Precision Reuse for Efficient Regression Verification [†]

Dirk Beyer [1], Stefan Löwe [1], Evgeny Novikov [2], Andreas Stahlbauer [1], and Philipp Wendler [1]

[1] University of Passau, Germany
[2] Institute for System Programming (ISP RAS), Russia

ABSTRACT

Continuous testing during development is a well-established technique for software-quality assurance. Continuous model checking from revision to revision is not yet established as a standard practice, because the enormous resource consumption makes its application impractical. Model checkers compute a large number of verification facts that are necessary for verifying if a given specification holds. We have identified a category of such intermediate results that are easy to store and efficient to reuse: *abstraction precisions*. The precision of an abstract domain specifies the level of abstraction that the analysis works on. Precisions are thus a precious result of the verification effort and it is a waste of resources to throw them away after each verification run. In particular, precisions are reasonably small and thus easy to store; they are easy to process and have a large impact on resource consumption. We experimentally show the impact of precision reuse on industrial verification problems created from 62 Linux kernel device drivers with 1 119 revisions.

Categories and Subject Descriptors: D.2.4 [*Software Engineering*]: Software/Program Verification F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying, Reasoning about Programs

General Terms: Verification, Reliability, Languages

Keywords: Formal Verification, Regression Checking

1. INTRODUCTION

Reliable software is essential both for convenience and safety in our daily lives and for the revenue in the economy. Producing reliable software is costly; and speeding up testing and formal verification of software can save huge amounts of time and money. Economic pressure requires companies to come up with innovations more quickly by introducing more features in shorter release cycles — software is a key contributor to today's innovations. However, the problem of extending software, e.g., by introducing a new feature, is that this might break existing features — bugs get introduced. This is known as regression. To avoid regression, developers execute automated tests before a new revision of a piece of software is checked-in, in the hope that the tests alarm the developer of any new bug. The quality of the software (in terms of correctness) depends on the coverage of the regression test set. Regression testing is an established and well-investigated technique since many years (e.g., [24,32,34]).

The confidence of correctness can be increased by augmenting the development process with formal verification, i.e., regression verification [18, 25, 27, 35, 37]. Formal verification exhaustively checks the program for bugs, but at the same time consumes large amounts of computation resources (time and memory), in particular when applied to industrial-size software. Regression verification applies formal verification techniques to continuously check development revisions in order to identify regressions early. Innovations in this field pave the road that leads from regression testing to regression verification, and from simply finding bugs to actual proofs of correctness during the whole software-development process.

Verification tools spend much effort on computing intermediate results that are needed to check if the specification holds. In most uses of model checking, these intermediate results are erased after the verification process — wasting precious information (in failing and succeeding runs). There are several directions to reuse (intermediate) results [16]. *Conditional model checking* [7, 20] outputs partial verification results for later re-verification of the same program by other verification approaches. *Regression verification* [18, 25, 27, 35, 37] outputs intermediate results (or checks differences) for re-verification of a changed program by the same verification approach.

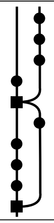
The contribution of this paper is to reuse *precisions* as intermediate verification results. In program analysis, e.g., predicate analysis, shape analysis, or interval analysis, the respective abstract domain defines the kind of abstraction that is used to automatically construct the abstract model. The *precision* for an abstract domain defines the level of abstraction in the abstract model, for example, which predicates to track in predicate analysis [9], or which pointers to track in shape analysis [8]. Such precisions can be obtained automatically; interpolation is an example for a technique that extracts predicate precisions from infeasible error paths.

Precisions are a good choice for reuse in regression verification, because they are technically easy to use and do not require much extra computation effort before they can be reused, they have a small memory footprint, and they are, as we show, rather insensitive to changes in the program source code. We performed an extensive experimental study

[†]A preliminary version appeared as technical report [14].

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491429

Table 1: **Verification of driver `extcon-arizona` without and with precision reuse; time in seconds of CPU usage**



		Specification 1: 'Spinlocks lock/unlock'							Specification 2: 'Mutex lock/unlock'						
Rev.	Commit Message	Result	**Refinements**	with Reuse	**Abstractions**	with Reuse	**Analysis Time**	with Reuse	Result	**Refinements**	with Reuse	**Abstractions**	with Reuse	**Analysis Time**	with Reuse
3	Implement button detection support	safe	24	24	792	792	4.7	4.6	unsafe	8	8	38	38	1.1	1.0
4	Free MICDET IRQ on error during probe	safe	24	0	792	27	4.6	.99	unsafe	8	0	38	14	1.1	.85
5	fix typos in extcon-arizona	safe	24	0	792	27	4.7	1.0	unsafe	8	0	38	14	1.1	.86
6	Use bypass mode for MICVDD	safe	4	0	10	3	.86	.68	unsafe	1	0	3	2	.59	.58
7	Merge tag 'driver-core-3.6' of git://git.kernel.org/...	safe	24	0	792	27	4.6	1.0	unsafe	8	0	38	14	1.0	.84
8	**unlock mutex on error path in arizona_micdet()**	safe	24	0	792	27	4.5	1.0	**safe**	43	16	571	524	5.0	4.5
9	remove use of __devexit	safe	24	0	792	27	4.6	1.0	unsafe	8	0	38	22	1.1	1.2
10	remove use of __devinit	safe	24	0	792	27	4.6	1.0	unsafe	8	0	38	22	1.1	1.2
11	remove use of __devexit_p	safe	24	0	792	27	4.6	1.0	unsafe	8	0	38	22	1.1	1.2
12	Merge tag 'pull_req_20121122' of git://git.kernel.org/...	safe	24	0	792	27	4.6	1.0	safe	43	0	571	27	5.0	.97

on industrial code, in order to show the significant impact of precision reuse on regression verification (in terms of performance gains and increased number of solvable verification tasks). The benchmark verification tasks were extracted from the Linux kernel, which is an important application domain [15], and prepared for verification using the Linux Driver Verification toolkit (LDV) [29, 31]. Our study consisted of a total of 16 772 verification runs for 4 193 verification tasks, composed from a total of 1 119 revisions (spanning more than 5 years) of 62 Linux drivers from the Linux kernel repository.

Verifying large numbers of program revisions often takes several hours or even days. Our approach of precision reuse can speed this up by a factor greater than 4 (in terms of CPU time for the analysis) on average for predicate analysis.

Example. We consider ten revisions of the Linux device driver `extcon-arizona` for which a bug was discovered using formal verification by the LDV team[1]. Table 1 lists the revisions and the corresponding commit messages (in bold: the commit that fixes the above-mentioned bug). We verify two specifications with a CEGAR-based predicate analysis: (1) 'Spinlocks lock/unlock', and (2) 'Mutex lock/unlock'. Revisions 3 to 7 and 9 to 11 violate specification 2. Tasks that violate the specification generally need less refinements and abstraction computations, because the analysis terminates as soon as a bug is found. In cases where the specification holds, the whole state space of the program has to be analyzed; mostly a large number of refinements (> 20) and expensive abstraction computations (> 500) have to be performed.

The columns titled 'with Reuse' show the results with precision reuse. For cases where a complete reusable precision from a successful verification of a previous revision is not available (revision 3 for specification 1, revisions 3 to 8 for specification 2) because the whole state space was not yet analyzed before, there is no speedup. For most cases where a complete state-space analysis was done in a previous run (with result 'safe'), and thus a large precision is available for reuse, a speedup of 4 can be achieved (CPU time for analysis around 1 s instead of over 4 s). Refinements are eliminated completely because all necessary verification facts are already specified by the reused precision that is given as input.

