2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Cooperative Software Verification via Dynamic Program Splitting

Cedric Richter
Carl von Ossietzky
Universität Oldenburg
Oldenburg, Germany
cedric.richter@uol.de

Marek Chalupa
Institute of Science
and Technology Austria
Klosterneuburg, Austria
mchalupa@ist.ac.at

Marie-Christine Jakobs
LMU Munich
Munich, Germany
m.jakobs@lmu.de

Heike Wehrheim
Carl von Ossietzky
Universität Oldenburg
Oldenburg, Germany
heike.wehrheim@uol.de

***Abstract*—Cooperative software verification divides the task of software verification among several verification tools in order to increase efficiency and effectiveness. The basic approach is to let verifiers work on different parts of a program and at the end join verification results. While this idea is intuitively appealing, cooperative verification is usually hindered by the fact that program decomposition (1) is often static, disregarding strengths and weaknesses of employed verifiers, and (2) often represents the decomposed program parts in a specific proprietary format, thereby making the use of *off-the-shelf* verifiers in cooperative verification difficult.**

In this paper, we propose a novel cooperative verification scheme that we call *dynamic program splitting (DPS)*. Splitting decomposes programs into (smaller) programs, and thus directly enables the use of off-the-shelf tools. In DPS, splitting is *dynamically* applied *on demand*: Verification starts by giving a verification task (a program plus a correctness specification) to a verifier V_1. Whenever V_1 finds the current task to be hard to verify, it splits the task (i.e., the program) and restarts verification on subtasks. DPS continues until (1) a violation is found, (2) all subtasks are completed or (3) some user-defined stopping criterion is met. In the latter case, the remaining uncompleted subtasks are merged into a single one and are given to a next verifier V_2, repeating the same procedure on the still unverified program parts. This way, the decomposition is steered by what is hard to verify for particular verifiers, leveraging their complementary strengths. We have implemented dynamic program splitting and evaluated it on benchmarks of the annual software verification competition SV-COMP. The evaluation shows that cooperative verification with DPS is able to solve verification tasks that none of the constituent verifiers can solve, without any significant overhead.

***Index Terms*—Software verification, cooperation, program splitting, off-the-shelf tools.**

I. Introduction

Software verification is the task of showing that programs adhere to given specifications. While recent years have seen continuous research innovation and increasing tool development in software verification [7], there are still numerous programs that are in principal verifiable, but none of the existing tools can automatically solve them.

Cooperative verification [17] aims at improving this situation by having tools (verifiers) cooperate on the task of software verification. Cooperative verification is inspired by the observation that verifiers have their specific strengths and weaknesses: there are *tasks* (i.e., programs with specifications) on which one verifier succeeds and some other fails, and vice versa. The basic scheme of cooperation proceeds by letting verifiers work on different parts of a program, and at the end join verification results. In this, program parts are in the majority of cases *statically* calculated, based on program structure: an upfront decomposition determines program blocks [15], modules [60], control-flow paths [74] or ranges of execution paths [37], [47], [75]. This decomposition often either results in specially annotated or heavily instrumented subprograms (e.g., [46]) or in no subprograms at all (e.g., only test inputs for path ranges [47], [75]).

Dynamic decomposition, i.e., a decomposition occurring *during* verification runs based on progress, is less frequent. We are only aware of two types of dynamic decomposition: (1) Approaches building specific verifiers documenting their progress in specific formats (e.g., [6], [11], [23], [42], [81]), and (2) approaches based on work stealing concepts where one verifier "steals" subtasks of others when it observes insufficient progress of others [25], [84]. Both types of approaches prohibit cooperation between off-the-shelf tools, the first type because it requires at least one specialized verifier documenting verification progress (and possibly a second understanding these documents) and the second type because it requires verifiers to monitor each other's progress.

In this paper, we propose a cooperative verification scheme called *dynamic program splitting* (DPS) allowing for a cooperation of arbitrary off-the-shelf verifiers. This scheme dynamically applies program splitting on demand whenever a verifier discovers that its current verification task is hard to verify. "Hardness" is therein defined by (user definable) upper limits on verification time. Splitting itself is carried out on branches of programs' execution trees and the result of splitting is a number of subprograms (i.e., subtasks). The DPS technique is successively applied on subtasks until (a) one subtask ends with the verdict "specification violated" or (b) all subtasks end with verdict "specification fulfilled" or (c) some user-defined stopping criterion is reached. In case of (c), the remaining (uncompleted) subtasks are *merged* into a single task. DPS is subsequently applied to this remaining task, employing the next verifier in a given list. As all inputs to verifiers in this

This work is partially supported by the German Research Foundation (DFG) – WE2290/13-2 (Coop2), and in part by the ERC-2020-AdG 101020093.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00092

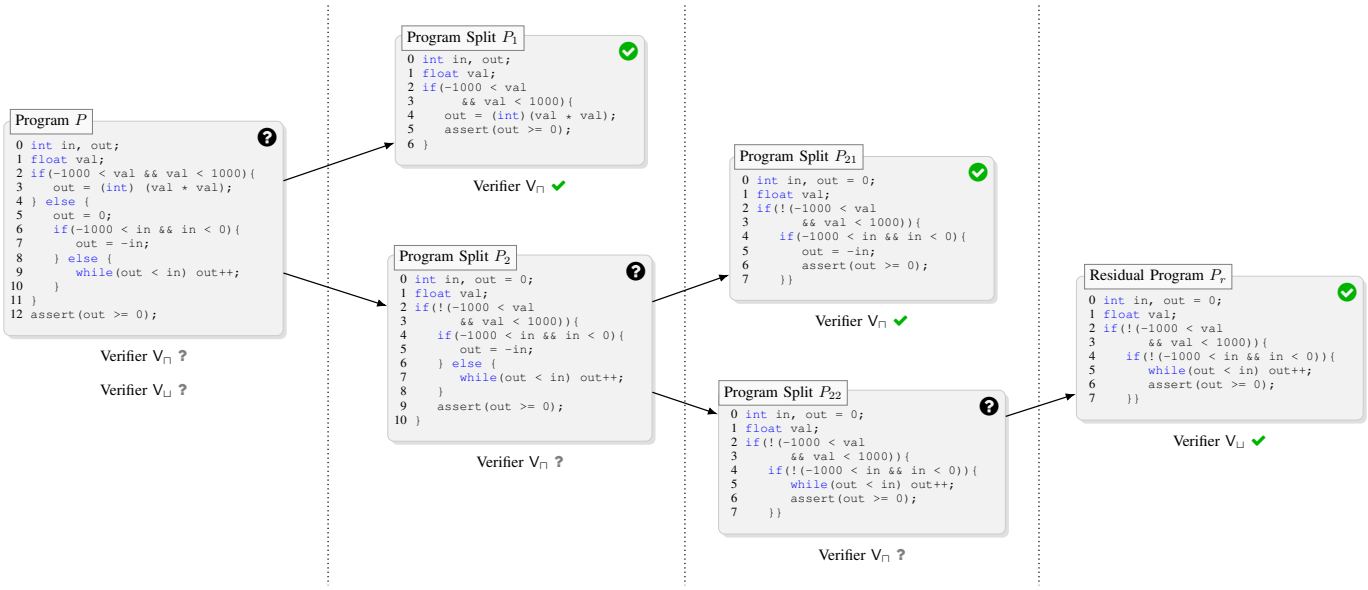

Fig. 1: **Cooperation through Dynamic Splitting.** Both $V_\sqcap$ and $V_\sqcup$ fail to verify P for different reasons (?). By iteratively splitting P, we can independently verify parts of P with $V_\sqcap$ and $V_\sqcup$.

scheme are programs, cooperation between off-the-shelf tools is directly achieved.

We have implemented the cooperation scheme with dynamic program splitting for C programs and have evaluated it on benchmarks of SV-COMP [7]. The evaluation shows that the choice of verifiers to be used in cooperation and the order in which they are employed in this scheme has significant impact on the effectiveness and efficiency of DPS. We have observed the best results when the first verifier employs an underapproximating analysis (e.g., symbolic execution [59][1]), fast at finding bugs, and the second verifier an overapproximating analysis (e.g., some form of abstract interpretation [27]). For such combinations, there are verification tasks in the benchmark set which are *uniquely* solved by the cooperative scheme.

