Block-Abstraction Memoization for Symbolic Memory Graphs

Oleg Petrov
Ivannikov Institute for System Programming of the Russian Academy of Sciences
Alexander Solzhenitsyn st., 25, Moscow, 109004, Russian Federation
o.petrov@ispras.ru

***Abstract*—This paper considers program analysis based on symbolic memory graphs (SMG) and Block-Abstraction Memoization (BAM) technique which enables procedure summarization. Both SMG and BAM are implemented in static software verification framework CPAchecker.**

SMG analysis models program memory as a graph with edges between objects in memory and symbolic values. Using it, violations of memory safety can be found in real-world programs such as Linux drivers.

With BAM, any given analysis can make a summary of a block (a function or a loop) and reuse the block summary when it enters a block with similar enough context.

This paper enables SMG to work with BAM. Several ways to distinguish context are introduced and compared against each other and baseline SMG analysis without BAM. We hope to improve efficiency of the analysis using BAM in future work.

***Index Terms*—formal software verification, software model checking, CPAchecker, symbolic memory graph, block abstraction, procedure summarization**

I. Introduction

Bug finding is a crucial stage in the development of widely adopted software, and it often cannot be performed manually due to the ever-growing scale and complexity of software. Testing may be the most popular approach to discovering problems in software due to its efficiency, but critical settings (such as operating systems) require a more thorough approach.

Software verification methods aim to either find a violation of a given formal specification or prove the correctness of the analyzed source or binary code against the specification. Although thorough, these methods can be memory and time-intensive or even undecidable. Other static methods, such as data flow analysis, make program analysis more efficient at the cost of precision, which can result in a high number of false positives and require additional manual effort to inspect the results.

CPAchecker[1] is a static software verification framework built on the concept of configurable program analysis (CPA) [1] which combines the efficiency of data flow analysis with the effectiveness of model checking and allows configuring the trade-off between the two [2]. Analyses with various abstract domains, including explicit values and predicates, can be easily introduced and discover the program's state space simultaneously. The tool has consistently won medals at the annual Competition on Software Verification (SV-COMP)[2], including a silver medal in the summary category *Overall* last year [3], [4].

CPAchecker includes a shape analysis with a symbolic memory graph (SMG) domain [5], [6]. Using it, CPAchecker has won medals in the *MemSafety* category at SV-COMP and has detected several hundred memory safety violations in the source code of the Linux operating system device drivers[3] [7], [8].

One popular technique for improving program analysis scalability is function summarization. CPAchecker incorporates Block-Abstraction Memoization (BAM), an example of this approach. BAM has been shown to be competitive with other software verification approaches while being applicable to different domains [9], [10]. We have implemented BAM operators for the SMG domain, allowing them to be used in conjunction.

The outline of this paper is as follows. Section II provides a motivating example. Section III offers an overview of several tools for memory safety static verification. Section IV details both the SMG analysis and the BAM technique. Section V presents several variants of the reduce (capture block context) and expand (apply block summary) operators needed to make the SMG domain compatible with BAM. In section VI, the variants and the baseline SMG-LDV analysis are evaluated on SMG integration tests and 845 Linux driver modules. While up to 30 verdicts were lost, the analysis with BAM was able to obtain over 100 new verdicts. Finally, the results are discussed.

II. Motivation

We want the verification tool to avoid reanalyzing a procedure call if it leads to the same result as a previously analyzed call. Although many calls depend on context, we can still use this approach if the contexts are similar enough to yield identical analysis results.

Consider the following example in Fig. 1. The program allocates the double array `matrix` in two steps: 1) an array of pointers to rows is allocated in the function `main`; 2) the call to the function `bad_row_of_integers` allocates an array of integers for each row in the loop in `main`. However, `bad_row_of_integers` also calls the function

[1] https://cpachecker.sosy-lab.org; https://gitlab.ispras.ru/verification/cpachecker

[2] https://sv-comp.sosy-lab.org/

[3] http://linuxtesting.org/ldv

```
1  #define N 9
2
3  void waste_analysis_time() {
4    char ***p3 = malloc(N * sizeof(char**));
5    for (int i = 0; i < N; i++) {
6      char **p2 = malloc(N * sizeof(char*));
7      p3[i] = p2;
8      for (int j = 0; j < N; j++) {
9        char *p1 = malloc(N * sizeof(char));
10       p3[i][j] = p1;
11       for (int k = 0; k < N; k++) {
12         char p0 = i + j + k;
13         p3[i][j][k] = p0;
14       }
15       free(p1);
16     }
17     free(p2);
18   }
19   free(p3);
20 }
21
22 int *bad_row_of_integers() {
23   waste_analysis_time();
24   return malloc(N * sizeof(int));
25 }
26
27 int main() {
28   int **matrix = malloc(N * sizeof(int *));
29   for (int i = 0; i < N; i++)
30     matrix[i] = bad_row_of_integers();
31   return 0;
32 }
```