Contributions. We make the following novel contributions:

- We identify the abstraction precisions as intermediate results that are valuable for reuse in regression checking.
- We define a tool-independent format for persistent storage and exchange of precisions.
- We extend the existing software-verification tool CPAchecker [11] in order to support regression verification with precision reuse.
- We prepare and consolidate a benchmark set for regression verification that is based on industrial source code from the Linux kernel and consists of thousands of benchmarks.
- In an extensive experimental study, we show that precision reuse leads to significant performance improvements and causes almost no overhead for the verification tool as well as for the benchmarking infrastructure (and thus, no additional barriers in a software-development process).

Related Work. The desire of constructing efficient tools for incremental formal verification exists since more than 15 years [25, 36]. In the literature, there are two main directions to approach the problem of regression verification: (1) based on analyzing the difference between the program and another program that was successfully verified in a previous verification run, and (2) based on reuse of intermediate results that were costly computed in previous verification runs.

Verification of Differences. The first group of approaches to efficient regression verification takes two programs as input and analyzes the differences in order to verify whether the specification is still fulfilled. An input condition can restrict the analysis to certain relevant parts of the state space [17, 22]. These approaches can be seen as conditional model checking [7], where the input condition instructs the verifier to perform a partial verification. The parts of the program that were identified as not being affected by modifications can be skipped [22, 33] during the verification process. A technique for proving conditional equivalence of two programs [18, 22] isolates and abstracts the functions of both versions using uninterpreted functions and then proves their equivalence. Checking the equivalence of two programs can be reduced to checking just those parts impacted by changes [1].

Reuse of Verification Results. The other group of approaches reuses state-space graphs [10, 27, 30], constraint solving results [38, 40], function summaries [35], or counterexample traces [16]. To ensure that the information is valid to be reused, those parts of the information that were affected by changes (to the analyzed program or its specification) have to be re-validated. The check for reusability is done either before the actual verification process is started [35, 39, 40] or immediately before certain information should be reused [27, 30]. Extreme model checking [27] is the only existing approach that uses unbounded model checking with lazy abstraction and predicate analysis for

[1]`https://patchwork.kernel.org/patch/1694901/`

regression verification. eVolCheck [35] is based on bounded model checking; the reuse is focused on function summaries, enhanced by a syntactic analysis of the differences between the program versions for improved performance. Another way of information reuse is to not store the concrete data, but its hash value. One such approach [25] stores hashes of verified models; these models are constructed by reducing a program to those parts that are relevant to prove a property. To be efficient, model construction must be less expensive than verifying the model. For formal regression verification of hardware using the ic3 algorithm, the reuse of correctness proofs and counterexamples has been proposed [19]. A more general fashion of reuse is to store and reuse canonicalized constraint-solver queries and the corresponding results [38].

Our approach belongs to the second category: we reuse abstraction precisions as intermediate results and do not (explicitly) analyze the differences in the program code (our approach implicitly spends more effort on changed parts). This is the first work that reuses abstraction precisions.

In general, it is possible to combine the reuse of different kinds of information, for example, state-space graphs, abstraction precisions, and constraint-solver results. Approaches for reusing verification results can also be preceded by (inexpensive syntactic) checks for differences between program versions for improved performance.

2. BACKGROUND

Abstract Reachability Graph. The class of analyses that we consider in our work is based on creating an abstract model of the program in form of an abstract reachability graph (ARG). An example for such an analysis is Blast [6]. The ARG is constructed iteratively by unrolling the control-flow automaton (CFA) of the program, creating an abstract successor state for the next location whenever the control flow passes through an edge of the CFA. The creation of abstract-successor states is usually over-approximating and guided by some form of precision that instructs the analysis which facts should be tracked and which facts should be omitted by abstraction. The abstract domain determines the characteristics of the precision. For example, if the abstract domain tracks information on program variables explicitly, then the set of relevant program variables to consider at a program location is a suitable precision for the analysis [13]. The precision in use should require the tracking of just enough information to prevent false alarms, while at the same time be as concise as possible for an efficient analysis.

Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR [21] is a well-established technique for automatically finding a suitable precision that matches the above criteria. Beginning with an initial coarse or even empty precision, the ARG is created based on this initial precision. If no state violating the specification is found, the program is proved safe. If a violation of the specification is found, the concrete path of this counterexample is analyzed for feasibility. If it is feasible, the program is unsafe and the analysis terminates. Otherwise the abstract model of the program was too coarse, so the precision needs to be refined to exclude this infeasible counterexample from future explorations. Depending on the abstract domain, the facts necessary to rule out this counterexample are extracted from the proof of infeasibility and added to the precision. Then the CEGAR loop is continued with this newly-refined precision.

Lazy Abstraction. The efficiency of CEGAR-based analyses can be increased by using lazy abstraction [28]. Instead of always restarting the analysis from scratch after an infeasible counterexample was found, the abstract model is refined in a "lazy" style. That is, during counterexample analysis the newly-learned facts that are extracted from the counterexample are only added where necessary. Then only those parts from the ARG that were computed with a too coarse precision are removed and scheduled for re-exploration. The remainder of the ARG, for example, a prefix of the current counterexample path, or other paths not related to the current counterexample, are kept and are neither removed nor re-explored. This does not only reduce unnecessary re-computations, but also reduces computation effort by lazily applying the new, stronger, precision only to those states of the ARG where it is needed. States on unrelated paths of the ARG are still computed with the old, weaker, and thus more efficient, precision. A further improvement is to use different precisions for each program location in order to track as little information as possible (and for example to erase information during path exploration when reaching a location after which the information is not needed any more).

Predicate Analysis. One technique which is used widely together with the above concepts is predicate abstraction [23]. Given the set X of program variables, and the set $\mathcal{P}$ of quantifier-free predicates over variables from X, the abstract domain here is the set of boolean combinations of predicates from $\mathcal{P}$. The precision π is a set of predicates from $\mathcal{P}$. When constructing the ARG, abstract successor states are created by computing either the cartesian or the boolean abstraction of the current state using the predicates from π with an SMT solver. Using Craig interpolation, predicates can be generated fully automatically from a proof of unsatisfiability for the formula representing a spurious counterexample [26].

The performance of predicate abstraction can be improved by adjustable-block encoding (ABE) [12]. This technique groups program statements into blocks and computes abstractions only at the end of each block instead of at all program locations. Furthermore, if control flow merges within a block, paths in the ARG are also merged so that sets of paths are considered instead of single program paths. When using ABE-Loops (which encodes loop-free parts of the program into blocks), abstractions will be computed only at loop-head locations. Thus, predicates will be relevant only at these locations, and no precision is used at all other locations.

Explicit-Value Analysis. Another domain that can utilize a precision is explicit-value analysis [13], which tracks the current value for each program variable explicitly. Within this analysis, an abstract state is represented as an abstract variable assignment $X \to \mathbb{Z} \cup \{\top, \bot\}$, where X denotes the set of program variables of a program. The value $\top$ represents a variable valuation that is unknown, e.g., due to an uninitialized variable; the value $\bot$ represents a variable valuation that is impossible. Abstract successor computations are performed by evaluating program operations and assigning the resulting values to the respective program variables in abstract variable assignments explicitly — in contrast to modeling them symbolically as in the predicate domain.

The precision for an abstract variable assignment is defined as a set π of variables from X, which is used to restrict an abstract variable assignment to variables from that precision π. For example, applying the precision $\pi = \{b\}$ to the abstract variable assignment $v = \{a \mapsto 4, b \mapsto 15\}$ would result in the abstract variable assignment $v^\pi = \{b \mapsto 15\}$. Experiments show that a variable that is relevant for one path, is often relevant on similar paths as well, and thus it is beneficial to add a newly-found relevant variable to the precision for all locations of the function in which it is relevant. This reduces the number of refinements, because similar paths can be eliminated often without further refinements.

