✓ Artifact evaluated by FSE

On-the-Fly Decomposition of Specifications in Software Model Checking

Sven Apel [1], Dirk Beyer [2], Vitaly Mordan [3], Vadim Mutilin [3], and Andreas Stahlbauer [1]

[1] University of Passau, Germany [2] LMU Munich, Germany [3] ISP RAS, Russia

ABSTRACT

Major breakthroughs have increased the efficiency and effectiveness of software model checking considerably, such that this technology is now applicable to industrial-scale software. However, verifying the full formal specification of a software system is still considered too complex, and in practice, sets of properties are verified one by one in isolation. We propose an approach that takes the full formal specification as input and first tries to verify all properties simultaneously in one run of the verifier. Our verification algorithm monitors itself and detects situations for which the full set of properties is too complex. In such cases, we perform an *automatic* decomposition of the full set of properties into smaller sets, and continue the verification seamlessly. To avoid state-space explosion for large sets of properties, we introduce *on-the-fly property weaving*: properties get weaved into the program's transition system on the fly, during the analysis; which properties to weave and verify is determined dynamically during the verification process. We perform an extensive evaluation based on verification tasks that were derived from 4 336 Linux kernel modules, and a set of properties that define the correct usage of the Linux API. Checking several properties simultaneously can lead to a significant performance gain, due to the fact that abstract models share many parts among different properties.

CCS Concepts

•Software and its engineering → Formal software verification;

Keywords

Software Model Checking, Program Analysis, Multi-Property Verification, Specification, Formal Methods, Decomposition

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950349

1. INTRODUCTION

Software model checking is an automatic, exhaustive, and precise approach for verification. A software model checker takes a program and a formal specification as input, constructs an abstract model of the program to verify whether the program adheres to the specification, and provides the verification result TRUE or FALSE (ideally accompanied by a witness). The challenge in model checking is to represent huge state spaces by sound and complete approximations that avoid the state-space explosion problem. Abstraction is crucial to construct an abstract state space of tractable size [19, 27, 33]; the *abstraction precision* [10, 14] defines the level of abstraction. The model of a program has to be abstract enough to allow for an efficient verification process, and precise enough to be able to prove, or refute, that the specification is satisfied [2, 27]. The complexity of the model, and the resources required for computing an abstraction, increase with the complexity of the specification to verify.

The formal specification of a software system is typically described by dozens, or hundreds, of properties. For example, given a Linux kernel module, there is a set of API usage rules that the module must fulfill; each of these rules is a safety property. Defining each property in isolation respects the principle of separation of concerns, and ensures maintainability and comprehensibility of the formal specification. Due to the size and structure of an elaborate software system, such as the Linux kernel, it is state-of-the-art to *decompose* the specification *before* the verifier starts, and to verify each part (property) of the specification in isolation, in a new instance of the verification tool. Verifying many properties simultaneously can exhaust computing resources, in particular due to state-space explosion, expensive solver queries, and complex abstraction computations. Thus, small sets of properties are preferred to reduce the complexity of the abstract model (a lower precision is sufficient). Several significant improvements in the last years (most prominently, the performance breakthrough of SMT solving [4]) make us believe it is time to address the challenge of verifying large sets of properties at once. Verifying a set of properties in one run has major advantages over the state-of-the-art approach, especially in industrial practice: (1) the program is parsed only once, (2) invariants can be reused across different properties, and (3) the overall number of expensive satisfiability checks can be reduced. We developed a set of techniques as a foundation for verifying many properties simultaneously in one run of a software model checker, including on-the-fly property weaving and dynamic specification decomposition.

Specification Automata. A required property [1] is usually represented by a monitor automaton [7,42]. Generally, there are two alternative methods to verify a property for a program with software model checking: (a) the monitor automaton is weaved into the source code of the program and thus reduced to the reachability of a certain program location [3,7] and (b) the monitor automaton runs in parallel and the current state of the automaton is represented separately with an abstract domain over the automaton states [20,43]. Method (a) is easy to implement in a pre-processing step, for example, by an aspect weaver [39], but has the drawback that the source code is changed, blown-up considerably, and traceability is more difficult to ensure. If applied to a multi-property setup, instrumenting all properties of a specification into the program code introduces a lot of noise in the program, especially in cases where only a subset of the properties is verified; this adds another burden to the abstraction-refinement procedure, makes calls to the SMT solver more expensive, and makes the program-comprehension task for the verification engineer harder. Method (b) is conceptually much more elegant, but has the following drawback: because each property is equivalent to an automaton (that is, each property has an own state space, which needs to be tracked in addition to the current state in the program exploration), and the combination of the properties can be seen as a product construction, there is a new kind of state-space explosion: a huge number of different states in the specification.

Both methods have advantages that seem mutually exclusive: Method (a) allows us to encode the specification state space into the program's control flow and thus benefit from the same symbolic representation as in the program state space. It helps to avoid an explicit representation of the automaton states of the resulting product specification. Method (b) allows us to integrate the concept of abstraction precisions [10], which makes it possible to enable and disable properties on the fly. We can view the process of reducing the number of properties as abstraction and the process of adding properties as refinement of the abstract state space.

On-the-Fly Property Weaving. We have solved the above dilemma by designing the Loom analysis, which is a flexible and configurable analysis preventing state-space explosion that arises from the specification. This analysis *weaves* only those properties *on-the-fly* into the program's transition system that are to be verified, and allows us to use the concept of dynamic precision adjustment to dynamically enable and disable properties during the verification process.

While we would like to leverage reuse between properties (abstract models for different properties often overlap), we sometimes have to reduce the number of simultaneously verified properties to keep the complexity of the abstract model to construct manageable.

Dynamic Specification Decomposition. Our goal is to *automatically decompose* large sets of properties into smaller sets, instead of requiring the decomposition from the verification engineer. A multi-property model checker takes a set of properties (the specification) as input, and provides a separate result for each property as output. The verifier automatically decides which properties to verify in combination, or individually. The goal is to decompose the specification such that the overall effectiveness and efficiency of the verification process is improved.

Table 1: Simultaneous verification of different sets of properties of a Linux kernel module for a network device; verifying all properties simultaneously runs into a timeout (1.5 hours)

Property 132a	134a	43a	68a	68b	Analysis Speedup	Total Speedup	Analysis CPU Time (s)	Total CPU Time (s)	Loop Unrollings	Refinements
•	–	–	–	–	1.0	1.0	110	140	7	75
–	•	–	–	–	1.0	1.0	15	48	1	2
–	–	•	–	–	1.0	1.0	18	50	2	5
–	–	–	•	–	1.0	1.0	25	61	2	5
–	–	–	–	•	1.0	1.0	5.9	40	1	0
•	•	–	–	–	1.0	1.2	120	150	7	81
•	–	•	–	–	.68	.88	180	220	7	100
•	–	–	•	–	.58	.76	230	270	12	65
•	–	–	–	•	1.0	1.2	110	150	7	75
–	•	•	–	–	1.5	1.8	21	55	3	7
–	•	–	•	–	1.5	1.8	27	62	3	9
–	•	–	–	•	1.3	1.7	16	51	1	2
–	–	•	•	–	.90	1.3	48	88	4	23
–	–	•	–	•	1.2	1.7	19	54	2	5
–	–	–	•	•	1.2	1.7	26	60	2	5
•	•	•	–	–	.64	.93	220	260	7	102
•	•	–	•	–	.56	.83	260	310	12	65
•	•	–	–	•	1.0	1.4	120	160	7	81
•	–	•	•	–	.048	.077	3200	3300	12	113
•	–	•	–	•	.66	.99	200	240	7	100
•	–	–	•	•	.57	.86	250	280	12	65
–	•	•	•	–	.92	1.5	63	110	4	27
–	•	•	–	•	1.9	2.6	20	54	3	7
–	•	–	•	•	1.8	2.5	25	60	3	9
–	–	•	•	•	.96	1.6	51	92	4	23
•	•	•	•	–	–	–	–	–	–	–
•	•	•	–	•	.61	1.0	240	280	7	102
•	•	–	•	•	.47	.80	330	370	12	65
•	–	•	•	•	.046	.083	3400	3600	12	113
–	•	•	•	•	1.0	1.8	63	110	4	27
•	•	•	•	•	–	–	–	–	–	–

We developed a technique that allows us to dynamically (on-demand and on-the-fly) adjust the specification by switching on and off individual properties using the concept of dynamic precision adjustment [10]. To determine when and where to decompose, our analysis algorithm *monitors itself* to detect situations in which the abstract model for a certain set of properties becomes too complex to be efficiently constructed. The approach requires heuristic estimates for choosing a promising decomposition alternative.