Example. In Fig. 1, we provide an example that shows how cooperation between *off-the-shelf* verifiers could be achieved via splitting. Consider the program P in Fig. 1. The program computes a value for the variable `out` based on the input `in` (of type `int`) and `val` (of type `float`). The assertion at line 12 gives the correctness specification and specifies the expected postcondition to be verified, namely that `out` is non-negative at the end of the program (which holds). The program contains two constructs which could be challenging during the verification process: (1) a floating point multiplication at line 3 and (2) a (potentially unbounded) loop at line 9. Now, assume that we have two verifiers $V_\sqcap$ and $V_\sqcup$. $V_\sqcap$ employs an underapproximating analysis (e.g. symbolic execution) that can easily handle floating point operations, but can get stuck in the loop. $V_\sqcup$ employs an analysis that overapproximates the state space of the loop. It however fails to verify the program due to the floating point multiplication at line 3. By splitting the program P into two subprograms P_1 and P_2, we can independently verify the part of the program that includes floating point multiplication with $V_\sqcap$ and the part of the program with a loop with $V_\sqcup$.

Our approach starts by giving the initial program P (with included specification) to the first verifier $V_\sqcap$. It fails to solve the entire task, and hence DPS splits the program into P_1 and P_2. Verifier $V_\sqcap$ succeeds on P_1 (with verdict "specification fulfilled", ✔), but fails on P_2 (verdict ?). Thus DPS splits P_2 once more leading to subtasks P_{21} and P_{22}. This procedure is repeated until a stopping criterion is reached. Here, we use the maximum splitting depth 2 as the stopping criterion, so when verifier $V_\sqcap$ succeeds on P_{21} (with ✔) and fails on P_{22} (?), we do not split P_{22} once more. Instead, we give the *residual program* P_r (which is P_{22}) to the second verifier $V_\sqcup$. $V_\sqcup$ succeeds on P_r with verdict ✔, which is then also the overall verdict for P.

Novelty. We provide the following novel contributions:

- We conceptually develop a cooperative verification technique via dynamic program splitting, which determines splits on demand based on verifier capabilities and which enables usage of off-the-shelf verifiers.
- We prove *soundness* of the technique.
- We implement dynamic program splitting for C programs.
- We carry out a *rigorous* experimental evaluation using benchmarks of 4 771 verification tasks (written in C), employing 4 different types of off-the-shelf software verifiers.

[1]We consider symbolic execution to be an underapproximating technique because it cannot inspect all execution paths in case of programs with unbounded loops.

Significance and Potential Impact. Competitions on software verification and evaluations in research literature have shown that existing verification tools indeed have their individual strengths and weaknesses, and automatic verification fails for some programs where it should – in principle – not fail. Cooperative verification has emerged as a way out of this problem, but could so far not completely fulfill its promises. By allowing arbitrary off-the-shelf tools to cooperate on verification tasks, specifically running tools on parts hard for others, we see our contribution as a significant step towards unleashing the full potential of cooperative verification. Our dynamic splitting scheme is moreover directly applicable to other types of software analyses, e.g. testing.

Artifact. All our proofs, implementations and data are archived and available at Zenodo[2] [72].

II. Background

We start by introducing some basic notations about syntax and semantics of programs as well as the task of software verification.

Programs. In our presentation, we consider simple, imperative programs that execute assignments, conditionals (if-then-else), and loops using a set $\mathcal{X}$ of integer variables; our implementation supports C programs. We formally model a program as a *control-flow automaton* (CFA) $P = (L, \ell_0, G)$ with program locations L, an initial location $\ell_0 \in L$, and control-flow edges $G \subseteq L \times Ops \times L$ (where the set Ops contains all possible program statements). In that, if-then-else and loops are encoded using assume operations on boolean expressions for conditions. We let $\mathcal{P}$ be the set of all control-flow automata, i.e., all programs. From a CFA we can derive the syntactically possible paths. A *syntactic program path* $\pi = \ell_0 \xrightarrow{g_1} \dots \xrightarrow{g_n} \ell_n$ is a sequence of program locations and control-flow edges such that ℓ_0 is the initial program location and for all i, $1 \leq i \leq n$, we have $g_i = (\ell_{i-1}, \cdot, \ell_i)$. For the semantics of programs, we need to extend syntactic program paths with information on variable values. To this end, we represent program states by pairs (ℓ, d) of program location ℓ and data state $d : \mathcal{X} \to \mathbb{Z}$ from the set $\mathcal{D}$ of all data states. Then, a *program execution path* $ex = (\ell_0, d_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (\ell_n, d_n)$ is a sequence of program states and control-flow edges such that $\ell_0 \xrightarrow{g_1} \dots \xrightarrow{g_n} \ell_n$ is a syntactic program path and for all i, $1 \leq i \leq n$, the data states adhere to the semantics of g_i's operation op_i. This means, assume operations $op_i = assume(b)$ need to evaluate to true ($d_{i-1} \models b$) and do not change the data state ($d_{i-1} = d_i$), while assignments change the data state according to the strongest postcondition ($d_i = SP_{op_i}(d_{i-1})$). Given a program execution path $ex = (\ell_0, d_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (\ell_n, d_n)$ with $g_i = (\ell_{i-1}, op_i, \ell_i)$ ($1 \leq i \leq n$), we call the sequence $\tau = d_0 op_1 d_1 \dots op_n d_n$ a *program trace*. The set $tr(P)$ denotes the set of all program traces of program P.

Safety. We use program traces to define safety of programs. A *safety specification* $\varphi \subseteq \mathcal{D} \times (Ops \times \mathcal{D})^*$ describes sequences $d_0 op_1 d_1 \dots op_m d_m$ of allowed program traces. We let Φ be the set of all specifications. In practice, a specification φ may e.g. be defined by all traces (a) fulfilling an assertion (as in the case of the example in Fig. 1), (b) not causing integer overflows, or (c) adhering to particular operation sequences (e.g., typestate specifications [79]). A *program P is* safe *wrt. specification φ* if and only if φ allows all of its program traces, i.e., $tr(P) \subseteq \varphi$, otherwise it is *unsafe*. In case a program P is unsafe wrt. specification φ, there exists at least one *error trace* $\tau \in tr(P) \setminus \varphi$.

[2]https://doi.org/10.5281/zenodo.13142908

Algorithm 1 Dynamic Program Splitting (*DPS*)

Input: Verifier V
Program P
Specification φ
Output: Verdict
Set of Programs

1: $verdict \leftarrow \mathsf{V}(P, \varphi)$;
2: **if** $verdict = ✔$ **then return** ✔, $\emptyset$;
3: **if** $verdict = ✘$ **then return** ✘, $\{P\}$;
4:
5: $splits \leftarrow \{P\}$;
6: **while** $|splits| > 0 \wedge \neg stop$ **do**
7: choose $P_{\text{split}} \in splits$; $splits \leftarrow splits \setminus \{P_{\text{split}}\}$;
8: $P_1, \dots, P_n \leftarrow split(P_{\text{split}})$;
9: $r_1, \dots, r_n \leftarrow \|_{P_i} \mathsf{V}(P_i, \varphi)$; ▷ *in parallel*
10: **if** ✘ $\in \{r_1, \dots, r_n\}$ **then**
11: **return** ✘, $\{P_{\text{split}}\} \cup splits$;
12: $splits \leftarrow splits \cup \{P_i \mid r_i = ?\}$;
13:
14: **if** $splits = \emptyset$ **then return** ✔, $\emptyset$;
15: **return** ?, $splits$;

Verifier. Given a program P and a specification φ, the task of a software verifier is to check whether a program is safe wrt. φ. Formally, we see a verifier as a function $\mathsf{V} : \mathcal{P} \times \Phi \to \{✔, ✘, ?\}$, and V is *sound* if $\mathsf{V}(P, \varphi) = ✔$ implies $tr(P) \subseteq \varphi$ and $\mathsf{V}(P, \varphi) = ✘$ implies the existence of an error trace $\tau \in tr(P) \setminus \varphi$. The third value ? (unknown) is necessary here since verification is in general undecidable, and verifiers may fail to prove the existence or the absence of disallowed paths. In the following, we refer to the elements in $\{✔, ✘, ?\}$ as *verdicts*.

III. Cooperation via Dynamic Splitting

Our novel cooperative verification scheme achieves cooperation through dynamic program splitting. We start by introducing the algorithm behind DPS and we show that it has some useful properties enabling the use of *off-the-shelf* components in cooperative verification.