Fig. 1. An example of a program wasting the time of the analysis

`waste_analysis_time` which allocates, writes to, and deallocates a triple array **`char`** `***p3`. The function is correct but useless code, and the SMG analysis reanalyzes it every time `bad_row_of_integers` is called. As the analysis unrolls loops, it exceeds the 15-minute CPU time limit.

With BAM, the SMG analysis is able to traverse the functions `waste_analysis_time` and `bad_row_of_integers` once, and then use the latter summary for the remaining eight calls. In total, CPAchecker takes less than 20 s to detect the memory leak and produce the verdict *unsafe*. The use of BAM for loop bodies instead of unrolling is left for future work.

III. Related Work

This section discusses four automatic software verifiers, including CPAchecker, and their techniques for efficiently verifying program memory safety. These tools have won various categories in recent years at SV-COMP, including overall, memory safety, and software systems.

One of the common problems in program analysis is pointer behavior. A pointer analysis aims to statically determine the possible runtime values of a pointer [11]. A shape analysis is a more specialized and typically more expensive form of a pointer analysis that attempts to determine the behavior of a program that manipulates complex dynamic data structures.

A shape analysis based on symbolic memory graphs was introduced in the Predator[4] tool in 2013 to automatically verify C programs that use lists and low-level memory operations [5]. PredatorHP [12] and the SMG CPA in CPAchecker both abstract sequences of singly or doubly linked memory regions into appropriate kinds of list segments. SMG was also extended in CPAchecker to track explicit values and simple relations (e.g. $sym_1 < sym_2$) in the form of predicates [6]. Predator, on the other hand, uses bounded intervals for values, including offsets and memory region sizes.

Predator verifier aims to verify the program using expensive SMG analysis; to improve efficiency, *Predator hunters*[5] run in parallel, searching for violations both in depth and in breadth without using more expensive features of the SMG analysis.

Contrary to the sophisticated shape analysis, the Symbiotic tool[6] uses lightweight pointer analysis, instrumentation and static slicing to find reduced traces of possible violations and then uses symbolic execution to confirm them [13]. To find memory safety violations, it employs instrumentation, but optimizes the placement of checks using pointer analysis. The tool uses a data flow analysis with the values `unknown`, `null`, and `invalidated`, the latter of which is used to track when a pointer becomes invalid, as opposed to a typical pointer analysis.

One of the well-known model checking approaches is counterexample guided abstraction refinement (CEGAR) [14]. A tool constructs a coarse abstract model of the program and then increases its precision – refines it – each time a spurious error path – a *counterexample* – is found. The precision of the model can be represented by tracked variables for a value analysis, or tracked predicates for a predicate analysis. The resulting abstract model is a proof of program correctness if it has no error states, or provides a violation witness if it has a feasible path to an error state.

Summarization is a commonly used approach in static analysis. It can be used for efficiency and scalability of analysis or to allow analysis of a program with recursive calls. Block-Abstraction Memoization can be viewed as a generalization of several block-based summarization approaches, including context inference and function summarization [10]. It was first introduced in CPAchecker in conjunction with predicate analysis in 2012 [9].

In principle, the technique is not limited to any specific analysis domain, and later it was implemented to work with any compatible CPA, e.g. explicit value analysis and its combination with predicate analysis. The technique itself has also been improved to analyze programs with recursive calls and to enable analysis of blocks in parallel.

The different combinations of the technique with value and predicate analyses (e.g. using region-based memory model to

[4] https://www.fit.vutbr.cz/research/groups/verifit/tools/predatorhp/
[5] hence the name PredatorHP: Predator Hunter Party
[6] https://github.com/staticafi/symbiotic

simplify predicates [15]) were successfully used for reachability and memory safety tasks.

Since Symbiotic does not create an abstract model, it does not use CEGAR. While Predator does use abstractions, it does not use CEGAR. Symbiotic uses imprecise pointer analysis, which gives already summarized information, and Predator uses function summaries.