To discuss this aspect in more detail, we show results of verifying a Linux kernel module in Table 1, with all possible decompositions of a set of properties. The module provides access to a family of Siemens Gigaset devices.[2] We chose this module for discussion because it is one of the most challenging for finding a good decomposition, which is a central problem to be solved—we obtain significant speedups for other kernel modules as well (see Fig. 2). The analysis CPU time only includes the time for model checking (model construction, refinement based on infeasible error paths), the CPU time for parsing the input program, constructing the

[1] We focus on safety properties that define the correct usage of an operating-system API [1,32]. In our experiments, we verify properties of Linux kernel modules, due to their open-source availability, and their well-understood requirements.

[2] http://kernel.org/doc/Documentation/isdn/README.gigaset

control-flow graph, and other preparation and initialization steps is included in the total CPU time.

Verifying all five properties simultaneously is not possible because we run into a timeout (last row of Table 1). If we choose a decomposition where property `132a` is verified in isolation and the other four properties simultaneously, we spend a total analysis time of 110 s + 63 s = 173 s. The decomposition {{`134a`, `68a`, `68b`}, {`132a`}, {`43a`}} needs an analysis CPU time of 25 s + 110 s + 18 s = 153 s. Admittedly, this is not a too big speedup compared to 174 s for verifying each property separately: the overall cost is dominated by the one hard property `132a`.

The potential speedup of verifying several properties simultaneously depends on the program and on the properties. Some program parts require a high precision for verifying several properties, whereas the precision can be coarse for other parts (because they are not relevant for any property). Different properties have different scopes, and thus, could benefit from intermediate verification results that are available in the scope from other properties. Overlapping scopes of properties can also have a negative impact on the verification performance: the state space might in some cases explode for those regions. Our approach tries to detect such situations early and verify certain properties in isolation.

Related Work. We classify the related work into approaches related to multi-property verification and approaches related to specification automata and specification weaving.

Multi-Property Verification. Checking properties individually has already been identified to be impractical in the context of hardware verification [17, 23, 36]. Some of the discussed problems and ideas are applicable to verifying software: (1) grouping properties according to their potential of reuse of intermediate results, (2) reuse of learned facts, and (3) projection of constraints, and reachable states, between properties based on a cone-of-influence analysis. The tool VARVEL [29], which uses abstract interpretation and bounded model checking, has been used for verifying several properties of software; it is applied function-wise and does not perform a whole-program analysis; function contracts can be used to reason about whole programs [30]. Multi-aspect verification [37] is the work closest to our approach; it was inspired by the positive effects of precision reuse [14] and led to our new concepts and experiments. The idea is to use the LDV toolkit to produce, for a given program source code and a set of properties, a source code that is instrumented with the given properties. The model checker (CPACHECKER) is repeatedly queried to verify such instrumented programs.

The potential of reuse for similar queries was explored in the area of SAT solving [31, 41, 45]. Large sets of test goals are considered in the context of dynamic test generation [25], or in dynamic analysis [22]. In general, test generation can benefit from reusing parts of abstract reachability graphs [11]. Also regression verification can benefit from the reuse of abstract reachability graphs [26] or of abstraction precisions [14] to reduce the verification effort over several verification runs.

Our work complements the existing approaches by automatic specification decomposition with self-monitoring for the case that the full specification is too complex, an implementation using unbounded software model checking, verifying full programs as a whole, and by avoiding pre-processing (combine or weave) of properties. We conduct an extensive study on the effects, and specifics, of verifying several properties in one run of a software model checker.

Specification Weaving and Automata. In the past two decades, several languages for expressing formal software specifications have been proposed [3, 5, 7, 18]. Our specification language is inspired by the BLAST query language [7, 18]. Every safety property can be reduced to the reachability of an error label; a new version of the program, where the specification is part of the program code, is generated in a pre-processing step and a verifier checks then whether an error label that represents a violating program location is reachable. Aspect-oriented programming is one way of instrumenting a program with its specification [39]. In the spirit of configurable software verification [9], it is desirable to integrate the specification automaton as a configurable program analysis that runs simultaneously with the main analysis (on-the-fly product construction of program and specification) [43]. Our specification automata can encode several properties, similar to supermonitors [40]. Our specification analysis uses dynamic precision adjustment [10] to enable or disable certain properties on demand. A work that is closely related to our approach for on-the-fly weaving can be found in the context of performance monitoring [28]. Code instrumentation is added or removed dynamically at any point during the program execution; the main goal of that work is to collect performance data. Our LOOM analysis applies the idea of dynamic instrumentation to static analysis, that is, at verification time, without changing the actual program.

Contributions. We summarize our results as follows:

- We propose a fully-integrated approach for verifying sets of properties in one run of a software model checker.
- We developed the LOOM CPA, a technique that makes the simultaneous verification of large sets of properties efficient, by weaving properties on-the-fly into the program.
- We developed a configurable algorithm that dynamically decomposes a set of properties into smaller sets, whenever self-monitoring detects that the verification task is too complex for the current set of properties.
- We perform an extensive experimental evaluation of our approach to illustrate how an on-the-fly specification decomposition can increase both the efficiency and the effectiveness of the overall verification process. Our benchmark suite comprises 4 336 Linux kernel modules.
- We provide a replication package [3] that contains all data and a detailed description for reproducing our experiments. Our implementation is based on the open-source software-verification framework CPACHECKER; the source code is available under the Apache 2 license.

2. PRELIMINARIES

Before we introduce our new approach and the corresponding techniques, we start with some preliminary definitions.

Control-Flow Automata. We represent a program as *control-flow automaton* (CFA), which is a tuple (L, l_0, G) consisting of a set L of program locations, the entry location $l_0 \in L$, and a set $G \subseteq L \times Ops \times L$ of control-flow edges; an operation in *Ops* represents an assignment operation or an assume operation, in the programming language and (if available) its abstract syntax tree; the subset $R \subset Ops$ represents all assume operations (Boolean expressions); the subset $S \subset Ops \setminus R$ represents all assignment operations. For a set M, we write M^* for the set of all words over M,

[3] http://sosy-lab.org/research/spec-decomposition/

and ϵ for the word of length zero. The concatenation of two words w_1 and w_2 is written as $w_1 \circ w_2$.

Properties and Partitions. The set $\mathbb{P} = \{p_1, \ldots, p_n\}$ consists of all *properties* of a program that are to be verified. A *partitioning* $\mathscr{P} \subseteq 2^X$ of a set X is a set of non-empty sets, where all elements $P_1, P_2 \in \mathscr{P}$, with $P_1 \neq P_2$, are pairwise disjoint ($P_1 \cap P_2 = \varnothing$), and $X = \bigcup_{P \in \mathscr{P}} P$.

Configurable Program Analysis. Our work builds on the concept of configurable software verification [9] with dynamic precision adjustment [10]. CPAs of the form $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec}, \mathsf{target})$ are the central building blocks of this formalism. The full-fledged program analysis is composed of several component CPA to form a *composite analysis*. The abstract domain $D = (C, \mathscr{E}, [\![\cdot]\!])$ with the semi-lattice $\mathscr{E} = (Z, \sqsubseteq, \sqcup, \top)$ defines the type of abstract representation of concrete states from C, where C is the set of all concrete states of the program; one abstract state $e \in Z$ represents a set of concrete states (*region*), which can be obtained by the concretization function: $[\![e]\!] \subseteq C$. A precision $\pi \in \Pi$ defines certain aspects of the state space that should be represented by abstract states in a given abstract domain. The transfer relation $\rightsquigarrow$ defines, for each abstract state, a set of successor abstract states. The operator merge can combine two abstract states such that both abstract states are subsumed by the resulting abstract state; given an abstract state e, and a set $\mathscr{R} \subseteq Z$ of abstract states, the coverage-check operator stop returns *true* if e represents only concrete states that are already represented in $\mathscr{R}$, that is, $[\![e]\!] \subseteq \bigcup_{e' \in \mathscr{R}} [\![e']\!]$, and returns *false* otherwise. Given an abstract state e, a precision π, and a set $\mathscr{R} \subseteq Z$, the precision adjustment operator prec can provide a new abstract state e' with an adjusted precision π'. The operator $\mathsf{target} : Z \to \mathbb{B}$ returns *true* if a given abstract state is the goal of the reachability analysis, that is, if the abstract state violates the specification, and returns *false* otherwise. The strengthening operator $\downarrow$ of a CPA can be used to get information from the abstract state of another component CPA to compute a stronger successor abstract state; it is called within the transfer relation of the wrapping composite CPA after all component CPAs have done their transfers. More details on the CPA formalism can be found in the literature [9, 10, 12].