3. PRECISION REUSE

Definitions. A *precision* is the information that an abstraction-based analysis uses to guide the abstraction computation for creating abstract states. Given an analysis, we write Π for the set of possible precisions, and π for one element thereof. The *empty precision* is the coarsest precision from Π (usually, this precision defines that all information is abstracted). The *union* of two precisions from Π is defined in the intuitive way. For example, for predicate abstraction, a precision is a set of predicates over program variables, and the union of two precisions is the union of the two sets of predicates.

In order to use lazy abstraction, which supports different precisions at different program locations, we define a *program precision* as a mapping $L \to \Pi$ from the set L of program locations to the set of precisions Π. The union of two program precisions p_1 and p_2 is the program precision that maps every location l to the union of $p_1(l)$ and $p_2(l)$.

Format for Program-Precision Files. In order to write and read precisions to and from persistent storage, we define a simple text-based file format that describes program precisions in a human-readable and tool-independent way. A formal definition of the format is given in a technical report [14]. The basic structure is the same for all analyses. The file starts with a header (the content depends on the analysis). After the header, an arbitrary number of sections follow, each consisting of one line of scope selectors, and an analysis-dependent precision. There are three kinds of scope selectors: the literal `*` (representing all program locations), the name of a function in the program (representing all locations inside this function), and the id of a program location (representing this single location). The precision given in a section is used at all locations represented by the specified scope selector. The effective precision for any given program location is the union over the precisions from all sections with a scope selector matching that location.

For explicit-value analysis, the header is empty and each precision is a list of variables that occur in the program. For predicate analysis, each precision is a list of predicates given in the syntax of the SMT-LIB 2 standard [3] (a standard for SMT-solver interfaces supported by state-of-the-art SMT solvers); the header contains a sequence of term declarations which may be referenced by the formulas in the sections of the precision file.

Example. Consider a C program that contains two variables `lock` and `x`, both of which are relevant for proving the safety of the program. Variable `lock` may be relevant at all locations, whereas variable `x` may be relevant only in the functions `main` and `f`. An example program-precision file for explicit-value analysis that encodes this information is given in Fig. 1 (left). An example for predicate analysis is shown in Fig. 1 (right), assuming that the model checker encodes these variables as real numbers, and the predicates $lock = 0$ and $x \leq 1$ are relevant.

```
*:
lock

main f:
x
```

```
(declare-fun |lock|() Real)
(declare-fun |x|() Real)
(define-fun t1() Bool (= |lock| 0))
(define-fun t2() Bool (<= |x| 1))

*:
(assert t1)

main f:
(assert t2)
```

Figure 1: Example program-precision files; left: explicit-value analysis; right: predicate analysis

Generating Program-Precision Files. In order to enable the reuse of precisions, we collect all program precisions that are created during the analysis (typically, there is one for each refinement step), and create the union over all these program precisions. This resulting program precision is written to a file. For each program location, the precision is written once with the id and once with the function name of the program location as scope selector (not written if empty).

Precision Reuse. In order to reuse a precision for the subsequent analysis of the same or a similar program, an initial program precision for the analysis is created by interpreting the contents of a previously stored program-precision file. There are three possibilities to construct such a precision. First, precisions can be function-scoped, such that a precision is created for each function of the program, by taking the union of all precisions labeled with the function name. The result is assigned to all locations of the respective function. Note that this will widen the scope of precisions (thus potentially leading to a more precise abstraction), and also loose precisions if functions are renamed. This precision assignment is insensitive to changes of the control-flow structure within functions of a program. Second, precisions can be location-scoped such that the location ids in the file are used to identify the locations to which the read precision is assigned in the resulting program precision. For all program locations that do not occur in the file, the empty precision is assigned. Note that location ids may change if the program code was changed, and thus, precisions get assigned to locations that do not semantically correspond to the original location in the previous program. Third, precisions can be global-scoped by taking the union of all precisions in the file and assigning the result to all locations of the program. This will not loose any precision from the previous analysis, but might apply precisions to locations where they are not necessary (and thus make the analysis more expensive). In any case, precision elements that reference removed program variables or functions and are thus irrelevant can be identified and ignored with a quick syntactical check.

After the creation of the initial program precision, the analysis is started as usual. No change to the analysis itself is necessary. If the provided precision is strong enough to prove the program safe, no further refinement effort will be needed. If the input precision contains only a part of the necessary precision to be tracked, spurious counterexamples will be detected and subsequent refinements will strengthen the precision. Note that even in this case the input precision likely reduces the effort by decreasing the number of necessary refinements. This process may be iterated by writing again the program precision that was further refined by the second analysis to file, and using this as the input for a further analysis, possibly on a newer version of the program.

Discussion. One significant effect of reusing precisions from a previous verification run is a reduced number of refinements. These are usually among the most expensive operations executed by a model checker (for example involving satisfiability checks and interpolation queries over formulas that represent sets of complete program paths from the entry point to the error state). The second significant effect is that fewer refinements lead to a reduced effort on (partial) ARG pruning and re-construction. This is especially important for analyses that perform expensive operations during this phase, for example in predicate analysis, which needs SMT-solver queries to compute abstractions. While the introduction of large-block encoding [5, 12] has reduced the number of such computations by only abstracting at loop-head locations and not at every program location, the need to use boolean abstraction still makes this costly.

Precision reuse is an elegant and conceptually simple approach, because it integrates naturally into the techniques that are used by many successful model checkers. These techniques can be applied as they exist without any change, to the first, initial, verification run (when no reusable information is present), and also to the subsequent re-verification runs. Furthermore, this makes precision reuse applicable not only for the two presented analyses, but also to any analysis and abstract domain that is based on CEGAR and incorporates an abstraction step that is guided by some form of precision. For example, precision reuse would extend naturally to the abstract domain of shape analysis [8].

Precision reuse is easy to implement in existing model checkers that are based on CEGAR and abstractions. Only the import and export of precisions before and after the actual analysis needs to be added. Complex algorithms, as required for comparing two revisions of a program and detecting similar and changed code, are not necessary in our approach. The format we defined is easy to parse and write, and could be supported by a variety of model checkers, thus even enabling the reuse of precisions across different tools.

Furthermore, precision reuse is also user-friendly: a user that is already familiar with using one model checker will not need to learn how to use new concepts or tools. Exporting precisions as part of the analysis result should be enabled by default in most tools, and thus the only necessary action by the user is to supply the previously written program-precision file as an additional input to the next verification run. Even if the user mistakenly specifies a wrong program-precision file as input, the analysis will still work correctly (the result will be valid), and only the performance might degrade slightly. In order to employ precision reuse, it is not necessary to have access to previous program revisions; the only information needed is the (small) generated program-precision file.

Applicability of Precisions. The precisions from the previous verification run can be applied to the program locations of the program's next revision using three strategies, which differ in how they widen the scope of the precisions. A location-scoped precision is applied at exactly those locations stated in this precision. For example, consider a precision that is relevant at locations 5 to 10 of a program. Now, a change is made to the program, and a statement that is unrelated to the safety of the program is introduced right after location 6. Thus, the previous locations 5 to 10 now correspond to the new locations 5, 6, 8–11. The previous precision is not applied to location 11 and the analysis would find an infeasible error path, thus needing at least one refinement to rediscover the missing facts. Function-scoped precisions are insensitive to such changes. Even changes due to cross-cutting concerns that affect code locally in many functions are expected to be verifiable without many further refinements. Changes to the call graph of the program, however, might still generate a similar need for refinements, for example, if code that is relevant to the safety of a program is moved to another function. Global-scoped precisions reduce this problem further, making refinements only necessary if code referenced by the precision is changed directly (for example, if variables are renamed).