A. Dynamic Program Splitting

Algorithm 1 describes the process of dynamic program splitting using a single verifier V. We first run V on P. If

it completes the verification (verdict ✔ or ✘), we are done and can return the outcome together with either an empty set or with the set $\{P\}$. The returned set of programs in both cases includes the error traces of P. If the verdict of V is ?, program P is hard to verify for V and splitting starts with $splits = \{P\}$. At the beginning of each loop iteration (line 6), *splits* contains all subprograms that still must be verified. Within the loop (lines 7 to 12) we choose one program from *splits*, split it once more and run V in *parallel* on all newly generated subprograms (line 9). When a subprogram P_i does not get verified (r_i = ?), the subprogram is included into *splits* (line 12). Finally, we end the loop if (1) all splits have been verified (i.e., *splits* is empty), (2) a stopping criterion is fulfilled (e.g., a timelimit is exceeded) or (3) a property violation is detected (line 10). In all cases, we return the remaining splits together with a verification verdict.

Trace Splitting. A central part of DPS is the function *split* that we use for program splitting. The task of *split* is to divide a given program P into a finite number of subprograms $P_1, \ldots, P_n$ such that we can verify each subprogram individually. To achieve this, we adopt a *trace splitting strategy* that preserves the set of program traces. Formally, we thus define *split* as a function $split : \mathcal{P} \to 2^{\mathcal{P}}$ satisfying the following condition:

$$\forall P \in \mathcal{P} : \quad tr(P) = \bigcup_{P_i \in split(P)} tr(P_i)$$

Note that a function returning the original program (i.e., $split(P) = \{P\}$) is already a valid splitter, though not a very useful one. In practice, we use more sophisticated instantiations of *split* (see Sec. III-C).

Properties. DPS has some useful properties when used with a function *split* that satisfies the aforementioned condition.

Lemma 1: Let P be a program, φ a specification and V a sound verifier. If $DPS(\mathsf{V}, P, \varphi)$ returns $verdict, progs$, then

$$tr(P) \setminus \varphi \subseteq \bigcup_{P_i \in progs} tr(P_i) \subseteq tr(P)$$

Lemma 1 states that the set of (sub)programs returned by Alg. 1 includes all unverified parts of the program. In particular, the set of subprograms always include all error traces of the original program.

Lemma 2: Let P be a program, φ a specification and V a sound verifier. If $DPS(\mathsf{V}, P, \varphi)$ returns $verdict, progs$, then

- $verdict$ = ✔ implies $progs = \emptyset$, and
- $verdict$ = ✘ implies $tr(P) \setminus \varphi \neq \emptyset$.

Lemma 2 basically states that Algorithm 1 always returns a *sound* result.

Proof: The proofs of Lemma 1 and Lemma 2 can be found in the accompanying artifact [72].

B. Cooperation via Splitting

To enable cooperation between verifiers, we first employ DPS to identify the parts of a program that are hard to analyze for a particular verifier. We then use the results of DPS to construct a *residual program* P_r, which can be processed by the next verifier. Algorithm 2 describes the overall procedure of this cooperation scheme. The algorithm basically defines a new verifier based on a non-empty list of existing verifiers $\mathcal{V}$. The goal is to verify the program P wrt. specification φ. The algorithm starts by considering the first verifier V_{head} at the head of the list. If there are no subsequent verifiers in $\mathcal{V}_{tail}$, we do not need to generate a residual program and therefore only execute V_{head} on the given program (line 2). Otherwise, we use *DPS* to compute an intermediate verdict and a set of unverified program parts. If the verdict is conclusive, we stop and return the current verdict. If we need to continue the verification process ($verdict$ = ?), we compute a residual program P_r by merging all unverified parts of the original program. The residual program is then processed by the next verifier in $\mathcal{V}_{tail}$.

Algorithm 2 *CoopVeri*

Input: Verifiers $\mathcal{V} = \langle \mathsf{V}_{head} \rangle \circ \mathcal{V}_{tail}$
Program P
Specification φ
Output: Verdict
1: **if** $\mathcal{V}_{tail} = \langle\rangle$ **then**
2: **return** $\mathsf{V}_{head}(P, \varphi)$;
3: $verdict, progs \leftarrow DPS(\mathsf{V}_{head}, P, \varphi)$;
4: **if** $verdict \neq$? **then return** $verdict$;
5: $P_r \leftarrow merge(progs)$;
6: **return** $CoopVeri(\mathcal{V}_{tail}, P_r, \varphi)$;

Trace Merging. To compute P_r, we use the function *merge* for merging the traces of the subprograms. The task of *merge* is to rejoin a given set of programs $\{P_1, \ldots, P_n\}$ into a single program P while keeping the overall set of program traces of all subprograms. Formally, we define *merge* as a function $merge : 2^{\mathcal{P}} \to \mathcal{P}$ satisfying the following condition:

$$\forall \Pi \in 2^{\mathcal{P}} : \quad tr(merge(\Pi)) = \bigcup_{P_i \in \Pi} tr(P_i)$$

Soundness. Given a set of sound verifiers, we can now show that our cooperative verification scheme is sound.

Theorem 1: Let P be a program, φ a specification and let $\mathsf{V}_1, \ldots, \mathsf{V}_n$ be sound verifiers. Then $\mathsf{CV} : \mathcal{P} \times \Phi \to \{✔, ✘, ?\}$ defined by $\mathsf{CV}(P, \varphi) := CoopVeri(\langle \mathsf{V}_1, \ldots, \mathsf{V}_n \rangle, P, \varphi)$ is a sound verifier.

Proof: The proof can be found in the accompanying artifact [72].

Note that if at least one verifier is complete[3], then our cooperative verifier is also complete for all trace-based safety specifications.

[3]A verifier is complete if it can prove the existence or absence of error traces for every possible program.

```
0 if(b){ <- split here
1   // CODE BLOCK A
2 } else {
3   // CODE BLOCK B
4 }
5 // CODE BLOCK C
```

(a) Example program P

```
0 if(b){
1   // CODE BLOCK A
2 } else {
3   abort(); <- stop here
4 }
5 // CODE BLOCK C
```

(b) Split program $P_{\texttt{b}}$

```
0 if(b){
1   abort(); <- stop here
2 } else {
3   // CODE BLOCK B
4 }
5 // CODE BLOCK C
```

(c) Split program $P_{\texttt{!b}}$

Fig. 2: A simplified program P and the obtained splits $P_{\texttt{b}}$ and $P_{\texttt{!b}}$ after splitting P with split condition `b`.

C. Instantiations of Split and Merge

We next discuss some concrete instantiations of the functions *split* and *merge* used for trace splitting and trace merging.

Control-Flow Splitting. For implementing the *split* function, we employ a *control-flow* based splitting technique. The algorithm is shown in Alg. 3. We start by identifying potential *split locations* in the program $P = (L, \ell_0, G)$. A split location $\ell_{sp} \in L$ is a program location in P that (1) has multiple successor locations in G (a branching location) and (2) cannot be (re-)reached again in the CFA after being visited for the first time (e.g., a location belonging to an `if` but not to a `while` statement). To find a potential split location, we first search for a branching location in G (lines 1 to 7) and then check whether the branching location can be reached again (*rereach* in line 9). We abort (line 7) if no branching location can be found.

It is important to note that a split location always corresponds to a branching decision taken at most *once* in a program execution. Thus, syntactically splitting the program at a given split location does not influence future (semantic) branching decisions. This property is crucial for the *soundness* of *split*: Splitting a program at a location that cannot be reached again ensures that the set of traces of P can be represented by a *finite* set of subprograms P_i. Splitting at branching locations of loops and in functions that are called multiple times would potentially require a split into infinitely many subprograms (for each of the potentially infinitely many branching decisions) and is therefore not allowed. To still be able to split program location that are reached again in the original CFA (lines 10 to 11), we employ several program unfolding strategies which we describe below.

To then finally compute a concrete split (in lines 13 to 16), we construct a split condition `b`[4] such that all executions that reach ℓ_{sp} and that satisfy `b` follow the same branch in the program, i.e., follow the outgoing edge $(\ell_{sp}, assume(\texttt{b}), \cdot) \in G$. We then construct two subprograms $P_{\texttt{b}}$ and $P_{\texttt{!b}}$ such that $P_{\texttt{b}}$ contains all execution paths that (1) either do not reach ℓ_{sp} at all, or (2) reach ℓ_{sp} and satisfy `b`; $P_{\texttt{!b}}$ contains all remaining execution paths. An example program and the corresponding splits are shown in Fig. 2. A proof that our control-flow based splitting strategy implements a valid *split* function can be found in the accompanying artifact [72].