Abstract Reachability Graph. To verify properties of a program, we run a program analysis [10] (Alg. 1) that is defined by a CPA $\mathbb{D}$. Starting from all initial abstract states with precision (from W_0), we compute all successor abstract states until we have reached the fixed point. All abstract states that have been reached so far are stored in the set reached; the set waitlist represents the frontier, that is, the set of abstract states for which the successor abstract states still have to be computed. The abstract states in the set reached are the nodes of an *abstract reachability graph* (ARG), a directed graph that is rooted at the initial abstract states with precision (from W_0). *Lazy abstraction* [27] proposes to use different precisions for different parts of the ARG; both the set reached and waitlist consist of pairs of an abstract state with abstraction precision.

3. LOOM CPA: ON-THE-FLY WEAVING

This section introduces (1) our specification automata, (2) our specification analysis, which provides on-the-fly specification decomposition, and (3) the LOOM analysis, which provides on-the-fly weaving of operations from the specification automata into the transition relation of the analysis.

Algorithm 1 $\mathit{CPAalg}_{\mathbb{D}}(R_0, W_0)$, adopted from [10]

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec}, \mathsf{target})$,
a set $R_0 \subseteq E \times \Pi$ of abstract states with precision,
a subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
where E denotes the elements of the lattice of D
Output: a set of reachable abstract states with precision,
a subset of frontier abstract states with precision
Variables: a set $\mathsf{reached} \subseteq E \times \Pi$, a set $\mathsf{waitlist} \subseteq E \times \Pi$

1: $\mathsf{reached} := R_0$; $\mathsf{waitlist} := W_0$
2: **while** $\mathsf{waitlist} \neq \varnothing$ **do**
3: $(e, \pi) := \mathsf{choose}(\mathsf{waitlist})$
4: **for each** e' with $e \rightsquigarrow (e', \pi)$ **do**
5: $(\hat{e}, \hat{\pi}) := \mathsf{prec}(e', \pi, \mathsf{reached})$
6: **for each** $(e'', \pi'') \in \mathsf{reached}$ **do**
7: $e_{new} := \mathsf{merge}(\hat{e}, e'', \hat{\pi})$
8: **if** $e_{new} \neq e''$ **then**
9: $\mathsf{waitlist} := (\mathsf{waitlist} \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
10: $\mathsf{reached} := (\mathsf{reached} \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
11: **if** $\neg\ \mathsf{stop}(\hat{e}, \{e \mid (e, \cdot) \in \mathsf{reached}\}, \hat{\pi})$ **then**
12: $\mathsf{waitlist} := \mathsf{waitlist} \cup \{(\hat{e}, \hat{\pi})\}$
13: $\mathsf{reached} := \mathsf{reached} \cup \{(\hat{e}, \hat{\pi})\}$
14: **if** $\mathsf{target}(\hat{e})$ **then**
15: **return** $(\mathsf{reached}, \mathsf{waitlist})$
16: **return** $(\mathsf{reached}, \varnothing)$

A key design decision of our approach is to give the formal specification and the program code to the verifier as separate objects. Instead of instrumenting the specification into the program code before running the verifier, the verifier weaves the specification *on-the-fly* into the transition relation of the program, during the analysis. Whenever a different (sub-) set of properties is to be verified, we do not have to re-instrument the input program and restart the verifier. In our method of on-the-fly specification weaving, only selected properties are weaved, and only for a specific part of the state space (which properties, and where to weave them, can be dynamically adjusted, e.g., via lazy abstraction [27]). Irrelevant noise in terms of unnecessarily instrumented properties is avoided and the algorithm (and the engineer who tries to understand an error path) deals with 'clean' abstract program paths.

A formal specification that represents a set of program executions can be expressed as a temporal-logic formula or by an automaton that accepts the same set of executions [16, 38, 44]. We concentrate on temporal *safety* properties, which can be expressed using finite automata [34, 42, 44].

Specification Automata. A specification automaton encodes a set of properties and observes, but not restricts, the state space of an analysis run. A specification automaton $(Q, \Sigma, \delta, q_0, F)$ for a given CFA (L, l_0, G) is a nondeterministic finite automaton with a finite set Q of control states, an alphabet $\Sigma \subseteq 2^G \times S^* \times R^*$, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, an initial control state $q_0 \in Q$, and a set F of accepting control states. Each $q_p \in F$ represents that property $p \in \mathbb{P}$ is violated (i.e., a path through the program that violates p is found). We write $q \xrightarrow{\sigma} q'$ if $(q, \sigma, q') \in \delta$ holds. A symbol $(\gamma, s, r) \in \Sigma$ consists of a set γ of control-flow edges, a sequence s of assignment operations to weave, and a sequence r of assume operations.

To increase traceability and to support enabling and disabling automaton transitions that are irrelevant for specific properties, we calculate two maps that assign to each control state $q \in Q$ and to each transition $\tau \in \delta$ the sets of relevant properties $\mathbb{P}(q)$ and $\mathbb{P}(\tau)$, respectively.

Our specification automata support three modes of expressing and encoding properties. The *pure automata-based mode* tracks every possible state of the specification explicitly, and thus contributes to the explosion of the (specification) spate space. In the *pure weaving mode*, the specification that is represented by the automaton gets weaved into the program completely. The set of control states only consists of the initial control state and a number of accepting control states, each representing a different property. The *hybrid mode* combines the first two modes. Different control states are used for guiding the process of weaving the given set of properties into the transition relation of the underlying analysis. A control state of an automaton typically models the current context of the program.

To not affect the completeness or soundness of the program analysis, the operations that are introduced by the weaving process must never modify or restrict the state space of the program under analysis: (1) assignment operations are allowed to assign values to only such variables that were introduced by the automaton itself, and (2) for each control state, the disjunction of all predicates from assume operations on the outgoing transitions must evaluate to *true*.

Property Relevance. A property p is *relevant* for a given program if the specification automaton has a transition τ to control state q' with $p \in \mathbb{P}(q')$ or $p \in \mathbb{P}(\tau)$ and transition τ syntactically matches a control-flow edge of the program. Situations where a property is not relevant for a program can also indicate a flaw in the specification.

Specification Analysis. For each specification automaton, we instantiate one specification analysis. The specification analysis (1) keeps track of the current state of the automaton and determines its successors based on the transition relation δ and the current control-flow edge of the CFA, (2) it provides operations that should later be weaved into the control flow, (3) it provides assumptions on the state space for strengthening the composite abstract state, (4) it can disable the verification of certain properties on-the-fly for some region of the state space, and (5) it is responsible for determining whether a violating state has been reached. By running several specification analyses in parallel, we lazily accept the union of words without any explicit automaton construction.

The specification analysis for a specification automaton is a CPA $\mathbb{D}_{\mathbb{S}} = (D_{\mathbb{S}}, \Pi_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, \mathsf{merge}_{\mathbb{S}}, \mathsf{stop}_{\mathbb{S}}, \mathsf{prec}_{\mathbb{S}}, \mathsf{target}_{\mathbb{S}})$. A precision $\pi \in \Pi_{\mathbb{S}}$ of this CPA is a set of properties to verify; it implicitly defines the subset δ^{π} of transitions of the specification automaton that are relevant for verifying properties from π: $\delta^{\pi} = \{\tau \in \delta \mid \mathbb{P}(\tau) \cap \pi \neq \emptyset\}$. The specification analysis interprets the transitions of the specification automaton according to the precision: for a given transition $q \xrightarrow{\sigma} q'$, it uses the *precision-adjusted* transition $q \xrightarrow{\sigma}_{\pi} q'$ (i.e., it uses only transitions that are relevant for verifying a property $p \in \pi$). The CPA is defined as follows:

1. The abstract domain $D_{\mathbb{S}} = (C, \mathcal{Q}, [\![\cdot]\!])$ consists of the set C of concrete states, a semi-lattice $\mathcal{Q}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{Q} = (Z, \sqsubseteq, \sqcup, \top_{\mathcal{Q}})$ is a flat lattice on the set of abstract states $Z = (Q \cup \top) \times Ops^*$, where one abstract state $(q, o) \in Z$ consists of an automaton state q and a sequence o of operations.

2. The transfer relation $\rightsquigarrow_{\mathbb{S}}$ has the transfer $(q, o) \overset{g}{\rightsquigarrow}_{\mathbb{S}} ((q', o'), \pi)$, if the specification automaton A has a precision-adjusted transition $q \xrightarrow{\sigma}_{\pi} q'$, with $\sigma = (\gamma, s, r)$ and $g \in \gamma$, and $o' = r \circ s$. If no transition is applicable, it has the (stuttering) transfer $(q, o) \overset{g}{\rightsquigarrow}_{\mathbb{S}} ((q, o), \pi)$. An analysis shall not stop exploring a program path after a property violation; other properties could be violated later along the path as well (soundness).