We consider location-scoped precisions to be too sensitive to program-code revisions. Which of the other two strategies performs better depends on the class of program changes (e.g., whether heavy refactorings changing the functions of the program are common), and how expensive an unnecessarily fine precision is for the analysis. Often, the latter has less effect than one would intuitively consider. For example, specifying local variables from a function f in the precision of a function g has no effect because the local variables in f are out of scope in g. The policy of most projects is to create small commits with mostly local changes, thus, we expect function-scoped precisions to be most promising in practice.

4. EXPERIMENTAL EVALUATION

In order to evaluate the impact of precision reuse on the effectiveness and efficiency of regression verification, we performed an extensive experimental evaluation. We used industrial software for our experiments: in total, we prepared 4 193 verification tasks from 1 119 revisions of 62 device drivers from the Linux kernel. We started verification runs on all those problems with both an explicit-value analysis and a predicate analysis, each with and without precision reuse (a total of 16 772 runs). Our tool implementation, the C source code of the device drivers, and the full benchmark results are available on our supplementary web page: `http://www.sosy-lab.org/~dbeyer/cpa-reuse/`. During our experiments, we found an actual bug in the Linux kernel[2].

Implementation. Our implementation is based on the open-source verification framework CPAchecker[3] [11], which is available under the Apache 2.0 license. CPAchecker provides implementations of explicit-value analysis [13] and predicate analysis with ABE [12]. Both approaches are based on CEGAR and use a precision to define the level of abstraction. Thus we only had to add support for writing the program precision to file after a verification run, and reading a previously written program precision to be used as initial precision before a verification run. The format for persistent storage of program precisions is described in Sect. 3. Further changes to the verification tool were not necessary, in particular, the verification algorithm and the abstract domains were not changed. Our extension for precision reuse is integrated into the trunk of the project's source-code repository[4].

Verification Tasks. A verification task is a fully specified verification input, which is referred to by a triple that consists of the name of the driver, the specification that the driver has to satisfy, and the revision number from the repository.

[2]`https://patchwork.kernel.org/patch/2204681/`

[3]`http://cpachecker.sosy-lab.org`

[4]`https://svn.sosy-lab.org/software/cpachecker`

Preparation of an Industrial Benchmark for Regression Verification. We started our selection process by considering the verification tasks from the category 'DeviceDrivers64' of the 2nd Intl. Competition on Software Verification (SV-COMP'13) [4], which is a benchmark set that consists of 1 237 verification tasks. From this set of verification tasks, we selected those device drivers that met the following two criteria: (1) CPAchecker, in revision 7481, needed more than 20 s of CPU time to report either SAFE or UNSAFE and (2) at least one refinement was necessary to verify the driver. This selection process helped us to omit trivial tasks and those for which precisions are not needed.

In total, 62 device drivers from the SV-COMP'13 benchmarks fulfilled the criteria above. We extracted the sources for all available revisions of those drivers from the official Linux kernel repository[5]. Each of these device drivers consists of several header and source files, each having its own revision history. We considered all commits to all C source files of the device driver, in chronological order, starting with the revision in which the device driver was added to its directory in the kernel repository (if the driver resided in the "staging" area of the kernel before being accepted into the main area, these revisions were not considered). In order to obtain a linear history of changes we excluded commits that occurred on branches that were created during the development of a driver (the merge commits that reintegrated such branches are included, and thus no changes are lost). Our oldest revisions date back to the year 2007, and the latest revisions to the end of 2012.

In order to obtain verification tasks, we also need specifications. We used as specifications six different rules for correct Linux kernel core API usage (cf. Table 2). We composed each revision of the 62 selected drivers with each specification. The composition was done using the LDV toolkit[6] [29, 31] and consisted of: (1) adding a main function that simulates calls to the device driver from the Linux kernel core, (2) weaving in one of the six specifications (reducing the rule-based specification of the property into a reachability property by instrumenting the driver with a monitor automaton), and (3) combining all files that the device driver (in the particular revision) consists of, into a single file (using Cil pre-processing). The result of this composition process is a verification task that consists of a single verifiable C file, for each revision.

We omitted tasks where the specification is trivially satisfied, e.g., specification "Module get/put" for drivers that do not call the functions try_module_get() and module_put(). For evaluating the effect of our approach, we need to consider those verification tasks for which the precision needs to be fully discovered and where repeated application of the verifier yields deterministically the same precision. This is not the case for verification tasks with a known specification violation, because the analysis can terminate as soon as finding a counterexample, skipping parts of the state space. Of course, precision reuse is applicable in such cases as well (witnessed by the bug we found), but in our benchmarks the numbers would not be comparable. Therefore, we remove from our benchmark set all verification tasks with the expected result UNSAFE. The resulting benchmark set[7] for regression verification consists of a total of 4 193 industrial-strength verification tasks, which allows us to perform a significant experimental study. Its high quality and usefulness has been acknowledged by the ESEC/FSE Artifact Evaluation Committee.

[5] `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`

[6] `http://linuxtesting.org/project/ldv`

[7] `http://www.sosy-lab.org/~dbeyer/cpa-reuse/regression-benchmarks/`

Table 2: Considered specifications (LDV rules)[8]

Name	Description
08_1a	*Module get/put.* For each successful call to try_module_get() a corresponding call to module_put() that unblocks the module must exist.
32_1	*Mutex lock/unlock.* A less accurate implementation of specification 32_7a.
32_7a	*Mutex lock/unlock.* A mutex must not be acquired or released twice. A mutex must not be released without prior acquiring. Finally, all mutexes must be released.
39_7a	*Spinlocks lock/unlock.* A spin lock must not be acquired or released twice. A spin lock must not be released without prior acquiring. Finally, all spin locks must be released.
43_1a	*Memory allocation inside spinlocks.* The flag for atomic allocation operations must be used whenever a memory allocation function call is done while a spin lock is held.
68_1	*USB alloc/free urb.* For each allocation of an USB Request Block (URB) using usb_alloc_urb() a corresponding call to usb_free_urb() must exist.

Differences between Verification-Task Revisions. While normally source-code changes for the device drivers are rather limited from revision to revision, our benchmark set has quite large source-code differences between revisions, which is (not by design, rather as a side-effect) good to evaluate insensitivity to changes. We explain the main three reasons in the following: (1) Whenever commits occurred in branches, we did not include the corresponding revisions along the branch; instead, we extracted only revisions from the mainline branch. The revisions after a merge into the mainline branch result from a single (generally larger) commit. (2) Another reason for a large difference between revisions is the omission of revisions with a known specification violation. Thus, the changes from such revisions appear together with the changes of the next commit, in the succeeding revision without a specification violation. (3) Another cause for large differences is that we took one snapshot of the code for each revision in which one of the C source files of the device driver was changed. However, in the kernel project there are many other (header) files that influence the code of a particular file, by being included from the file, and by defining macros, types, inline functions etc., which are used in the code. Thus, the change between two revisions incorporates not only the changes to the C source files of the device driver, but also the changes to all other kernel (header) files since the last revision. The latter changes are sometimes even larger in size and effect than changes to the driver. For example, the introduction of the kernel feature `CONFIG_BRANCH_TRACER` (profiling of unlikely and likely branches in the code by code instrumentation) added several lines of auxiliary variables per `if` statement, and this additional code appears as new code in the next revision that was made for each driver after the feature was introduced. Our benchmark set of verification tasks has an average of 688 changed lines of source code between subsequent revisions. Our results, presented in the following, show that precision reuse is quite insensitive to such large differences between revisions.

