Our variant of control-flow splitting was mainly inspired by Mooly Sagiv's invited talk "Scaling Formal Verification to Realistic Code with Applications to DeFi" at ETAPS 2023. Our implementation however splits C programs, not Solidity contracts.

```c
int main(void) { // P+
  int x = nondet();
  assume( x >= 1000 );
  abort();
}
```

```c
int main(void) { // P-
  int x = nondet();
  assume( !(x >= 1000) );
  if( x <= 10 ) ...
}
```

Fig. 2. Result of splitting the program from Figure 1 at the first branching point.

loop folding (BSELF) [8] which allows us to generate loop invariants and prove programs correct.

SE can very quickly identify easy-to-verify problems, so we use it with a short timeout as the weak verifier during splitting. Strong verifiers in the main verification phase are selected based on the property. For the property *unreach-call*, we use BSELF and SE (with no timeout) in parallel – BSELF to prove programs correct and SE to (mainly) find bugs. Other properties are not supported by BSELF. For checking termination properties, we run SE and *termination with inductive invariants with progress* (TIIP) [7]. For checking memory safety, we use only SE. Note that the splitting phase is executed for all properties.

2 Software architecture

BUBAAK runs verification tools in a combination of sequential and parallel portfolio. The verifiers are not composed into a fixed scheme, but they are invoked dynamically based on the information gathered during the verification process. In a bit more detail, the architecture of BUBAAK is inspired by *process algebras* [4] and is centered about *tasks* and their *rewriting*. The tool starts with the execution of a set of initial tasks; upon finishing, each task either yields a result, or rewrites itself into a new task or a set of new tasks. Whenever a task rewrites itself into a set of new tasks, it also specifies how the results of the new tasks should be aggregated into a single result. The important feature is that generating new tasks is not fixed in a static scheme: a task can rewrite itself into new tasks based on the context and information hitherto gathered about the program during the verification process.

What tasks are executed and how they are being rewritten is defined by a selected *workflow*. The workflow for splitting in SV-COMP 2024 is depicted in Figure 3. It defines the task *Split(P)* that takes program P and splits it into two parts as described in Section 1. This task is invoked as the initial task on the input program. After splitting the program, *Split* rewrites itself into two identical tasks *CCAndCheckWeak* that are invoked on those two splits. As the name suggests, the input split is compiled (into LLVM [1]) and the weak verifier is ran on it to check if the split is easy to solve. If a split is not easy to solve, the task *Split* is invoked on the split recursively, and this process continues until a pre-defined depth is reached, at which point instead of splitting further the workflow invokes the strong verifier.

Split(P) : $P^+, P^- :=$ `split`$(P) \to \wedge$ → $\mathit{CCAndCheckWeak}(P^+)$; → $\mathit{CCAndCheckWeak}(P^-)$

CCAndCheckWeak(P) : $bc :=$ `compile`$(P) \longrightarrow \mathit{CheckWeak}(bc, P)$

CheckWeak(bc, P) : `bubaak-lee`(bc) —[$\mathit{time} \geq 2s \wedge \mathit{depth} < 2$]→ $\mathit{Split}(bc, P)$; —[$\mathit{time} \geq 2s \wedge \mathit{depth} \geq 2$]→ $\mathit{CheckStrong}(bc)$

CheckStrong(bc) : $\longrightarrow \vee$ → `bubaak-lee`(bc) ; → `slowbeast-bself`(bc)

Fig. 3. The workflow of Bubaak-SpLit for SV-COMP 2024. For brevity, the scheme does not show errors handling and the result propagation.

Workflows are only an abstraction: internally, task execution and rewriting is implemented using an event loop that handles events coming from tasks, task creation and destruction, and the results aggregation.

The weak verifier is based on Bubaak-Lee and we run a combination of Bubaak-Lee and Slowbeast during the main verification phase. Bubaak and Slowbeast are implemented in Python, Bubaak-Lee is in C++. Both Slowbeast and Bubaak-Lee use Z3 [9] as the SMT solver.

3 Strengths and Weaknesses

Program splitting has been shown to improve the *runtime efficiency* [15] and *verification effectiveness* [11] of symbolic execution engines. By splitting the program into several parts, SE and BSE can analyze different parts of the program at the same time, which can lead to results being decided more quickly. In SV-COMP 2024 [6], Bubaak-SpLit was able to solve 60 benchmarks that Bubaak was not able to solve and 456 benchmarks were solved faster, often significantly. In comparison to Bubaak, Bubaak-SpLit misses several violations on the *ReachSafety* benchmarks. Most of them are due to the fact that we severely limit the execution time of SE during verification. Another problem is the scalability of our approach in the restricted setting of the SV-COMP. By splitting the program up to n times, we currently run up to 2^n verifiers at the same time. While in praxis this might significantly reduce the walltime, it also significantly reduces the cputime available to each verifier. Overall, Bubaak-SpLit inherits the strengths of the underlying analyses which allows the tool to perform well in the categories *ReachSafety* and *SoftwareSystems*. After SV-COMP 2024, we have found and fixed several bugs in the implementation of Bubaak-SpLit which might have severely limited its performance.

Acknowledgements This work was partially supported by the ERC-2020-AdG 10102009 grant.

Data-Availability Statement The submitted version of our tool contribution is archived and available at Zenodo [2]. The source code is also available on GitLab [3].

References

1. llvm.org. https://llvm.org, accessed: 2023-12-21
2. Bubaak-SpLit artifact (2023). https://zenodo.org/records/10202207
3. Bubaak-SpLit repository (2023), https://gitlab.com/mchalupa/bubaak
4. Baeten, J.C., Weijland, W.P.: Process algebra. Cambridge university press (1991)
5. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_1
6. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
7. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers - (competition contribution). In: TACAS 2023. LNCS, vol. 13994, pp. 535–540. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
8. Chalupa, M., Strejcek, J.: Backward symbolic execution with loop folding. In: SAS 2021. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
9. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS 2008. pp. 337–340. Springer (2008)
10. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 103–134. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_4
11. Haltermann, J., Jakobs, M., Richter, C., Wehrheim, H.: Parallel program analysis via range splitting. In: FASE 2023. LNCS, vol. 13991, pp. 195–219. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_11
12. Haltermann, J., Jakobs, M., Richter, C., Wehrheim, H.: Ranged program analysis via instrumentation. In: SEFM 2023. LNCS, vol. 14323, pp. 145–164. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_9
13. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: SAS 1998. LNCS, vol. 1503, pp. 200–214. Springer (1998). https://doi.org/10.1007/3-540-49727-7_12
14. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
15. Siddiqui, J.H., Khurshid, S.: Scaling symbolic execution using ranged analysis. In: OOPSLA 2012. pp. 523–536. ACM (2012). https://doi.org/10.1145/2384616.2384654

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.