Our specification language supports patterns variables: the sequences s and r of assignment and assume operations, respectively, have to be instantiated within the transfer relation based on the current automaton state and the control-flow edge g. The set of matching control-flow edges γ can be defined by patterns like `$1 = malloc($2)`, which matches for example the control-flow edge with the assignment operation `ptr = malloc(512)`. The expressions that match the pattern variables can then be referenced for instantiating new sequences of assignment or assume operations. Continuing the example, the pattern `$1 == NULL`, would be instantiated as the assume operation `ptr == NULL`.

3. The precision $\pi_{\mathbb{S}}$ of the analysis determines which properties are verified for which part of the state space. The precision adjustment operator $\mathsf{prec}_{\mathbb{S}} : Z \times \Pi_{\mathbb{S}} \times 2^{Z \times \Pi_{\mathbb{S}}} \rightarrow Z \times \Pi_{\mathbb{S}}$ is central in the on-the-fly decomposition of a specification.

Based on several measures of the verification effort spent on each property, we dynamically decide whether we should stop verifying a property starting from a given abstract state onwards. The operator $\mathsf{exceeds} : \mathbb{P} \rightarrow \mathbb{B}$ returns *true* if a specific budget for a given property is exceeded, and *false* otherwise. A property p is removed from the precision $(\mathsf{prec}_{\mathbb{S}}(e, \pi_{\mathbb{S}}, \mathsf{reached}) = (e, \pi_{\mathbb{S}} \setminus \{p\}))$ if $\mathsf{exceeds}(p) = true$.

4. The operator $\mathsf{merge}_{\mathbb{S}} : Z \times Z \rightarrow Z$ keeps two abstract states always separate: $\mathsf{merge}_{\mathbb{S}}(e, e') = e'$.

5. The operator $\mathsf{stop}_{\mathbb{S}} : Z \times 2^{Z} \rightarrow \mathbb{B}$ checks whether there is already an abstract state that subsumes a given state: $\mathsf{stop}_{\mathbb{S}}(e, \mathscr{R})$ returns *true* if $\exists e' \in \mathscr{R} : e \sqsubseteq e'$, otherwise *false*.

6. The operator $\mathsf{target}_{\mathbb{S}} : Z \times \mathbb{B}$ returns *true* if the specification automaton is in an accepting control state $q_p \in F$, which signals the violation of property p: given an abstract state $(q, \cdot) \in Z$, it returns *true* if $q \in F$, otherwise *false*.

On-the-fly Weaving with the* Loom *Analysis. The Loom analysis is a composite [9] CPA $\mathbb{D}_{\mathbb{L}\#}$ that weaves sequences of operations from the specification CPA into the transition relation of the analysis. The definition of specification automata ensures that this process does not make the data-flow analysis for the program unsound.

The Loom analysis is composed of (at least) the location CPA as well as the instances of the specification CPA. Given the composite state $e = (l, \ldots, (q_1, o_1), \ldots, (q_m, o_m))$ and the concatenation $o_c = o_1 \circ \ldots \circ o_m = (op_1, \ldots, op_n)$ of all operation sequences, with $o_c \neq \epsilon$, we add new control-flow edges to G: $(l^{\#}, op_1, l_1), \ldots, (l_{n-1}, op_n, l)$, where the last control-flow edge leads to the original location l; the transfer relation $\rightsquigarrow_{\mathbb{L}\#}$ has the transfer $(l, \ldots) \rightsquigarrow_{\mathbb{L}\#} (l^{\#}, \ldots)$, which is taken after all component CPAs have performed their transfers and strengthenings.

The only operator that is allowed to modify the components of the composite state is the strengthening operator. The strengthening of the location CPA [9] is used to modify the current location based on the operations to weave that are provided in the states of the specification CPAs. Without loss of generality, we assume that no transition $g \in G$ of the CFA leads to the entry location $l^{\#}$ of the sequence of newly introduced control-flow edges. Figure 1 illustrates this process on an example.

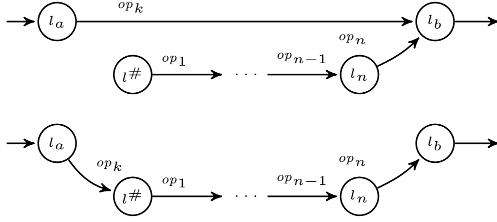

Figure 1: Given an abstract state $e = (l_b, \dots, (q, o), \dots)$ created based on the control-flow operation op_k, with $o = (op_1, \dots, op_n)$, the Loom CPA introduces a sequence of control-flow edges that correspond to the operations o. The program analysis is redirected to their entry location $l^\#$ such that $e' = (l^\#, \dots, (q, \epsilon), \dots)$.

4. DECOMPOSING PROPERTY SETS

Different properties exhibit different characteristics [21], which determine whether some properties should be verified on the same abstract model or separately. Ideally, a decomposition strategy separates those pairs of properties that, if verified simultaneously, lead to a decrease of efficiency, and it bundles those properties that benefit from simultaneous verification. We propose run-time strategies for specification decomposition, which can consider measures of the verification process and the state space (under construction), for deciding which properties to verify separately or in a specific combination. Depending on the chosen abstract domain or analysis configuration, different measures might be appropriate. Such measures can help to identify properties whose scope spans across loops, properties that involve an undecidable theory, or any property that would cause a blow-up of the abstract reachability graph.

We define three points in time for providing or performing a decomposition of the specification: (1) a domain expert can provide an *authoritative* decomposition of the specification before starting the verifier, (2) the decomposition can be performed *dynamically* each time before running the model-checking algorithm, and (3) it can be performed *on the fly* during the execution of the model-checking algorithm.

4.1 Decomposition Framework

Model-checking algorithms found in the literature are neither designed for decomposing a specification, nor are they able to handle the fact that only a fraction of the specification might have been verified. Algorithm 2 *SDC* (Specification DeComposition) was particularly designed for this concern: it wraps a standard analysis algorithm analyze[4], and its specifics are defined by a decomposition strategy $\mathbb{S}$.

The algorithm takes four parameters. The initial abstract state e_0 with the adjusted (line 4) initial precision π_0 defines the initial frontier (line 5), which is the starting point for the analysis to compute successor states until a fixed point is reached. The initial partitioning of the specification and the initial resource budget define the initial decomposition process. If no property is left for verification (line 18), the algorithm terminates with a pair of satisfied properties and violated properties. Properties are removed from the set remaining if a feasible error path was found (line 9), or a fixed point was reached with the properties enabled for verification (line 14), or if the analysis resigns to verify them (line 17).

Resource Budgets. Due to the general undecidability of the verification problem, some properties cannot be verified, even with an arbitrary amount of resources; the algorithm would terminate without reaching a fixed point, returning the result *unknown*. Appropriate resource budgets for specific properties or sets of properties are therefore crucial for the overall efficiency and effectiveness of verification with specification decomposition.

Different resources H can be taken into account for defining budgets on properties, or the overall verification process. A *resource* $\kappa \in H$ can be, for example, the CPU time that is spent on a specific operation of the analyzer or the full (enclosing) analysis algorithm, or the number of precision elements with specific characteristics, or the number of branchings in the ARG (this list is not exhaustive).

A *resource utilization monitor* $\mathbb{U} = (\mathsf{ut}, \mathsf{ut}_{\mathbb{P}})$ tracks the amount of resources consumed by the analyzer. The utilization operator $\mathsf{ut} : H \to \mathbb{Z}$ returns the total amount $\mathsf{ut}(u)$ for a given resource $\kappa \in H$ that was consumed between the start of the monitor and the time the operator was invoked. The utilization operator $\mathsf{ut}_{\mathbb{P}} : \mathbb{P} \times H \to \mathbb{Z}$ returns the amount $\mathsf{ut}_{\mathbb{P}}(u)$ for a given resource $\kappa \in H$ that was consumed for a specific property $p \in \mathbb{P}$. We denote the set of all resource utilization monitors by $\mathscr{U}$. Whenever the specification-decomposition algorithm starts with a new set of properties, the resource utilization monitor is reset (all utilization operators are set to *zero* whenever operator init is called).

A *resource budget* $\beta : \mathscr{U} \to \mathbb{B}$ is an operator that returns *true* for a given pair of ut and $\mathsf{ut}_{\mathbb{P}}$ if there are sufficient resources left for a given property $p \in \mathbb{P}$, and *false* otherwise. We denote the set of all budgets by $\mathscr{B}$; budget $\beta_\top$ does not restrict; budget $\beta_\bot$ declares all resources unavailable.