Unfolding Strategy. In general, we apply splitting on the first split location reachable from ℓ_0. As not all programs contain split locations, we might, however, first need to apply one of the following two unfoldings to generate split locations and thereby enable splitting (*unfold* in line 10 of Alg. 3).

Loop unrolling We unroll loops. This generates new `if` statements that can serve as split locations.

Function cloning We clone functions that are called multiple times. To this end, we copy the function definition, replace the function name of the cloned function with a new function name and modify the function call accordingly. This generates a new function that is only called once and can therefore be split.

We apply loop unrolling or function cloning depending on what we see first in the program, a loop or a function called multiple times. When neither technique helps with generating a split location, e.g., because the program contains no further loops, splitting simply returns the original program. When we unroll a loop, we mark it as unrolled and do not unroll it in successive splits. Once all loops in the program are marked, we remove the markings so that the loops can be unrolled again. This way we balance unrolling the loops. Most importantly, we avoid unrolling only the same loop over and over again.

Merging. To implement the *merge* function, we exploit the structure of the DPS algorithm.

DPS Inversion. DPS implicitly defines a tree of program splits. Each parent program has a set of child program splits and the leaves of the tree are the final splits. Each final split is either solved or left open for the next verifier. To merge the remaining subprograms, we employ an iterative merging strategy. Starting from the leaves, we compute a merged program for each split operation. If all children are verified, we return an empty program and mark it as solved. If only one child remains unsolved, we propagate the child up the tree and mark it as unsolved. If two or more children remain unsolved, we merge them via a control-flow-based merging strategy (described next). After merging all subprograms, we end up with a residual program P_r, which can be directly processed by the next verifier.

Control-Flow Merging. For the actual merging, we exploit the fact that the programs to be merged are all splits of one

[4]The split condition `b` often corresponds to the condition of an `if`. Switch statements require extra care.

Algorithm 3 Control-Flow Splitting *split*

Input: Program $P = (L, \ell_0, G)$
Output: Set of Programs

$\ell_{sp} \leftarrow \ell_0$;
$visited \leftarrow \{\ell_0\}$;
while $\neg isBranch(P, l_{sp})$ **do**
  if $\exists \ell'_{sp} \in L : (\ell_{sp}, \cdot, \ell'_{sp}) \in G \wedge \ell'_{sp} \notin visited$ **then**
    $\ell_{sp} \leftarrow \ell'_{sp}$; $visited \leftarrow visited \cup \{\ell'_{sp}\}$;
  else
    return {P}; ▷ *No split location found*

if $rereach(P, \ell_{sp})$ **then**
  $P' \leftarrow unfold(P, \ell_{sp})$; ▷ *Unfold program*
  return $split(P')$;

choose $(\ell_{sp}, op, \cdot) \in G$ with $op = assume(\mathtt{b})$;
$G_{\mathtt{b}} \leftarrow G \setminus \{(\ell_{sp}, op', \ell''_{sp}) \mid op' \neq op\}$;
$G_{!\mathtt{b}} \leftarrow G \setminus \{(\ell_{sp}, op', \ell''_{sp}) \mid op' = op\}$;
$P_{\mathtt{b}}, P_{!\mathtt{b}} \leftarrow (L, \ell_0, G_{\mathtt{b}}), (L, \ell_0, G_{!\mathtt{b}})$;

return $\{P_{\mathtt{b}}, P_{!\mathtt{b}}\}$;

common start program P. Hence, two programs P_1 and P_2 to be merged always consist of some first (potentially empty) part that they share followed by parts in which they differ. To merge P_1 and P_2, merging needs to first find the *merge locations* ℓ_{mg}, which are the locations at which the common part of the two programs ends. Technically, we identify merge locations on the product automaton of the CFAs of P_1 and P_2, and sometimes we also need to employ loop unrolling and function cloning to be able to find merge locations at all.

Merging P_1 and P_2 at a found merge location ℓ_{mg} should then ideally construct a merged program P_m such that ℓ_{mg} is a split location of P_m and splitting at this location gives us programs P_1 and P_2. Further recall that the objective of merging is to guarantee that the traces of P_m is exactly the union of traces of P_1 and P_2. Therefore, to construct P_m, we start by setting P_m to be P_1. We then successively search for merge locations and merge parts of P_2 into P_m whenever we notice that traces of P_2 are missing in P_1. In this case, the merge location fixes the point in the program at which the additional behavior of P_2 is added. In our example in Fig. 2, the `if` at line 0 is a merge location for $P_{\mathtt{b}}$ and $P_{!\mathtt{b}}$, and merging indeed gives P again. We employ various optimizations to obtain a merged program that is as similar as possible to the original program before any splitting. For example, our merger detects potential join locations (e.g., the start of `CODE BLOCK C` in Fig. 2) from where the splits may be the same and tries to join them together during the merging process.

Whenever we are provided with more than two subprograms, we incrementally merge them together.

IV. Evaluation

Our DPS-based cooperative verification scheme enables cooperation between off-the-shelf verification tools. Our focus in the evaluation is thus on examining whether this brings us any practical advantages wrt. the effectiveness and efficiency of verification. Our evaluation is guided by the following research questions:

RQ1 How does DPS-based cooperative verification compare with non-cooperative approaches?
RQ2 How does DPS-based cooperation compare with existing cooperation schemes?
RQ3 How does the unfolding strategy impact the ability of DPS-based cooperation to uniquely solve verification tasks?

A. Implementation

We implemented our DPS-based cooperation scheme within Bubaak[5] [20]. Bubaak is a tool mainly designed for the *dynamic* composition of verification tools. Tools in Bubaak are not composed in a static scheme, but invoked based on information gathered during the verification process. We implemented Algorithm 2 as a new type of dynamic composition. For our instantiations of split and merge, we furthermore required program analyses that can directly process and transform C code. For this, we implemented pycpa[6], a novel program analysis framework for C written in Python. pycpa directly operates on a code representation (i.e., CFA) that is *grounded*[7] in the actual C code. Therefore, code modifications like splitting or loop unrolling can be done on the code and are then directly reflected in the CFA. pycpa is implemented in around 6K lines of Python code and supports the analysis and transformation of ANSI C programs.

B. Tools and Techniques

Modern verification tools often employ a variety of verification techniques [7]. To evaluate the impact of our cooperation scheme on the verification process, we employ implementations of the most common techniques, focusing on techniques with complementary strengths.

Symbolic Execution. Symbolic Execution (SE) [59] symbolically executes programs, enumerating feasible program execution paths. Hence, SE is good at finding specification violations, but has difficulties in proving the safety of programs with large or unbounded number of execution paths. For our evaluation, we employ Bubaak-Lee [20], which is a fork of symbolic executor Klee [19] for LLVM bitcode. We characterize SE as an *underapproximating* technique (**UA**).

Bounded Model Checking. Bounded Model Checking (BMC) [38] constructs a logical formula encoding program paths and examines its satisfiability in conjunction with a formula for a specification. To ensure that the program formula

[5]https://gitlab.com/mchalupa/bubaak
[6]https://github.com/cedricrupb/pycpa
[7]I.e., we maintain a one-to-one correspondence between code and CFA.

can be constructed, it bounds loop iterations. Hence, BMC can prove a program to be safe up to a given loop bound. In our experiments, we employ ESBMC [38], a verification tool specialized for SMT-based BMC and k-Induction. We configure ESBMC to run incremental BMC, i.e., a variant of BMC that incrementally increases the given loop bound. BMC is an underapproximating technique (**UA**).

k-Induction. k-Induction (KI) [5], [32] is an extension of BMC for verifying programs with unbounded loops. KI first performs BMC with a given bound on loop iterations, and then tries to prove that a given property holds also for all future iterations. Therefore, KI can be used to prove the safety of programs with bounded and/or unbounded loops. In our experiments, we employ ESBMC-kInd [38] which is a version of ESBMC specifically configured to run k-Induction. KI is an *overapproximating* technique (**OA**).

Predicate Analysis. Predicate Analysis (PA) [14] is a technique based on abstract interpretation. To prove the safety of programs, PA abstracts the concrete execution state with a predicate-based abstraction. Therefore, PA can analyze programs with unbounded loops if their behavior can be abstracted with a finite set of predicates. To compute the set of predicates used for the abstraction, we use PA with counterexample guided abstraction refinement (CEGAR) [26]. For our evaluation, we employ PA as implemented in the verification framework CPAchecker [13]. PA is another overapproximating technique (**OA**).