[8] `http://linuxtesting.org/ldv/online?action=rules`

Table 3: Results for driver `dvb-usb-az6007` using predicate analysis; time in seconds of CPU usage

			without Precision Reuse			*with* Precision Reuse		
Spec.	n-th Rev.	Diff. Lines	Analysis Time	Refinements	Abstractions	Analysis Time	Refinements	Abstractions
08_1a	32	-	6.3	2	1352	6.1	2	1352
	33	593	6.0	2	1352	.78	0	24
	34	707	15	3	5971	7.2	1	683
	35	478	14	3	5971	3.2	0	156
	36	2	15	3	5971	3.2	0	156
		Total	56	13	20617	20	3	2371
32_7a	32	-	3.5	27	186	3.4	27	186
	33	752	3.8	28	210	1.6	1	48
	34	961	27	264	2351	18	22	977
	35	462	27	264	2351	5.8	0	156
	36	2	27	264	2351	5.9	0	156
		Total	88	847	7449	35	50	1523
39_7a	32	-	9.9	10	8432	9.5	10	8432
	33	595	10	10	8432	.89	0	24
	34	707	35	19	36659	15	9	3171
	35	462	35	19	36659	3.6	0	156
	36	2	35	19	36659	3.6	0	156
		Total	120	77	126841	33	19	11939

Setup. All experiments were performed on machines with a 3.4 GHz Quad Core CPU (Intel Core i7-2600) and 32 GB of RAM. We used Ubuntu 12.04 (64-bit) with Linux 3.2 and OpenJDK 1.7. We used CPAchecker, revision 8112. The predicate analysis uses MathSAT 5.2.5 as SMT solver. Each verification run was limited to 15 minutes of run-time and 15 GB of RAM; the Java heap size was limited to 10 GB. This is a similar environment to the community-agreed setting of SV-COMP'13. The analysis time that we report refers to the CPU time of the analysis phase of the verification tool (excluding startup and program parsing), and is given in seconds with two significant digits. The size of code differences between two revisions of one program is given as the number of differing lines excluding whitespace changes (calculated with `diff --ignore-all-space | diffstat`).

Results. We experiment with the reuse of precisions across a sequence of different revisions of a program. For this we start the verification of the first revision with the empty precision, save the generated precision and use it as the initial precision for the verification of the second revision. The final precision of the second verification run is used as input precision for the verification of the third revision and so on. We compare the time needed for this process against the time that is needed for verifying all revisions individually (using the empty precision as the input for each run and without generating program-precision files).

Results for a Single Driver. The results for a single driver (`dvb-usb-az6007`) from the Linux kernel are shown in Table 3. There are five revisions for this driver, and we show the verification of three specifications using predicate analysis. The column "Diff. Lines" shows the number of lines differing in one revision compared to the previous revision. The lines "Total" show the sum of the respective values for all revisions with one specification.

As expected, the run time for verifying the first revision is not decreased by the reuse of precisions (there is no precision to reuse); also, there is no significant overhead for writing the precision to the output file. For the remaining revisions, the run time results show a clear improvement of performance if the precision from the previous revision is reused. This is achieved by almost completely eliminating the need for refinements, and by lowering the number of (costly) boolean-abstraction computations considerably, compared to the verification of the same program without precision reuse. It is interesting to observe the third revision of this driver: This revision affected the program source in a way that made additional predicates necessary for all specifications (witnessed by the increase in the number of refinements). In such a case, the analysis with precision reuse also has to perform refinements, because these additional predicates are not yet known. However, those refinements that were necessary to discover predicates for the first revision, are not necessary, because the results are read from the precision file. Thus, the run time is still much better than without precision reuse.

Results for all Device Drivers and Specifications. Tables 4 and 5 show the results of verifying all revisions of all 62 device drivers against all appropriate specifications from Table 2, with predicate analysis and explicit-value analysis, respectively. Due to space reasons we restrict these tables to the 50 best and 10 worst cases out of the total 259 driver/specification pairs (sorted by column "Analysis Speedup"). We also show all cases where due to precision reuse, the analysis is able to verify more revisions. The complete tables are available on the supplementary webpage.

The columns "Analysis CPU Time" show the accumulated CPU time used by the analysis to verify all revisions of the device driver against the given specification (excluding revisions for which a timeout or an out-of-memory occurred). In addition, the columns "Total CPU Time" show the total CPU time used by the tool (including program-startup and parsing time). The columns "1st Rev." show the time needed for verifying only the first revision (this is the same with and without reuse). The column "Solved Tasks" shows the number of successfully verified revisions out of the total number of revisions for this driver (the remaining cases were either timeout or out-of-memory, there were no incorrect verification results). If the value in this column is of format "$N + M$", this means that without precision reuse only N revisions could be verified, whereas with precision reuse $N + M$ revisions could be verified; otherwise the number of successfully verified revisions is the same. There were no cases where a revision could be verified without reuse, but not with precision reuse. The column "Analysis Speedup" gives the average speedup for the task of verifying a single revision of the driver if a precision from a previous revision is reused in relation to the case where no information is reused (based on the analysis CPU time without program startup and parsing). The analysis time of the first revision of each driver is not taken into account for calculating the speedup, in order to make this value independent from the number of revisions per driver (otherwise a driver with more revisions would in general show a higher speedup because the cost of the verification of the first revision has less impact on the speedup). We also excluded from calculating the speedup such revisions that could not be verified by the configuration without reuse. In the last column, we report the (maximal) size in bytes of the final program-precision file that was produced during the verification of the revisions of this driver. Note that our file format is purely text-based, thus, this number gives a coarse over-approximation of the amount of information that is reused between verification

Table 4: Best and worst results for predicate analysis (details for highlighted line in Table 3)