CPAchecker employs CEGAR for several CPAs, with CEGAR and BAM capable to work together [16]. However, SMG CPA has not been used in conjunction with BAM. The SMG CPA has never been adapted to work with CEGAR, reinforcing the need for other methods to improve the analysis scalability.

IV. Overview

This section gives an overview of two key topics: (a) the SMG CPA, specifically the configuration `-smg-ldv` which serves as the baseline, and (b) the BAM technique ready for use with compatible CPAs.

A. Symbolic Memory Graphs

A symbolic memory graph is a bipartite graph with edges between objects in memory and symbolic values. A *has-value* edge maps an object, offset, and size to a specific value in that memory, while a *points-to* edge maps a value to an object at that address. As memory objects can be abstract, a *points-to* edge can have a tag specifier in addition to a regular offset.

The SMG for the example program in Fig. 1 after all allocations in the `main` function is as follows. A stack variable `matrix` has a has-value edge leading to a symbolic value. This value has a points-to edge leading to a heap object that represents the array of pointers allocated on line 28. Nine edges leave from this object to different symbolic values, each of which points to a different heap object that represents a row allocated inside the loop by the function `bad_row_of_integers`. Since the elements of the `matrix` have not been initialized, there are no has-value edges leaving from any of them.

One of the challenges in software verification is the complexity of data structures, such as singly and doubly linked lists. To address this issue, the considered analysis identifies lists within heap objects and abstracts (collapses) their tail or middle elements. This enables the states with longer lists to be covered by an existing abstract state, which in turn facilitates real-world software bug detection.

The analysis also collects information on the symbolic values present in the graph, including their explicit values and some relations.

B. Block-Abstraction Memoization

Block-abstraction memoization uses a given analysis to create a summary of a procedure or loop block. When the analysis enters a block, BAM *reduces* the entry state (the abstract state before entering the block) to include only the context important for analyzing that block. The underlying analysis then explores the block starting from the reduced entry state and producing the exit states, i.e. the states before leaving the block. These exit states are the block summary, and are stored in a cache using the reduced entry state as the key. To continue the analysis, the exit states are applied to the full entry state using the expand BAM operator.

When the underlying analysis re-enters the same block with a sufficiently similar context, the reduced entry state is found in the cache. The retrieved summary is applied to the new entry state, i.e. the old exit states are each *expanded* using the new full entry state. Obviously, the more effective the reduce operator, the higher the cache hit rate is and the more efficient the analysis using BAM is. However, a stronger reduce operator may lead to a less precise analysis.

To use Block-Abstraction Memoization with the SMG analysis, one needs to define two operators.

- $\mathsf{reduce}(e, B, n, V)$ returns the reduced block entry state, i.e. for a given block B, its entry node n, and the variables V referenced in it, the operator abstracts the entry state e from information unnecessary to analyze the block. The result is the context necessary for the analysis to create a block summary. The entry node n is provided as BAM supports block partitioning not limited to function blocks, and e.g. loop blocks can have multiple entries.
- $\mathsf{expand}(e, B, r)$ returns the expanded block exit state. It applies the exit state r to the given full entry state e for the given block B. To obtain the block exit state r, the underlying analysis is run as usual, starting from the reduced entry state. This may produce multiple exit states r, as there can be multiple abstract states at the block exit. In this case, expand is applied to the entry state e for each of the exit states.

BAM can enable analysis of recursive procedures, but this is left for future work.

V. Implementation

This section presents the issues involved in making the SMG analysis compatible with the BAM technique. It presents several variants of the BAM operators, i.e. different approaches to producing the block context and applying the block summary. It then discusses the impact of these variants on the analysis.

A. SMG implementation details

The abstract state includes a symbolic memory graph itself and the value information collected. The graph contains its objects, values, has-value edges leading from objects to values and points-to edges in the opposite direction. The value information consists of a not-equals relation between the values, a mapping from the symbolic values to their explicit values, a path predicate, and an error predicate. As a full-scale predicate analysis was found to be too expensive, the predicates consist of inequality relations between the values (such as $a < b$ and $a \geqslant 1$) [6].

SMG contains global objects (including auxiliary objects that are introduced by analysis itself), call stack and stack objects (variables and return object where necessary) allocated in respective frames, heap objects (that can be a number of

abstracted memory objects aside from a concrete region) and separate collections for valid and externally allocated objects.