As part of our framework, we define a set of operators that take the resource monitors and the resource budgets into account for performing the specification decomposition, dynamically, on-the-fly: the precision-adjustment operator $\mathsf{prec}_{\mathbb{S}}$ of the specification analysis and the operators of the decomposition strategy (the exhaustion operator resigned and the partitioning operator adjPrecision).

Decomposition Strategy. Algorithm 2 *SDC* for specification decomposition is configured by a decomposition strategy $\mathbb{S} = (\mathsf{adjPrecision}, \mathsf{resigned})$:

1. The operator $\mathsf{adjPrecision} : \Pi \times 2^{\mathbb{P}} \to \Pi \times \mathscr{B}$ for a given decomposition strategy $\mathbb{S}$ takes as input a precision and a set of remaining properties (that are still to be verified), and returns as output a pair of precision π and budget β. The precision defines which properties to verify simultaneously in the next run of the analyzer. Our (stateful) implementation of operator adjPrecision internally maintains a decomposition of the specification, which is represented as partitioning $\mathscr{P}$ of properties. The partitioning is iteratively adjusted based on the verification progress (in terms of remaining properties and consumed resources) and the next partition is returned as part of π. The new resource budget β limits the CPU time, number of transitions, number of refinements for the next run of the analyzer.

[4]This can be a standard program analysis, such as *CPAalg* (Alg. 1), or even testing; algorithm analyze must satisfy the interface: operate on sets reached and waitlist, and return a set of error paths. While we base our implementation on CPAchecker, our concepts are not bound to this framework.

Algorithm 2 $SDC_{\mathbb{S},\mathsf{analyze}}((e_0,\pi_0),\mathscr{P}_0,\beta_0)$

Input: a decomposition strategy $\mathbb{S} = (\mathsf{adjPrecision}, \mathsf{resigned})$,
an analysis algorithm analyze,
an initial state $e_0 \in E$ with precision $\pi_0 \in \Pi$,
an initial partitioning $\mathscr{P}_0$,
the initial resource budget β_0
Output: set of satisfied properties, set of violated properties
Variables: a set reached $\subseteq E \times \Pi$, a set waitlist $\subseteq E \times \Pi$,
a precision π that determines which properties to track,
a set remaining $\subseteq \mathbb{P}$ of remaining properties,
a set satisfied $\subseteq \mathbb{P}$ of satisfied properties,
a set violated $\subseteq \mathbb{P}$ of violated properties,
a set cexs of counterexamples (abstract program paths),
which violate properties $\mathbb{P}(\mathsf{cexs}) \subseteq \mathbb{P}$

```
1:  π := π0; β := β0
2:  satisfied := ∅; violated := ∅; remaining := ℙ
3:  repeat
4:     (π, β) := adjPrecision_{𝒫0,β0}(π, remaining)
5:     waitlist := {(e0, π)}; reached := waitlist
6:     repeat
7:        (reached, waitlist, cexs) := analyze_β(reached, waitlist)
8:        violated := violated ∪ ℙ(cexs)
9:        remaining := remaining \ violated
10:       waitlist := disable(waitlist, violated)
11:    until cexs = ∅
12:    if waitlist = ∅ then
13:       satisfied := satisfied ∪ active(reached)
14:       remaining := remaining \ satisfied
15:    else
16:       // Resource budget β exhausted!
17:       remaining := remaining \ resigned(reached)
18: until remaining = ∅
19: return (satisfied, violated)
```

2. The operator resigned : $2^{E\times\Pi} \to 2^{\mathbb{P}}$ takes as input a set of abstract states with precision (reached) and returns a set of properties that are considered not verifiable (within the given resource constraints) and should no longer be considered in any further iterations of the decomposition algorithm.

The auxiliary function disable : $2^{E\times\Pi} \times 2^{\mathbb{P}} \to 2^{E\times\Pi}$ returns for a given set of abstract states with precision and a set of properties P, a new set of abstract states with precision from which all properties from P are removed from the precision of the specification analyses.

The helper function active returns, for a given set of reachable abstract states with precision, a set of properties that the analysis was still verifying in the last run of the analyzer (that is, the properties that were still active).

4.2 Decomposition Strategies

Next, we describe how specification-decomposition strategies can be instantiated within our framework. An experimental evaluation of our strategies can be found in Sect. 5.

The strategies are different in decomposition operator adjPrecision, the initial partitioning $\mathscr{P}_0$, and the initial budget β_0. The strategies consider the set $H = \{at, rc, sc\}$ of resources, were at, rc, sc denote the analysis CPU time, the number of refinements, and the number of transitions taken so far by the analyzer, respectively. All strategies define a budget for the analysis CPU time $at \in H$ (for example, the budget $\beta_{900} = \mathsf{ut}(at) \leqslant 900$ limits the analysis CPU time to 900 s). The operator exceeds, which is used in the precision adjustment operator $\mathsf{prec}_{\mathbb{S}}$ of the specification CPA, returns *true* if any of the resources is exhausted in β. The operator $\mathsf{resigned}_{One}$ returns *true* if only one property was enabled for verification in the given set of abstract states, otherwise, it returns *false*.

Strategy $\mathbb{S}_0 = (\mathsf{adjPrecision}_{\mathbb{S}_0}, \mathsf{resigned}_{\mathbb{S}_0})$***.*** This strategy does not perform any specification decomposition (it places all properties in one single partition) and does not specify any resource limit. This strategy is used as a baseline for evaluating the performance of more advanced strategies. We use this strategy with the initial partitioning $\mathscr{P}_0 = \{\mathbb{P}\}$ and the initial resource budget $\beta_0 = \beta_\top$. After the analysis terminates for the initial partitioning $\mathscr{P}_0$, the operator $\mathsf{adjPrecision}_{\mathbb{S}_0}$ returns $(\pi, \beta_\bot)$: we signal the exhaustion of all resources. The operator $\mathsf{resigned}_{\mathbb{S}_0}$ returns the full set $\mathbb{P}$ as result, which leads to the immediate termination of the specification decomposition algorithm.

Strategy $\mathbb{S}_1 = (\mathsf{adjPrecision}_{\mathbb{S}_1}, \mathsf{resigned}_{One})$***.*** This strategy starts by verifying all properties simultaneously, and continues to verify the remaining properties in isolation (one by one). This simple strategy is useful if no measure is available for deciding which properties to verify in the same partition. The goal of this strategy is to not lose verification results in cases where verifying all properties simultaneously is not possible. We use this strategy with the initial partitioning $\mathscr{P}_0 = \{\mathbb{P}\}$ and the initial resource budget $\beta_0 = \beta_{900}$. After the analysis is finished for the initial partitioning $\mathscr{P}_0$, the call $\mathsf{adjPrecision}_{\mathbb{S}_1}(\pi, \mathsf{remaining})$ returns (π', β_{900}) where π' enables exactly one arbitrarily (but deterministically) chosen property $p \in \mathsf{remaining}$. The operator $\mathsf{resigned}_{One}$ ensures that a property is verified only once in isolation.

Strategy $\mathbb{S}_2 = (\mathsf{adjPrecision}_{\mathbb{S}_2}, \mathsf{resigned}_{One})$***.*** Strategy $\mathbb{S}_2$ builds on strategy $\mathbb{S}_1$ but restricts the budget for specific resources. We have configured the strategy to stop verifying a property if it causes, relative to the other properties, significantly more refinement iterations (which might correlate with a much higher abstraction precision). A property p is removed during the verification process from the precision π if there have been, relative to the maximum number of refinements for the other properties in π, at least twice as much, but at least ten, refinements: Given the number of refinements $rc_p = \mathsf{ut}_{\mathbb{P}}(p, rc)$, we use the initial budget $\beta_0 = \mathsf{ut}_{\mathbb{P}}(p, rc) < 10 \vee \mathsf{ut}_{\mathbb{P}}(p, rc) < 2 \cdot \max_{p' \in (\pi \setminus p)} \mathsf{ut}_{\mathbb{P}}(p', rc)$.

The background is that our program analysis uses counterexample guided abstraction refinement (CEGAR) [19] for constructing the abstract model of the program, that is, we use CEGAR to determine the abstraction precision that is necessary to rule out infeasible error paths. An error path cex witnesses the violation of a set of properties $\mathbb{P}(cex)$. The idea of strategy $\mathbb{S}_2$ is to use the number of infeasible error paths (that have to be ruled out by an abstraction refinement) as an indicator for the cost of verifying a specific property. The number of refinement iterations is a critical factor that affects the performance of an analyzer that is based on CEGAR [14].