Device Driver	Spec.	Tasks	Avg. Diff. Lines	**Refinements**		**Total CPU Time**			**Analysis CPU Time**			**Solved Tasks**	**Analysis Speedup**	Max. Size of Prec.
				no Reuse	Reuse	1st Rev.	no Reuse	Reuse	1st Rev.	no Reuse	Reuse			
leds-bd2802	43_1a	4	426	210	6	220	2000	250	210	2000	240	3+**1**	63	640
leds-bd2802	08_1a	14	504	960	8	200	3200	350	190	3200	320	14	24	471
mos7840	39_7a	57	621	35865	789	140	8600	780	140	8400	590	57	18	3307
dp83640	39_7a	16	557	2256	140	78	1500	220	74	1400	170	16	14	3516
farsync	08_1a	5	984	154	32	20	85	37	17	71	21	5	14	815
i2c-algo-pca	68_1	7	477	238	35	8.5	70	28	5.8	53	9.3	7	14	917
i915	39_7a	79	842	3472	72	140	3500	870	140	3000	370	79	12	3075
i2c-algo-pca	32_1	7	223	131	19	6.5	47	22	4.4	32	6.6	7	12	668
dmx3191d	08_1a	2	1432	20	10	52	110	59	49	99	53	2	11	514
vsxxxaa	68_1	2	1354	28	14	11	20	15	7.9	14	8.6	2	9.5	706
it87	43_1a	15	612	105	7	8.4	150	60	5.6	100	16	15	9.4	405
videobuf-vmalloc	68_1	18	490	232	14	5.8	110	57	3.0	60	9.8	18	8.4	750
dvb-usb-vp7045	68_1	2	1831	20	10	6.4	13	9.9	3.7	7.5	4.2	2	7.8	512
xilinx_uartps	39_7a	3	352	531	177	14	41	22	11	33	14	3	7.7	2248
mos7840	08_1a	60	795	722	15	39	2600	570	36	2400	370	60	7.3	889
arkfb	39_7a	22	447	1320	70	360	1500	580	360	1400	500	22	7.1	2322
vsxxxaa	32_1	2	755	14	7	8.3	16	11	5.8	11	6.4	2	7.0	474
i915	08_1a	79	731	1264	20	57	1900	670	53	1400	250	79	6.9	527
vsxxxaa	43_1a	2	786	11	5	6.4	14	9.2	4.1	9.0	4.8	2	6.8	1007
spcp8x5	39_7a	37	481	4701	348	30	920	250	27	810	140	37	6.8	1847
i2c-algo-pca	39_7a	14	367	236	17	5.6	77	45	3.0	46	9.4	14	6.6	810
cp210x	39_7a	71	256	424	34	850	18000	3600	850	18000	3500	33+**8**	6.4	2105
dvb-usb-rtl28xxu	39_7a	10	173	154	10	7.6	120	50	4.5	92	18	10	6.3	1820
it87	39_7a	54	462	1358	37	18	1900	470	16	1700	290	54	6.2	2091
sym53c500_cs	39_7a	19	468	1947	113	13	290	110	10	230	46	19	6.1	2634
mISDN_core	39_7a	59	1265	2674	38	13	930	470	6.4	560	99	59	6.0	2691
rtc-pcf2123	68_1	2	46	18	10	7.1	13	10	4.4	7.6	5.1	2	5.9	567
i915	32_7a	79	777	1184	24	6.1	1700	630	2.7	1300	230	79	5.8	1020
dmx3191d	39_7a	2	1597	104	57	140	280	170	130	270	160	2	5.6	3321
uartlite	39_7a	9	326	198	22	8.1	71	36	5.5	48	13	9	5.6	2151
budget-patch	39_7a	9	1669	205	27	7.4	74	41	4.2	42	11	9	5.5	2290
it87	32_7a	59	463	860	25	16	1600	480	13	1400	270	59	5.4	1696
sym53c500_cs	68_1	8	522	120	20	8.0	63	36	4.9	41	11	8	5.4	769
cp210x	32_1	14	219	1473	227	66	1200	310	62	1200	270	14	5.3	693
twidjoy	39_7a	2	1458	46	26	7.0	13	10	4.2	8.8	5.0	2	5.3	2159
mtdoops	68_1	20	345	157	15	5.2	93	63	2.4	46	11	20	5.3	511
cp210x	68_1	14	538	954	162	330	4300	1100	330	4200	1100	14	5.3	938
i2c-matroxfb	39_7a	7	617	120	15	3.2	36	21	.90	17	4.0	7	5.3	1426
sym53c500_cs	32_1	8	507	60	10	6.8	53	31	4.2	32	9.5	8	5.2	503
rtc-pcf2123	39_7a	9	769	138	15	6.3	54	31	4.0	33	9.6	9	5.2	1398
i915	68_1	79	354	900	18	23	970	540	20	550	120	79	5.1	825
wm831x-dcdc	32_7a	34	298	84	4	2.8	220	110	.56	140	28	34	5.1	762
dvb-usb-rtl28xxu	32_7a	10	173	58	4	6.2	90	46	3.2	59	15	10	4.9	724
metro-usb	39_7a	25	158	351	15	7.3	190	100	4.4	120	28	25	4.9	1417
dvb-usb-az6007	39_7a	5	353	77	19	13	140	50	9.5	120	33	5	4.9	1856
ar7part	68_1	2	677	16	8	3.2	6.9	6.0	.75	1.8	.97	2	4.9	409
spcp8x5	68_1	13	740	508	46	9.6	250	86	6.9	210	49	13	4.8	1385
ssu100	39_7a	28	337	791	44	11	390	160	8.2	310	72	28	4.8	2417
panasonic-laptop	68_1	4	775	47	32	9.0	24	19	6.5	13	8.0	4	4.8	854
metro-usb	32_7a	25	184	104	8	3.6	130	77	1.2	66	15	25	4.8	1115
⋮				For full results cf. `http://www.sosy-lab.org/~dbeyer/cpa-reuse/predicate.html`										⋮
cp210x	32_7a	71	257	600	20	43	5000	2000	39	4800	1800	56+**15**	2.7	1639
⋮														⋮
farsync	32_7a	9	889	0	0	4.0	48	46	1.3	21	20	9	1.0	2
slram	08_1a	9	563	60	6	3.3	31	31	1.2	11	11	9	1.0	490
cfag12864b	43_1a	2	74	0	0	2.4	4.7	4.8	.49	.97	.97	2	1.0	2
i2c-algo-pca	43_1a	7	478	0	0	2.4	19	19	.30	3.0	3.0	7	1.0	2
magellan	32_7a	2	1267	10	9	3.9	7.4	7.5	1.6	3.0	3.1	2	.93	1209
wl12xx_sdio	08_1a	38	258	38	2	3.2	120	130	.65	24	27	38	.87	579
mtdoops	08_1a	41	264	47	4	4.8	110	110	2.3	19	22	41	.86	539
wl12xx_sdio	32_7a	38	261	42	3	3.2	120	130	.64	24	27	38	.86	776
slram	32_7a	9	625	34	15	2.4	28	30	.28	8.1	10	9	.78	1578
mos7840	43_1a	25	1018	658	73	220	1700	2500	220	1700	2400	12+**6**	.66	3973
Sum		4 193		90 190	5 197	4 700	99 000	38 000	4 000	83 000	23 000	4 048+**30**		245 276
Average		16	688	361	21	19	390	150	16	330	90	16	4.3	981

runs. The highlighted row shows the driver `dvb-usb-az6007`, for which further details are available in Table 3 (the line here corresponds to the line labeled "Total" for specification `39_7a` in Table 3). The bottom rows report the sum and the average of the respective values per driver/specification pair.

Precision reuse not only increases the efficiency, but also the effectiveness: For five pairs of driver and specification, the number of successfully solved verification tasks was increased by our approach. This is possible if an early revision of a driver is verifiable, and a later revision results in a timeout. With precision reuse, the verification of the later revision is easier, because a large part of the precision is given as input; sometimes making it possible to successfully verify tasks that could not be verified before.