The objects in an SMG are divided into three categories: global objects, stack objects, and heap objects. Global objects include global variables from the program and auxiliary objects introduced by the analysis itself. Stack objects are stack variables (and return objects) organized as a call stack. Heap objects include dynamic memory, both concrete regions and abstracted objects. In addition, SMG maintains a record of which objects are valid and externally allocated.

The context sufficient for a sound and precise analysis of the block comprises 1) subgraphs reachable from the memory objects that correspond to the variables referenced in the block; 2) the value information for all symbolic values in these subgraphs; and 3) information on the edges that point to these subgraphs from the remaining graph.

The value information is necessary to keep the analysis precise. The edges to the subgraphs are necessary for correct expand and to prevent the heap objects from being considered as leaked incorrectly. The described subgraphs may constitute a context of a larger size than necessary, but we leave this issue for future work.

B. Reduce

The reduction of a symbolic memory graph with respect to the given variables is a relatively straightforward process. Every global or top stack frame variable that is referenced in the block is retained together with the respective symbolic memory subgraph, including all edges, values, and objects that can be reached transitively from the variable.

With regard to the edges entering the retained subgraphs, any value pointing to a retained object is retained with the respective points-to edge. An auxiliary global object is added with a has-value edge to such a value. While some types of auxiliary objects (such as string literals) have to be retained for the analysis, these out-of-block reference objects should not be propagated to inner blocks.

Each pointer dereference, array access, and field access is represented in referenced variables by both the full expression and the comprising variables. The elaborate evaluation of the referenced objects using the full expressions to capture smaller subgraphs is a topic that will be addressed in future work. Such evaluation also appears to be necessary to use BAM instead of loop unrolling.

a) Call stack reduction: As the SMG implementation incorporates a call stack with stack objects allocated in the respective frames, there may be frames that are either originally empty or are emptied as a result of the reduction. Such frames may cause a cache miss, i.e. the entry states with different call stack prefixes will be considered different even if these prefixes consist of empty frames. Thus, we want to retain only those frames containing objects from the retained subgraphs. However, section VI shows that the removal of superfluous frames has a minimal impact on the analysis with BAM.

b) Value relation reduction: Another challenge is the reduction of value information. In the seminal work, SMG stores a not-equals relation to distinguish between non-zero values and values that may be zero. In CPAchecker, this has been extended with explicit values and simple predicates. An abstract state has a mapping of symbolic values to their explicit values and collects inequality relations (such as $a < b$ and $a \geqslant 1$) between symbolic values and between symbolic and explicit values.

The most basic approach is to keep the value information as is, exploiting the fact that most of the entry states do not have such information. A less straightforward option would be to eliminate trivial relations such as $a \leqslant a$ or $1 < 10$, but this is to be implemented. Discarding predicates with values not included in the context appears to have only a limited impact.

An opposing approach is to discard all value information, which results in a significant increase in the number of false positives. As SMG analysis is highly dependent on explicit value extension, it may be sufficient to retain the symbolic-to-explicit mapping while discarding predicates and not-equals.

The most interesting option is to produce canonized reduced value information. For example, $0 < a < b < c < d$ is the full path predicate but only 0 and b are present in the reduced SMG. Therefore, $b < c$ and $c < d$ are not necessary, while $0 < a$ and $a < b$ can be replaced with $0 \leqslant_2 b$ with weighted (distance) operator $\leqslant_d$, where d stands for required difference. This remains a topic for future work.

c) Metasymbolic values: A procedure can be entered with an SMG that differs from an already analyzed one only in symbolic values. This suggests the use of some kind of metasymbolic values to match to the previous results of analysis. As has-value edges are ordered by object and offset and contain all symbolic values, the order in the edges can be used in place of the value itself.

This entails according replacements in the value information. We expect metasymbolic values to work better with canonizing predicate reduction described above. Currently, the path predicate and not-equals contain symbolic values. Consequently, if this information is not discarded, the hash is distinct for states with different symbolic values.

d) Sophisticated SMG reduction: While the reductions described above are relatively straightforward and have been implemented and evaluated (except canonizing value information reduction), there is potential for a more sophisticated approach. We can consider isomorphic memory graphs (with additional conditions, e.g. matching object sizes) to be the same, i.e. reduce a concrete memory region to its size. This approach appears promising, but is left for future work.

C. Expand

Similarly to the reduction, objects are copied along with the respective subgraphs from the exit state r into the entry state e. Each valid object in the block summary is added to the block entry state, with their has-value edges updated where necessary. This way, the analysis does not miss a memory leak. If an object valid before the block is invalidated within the block, it is also invalidated in the resulting SMG.