Strategy $\mathbb{S}_3 = (\mathsf{adjPrecision}_{\mathbb{S}_3}, \mathsf{resigned}_{One})$***.*** This strategy tries to reduce the effort for verifying properties that are likely irrelevant for a given program, and to focus on the properties that have been identified as relevant. Whether a property p is considered relevant is determined based on the number of transitions $sc \in H$ to a control state q of a specification automaton with $p \in \mathbb{P}(q)$, which can be queried from the resource utilization monitor: $\mathsf{ut}_{\mathbb{P}}(p, sc)$. The set $\mathbb{P}_{Rel} = \{\{p \mid p \in \mathbb{P} \wedge \mathsf{ut}_{\mathbb{P}}(p, sc) > 0\}\}$ represents the set of properties that have already been identified to be relevant;

the set $\mathbb{P}_{Irr} = \mathbb{P} \setminus \mathbb{P}_{Rel}$, represents the properties that have not yet been identified to be relevant and that might be irrelevant for the given program. Each partition of properties is verified with the resource budgets β_{900}.

The strategy operates in three phases; we start with the initial partitioning $\mathscr{P}_0 = \{\mathbb{P}\}$, which defines the first phase of the strategy. The full set properties get verified simultaneously for 900 s without any limits on certain properties.

In the second phase of the strategy, we verify only those properties that have, so far, not yet identified to be relevant for the program; the second phase is skipped if $|\mathbb{P}_{Irr}| = 0$. Thus, we make sure that all properties that have not been identified to be relevant in the first phase are really irrelevant for the given program (at least with the given resource budget). If any new relevant property was identified during that phase, the set of relevant properties is corrected and the second phase is restarted.

In the third phase of our strategy, we verify all those properties separately that have still not been verified successfully, but have been identified to be relevant for the given program. That is, given the set remaining of remaining properties, the properties $\mathbb{P}_{Rel} \cap \mathsf{remaining}$ get verified in isolation with the resource budget β_{900}.

Strategy $\mathbb{S}_4$***.*** Strategy $\mathbb{S}_4$ combines strategies $\mathbb{S}_2$ and $\mathbb{S}_3$: It considers the relevance of properties to reduce the number of properties to verify separately and a resource budget on properties limits the number of refinements per property.

5. EVALUATION

In a series of experiments, we evaluate the potential of software verification with on-the-fly decomposition of specifications in terms of efficiency and effectiveness.

5.1 Research Questions

Our experimental evaluation is guided by five research questions, which are divided into three groups that provide different perspectives on our approach.

Simultaneous Verification of All Properties. We first investigate the performance benefit of verifying all properties of a specification in one verification run using one shared abstract model (one abstract reachability graph) of the program. This reuse of intermediate verification results can influence both the efficiency and the effectiveness of a software verifier. For now, we are only interested in the performance of the analysis procedure itself; the effort for all pre-processing steps that are associated with the verification run will be investigated separately.

RQ1.1: *How many verification tasks can be solved more* efficiently*, and what is the speedup in terms of CPU time for the analysis, by verifying all properties of a specification simultaneously compared to verifying each property in a separate run of the verifier?*

RQ1.2: *How many verification tasks can be solved more* effectively*, in terms of number of verification results, by verifying all properties of a specification simultaneously compared to verifying each property in a separate run of the verifier?*

Specification-Decomposition Strategies. Our initial example (see Table 1) illustrates that there are cases for which simultaneously verifying all properties of a specification with one abstract model is not feasible. But, it can still be beneficial to verify at least some of the properties simultaneously. This set of research questions aims at investigating the potential of automatic specification-decomposition strategies:

RQ2.1: *How many verification tasks can be solved more* efficiently *by applying automatic decomposition strategies ($\mathbb{S}_1$ – $\mathbb{S}_4$) for verifying several properties simultaneously in one run of the verifier?*

RQ2.2: *How many verification tasks can be solved more* effectively *by applying automatic decomposition strategies ($\mathbb{S}_1$ – $\mathbb{S}_4$) for verifying several properties simultaneously in one run of the verifier?*

Overall Performance. The above research questions have focused purely on the analysis procedure; we have not discussed the costs for the pre-processing steps. These steps include parsing the program, construction of the CFA, and setting up the different CPAs (which includes initializing solvers, etc.). Taking these steps into account results in the last research question:

RQ3.1: *What* overall effectiveness and efficiency *can we expect* in practice *(that is, including costs for the pre-processing steps), from a verifier with on-the-fly specification decomposition compared to traditional configurations that verify each property in a separate run of the verifier?*

5.2 Setup

Our benchmark suite consists of two sets of Linux kernel modules: the full set with 4 336 modules and a subset with 250 modules. Details on the benchmarking environment and the benchmark suite can be found in Sect. 8, where we describe the replication package. The replication package includes all tools and data for replicating our experiments, as well as a detailed description of all properties.

Presentation. If not stated otherwise, we report CPU time in hours, rounded to two significant digits. Analysis CPU time excludes the time taken by pre-processing steps that are performed before the analysis itself starts.

Analysis Domain. We have configured a composite analysis, where one of the components is a predicate analysis with adjustable-block encoding [13]. This analysis is used for representing central aspects of the state space of the program and the portion of the specification that was weaved (on-the-fly) with our LOOM analysis; assume operations that are provided by the specification automata get encoded within the strengthening operation of this analysis. The program counter, the call stack, and the control states of the specification automata, are tracked each by separate analyses. Our predicate analysis is configured to compute a Boolean predicate abstraction [35] for each function-call location and for each head of a loop (we perform a large-block encoding [8]).

Experiments. The *baseline* for all our discussions and experiments is an analysis where only one property is verified in one instance of the verifier. We have limited the analysis CPU time for one property to 0.25 h. Limiting the analysis CPU time instead of the total CPU time helps us to exclude costs for pre-processing, and focus on the costs of the state-space exploration. Given the set of 250 kernel modules (see Sect. 8) and the set of 14 properties, we run 3 500 single-property verification tasks; the analysis can provide results in 3 394 cases, from which 97 violate and 3 297 satisfy the property. The analysis CPU time for the solved verification tasks sums up to 52 h (includes the time for tasks that ran into a timeout).

To answer RQ1.1 and RQ1.2, we compare the baseline to a multi-property verification run that is configured to verify all properties in one partition, that is, we use the decomposition strategy $\mathbb{S}_0$ on 250 kernel modules for which we verify all 14 properties; the overall CPU time for analysis is limited to 3.5 h (14 × 0.25 h).

To answer RQ2.1 and RQ2.2, we perform two experiments, where we evaluate the decomposition strategies $\mathbb{S}_1$–$\mathbb{S}_4$. The first experiment is on the 250 kernel modules for which we verify all 14 properties. The second experiment evaluates the efficiency and effectiveness of our decomposition strategies on *"hard"* tasks, that is, tasks for which the baseline configuration is either able to provide results for more properties or is more efficient than strategy $\mathbb{S}_0$. For both experiments, we have limited the overall CPU time for the analysis to 3.5 h.

To answer RQ3.1, we run two experiments for which we do not force the Java Virtual Machine (JVM) to compile most of the bytecode during its startup. The first experiment is on the 250 kernel modules for which we verify all 14 properties. To confirm our results and increase their validity, the second experiment is on the larger set of verification tasks, that is, we verify the 14 properties of all 4 336 Linux kernel modules.

5.3 Results

This section presents the results of our experiments; raw data and more details are shipped with our replication package (Sect. 8).

RQ1.1: Efficiency of Multi-Property Verification. Verifying all properties simultaneously is faster for 80 percent (199 modules) of the modules compared to verifying each property individually. Overall, we gain an average speedup of 5.2 in terms of CPU time for the analysis. A graphical illustration of the results can be found in Fig. 2.

Several relevant properties can be verified without any refinements, purely based on the state of the specification automata. In the cases in which verifying the properties individually is more efficient, the simultaneous checks ran into a timeout for 14 modules, without providing any result; the individual checks were able to provide results for some of the properties of eleven of those modules.

RQ1.2: Effectiveness of Multi-Property Verification. The results illustrate that verifying all properties simultaneously without any decomposition strategy can lead a substantial loss of results: We lost 5.6 % (191) of the results; the goal of specification decomposition is to improve this. On the other hand, the analysis is able to provide five additional results (for different properties in different modules) because the resources are not divided across different partitions.

RQ2.1: Efficiency by Specification Decomposition. For the set of 250 modules, each decomposition strategy can provide an average speedup of at least 5.1. There are subsets of these modules for which specific decomposition strategies are the most efficient choice. Strategy $\mathbb{S}_0$ is the best choice (only those strategies are considered that provide results for all properties of a given module) for 20 %, $\mathbb{S}_1$ for 15 %, $\mathbb{S}_2$ for 18 %, $\mathbb{S}_3$ for 19 %, and $\mathbb{S}_4$ for 12 % of the modules. The baseline configuration provides the best efficiency for 12 % of the modules.