Table 5: Best and worst results for explicit-value analysis

Device Driver	Spec.	Tasks	Avg. Diff. Lines	**Refinements**		**Total CPU Time**			**Analysis CPU Time**			**Solved Tasks**	**Analysis Speedup**	Max. Size of Prec.
				no Reuse	Reuse	1st Rev.	no Reuse	Reuse	1st Rev.	no Reuse	Reuse			
cfag12864b	08_1a	4	326	344	86	510	2200	530	510	2100	520	4	210	374
cfag12864b	32_7a	4	369	246	82	540	1500	550	540	1500	550	3+**1**	210	405
cfag12864b	32_1	2	48	32	16	530	1000	530	530	990	530	2	180	267
com20020_cs	39_7a	2	524	18	9	4.6	9.7	7.4	1.9	3.8	2.1	2	10	304
mISDN_core	39_7a	59	1265	974	19	20	1400	480	14	1000	120	59	9.3	499
wl12xx_sdio	39_7a	38	266	372	11	4.8	190	120	2.2	85	12	38	8.8	352
slram	68_1	5	511	20	4	3.8	19	14	1.5	7.9	2.2	5	8.6	204
uartlite	39_7a	9	326	63	7	4.6	41	27	2.1	18	4.1	9	8.3	219
slram	08_1a	9	563	58	5	3.3	30	24	.88	11	2.2	9	7.4	193
sil164	39_7a	3	383	18	6	4.6	14	10	2.0	6.0	2.6	3	7.2	192
slram	39_7a	9	599	145	18	4.5	40	26	2.1	19	4.4	9	7.2	396
tcm_loop	39_7a	41	263	517	14	6.6	300	180	3.0	160	25	41	6.9	428
slram	32_1	5	450	15	3	3.1	17	13	1.0	5.7	1.7	5	6.9	181
i2c-matroxfb	39_7a	7	617	51	8	4.2	29	22	1.5	11	3.0	7	6.7	242
intel_vr_nor	39_7a	10	274	40	4	3.1	29	24	.90	7.7	1.9	10	6.7	141
mtdoops	39_7a	35	243	145	6	3.5	120	82	1.4	40	7.2	35	6.7	202
dvb-usb-rtl28xxu	39_7a	10	173	90	9	5.4	56	36	2.2	22	5.2	10	6.6	304
panasonic-laptop	39_7a	16	410	104	7	4.2	63	50	1.4	25	5.0	16	6.5	213
xilinx_uartps	39_7a	3	352	21	7	4.8	14	11	1.9	5.7	2.5	3	6.5	219
lms283gf05	39_7a	13	458	80	7	3.7	49	34	1.4	16	3.7	13	6.4	213
uio_sercos3	39_7a	5	897	29	8	3.7	18	13	1.5	6.3	2.3	5	6.2	254
cfag12864b	68_1	2	155	6	3	3.6	6.7	6.1	1.1	2.1	1.2	2	6.2	196
gpio-regulator	39_7a	20	193	80	4	3.2	70	46	1.0	21	4.4	20	6.0	141
dvb-usb-az6007	39_7a	5	353	45	9	5.7	33	20	2.8	18	5.4	5	6.0	304
i2o_scsi	39_7a	6	454	55	10	5.0	29	20	2.1	13	4.0	6	5.8	282
rtc-max6902	39_7a	8	890	44	7	4.3	29	24	1.7	10	3.2	8	5.8	213
metro-usb	39_7a	25	158	175	7	4.6	110	82	1.7	46	9.3	25	5.8	226
ems_usb	39_7a	21	666	199	10	5.6	120	74	2.5	55	12	21	5.8	325
keyspan_remote	39_7a	7	929	43	7	3.9	26	20	1.4	9.5	2.8	7	5.5	213
cx231xx-dvb	39_7a	13	577	127	10	6.0	89	51	3.1	45	11	13	5.5	325
dvb-usb-vp7045	39_7a	12	1001	110	11	4.9	62	42	2.1	28	6.9	12	5.4	359
budget-patch	39_7a	9	1669	98	13	4.4	45	31	1.7	19	5.0	9	5.3	421
pcc-cpufreq	39_7a	3	554	21	7	3.8	10	9.1	1.2	3.5	1.6	3	5.2	210
lms283gf05	32_7a	13	476	71	7	3.5	46	37	.99	15	3.8	13	5.2	228
budget-patch	43_1a	5	1239	20	4	5.5	29	18	2.9	14	5.1	5	5.1	178
dvb-usb-az6007	32_7a	5	435	940	99	7.9	59	28	5.1	43	13	5	5.1	869
keyspan_remote	43_1a	3	285	12	4	3.4	9.0	9.0	.79	2.4	1.1	3	5.1	261
rtc-pcf2123	39_7a	9	769	48	7	4.1	35	24	1.8	12	3.9	9	5.0	213
cp210x	39_7a	71	256	456	8	6.4	450	250	3.6	260	56	71	4.9	227
dp83640	39_7a	16	557	176	11	6.7	100	62	3.7	57	15	16	4.9	441
mISDN_core	68_1	26	2481	52	2	7.6	350	210	2.3	160	35	26	4.8	35
dvb-usb-vp7045	68_1	2	1831	4	2	3.8	7.7	7.0	.97	2.3	1.2	2	4.8	35
ssu100	39_7a	28	337	209	9	6.0	160	100	2.7	77	18	28	4.8	271
mt2266	39_7a	5	806	31	7	3.3	16	13	.95	4.9	1.8	5	4.7	213
ab8500-usb	39_7a	6	183	24	4	3.6	20	17	1.0	6.4	2.2	6	4.7	141
catc	39_7a	22	893	246	13	5.4	130	84	2.5	70	17	22	4.7	411
sym53c500_cs	39_7a	19	468	175	10	5.4	110	66	2.6	53	13	19	4.7	282
dvb-usb-rtl28xxu	32_7a	10	173	30	3	4.4	43	35	1.3	14	4.0	10	4.5	110
it87	39_7a	54	462	513	10	4.6	410	230	2.2	240	55	54	4.5	356
mtdoops	68_1	20	345	58	3	3.2	64	54	.89	19	4.9	20	4.5	206
⋮	For full results cf. `http://www.sosy-lab.org/~dbeyer/cpa-reuse/explicit.html`												⋮	
abyss	32_7a	4	2025	2	2	2.6	13	13	.51	3.5	3.5	4	1.0	72
twidjoy	32_7a	2	1268	2	2	2.3	4.6	4.5	.25	.45	.43	2	1.0	57
i915	43_1a	79	746	0	0	3.7	460	450	.34	36	36	79	1.0	3
dmx3191d	32_7a	2	1608	3	3	2.8	6.9	7.0	.17	1.6	1.6	2	.99	87
pt	32_7a	9	615	0	0	4.8	41	44	2.0	18	18	9	.99	3
farsync	32_7a	9	889	0	0	2.9	28	28	.21	2.9	3.0	9	.99	3
abyss	43_1a	3	1465	0	0	3.0	8.3	9.2	.52	1.5	1.6	3	.95	3
i2c-algo-pca	43_1a	7	478	0	0	2.4	16	18	.12	.91	1.0	7	.90	3
ar7part	43_1a	2	220	1	1	2.1	3.7	4.4	.04	.20	.23	2	.89	11
magellan	32_7a	2	1267	2	2	2.4	5.0	4.8	.27	.47	.51	2	.88	57
Sum		4 193		15 852	1 200	2 600	28 000	20 000	1 900	13 000	4 900	4 186+**1**		29 239
Average		16	688	62	5	10	110	78	7.3	52	19	16	3.8	113

The maximum analysis speedup is 63 for predicate analysis and 210 for explicit-value analysis, with average analysis speedups of 4.3 and 3.8, respectively. The time that the predicate analysis used for successfully verifying 4 048 verification tasks without precision reuse was 83 000 s, whereas with precision reuse, 4 078 verification tasks (30 more) were verified in only 23 000 s, less than a third of the time. The time used by the explicit-value analysis improved from 13 000 s to 4 900 s if our technique of precision reuse is applied to the 4 193 verification tasks. This gives evidence for the significant performance improvement of our approach.