Consequently, the analysis does not fail to identify an invalid dereference or double free.

Updating the subgraphs of global objects and top stack frame variables would be sufficient, but currently, stack variables can be marked as out-of-scope (and thus removed from the frame) before the call to expand. Because of this, subgraphs of stack variables from all frames are updated.

However, for procedures allocating new memory, it is not sufficient to simply copy the new memory objects each time the summary is applied to an entry state. To simulate the allocation, the new memory objects are copied and new symbolic values are introduced to point to the copies. Every time the block summary is applied to an entry state, new values and copies are used rather than the original ones.

D. Soundness and Precision

The idealistic goal of a software verification tool is to be sound, that is, to never produce false negatives. However, in practice, the complexity of the C language and programs in general leads to unsound shortcuts [17]. Given sound reduce and expand, BAM itself is sound, but can make analysis less precise than the underlying analysis alone [10]. Among the variants presented above, neither full stack nor reduced stack affects the soundness or precision of the analysis. In contrast, dropping value information makes analysis imprecise for a higher BAM cache hit rate.

VI. Evaluation

We evaluated the baseline SMG analysis alone and in conjunction with several BAM operator variants described above. The evaluation was conducted on two sets of programs: a subset of CPAchecker integration tests for a memory safety analysis and a subset of Linux 5.10.120 driver modules. We aim to improve the efficiency of the analysis, i.e. to reduce the verification time, while maintaining the number of verdicts produced.

The evaluated BAM operator variants include 6 variations, defined by the reduction of the call stack and value information: the reduce operator either keeps the call stack as is or removes the empty frames; the operator either keeps the value information as is, keeps only the explicit values, or discards all the value information. The baseline analysis configuration was used as the underlying analysis for BAM.

The integration test subset (536 programs) consists of programs from the Competition on Software Verification benchmarks[7], which are used for general SMG regression testing. Additionally, 41 simple programs have been specifically created to test the implementation of the reduce and expand operators. For each program, an expected verdict is provided. The set comprises smaller programs, with 108 programs (20%) having only 1 function, 277 programs (52%) having 2–6 functions, 131 programs (24%) having 7–29 functions, and other 16 programs (3%) having more than 60 functions.

The Linux driver set comprises USB and NET drivers (845 modules) for the 5.10.120 Linux operating system kernel, without known verdicts. Typical modules in this set consist of 50–300 functions. The Linux driver modules were prepared for the verification using the Klever framework[8] [18]. While not definitive, the verdicts provided by Klever are used to divide the driver set into three subsets: 374 “safe” modules (up to 700 functions), 134 “unsafe” modules (up to 1400 functions), and 337 “unknown” modules (up to 1900 functions).

TABLE I
The Produced Verdicts and Cumulative Resources for 536 integration tests

		keep all		keep expl.		discard all	
	baseline	full	red.	full	red.	full	red.
correct *safe*	208	206	205	205	205	140	140
correct *unsafe*	191	188	188	186	186	156	156
incorrect *safe*	12	12	12	12	12	9	9
inc. *unsafe*	41	43	43	47	47	156	156
timeout	59	59	60	58	58	49	49
recursion	6	6	6	6	6	3	3
unrec. code	12	17	17	17	17	17	17
exception	7	5	5	5	5	6	6
CPU time, min	108	126	126	125	125	113	113
memory, GB	113	114	114	114	114	111	112
analysis, min	–	44	44	43	43	31	31
reduce, min	–	0.8	0.8	1.2	1.2	0.5	0.5
expand, min	–	0.7	0.6	0.6	0.6	0.4	0.4

The evaluation was conducted using the BenchExec framework[9] [19] on a 64-bit machine with an 11th Gen Intel Core i7-11700 @ 2.50GHz 16-core CPU, 32 GB RAM and the Linux 20.04.6 LTS operating system. CPAchecker was constrained to one core and a CPU time limit of 1 minute for the integration subset and 2 minute for the drivers.

The implementation in the CPAchecker framework is available at the ISP RAS fork of the tool[10]. The driver modules, tables with detailed results, and instructions for replication are available in the reproduction package [20].

Table I shows the verdicts obtained and resources used during the integration tests verification. The columns labeled “full” correspond to the BAM operator variants that do not reduce the stack. The columns labeled “red.” (i.e. “reduced”) correspond to the variants that remove empty stack frames. The columns labeled “keep all”, “keep expl.”, and “discard all” correspond to the operator variants that keep all the value information, keep only the explicit values (i.e. remove the path predicates and not-equals relation), and remove all the value information, respectively.