Conditional Model Checking. Finally, for RQ2 we want to compare our approach to another cooperation scheme. For this, we have chosen *conditional model checking* (CMC) [6] as it also employs dynamic decomposition based on verification progress. In CMC, two verifiers cooperate by exchanging so-called verification *conditions*. A verification condition documents the progress of the first verifier by describing the already verified program part. To the best of our knowledge, CPAchecker [13] is currently the only tool to generate and understand verification conditions. The second verifier within CMC can still be an off-the-shelf tool, because there is a *reducer* technique for converting conditions into programs [11]. For the first verifier generating the condition, we can nevertheless only use CPAchecker.

C. Experimental Setup

We run our experiments on machines with Intel Core i5-1230, 8 cores, 33GB of memory and Ubuntu 22.04 LTS with Linux kernel 5.15. To increase reproducibility of our experimental results, we run our experiments with BenchExec [16]. Each verification run is restricted to 15 GB RAM, 4 CPU cores and 15 min CPU time. The setup is comparable to the setup used in SV-COMP. For evaluation, we employ verification tasks of SV-COMP 2024 [7], which represents the largest available benchmark set for verification of C programs. We included all 4 771 tasks from the categories *Arrays, BitVectors, ControlFlow, Floats, Heap, Loops, ProductLines, Recursive, Sequentialized, XCSP* and *Combinations*. A verification task is either safe (and contains no specification violation) or unsafe (contains a violation, viz. an error trace). We use CPAchecker version 2.3, BUBAAK-LEE version 3.0 and the version of ESBMC used in SV-COMP 2024.

V. RESULTS

We report about the results per research question.

A. RQ1: Cooperation via DPS vs. no cooperation

To answer our first research question, we run Alg. 2 with different lists of verifiers and compare it to the verifiers individually. Each list consists of two verifiers V_1 and V_2. Below we will also see results for lists with more than two verifiers. As *stop* function within Alg. 1 we use a timeout: We limit the runtime of V_1 to 16s[8], the overall execution time of the DPS algorithm to 100 s and let the verifier V_2 run until it finishes or a global resource limit is exceeded.

Results. Our experimental results are shown in Tab. I. We list the number of verification tasks that each verification approach solved correctly, incorrectly[9] and which it cannot solve (unknown). The correct and incorrect results are further categorized into proofs and alarms, i.e. whether the approach reported a proof or a specification violation respectively. Uniques are the number of verification tasks that the combination V_1+V_2 can solve, but not the baseline verifiers V_1 and V_2 individually. Due to a lack of space, we focus on combinations with SE when combining underapproximating (UA) with overapproximating (OA) analyses.

DPS Cooperation increases number of verified programs. We observe that in all cases the combination of any two verifiers V_1 and V_2 solve significantly more verification tasks correctly than verifier V_1 or V_2 alone. In some cases, e.g. SE+PA, the combination solves up to 794 and 528 tasks more than the baseline verifiers PA and SE, respectively. We moreover see that there are at least 4 to 35 tasks (row **Uniques**) that can only be correctly solved by a combination of two verifiers V_1 and V_2 but not by the individual verifiers V_1 and V_2 (within 900 s). These tasks require cooperation between V_1 and V_2 and would not be solved by a combination that simply executes the verifiers one after another on the same verification task (e.g. [62]). The number of incorrect results does not increase significantly, except for the combinations PA+SE and PA+KI. There, PA produces some incorrect alarms because it investigates execution paths after splitting (and wrongly attribute these as error traces) that would not have been considered at all without splitting due to a failure of PA on other, previously examined paths.

Tool choice and order matters. Table I also gives us results about useful combinations of verifiers and their ordering. UA tools are often more effective in finding specification violations. Therefore, the combination of two UA tools (SE+BMC)

[8]We experimented with different runtimes ranging from 2 s to 64 s. We found that 16 s is long enough for the verifiers to effectively guide DPS, while being short enough to ensure a sufficient number of program splits.

[9]Verifiers can produce unsound (incorrect) results due to bugs in their code.

TABLE I: DPS-based cooperation vs. no cooperation (Columns **UA** and **OA** show results of single verifiers, other columns show results of DPS-based cooperation with different lists of verifiers, ordering giving sequence of verifier list).

	UA		**OA**		**UA → UA**		**UA → OA**		**OA → UA**		**OA → OA**	
	SE	BMC	KI	PA	SE+BMC	BMC+SE	SE+KI	SE+PA	KI+SE	PA+SE	KI+PA	PA+KI
Correct	2 244	2 389	2 500	1 978	2 672	2 564	**2 816**	2 772	2 681	2 654	2 683	2 795
Correct proof	1 016	1 144	1 258	984	1 254	1 174	1 401	1 410	1 294	1 386	1 377	**1 491**
Correct alarm	1 228	1 245	1 242	994	**1 418**	1 390	1 415	1 362	1 387	1 268	1 306	1 304
Incorrect	1	5	5	33	5	1	6	32	2	45	32	50
Incorrect proof	**0**	5	5	**0**	3	1	4	**0**	2	**0**	2	5
Incorrect alarm	1	**0**	**0**	33	2	**0**	2	32	**0**	45	30	45
Unknown	2 526	2 377	2 266	2 760	2 094	2 206	1 949	1 967	2 088	2 072	2 056	**1 926**
Uniques	–	–	–	–	26	4	**35**	6	10	6	29	31

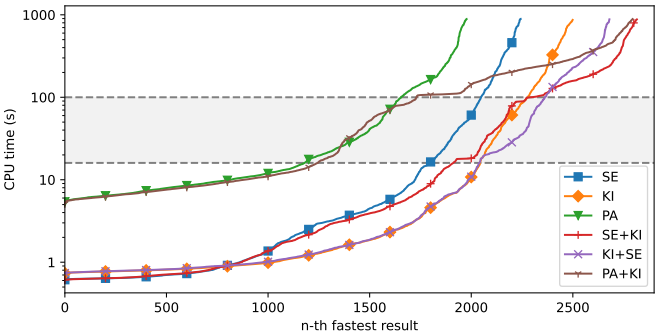

Fig. 3: Quantile plot for SE, KI, PA and their combinations SE+KI, KI+SE and PA+KI.

is able to find more than 100 violations more than any OA combination (KI+PA and PA+KI). As expected, OA tools are often more effective in producing safety proofs. We thus find that a combination of UA and OA tools (SE+KI) achieves the overall highest number of correctly solved tasks, and it is on average better to have the UA tool first in the list.

Efficiency. We are also interested in the runtime of verifiers alone and of the cooperation. Figure 3 shows quantile plots for the verifiers SE, KI, PA and their combinations SE+KI, KI+SE and PA+KI. A data point (x, y) means that the x fastest results are solved each within y seconds of CPU time. The gray box marks the time between 16 s and 100 s, i.e., the time where most of the splits are performed. We observe that both tool choice (SE or PA for V_1) and tool order (SE+KI or KI+SE) have a significant impact on efficiency, but except for PA+KI no significant difference to runtimes of tools alone shows.

More verifiers, more tasks solved. We finally investigate the impact of incorporating more tools in DPS-based cooperation. For this, we run Alg. 2 with verifier list ⟨SE, BMC, PA, KI⟩. Here, every verifier except KI (which is the last one) is restricted to a runtime of 100 s (still with the limit of 16 s per split). To investigate the impact of constituent verifiers, we also run Alg. 2 with suffixes of this list. Our results are shown in Fig. 4. We report the number of tasks that the DPS-based cooperation can correctly solve (in total, plus per proof and alarm) but not the base verifier KI. We find out that by increasing the number of diverse verification technologies in our DPS-based cooperation we also increase the number of correctly solved tasks.

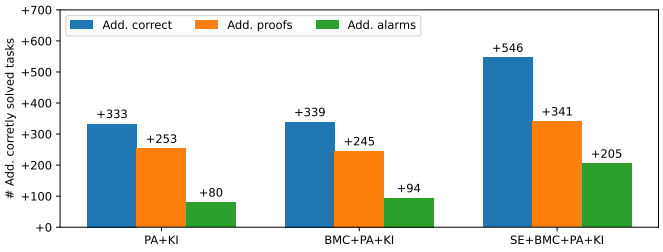

Fig. 4: Number of tasks that the DPS-based combinations can solve but KI cannot.