We also list all negative results; there are only a few. The last lines of the tables report the few cases for which the verification with precision reuse takes a bit more time than without. Most of these cases have only a low average CPU time per revision of about 1 s. There is only one case for which the time for verification with precision reuse is significantly higher (the last line in Table 4, driver `mos7840` with

Table 6: **Results for considering all revisions versus considering only every 4th revision**

Analysis	Revs.	Tasks	Avg. Diff. Lines	**Total CPU Time**		**Analysis CPU Time**		**Analysis Speedup**	**Solved Tasks**
				no Reuse	Reuse	no Reuse	Reuse		
Predicate	All	4193	688	99000	38000	83000	23000	4.3	4048+30
	4th	1090	1579	24000	13000	21000	9200	3.0	1055+7
Explicit-Value	All	4193	688	28000	20000	13000	4900	3.8	4186+1
	4th	1090	1579	6300	5100	2200	1000	2.5	1090

specification `43_1a`). However, note that precision reuse increased the number of successfully solved tasks from 12 to 18 for this case. We generally consider an increase in the number of solved programs to be more important than a performance difference. A detailed analysis of the `mos7840` driver code revealed that a single variable is used as loop counter in three different loops. Some predicates about this variable are added to the precision, and due to the function-scoped precisions in our benchmark configuration are now applied to all three loops, the analysis becomes more expensive. The verification of the same driver against the other specifications actually shows nice speedups (e.g., line 3 of Table 4).

Size of Precision. The size of the precision that is necessary to be stored between verification runs is small: usually just a few kBs in our uncompressed plain-text format. The average size for predicate analysis is 1 kB (max: 4 kB); for explicit-value analysis it is 110 bytes (max: 1 kB). The total amount of precision storage that was necessary for verifying all 4 193 verification tasks was 250 kB for predicate analysis and 30 kB for explicit-value analysis, which is orders of magnitude less compared to the size of the source code.

Scaling with Larger Changes. The changes between subsequent revisions in our benchmark set are rather large (affecting 688 lines on average) compared to typical developer commits. To find out how our approach scales with the size of changes per revision (change-size sensitivity), we created verification problems with even more changes: we consider only every 4th revision per driver/specification pair as an alternative benchmark set. Thus, the difference between two revisions in this benchmark set combines the differences of four actual driver revisions into one.

Table 6 shows the results for this experiment in the lines that are marked "4th" in column "Revs." (the lines marked "All" show the results from Tables 4 and 5 for comparison). The average size of differences between revisions increased from 688 to 1579 lines. As expected, the speedup decreases, but only from 4.3 to 3.0 for predicate analysis, and from 3.8 to 2.5 for explicit-value analysis. In both cases the speedup decrease is significantly smaller than the increase in the size of the code differences. This shows that our approach copes well even with massive changes to the analyzed code.

Threats to Validity. To obtain significant experimental results, we created a large benchmark set consisting of 4 193 verification tasks. For highly credible experimental data, and instead of relying on random or artificial benchmarks, our selection of verification tasks is based on hundreds of actual source-code commits to 62 different Linux device drivers. The characteristics of systems software, in particular kernel device drivers, might be similar and could have an impact on the validity of our experiments, but (Linux) driver verification is important enough to be representative on its own [15]. The Linux Driver Verification program of the Linux Verification Center [29] and Microsoft's Static Driver Verifier project [2] dedicate considerable resources to driver verification.

We used an experimental setup and environment that is identical to the infrastructure for the competition on software verification (community-agreed). Precision reuse has a different impact on different abstract domains. We included two very different analysis approaches in our experimental evaluation: a symbolic and an explicit model-checking approach. Our experiments are not based on one particular specification, but on six different, real-world specifications, with all showing a considerable speedup.

It would be interesting to compare precision reuse to other approaches for regression verification. Our implementation does not analyze for syntactical differences, in order to be able to identify the speedup that is achievable by precision reuse in isolation (such optimizations are orthogonal and could be used in combination with precision reuse as well).

5. CONCLUSION

Precisions, defining the abstraction level of an abstract model, are costly to compute and represent precious intermediate verification results. We propose to treat these abstraction precisions as reusable verification facts, because precisions are easy to extract from model checkers that automatically construct an abstract model of the program (e.g., CEGAR), have a small memory footprint, are tool-independent, and are easy to use for regression verification.

The technical insight of our approach is that reusing precisions drastically reduces the number of refinements. The effort spent on analyzing spurious counterexamples and re-exploring the abstract state space in search for a suitable abstract model is significantly reduced.

To confirm the effectiveness and efficiency of our approach, we derived an extensive collection of verification tasks from the Linux kernel, in order to enable a meaningful evaluation of regression-verification techniques. Our benchmark set consists of 4 193 single verification tasks and is publicly available for download on our supplementary web page.

Based on these benchmarks, we show that the reuse of precisions has a significant effect on the verification process. The approach drastically improves the performance on most verification problems, and does not have noticeable negative side effects. Besides improving the run time, we are also able to solve verification problems that were not solvable before, within the given time and memory limits.

Because the information that we reuse does not depend on source-code details, our approach is less sensitive to changes in the source code, compared to other approaches.

Precision reuse is applicable to all verification approaches that are based on abstraction and automatically computing the precision of the abstract model (including, e.g., CEGAR-based approaches). Both the efficiency and effectiveness of such approaches can be increased by reusing the precision.

As a result of the experiments for this paper, a previously unknown bug in the Linux kernel was discovered and a fix was submitted to the maintainers by the LDV team[2].

6. REFERENCES

[1] J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *Proc. SPIN*, LNCS 7976, pp. 99–116. Springer, 2013.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pp. 1–3. ACM, 2002.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. SMT*, 2010.

[4] D. Beyer. Second competition on software verification (Summary of SV-COMP 2013). In *Proc. TACAS*, LNCS 7795, pp. 594–609. Springer, 2013.

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

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

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

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Proc. CAV*, LNCS 4144, pp. 532–546. Springer, 2006.

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

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

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

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

[13] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pp. 146–162. Springer, 2013.

[14] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Reusing precisions for efficient regression verification. Technical Report MIP-1302, University of Passau, 2013. http://arxiv.org/abs/1305.6915.

[15] D. Beyer and A. K. Petrenko. Linux driver verification. In *Proc. ISoLA*, LNCS 7610, pp. 1–6. Springer, 2012.

[16] D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS 7976, pp. 1–17. Springer, 2013.

[17] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *Proc. ICSE*. IEEE, 2013.

[18] S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *Proc. VMCAI*, pp. 119–135. Springer, 2012.

[19] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *Proc. FMCAD*, pp. 135–143. FMCAD, 2011.

[20] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *Proc. FM*, pp. 132–146, 2012.

[21] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[22] B. Godlin and O. Strichman. Regression verification: Proving the equivalence of similar programs. *Software Testing, Verification, and Reliability*, 2009.

[23] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pp. 72–83. Springer, 1997.

[24] T. L. Graves, M. J. Harrold, J.-M. Kim, A. A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proc. ICSE*, pp. 188–197. IEEE, 1998.

[25] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane. Efficient regression verification. In *Proc. WODES*, pp. 147–150, 1996.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pp. 232–244. ACM, 2004.

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

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

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

[30] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. ICSE*, pp. 291–300. ACM, 2008.

[31] M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, and P. E. Shved. Using Linux device drivers for static verification tools benchmarking. *Programming and Comp. Softw.*, 38(5):245–256, 2012.

[32] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[33] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. *ACM SIGPLAN Notices*, 46(6):504–515, 2011.

[34] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, 1996.

[35] O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *Proc. FMCAD*, pp. 114–121. FMCAD, 2012.

[36] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proc. CAV*, LNCS 818, pp. 351–363. Springer, 1994.

[37] O. Strichman and B. Godlin. Regression verification — a practical way to verify programs. In *Proc. Verified Software: Theories, Tools, Experiments*, pp. 496–501. Springer, 2008.

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

[39] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *ICSM*, pp. 115–124. IEEE, 2009.

[40] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *ISSTA*, pp. 144–154. 2012.