For the subset of 51 "hard" modules, strategy $\mathbb{S}_4$ provides an average speedup of 1.1, and is able to provide a speedup for 33 % of these modules; without a specification decomposition, no speedup was possible for this set of modules.

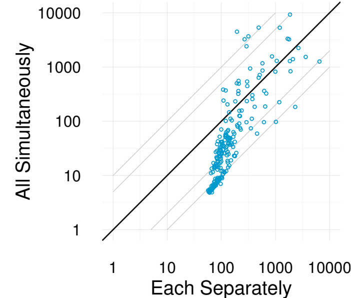

Figure 2: Scatter plot that compares the simultaneous verification of all properties (y-coordinate) to the classical approach where each property is verified in a separate instance (x-coordinate) of the verifier, in terms of CPU time for the analysis

Table 2: Results for different decomposition strategies; the first set of tasks covers 250 Linux kernel modules; the set of "hard" tasks is a subset of those tasks: tasks for which the baseline configuration is either able to provide results for more properties or is more efficient than strategy $\mathbb{S}_0$

	Strategy	Satisfied	Violated	Partitions	Analysis CPU Time (h)	Average Speedup	# Additional Results	# Lost Results	% Lost Results	% Speedup> 1
All Modules	$\mathbb{BL}$	3 297	97	3 500	53	1.0	0	0	0	0
	$\mathbb{S}_0$	3 104	99	250	97	5.2	5	196	5.6	80
	$\mathbb{S}_1$	3 284	96	790	53	5.2	3	17	.49	79
	$\mathbb{S}_2$	3 289	96	724	56	5.1	4	13	.37	78
	$\mathbb{S}_3$	3 281	96	403	33	5.2	0	17	.49	82
	$\mathbb{S}_4$	3 282	96	453	33	5.2	0	16	.46	84
Hard Modules	$\mathbb{BL}$	573	36	714	36	1.0	0	0	0	0
	$\mathbb{S}_0$	379	38	51	94	.43	4	196	27	0
	$\mathbb{S}_1$	572	35	538	44	.67	3	5	.70	14
	$\mathbb{S}_2$	575	35	355	42	.82	4	3	.42	24
	$\mathbb{S}_3$	571	35	188	27	.92	0	3	.42	18
	$\mathbb{S}_4$	572	35	185	24	1.1	0	2	.28	33

The strategies $\mathbb{S}_2$ and $\mathbb{S}_4$, which implement a decomposition heuristic based on the number of abstraction refinements, are more efficient than strategies $\mathbb{S}_1$ and $\mathbb{S}_3$ that do not use such a heuristic.

Table 2 provides more details on the efficiency of different decomposition strategies on the discussed sets of tasks.

RQ2.2: Effectiveness by Specification Decomposition. According to Table 2, all our strategies that perform some kind of specification decomposition can reduce the number of lost results to less than 1 %.

RQ3.1: Overall Performance. For estimating the practical potential of multi-property verification with specification decomposition and on-the-fly specification weaving, we consider the total process CPU time of strategy $\mathbb{S}_4$. Because the JVM is now allowed to take the full advantage (see Sect. 5.2)

of its just-in-time compiler, the results are slightly different from those for the earlier research questions.

For the set of 250 kernel modules, the overall process CPU time reduces, compared to the baseline, from 60 h to 25 h, with an average speedup of 5.5 (median 5.0). To confirm our results, and increase their validity, we evaluated our approach on a larger set of verification tasks that covers 4 336 kernel modules. The overall process CPU time is considerably reduced from 400 h to 130 h, with an average speedup of 8.0 (median 8.6), and a lower speedup for 99 % (4171 of 4 336 modules) of the modules. At the same time, 112 results (0.4 %) for separate properties were lost in comparison to the baseline. Note that this set of modules contains entries for which none of the properties is relevant, leading to the best possible speedup for them.

5.4 Discussion

Our results illustrate that multi-property verification is in many cases more efficient than verifying every property individually. The number of solved problems can be kept on a similar level by using a specification-decomposition strategy. Taking the relevance of properties into account helps to further improve the efficiency of the overall verification process. Measures of the verification process can indicate properties that are likely to cause high costs. The measures can be used to decide which properties to better verify later in another partition (or even in isolation).

On-the-fly weaving of specification automata, with dynamically adjustable precision, helps the verifier to focus on properties that are enabled for analysis at the moment, and the current state of the specification can be encoded in an appropriate abstract domain. Such an encoding is crucial for our approach: it helps avoiding state-space explosion considerably. An experimental comparison to traditional approaches that instrument the specification into the program code before the verifier starts may be promising but is outside the scope of this paper.

One hypothesis is that our approach can benefit from a large portion of intermediate verification results that are similar for different properties. As we use lazy abstraction, only those parts of the state space are modeled with higher precision that are relevant for at least one of the analyzed properties. Common states might be the result of expensive computations, such as the computation of a Boolean predicate abstraction. An indicator for the similarity of state spaces is their size, that is, the number of states in the set reached. We can compare the maximum number of reached states of verification runs that verify a single property to the size of the set reached of multi-property verification runs. This analysis reveals that the number of reached states is indeed similar in many cases, that is, we likely take advantage of this sharing potential; we leave a more elaborate analysis of this observation to future work.

Our decomposition strategies are configured to (re-) construct the state space from scratch for each partition of properties, which is (still) a waste of precious intermediate verification results. Re-using (parts of) previously constructed state space graph for successive partitions could lead to a significant improvement of the performance [11, 26].

5.5 Threats to Validity

Our benchmark suite consists of a substantial set of Linux kernel modules and a set of safety properties that are relevant in that domain. The size and diversity of our benchmark suite ensures that our conclusions are externally valid in that application domain. Different tools with different abstract domains and analysis techniques work differently in terms of sharing abstraction precisions and abstract states; similarly, the SMT solver is critical for the performance.

The chosen time limit of 0.25 h for a single property is chosen more-or-less from previous experience: Most verification tasks that we encounter can be solved within this time period by CPAchecker (cf. one of the reports on the International Competition on Software Verification [6]).

The speedup depends on the number of properties that are verified, and can be artificially increased by including many properties that are not relevant for a verification task. We use a subset of the properties that were defined in context of the Linux Driver Verification project; each property can be potentially relevant for each kernel module, or might become relevant by a minor change to its code.

The distribution of code that is relevant for proving a property is important. The scope of properties in a program influences the potential speedup of our approach. It is not possible to control this variable, thus, we increase internal validity by the large number of experiments on many different modules. Proving a property is in general considered harder than showing its violation. Our set of verification tasks has only a small percentage of violations, such that the overall picture is still valid.

6. CONCLUSION

We presented a set of enabling techniques for verifying formal specifications that can be further decomposed into sets of properties. First, we presented the Loom analysis, a new technique that on-demand and on-the-fly weaves properties into the transition system. This way, we can switch on and off, as needed during the verification process, properties, independent from other analysis components; the precision of the specification analysis defines the set of properties to verify for a specific part of the state space. Second, we developed several promising heuristics for self-monitoring the verification progress and reduce or increase the precision of the analysis (the set of properties to be verified) dynamically during the analysis. The combination of these concepts leads to an efficient and effective analysis of large sets of properties in one run of the verifier. The results of our experimental study are promising: Verifying several properties in one verification run can (in most cases) significantly increase the efficiency of the verification process; this complements the current practice where only single properties are verified in one run of the verifier. Our results open up a number of interesting research directions: Techniques that were successful for verifying single properties in one run might not be the best choice for verifying larger sets of properties.

7. ACKNOWLEDGMENTS

This work has been supported by the State of Bavaria, the Russian Science Foundation, and the German Research Foundation (AP 206/4 and AP 206/6).

8. ARTIFACT DESCRIPTION

To make our results easier to reproduce, we provide a replication package that includes all verification tasks, tools, and scripts for automatically re-running our experiments. The verification tasks have been derived from 4 336 Linux kernel modules, and a set of safety properties that define the correct usage of the Linux API. An implementation of our approach is part of the open-source software verification framework CPAchecker [12]; its source code is freely available under the Apache 2 license. The replication package—a large fraction of it can be reused for other research endeavors—is provided on a supplementary Web site [5] which contains detailed instructions for reproducing our experimental results.

Benchmark Suite. We evaluated our approach on a set of modules from the Linux kernel version 4.0-rc1. The modules were extracted and prepared using the Linux Driver Verification (LDV) toolkit [6] [32], which also takes care of enriching the modules with an environment model of the Linux kernel. Each module has several entry functions which represent, for example, different interrupt handlers; we only consider one entry function (`main0`) per module. We use two sets of modules for our experiments: the full set with 4 336 modules and a subset with 250 modules. The subset consists of randomly chosen tasks from the full set of modules, for which at least two properties are relevant. The pre-processed Linux kernel modules are licensed under GNU GPL 2.0.