DPS-based cooperation can significantly increase verification performance. Effectiveness and efficiency of the cooperation is often sensitive to tool choice, order, and the number of tools used in the combination.

B. RQ2: Cooperation via DPS vs. other cooperation schemes

We compare our approach with CMC-based cooperation. Similar to DPS, CMC-based cooperation performs dynamic decomposition based on verification progress. However, CMC can only utilize tools that can generate verification conditions, and hence in CMC we have to employ CPAchecker at least for the first verifier. To allow for variations of tool orderings, we hence also employ an implementation of SE within CPAchecker in addition to the PA implementation. For a fair comparison, we limit the runtime of the first verifier to 100 s in each combination.

Results. Table II presents the results of our comparison. We report the number of correctly and incorrectly solved tasks for the combinations SE+PA and PA+SE. We report results for a CMC-based cooperation (CMC, passing conditions from one to the next verifier), a reducer-based cooperation (Reducer, reducing the condition to a program before giving it to the next verifier) and a DPS-based cooperation (DPS-CPA), all using SE and PA implementations within CPAchecker. Finally, DPS-KLEE runs KLEE instead of the CPAchecker based symbolic execution.

DPS can be as effective as CMC. In comparison with CMC and Reducer, we observe that DPS-CPA performs comparably for SE+PA and slightly worse for PA+SE. For SE+PA, there

TABLE II: DPS-based vs CMC-based Cooperation

	SE + PA		PA + SE	
	Correct	**Incorrect**	**Correct**	**Incorrect**
CMC	2 186	**27**	2 254	50
Reducer	1 694	43	2 128	58
DPS-CPA	2 260	34	2 002	**45**
DPS-KLEE	**2 772**	32	**2 654**	**45**

TABLE III: Impact of Unfold Strategy on DPS Uniques

	SE + KI	PA + KI	SE + BMC
DPS (with all unfolding)	**35**	**31**	**26**
w/o loop unrolling	23	15	22
w/o function cloning	**35**	**31**	24

are 195 tasks that DPS-CPA correctly solves but neither CMC nor Reducer do. For PA+SE, there are 25 tasks that DPS-CPA solves but neither CMC nor Reducer do. However, there are also between 118 (SE+PA) and 280 (PA+SE) tasks that CMC and Reducer can solve but DPS-CPA cannot. For most of these tasks (103 and 268 tasks, resp.), the main reason of failure is the short runtime provided to the first verifier: As DPS restricts the runtime of the first verifier to 16 s during splitting, tasks that can be solved after 16 s are often missed. The remaining tasks (15 and 12, resp.) are uniques that are solved with the help of the generated condition.

Off-the-shelf tools can significantly improve performance. By employing off-the-shelf components, DPS-KLEE can solve significantly more verification tasks correctly than the other cooperation types. This highlights the importance of being able to use off-the-shelf tools: Without any modification, KLEE alone already solves 565 tasks that neither SE (CPAchecker) nor PA do. With DPS, we can exploit the effectiveness of KLEE in our cooperative scheme while maintaining the ability to uniquely solve verification tasks.

DPS-based cooperation achieves performance comparable to CMC and reducer-based cooperations. By employing efficient off-the-shelf verifiers, DPS can significantly increase the number of verification tasks solved.

C. RQ3: Impact of Unfold Strategy

For RQ3, we evaluate the impact of our unfold strategy on the best performing combinations in terms of overall correct results (SE+KI), overall correct proofs (PA+KI) and overall correct alarms (SE+BMC). We compare the effectiveness of DPS with both types of unfolding turned on and without them. In this, we specifically focus on the number of uniquely solved tasks per combination.

Results. Our results are shown in Tab. III. We find out that loop unrolling is necessary and deactivating it significantly reduces the number of uniquely solved tasks (with a reduction of 4 to 16 uniques). Deactivating function cloning has often a negligible impact on the number of uniquely solved tasks. This indicates that most of the tasks that are uniquely solved are solved by splitting at split locations that directly appear in the program or appear in loops. As all types of unfolding turned on works best, we employed DPS with both types of unfolding activated in our experiments for RQ1 and RQ2.

Both types of unfolding can have a positive impact on the number of uniquely solved tasks achieved with a DPS-based cooperation.

VI. Discussion

We briefly describe some insights about DPS we gained while inspecting specific results of the experiments, and discuss limitations of our approach as well as threats to the validity.

A. Lessons Learned

To gain a better understanding of *how* DPS impacts the verification process, we had a closer look at the tasks that could and could not be solved by the individual verifiers after running DPS. In the following, we discuss the lessons we learned in the process.

Already small changes can drastically change verifier behavior. A key assumption of our work is that software verifiers behave consistently. In particular, we assume that a verifier that is powerful enough to solve the original task is also powerful enough to solve all reduced variants. In this way, we can safely apply our cooperation scheme without making the task harder for the second verifier. However, we found that this is not always the case: Fig. 5 presents one extreme case where just unrolling the loop once lets the verifier (PA) fail. To understand why this happens, we manually analyzed the verification log of PA. We detected that due to loop unrolling PA can no longer compute the correct loop invariant (via interpolation) that is necessary to prove the task safe. Interestingly enough, we noticed that slight modifications of the program can also help us in the verification process. For example, at least 4 out of the 35 tasks uniquely solved by SE+KI are solved because the program merger encodes the control flow slightly differently.

Splitting can help verification. Recent works on *static* program splitting [37], [47], [75] have shown that program splitting can significantly improve the performance of underapproximating analyses in finding specification violations. Haltermann et al. [47] report that static program splitting has no significant impact on the ability of overapproximating analyses to prove programs safe. Therefore, we were quite surprised to find that the overapproximating analyses (PA and KI) were able to prove some tasks safe during DPS that they could not solve alone. For example for the combination KI+SE, we find that 5 out of 10 uniques are proofs that KI could only generate when running on program splits.

DPS is most effective for tasks with multiple verification goals. After analyzing the uniquely solved tasks for the combinations SE+BMC, SE+KI, KI+PA and, PA+KI, we noticed that the

```
0 int x, y;
1 assume(x == y && x >= 0);
2 // UNROLLED
3 if(x > 0){
4   x--; y--;
5   while (x > 0) { x--; y--; }
6 }
7 assert(y>=0);
```

Fig. 5: `benchmark37_conjunctive` after unrolling the while loop once.

majority of uniquely solved tasks (between 17 and 22 tasks) are tasks with multiple verification goals (e.g., multiple asserts or error locations). For these tasks, the first verifier is able to prove a subset of all verification goals. The residual program then contains the remaining verification goals which are solved by the second verifier.

B. Limitations and Threats to Validity

We have conducted our experiments using tasks from the `sv-benchmarks` of SV-COMP, one of the largest available collections for C program verification. Although it is widely used (especially in software verification competitions), our findings might not completely carry over to other real world C programs. In our experiments, we focused on a subset of 4 771 tasks of the ReachSafety category and three verification tools that implement four different verification techniques. Our results might therefore be biased to the specific selection of tasks and tools used in our experiments.

The correctness of our program transformations used for program splitting and merging is another threat. In principle, a bug in our implementation could lead to new tasks solved correctly (for the wrong reasons). Therefore, we manually checked whether all tasks that are newly solved are transformed correctly. Still, since our implementation is a prototype, it might contain bugs that impact the behavior of DPS on other tasks.

We found during our experiments that DPS significantly suffers from *redundancy of computation* and *trivial splits*. Because most off-the-shelf verifiers do not have an interface to share their verification progress, the verifiers have to re-explore a significant part of the program after each split. Trivial splits are program splits that do not significantly reduce the number of feasible traces per split. This could for example happen if the split condition is trivially true or false. While this does not affect the soundness of our approach, it could significantly reduce the effectiveness and efficiency of our approach. Still, as our evaluation shows, our approach improves the performance of existing verifiers.

VII. Related Work

In this work, we proposed a novel cooperative scheme that uses dynamic program splitting to enable the cooperation between off-the-shelf verification tools. In the following, we discuss the most closely related approaches in cooperative verification and program decomposition.

Cooperative verification. Many approaches have been proposed to combine the strengths of verification techniques [1], [2], [4], [5], [9], [12], [27], [28], [30], [33], [39], [40], [43]–[45], [51]–[54], [63], [68], [69], [77], [82], [83]. These approaches combine verification techniques by running them in parallel, in sequence or interleaved. Many of these approaches require some form of conceptual integration [2], [9], [27], [28], [43], [63], i.e., the approach tightly combines various verification components within one tool. Exchanging components or integrating new verification techniques in a conceptual integration is therefore often difficult (as it requires new implementations) or impossible (as it requires algorithmic innovation). For this reason, recent approaches [1], [4], [5], [39], [40], [44], [45], [49], [52], [68], [69], [77], [83] have explored ways to combine (black-box) verification components more loosely. For example, Conditional Model Checking (CMC) [6] combines multiple verifiers that exchange information about the progress of the verification. C-CEGAR [8] decomposes the CEGAR loop into multiple components that work together to solve a common verification task. Key to all of these approaches are verification artifacts (such as verification conditions) that can be generated and understood by the individual components. This naturally limits the types of verification components that can be used in a cooperation. Our technique employs C programs as the exchange format and only relies on the outputs naturally provided by a verifier (e.g. whether a program split is safe or not). Therefore, our technique can be used to combine arbitrary off-the-shelf verifiers.

Static Decompositions. Another way to combine verification components is via static (program) decompositions [3], [31], [36], [41], [50], [57], [58], [60], [61], [64]–[66], [70], [73], [74], [76], [78]. The key idea is to statically split the given program or its state space into multiple parts that can be analyzed individually. For example, ranged symbolic execution [75], SynergiSE [71], [80], ranged model checking [37], and ranged program analysis [46]–[48] split the program paths into multiple path ranges (i.e., consecutive paths) that can be analyzed in parallel. Distributed Summary Synthesis [15] splits the program into program blocks. Inverso and Trubiani [56] and EPA-Lazy-CSeq [35], [67] internally split the paths of a multi-threaded program based on thread scheduling. Most of these techniques utilize the computed splits internally to guide the verifier. Therefore, Haltermann et al. [46] recently proposed to encode the restriction to the range into the program via instrumentation statements, which allows the application of arbitrary analyses. They however noticed in a later work [48] that static splitting can easily lead to suboptimal splits such that the individual verifiers waste time on a subprogram that could be solved by another verifier. With dynamic program splitting, we can use a verifier to guide the splitting process.

Dynamic Decompositions. There are mainly two approaches to dynamic program decompositions in software verification:

(1) work stealing [18], [25], [75], [80], [84] and (2) verifier-guided decompositions [10], [11], [29], [55]. In work stealing, an idle verifier "steals" subtasks from another verifier. Subtasks can be subprograms that are split from the original task. In verifier-guided decompositions, the verifier explicitly reports information about its verification progress, which is then used to decompose the program in already verified and unverified parts. For example, Reducer [10], [11], [29] utilize the verification condition generated by the first verifier to construct a residual program containing all unverified parts. Huster et al. [55] split the task into verified and unverified proof obligations (as reported by the verifier). Further approaches document their achieved work in the program in the form of verification annotations [23], [24] or assume statements [34]. All of these approaches require that the verifier explicitly informs the decomposition procedure about the progress made on the program. As a result, all of these approaches require verifiers that are modified [10], [11], [23], [24], [29], [55] or specifically designed [34] for this purpose. With our approach, we can utilize the effectiveness of off-the-shelf components without any modification.

Finally, the idea of dynamic program splitting along control-flow has already been used in the recently proposed tool BUBAAK-SpLit [21]. BUBAAK-SpLit, however, only uses a fixed combination of tools (first KLEE, second SLOW-BEAST [22]). It moreover does neither employ merging nor different stop criteria for splitting. Our approach can thus be seen as a generalization and extension, enabling the use of arbitrary off-the-shelf verification tools that can be chained together to form a stronger verifier.

VIII. CONCLUSION

Software verification is an *important* but very *challenging* problem. It is important since we depend on the correctness and reliability of software systems and it is challenging as the problem is in general undecidable. To address software verification problems, many software verification tools have been developed that all come with their own strengths and weaknesses. Cooperative verification promises to combine their strengths, but so far no cooperative approach has been proposed that can fully utilize all publicly available *off-the-shelf* verification tools.

To address this shortcoming, we explored a novel cooperative scheme comprising dynamic program splitting. By *dynamically* splitting the verification task into multiple subtasks, our approach can identify which parts of the task are already verified and which have to be further processed by a second verifier. As each subtask is encoded within a program, our cooperation scheme can utilize arbitrary off-the-shelf verification tools. Our evaluation has shown the importance of this property: Not only can our cooperation scheme be used to improve the performance of a single verification tool, but by using off-the-shelf components, our approach is able to outperform existing cooperation techniques that rely on tighter integrated forms of cooperation.

In future work, we will investigate whether the inefficiencies of our method can be addressed without compromising its generality. A first idea would be to integrate a form of value analysis in the splitting process to directly avoid generating trivial splits.

REFERENCES

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From under-approximations to over-approximations and back," in *TACAS*, ser. LNCS 7214. Springer, 2012, pp. 157–172.

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *ICSE*. ACM, 2014, pp. 1083–1094.

[3] J. Barnat, L. Brim, and J. Chaloupka, "Parallel breadth-first search LTL model-checking," in *ASE*. IEEE, 2003, pp. 106–115.

[4] N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *ISSTA*. ACM, 2008, pp. 3–14.

[5] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," in *CAV*, ser. LNCS 9206. Springer, 2015, pp. 622–640.

[6] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *FSE*. ACM, 2012.

[7] D. Beyer, "State of the art in software verification and witness validation: SV-COMP 2024," in *TACAS*, ser. LNCS 14572. Springer, 2024, pp. 299–329.

[8] D. Beyer, J. Haltermann, T. Lemberger, and H. Wehrheim, "Decomposing software verification into off-the-shelf components: An application to CEGAR," in *ICSE*. ACM, 2022, p. 536–548.

[9] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *CAV*, ser. LNCS 4590. Springer, 2007, pp. 504–518.

[10] D. Beyer and M.-C. Jakobs, "FRed: Conditional model checking via reducers and folders," in *SEFM*, ser. LNCS 12310. Springer, 2020, pp. 113–132.

[11] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim, "Reducer-based construction of conditional verifiers," in *ICSE*. ACM, 2018, pp. 1182–1193.

[12] D. Beyer, S. Kanav, and C. Richter, "Construction of verifier combinations based on off-the-shelf verifiers," in *FASE*, ser. LNCS 13241. Springer, 2022, pp. 49–70.

[13] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification," in *CAV*, ser. LNCS 6806. Springer, 2011, pp. 184–190.

[14] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate abstraction with adjustable-block encoding," in *FMCAD*. IEEE, 2010, pp. 189–197.

[15] D. Beyer, M. Kettl, and T. Lemberger, "Decomposing software verification using distributed summary synthesis," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, 2024.

[16] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *STTT*, vol. 21, no. 1, pp. 1–29, 2019.

[17] D. Beyer and H. Wehrheim, "Verification artifacts in cooperative verification: Survey and unifying component framework," in *ISoLA*, ser. LNCS 12476. Springer, 2020, pp. 143–167.

[18] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys*. ACM, 2011, pp. 183–198.

[19] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX OSDI*. USENIX Association, 2008, pp. 209–224.

[20] M. Chalupa and T. A. Henzinger, "Bubaak: Runtime monitoring of program verifiers - (competition contribution)," in *TACAS*, ser. LNCS 13994. Springer, 2023, pp. 535–540.

[21] M. Chalupa and C. Richter, "Bubaak-SpLit: Split what you cannot verify (competition contribution)," in *TACAS*, ser. LNCS 14572. Springer, 2024, pp. 353–358.

[22] M. Chalupa and J. Strejcek, "Backward symbolic execution with loop folding," in *SAS 2021*, ser. LNCS, vol. 12913. Springer, 2021, pp. 49–76.

[23] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *ICSE*. ACM, 2016, pp. 144–155.

[24] M. Christakis, P. Müller, and V. Wüstholz, "Collaborative verification and testing with explicit assumptions," in *FM*, ser. LNCS 7436. Springer, 2012, pp. 132–146.

[25] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *OSR*, vol. 43, no. 4, pp. 5–10, 2009.

[26] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, ser. LNCS 1855. Springer, 2000, pp. 154–169.

[27] P. Cousot and R. Cousot, "Systematic design of program-analysis frameworks," in *POPL*. ACM, 1979, pp. 269–282.

[28] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "Combination of abstractions in the ASTRÉE static analyzer," in *ASIAN*, ser. LNCS 4435. Springer, 2008, pp. 272–300.