The specification consists of 14 safety properties that are relevant for the Linux kernel; a detailed description of these properties can be found in Table 4. Not all properties are relevant for all kernel modules, for example, property `77_1a` is relevant for only two modules from the subset with 250 elements. Table 3 provides an overview on the relevance of the properties for the two sets of kernel modules. For 1 989 of the kernel modules, at least one property is relevant; at least two properties are relevant for 1 059 modules.

Experimental Setup. We have implemented our approach on top of the CPAchecker framework. We used revision 21027 from the branch *muauto* for our experiments, with SMTInterpol as SMT solver. All experiments have been executed on machines with Linux 4.2 and Java 1.7, equipped with two Intel Xeon E5-2650 CPUs and 135 GB of RAM. As we assume that a software verifier can always make use of the full memory installed on a (typical) machine, we limit runs in which only single properties are verified (baseline) and runs where several properties are verified at once to the same amount of memory: 26 GB of Java heap and 30 GB for the process itself; each process was limited to use 4 cores.

Evaluation and Reproducibility. Since CPAchecker is written in Java, special characteristics of a JVM have to be considered [24]. In particular, a scenario where we compare multiple launches of the JVM to a single launch, but also comparing multiple iterations to a single iteration of an algorithm, requires special care. To mitigate the effects of the just-in-time compiler of the JVM, we force the JVM to already compile most of the byte code during its startup. The initial size of the Java heap is set to the maximum.

We measure and control computing resources using BenchExec [15], a framework for reliable and accurate benchmarking, which is freely available and licensed under

[5] http://sosy-lab.org/research/spec-decomposition/
[6] http://linuxtesting.org/project/ldv/

Table 3: Not all properties are relevant for every kernel module; the number of modules for which a property (table header) is relevant is given for two sets of modules: the full set *All* with 4 336 kernel modules, and a subset *Sub* consisting of 250 modules with at least two relevant properties each.

	08_1a	10_1a	32_1a	43_1a	68_1a	68_1b	77_1a	101_1a	106_1a	118_1a	129_1a	132_1a	134_1a	147_1a
Sub	29	119	129	191	31	14	2	4	19	19	20	13	43	23
All	129	783	846	1 054	150	65	6	10	111	82	125	52	200	123

Table 4: Safety properties that we verified for the Linux kernel modules

Property	Description
08_1a	Each module that was referenced with `module_get` must be released with `module_put` afterwards.
10_1a	Each memory allocation that gets performed in the context of an interrupt must use the flag `GFP_ATOMIC`.
32_1a	The same mutex must not be acquired or released twice in the same process.
43_1a	Each memory allocation must use the flag `GFP_ATOMIC` if a spinlock is held.
68_1a	All resources that were allocated with `usb_alloc_urb` must be released by `usb_free_urb`.
68_1b	Each DMA-consistent buffer that was allocated with `usb_alloc_coherent` must be released by calling `usb_free_coherent`.
77_1a	Each memory allocation in a code region with an active mutex must be performed with the flag `GFP_NOIO`.
101_1a	All structs that were obtained with `blk_make_request` must get released by calling `blk_put_request` afterwards.
106_1a	The modules `gadget`, `char`, and `class` that were registered with `usb_gadget_probe_driver`, `register_chrdev`, and `class_register` must be unregistered by calling `usb_gadget_unregister_driver`, `unregister_chrdev` and `class_unregister` correspondingly in reverse order of the registration.
118_1a	Reader-writer spinlocks must be used in the correct order.
129_1a	An offset argument of a `find_bit` function must not be greater than the size of the corresponding array.
132_1a	Each device that was allocated by `by usb_get_dev` must get released with `usb_put_dev`.
134_1a	The `probe` functions must return a non-zero value in case of a failed call to `register_netdev` or `usb_register`.
147_1a	RCU pointer/list update operations must not be used inside RCU read-side critical sections.

Apache 2. All requirements on the hardware, the command-line parameters for CPAchecker, and the verification tasks to use, are specified in benchmark definition files (XML), which are shipped with the replication package.

9. REFERENCES

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proc. EuroSys*, pages 73–85. ACM, 2006.

[2] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with Slam. *Commun. ACM*, 54(7):68–76, 2011.

[3] T. Ball and S. K. Rajamani. Slic: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

[4] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *J. Autom. Reasoning*, 50(3):243–277, 2012.

[5] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. Acsl: ANSI/ISO C specification language. 2008.

[6] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses. In *Proc. TACAS*. Springer, 2016.

[7] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *Proc. SAS*, LNCS 3148, pages 2–18. Springer, 2004.

[8] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32. IEEE, 2009.

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

[10] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.

[11] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *Proc. ESOP*, LNCS 7792, pages 472–491. Springer, 2013.

[12] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[13] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[14] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *Proc. ESEC/FSE*, pages 389–399. ACM, 2013.

[15] D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015.

[16] J. R. Büchi. *On a Decision Method in Restricted Second Order Arithmetic.* 1960.

[17] G. Cabodi and S. Nocco. Optimized model checking of multiple properties. In *Proc. DATE*, pages 543–546. IEEE, 2011.

[18] S. Chaudhuri and R. Alur. Instrumenting C programs with nested word monitors. In *Proc. SPIN*, LNCS 4595, pages 279–283. Springer, 2007.

[19] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pages 154–169. Springer, 2000.

[20] D. Dams and K. S. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. In *Proc. FMCO*, LNCS 4111, pages 138–160. Springer, 2005.

[21] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. ICSE*, pages 411–420. ACM, 1999.

[22] M. B. Dwyer, A. Kinneer, and S. G. Elbaum. Adaptive online program analysis. In *Proc. ICSE*, pages 220–229. IEEE, 2007.

[23] R. Fraer, S. Ikram, G. Kamhi, T. Leonard, and A. Mokkedem. Accelerated verification of RTL assertions based on satisfiability solvers. In *Proc. HLDVT*, pages 107–110. IEEE, 2002.

[24] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. OOPSLA*, pages 57–76. ACM, 2007.

[25] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proc. EMSOFT*, pages 207–216. ACM, 2008.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, LNCS 2772, pages 332–358. Springer, 2003.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[28] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proc. PACT*, pages 201–212. IEEE, 1997.

[29] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, T. Imoto, R. Pothengil, and M. Hussain. Scalable and scope-bounded software verification in Varvel. *Autom. Softw. Eng.*, 22(4):517–559, 2015.

[30] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. Dc2: A framework for scalable, scope-bounded software verification. In *Proc. ASE*, pages 133–142. IEEE, 2011.

[31] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna. Simultaneous SAT-based model checking of safety properties. In *Proc. HAV*, LNCS 3875, pages 56–75. Springer, 2005.

[32] A. V. Khoroshilov, V. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.

[33] G. A. Kildall. A unified approach to global program optimization. In *Proc. POPL*, pages 194–206. ACM, 1973.

[34] O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proc. CAV*, LNCS 1633, pages 172–183. Springer, 1999.

[35] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *Proc. CAV*, LNCS 4144, pages 424–437. Springer, 2006.

[36] C. Loiacono, Marco Palena, P. Pasini, D. Patti, S. Quer, S. Ricossa, and D. Vendraminet. Fast cone-of-influence computation and estimation in problems with multiple properties. In *Proc. DATE*, pages 803–806. ACM, 2013.

[37] V. O. Mordan and V. S. Mutilin. Checking several requirements at once by CEGAR. In *Proc. Ershov Memorial Conference 2015*, LNCS 9609, pages 218–232. Springer, 2015.

[38] D. E. Muller. Infinite sequences and finite machines. In *Proc. SWCT*, pages 3–16. IEEE, 1963.

[39] E. M. Novikov. An approach to implementation of aspect-oriented programming for C. *Programming and Computer Software*, 39(4):194–206, 2013.

[40] R. Purandare, M. B. Dwyer, and S. G. Elbaum. Optimizing monitoring of finite state properties

through monitor compaction. In *Proc. ISSTA*, pages 280–290. ACM, 2013.

[41] X. Qin, M. Chen, and P. Mishra. Synchronized generation of directed tests using satisfiability solving. In *Proc. VLSI*, pages 351–356. IEEE, 2010.

[42] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[43] O. Šerý. Enhanced property specification and verification in Blast. In *Proc. FASE*, LNCS 5503, pages 456–469, 2009.

[44] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS*, pages 332–344. IEEE, 1986.

[45] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proc. FSE*, page 58. ACM, 2012.