[29] M. Czech, M.-C. Jakobs, and H. Wehrheim, "Just test what you cannot verify!" in *FASE*, ser. LNCS 9033. Springer, 2015, pp. 100–114.

[30] M. Dangl, S. Löwe, and P. Wendler, "CPAchecker with support for recursive programs and floating-point arithmetic - (competition contribution)," in *TACAS*, ser. LNCS 9035. Springer, 2015, pp. 423–425.

[31] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *PLDI*. ACM, 2002, pp. 57–68.

[32] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," in *SAS*, ser. LNCS 6887. Springer, 2011, pp. 351–368.

[33] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *ICSE*. IEEE, 2007, pp. 3–12.

[34] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *FSE*. ACM, 2017, pp. 174–185.

[35] B. Fischer, S. L. Torre, and G. Parlato, "VeriSmart 2.0: Swarm-based bug-finding for multi-threaded programs with Lazy-CSeq," in *ASE*. IEEE, 2019, pp. 1150–1153.

[36] J. Fischer, R. Jhala, and R. Majumdar, "Joining dataflow with predicates," in *FSE*. ACM, 2005, pp. 227–236.

[37] D. Funes, J. H. Siddiqui, and S. Khurshid, "Ranged model checking," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[38] M. Y. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: An industrial-strength C model checker," in *ASE*. ACM, 2018, pp. 888–891.

[39] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *ISSRE*. IEEE, 2013, pp. 360–369.

[40] M. Gao, L. He, R. Majumdar, and Z. Wang, "LLSPLAT: Improving concolic testing by bounded model checking," in *SCAM*. IEEE, 2016, pp. 127–136.

[41] H. Garavel, R. Mateescu, and I. M. Smarandache, "Parallel state space construction for model-checking," in *SPIN*, ser. LNCS 2057. Springer, 2001, pp. 217–234.

[42] M. J. Gerrard and M. B. Dwyer, "ALPACA: A large portfolio-based alternating conditional analysis," in *ICSE*. IEEE / ACM, 2019, pp. 35–38.

[43] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *POPL*. ACM, 2010, pp. 43–56.

[44] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "SYNERGY: A new algorithm for property checking," in *FSE*. ACM, 2006, pp. 117–127.

[45] J. Haltermann and H. Wehrheim, "CoVEGI: Cooperative verification via externally generated invariants," in *FASE*, ser. LNCS 12649. Springer, 2021, pp. 108–129.

[46] J. Haltermann, M.-C. Jakobs, C. Richter, and H. Wehrheim, "Ranged program analysis via instrumentation," in *SEFM*, ser. LNCS 14323. Springer, 2023, pp. 145–164.

[47] ——, "Parallel program analysis via range splitting," in *FASE*, ser. LNCS 13991. Springer, 2023, pp. 195–219.

[48] ——, "Parallel program analysis on path ranges," *Science of Computer Programming*, vol. 238, p. 103154, 2024.

[49] J. Haltermann and H. Wehrheim, "Exchanging information in cooperative software validation," *Softw. Syst. Model.*, vol. 23, no. 3, pp. 695–719, 2024.

[50] M. Handjieva and S. Tzolovski, "Refining static analyses by trace-based partitioning using control flow," in *SAS*, ser. LNCS 1503, G. Levi, Ed. Springer, 1998, pp. 200–214.

[51] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski, "Ultimate Automizer and the search for perfect interpolants - (competition contribution)," in *TACAS*, ser. LNCS 10806. Springer, 2018, pp. 447–451.

[52] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in OPAL," in *FSE*. ACM, 2020, pp. 184–196.

[53] L. Holík, M. Kotoun, P. Peringer, V. Soková, M. Trtík, and T. Vojnar, "Predator shape analysis tool suite," in *HVC*, ser. LNCS 10028, 2016, pp. 202–209.

[54] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification," in *ASE*. IEEE, 2008, pp. 1–6.

[55] S. Huster, J. Ströbele, J. Ruf, T. Kropf, and W. Rosenstiel, "Using robustness testing to handle incomplete verification results when combining verification and testing techniques," in *ICTSS*, ser. LNCS 10533. Springer, 2017, pp. 54–70.

[56] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *PPoPP*. ACM, 2020, pp. 202–216.

[57] M.-C. Jakobs and H. Wehrheim, "Programs from proofs of predicated dataflow analyses," in *SAC*. ACM, 2015, pp. 1729–1736.

[58] P. Jalote, V. Vangala, T. Singh, and P. Jain, "Program partitioning: A framework for combining static and dynamic analysis," in *WODA*. ACM, 2006, pp. 11–16.

[59] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[60] K. Laster and O. Grumberg, "Modular model checking of software," in *TACAS*, ser. LNCS 1384. Springer, 1998, pp. 20–35.

[61] F. Lerda and R. Sisto, "Distributed-memory model checking with SPIN," in *SPIN*, ser. LNCS 1680. Springer, 1999, pp. 22–39.

[62] S. Löwe, M. U. Mandrykin, and P. Wendler, "CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses - (Competition Contribution)," in *TACAS*, ser. LNCS 8413. Springer, 2014, pp. 392–394.

[63] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*. IEEE, 2007, pp. 416–426.

[64] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *ESOP*, ser. LNCS 3444, S. Sagiv, Ed. Springer, 2005, pp. 5–20.

[65] S. McPeak, C. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," in *FSE*. ACM, 2013, pp. 554–564.

[66] D. Monniaux, "The parallel implementation of the Astrée static analyzer," in *APLAS*, ser. LNCS 3780, K. Yi, Ed. Springer, 2005, pp. 86–96.

[67] T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato, "Parallel bug-finding in concurrent programs via reduced interleaving instances," in *ASE*. IEEE, 2017, pp. 753–764.

[68] Y. Noller, R. Kersten, and C. S. Pasareanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *ISSTA*. ACM, 2018, pp. 322–332.

[69] Y. Noller, C. S. Pasareanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "HyDiff: Hybrid differential software analysis," in *ICSE*. ACM, 2020, pp. 1273–1285.

[70] F. Pauck and H. Wehrheim, "Together strong: Cooperative Android app analysis," in *FSE*. ACM, 2019, pp. 374–384.

[71] R. Qiu, S. Khurshid, C. S. Pasareanu, J. Wen, and G. Yang, "Using test ranges to improve symbolic execution," in *NFM*, ser. LNCS 10811. Springer, 2018, pp. 416–434.

[72] C. Richter, M. Chalupa, M.-C. Jakobs, and H. Wehrheim, "Replication package for article 'Cooperative Software Verification via Dynamic Program Splitting'," Aug. 2024, https://doi.org/10.5281/zenodo.13142908.

[73] X. Rival and L. Mauborgne, "The trace partitioning abstract domain," *ACM TOPLAS*, vol. 29, no. 5, p. 26, 2007.

[74] E. Sherman and M. B. Dwyer, "Structurally defined conditional dataflow static analysis," in *TACAS*, ser. LNCS 10806. Springer, 2018, pp. 249–265.

[75] J. H. Siddiqui and S. Khurshid, "Scaling symbolic execution using ranged analysis," in *SPLASH*. ACM, 2012, pp. 523–536.

[76] M. Staats and S. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *ISSTA*. ACM, 2010, pp. 183–194.

[77] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*. The Internet Society, 2016.

[78] U. Stern and D. L. Dill, "Parallelizing the mur*phi* verifier," in *CAV*, ser. LNCS 1254. Springer, 1997, pp. 256–278.

[79] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE TSE*, vol. 12, no. 1, pp. 157–171, 1986.

[80] G. Yang, R. Qiu, S. Khurshid, C. S. Pasareanu, and J. Wen, "A synergistic approach to improving symbolic execution using test ranges," *Innov. Syst. Softw. Eng.*, vol. 15, no. 3-4, pp. 325–342, 2019.

[81] B. Yin, L. Chen, J. Liu, J. Wang, and P. Cousot, "Verifying numerical programs via iterative abstract testing," in *SAS*, ser. LNCS 11822. Springer, 2019, pp. 247–267.

[82] L. Yin, W. Dong, W. Liu, and J. Wang, "Parallel refinement for multi-threaded program verification," in *ICSE*. IEEE, 2019, pp. 643–653.

[83] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: Better together!" in *ISSTA*. ACM, 2006, pp. 145–156.

[84] L. Zhou, S. Gan, X. Qin, and W. Han, "SECloud: Binary analyzing using symbolic execution in the cloud," in *CBD*. IEEE, 2013, pp. 58–63.