As the verdicts are known for this set, the obtained verdicts are marked as correct or incorrect. For 3–6 programs in the row “recursion”, the SMG analysis encountered a recursive function call. For 12–17 programs in the row “unrecognized code”, the SMG analysis does not recognize a source code feature, e.g. a variable-length array (VLA). For 5–7 programs in the row “exception”, a bug in CPAchecker occurred. The

[7] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[8] https://forge.ispras.ru/projects/klever

[9] https://gitlab.com/sosy-lab/software/benchexec

[10] https://gitlab.ispras.ru/verification/cpachecker/-/commit/6bf750a3066c57206f55e1edbe8b27bd7a7a98c7

TABLE II
THE PRODUCED VERDICTS AND CUMULATIVE RESOURCES FOR 374 DRIVERS WITH BASELINE VERDICT *safe*

	baseline	keep all full	keep all red.	keep expl. full	keep expl. red.
safe	357	365	366	356	358
unsafe	0	3	3	3	3
timeout	16	6	5	14	12
recursion	0	0	0	0	0
unrec. code	0	0	0	0	0
exception	0	0	0	1	1
CPU time, min	142	130	130	142	142
memory, GB	111	116	115	117	117
analysis, min	–	60	59	72	73
reduce, min	–	4.6	4.8	9.5	9.8
expand, min	–	4.7	4.5	4.7	4.6

TABLE III
THE PRODUCED VERDICTS AND CUMULATIVE RESOURCES FOR 134 DRIVERS WITH BASELINE VERDICT *unsafe*

	baseline	keep all full	keep all red.	keep expl. full	keep expl. red.
safe	0	2	2	2	2
unsafe	127	102	102	102	100
timeout	7	22	22	22	24
recursion	0	5	5	5	5
unrec. code	0	0	0	0	0
exception	0	3	3	3	3
CPU time, min	84	95	95	100	99
memory, GB	45	49	48	53	52
analysis, min	–	64	64	63	66
reduce, min	–	4.3	4.6	8.0	9.9
expand, min	–	4.9	4.8	3.7	3.8

rows “analysis”, “reduce”, and “expand” show the time spent for the analysis of blocks and the reduce and expand operators, respectively.

The analyses with reduce variants which retain the explicit values show results similar to the baseline, but a decreased number of verdicts *unsafe*. The results appear to be largely unaffected by the stack reduction and path predicate elimination, with nearly all verdicts and verifier errors matching. Conversely, the elimination of explicit values has an impact on precision. The analysis was completed for 10 additional programs, but half of all verdicts *unsafe* are erroneous. This renders discarding explicit values ineffective, and thus it was not evaluated on the driver modules.

Tables II, III, and IV present the results and resources for the driver set divided according to the Klever verdict (*safe*, *unsafe*, and no verdict, respectively). As the resources required for a verification of a program can vary somewhat from run to run, the baseline has lost or obtained verdicts for some modules in comparison to the Klever verdicts.

Table II (374 “safe” modules) shows that the use of BAM led to a slight decrease in the number of timeouts and a few changes in verdicts from *safe* to *unsafe*. The latter can be attributed to the inconsistencies in the used environment models, underlying analysis, or BAM operators. The variant without the value information reduction was able to obtain up to 9 additional verdicts *safe* while spending 8% less time than the baseline.

TABLE IV
THE PRODUCED VERDICTS AND CUMULATIVE RESOURCES FOR 337 DRIVERS WITH BASELINE VERDICT *unknown*

	baseline	keep all full	keep all red.	keep expl. full	keep expl. red.
safe	0	14	14	11	8
unsafe	1	41	43	42	42
timeout	319	259	257	259	262
recursion	15	18	18	17	17
unrec. code	2	2	2	2	2
exception	0	3	3	6	6
CPU time, min	657	583	585	585	587
memory, GB	149	181	175	180	174
analysis, min	–	515	517	513	515
reduce, min	–	29	30	50	52
expand, min	–	38	38	33	33

Table III (134 “unsafe” modules) shows that the use of BAM led to a noticeable increase in the number of timeouts (up to 17 additional timeouts), which can be attributed to an unfortunate search order in the program state space. The analyses with BAM required 13–19% more time, which can be attributed to a substantial number of the timeouts.

Table IV (337 “unknown” modules) demonstrates that BAM facilitated the analysis of over 50 modules. The analyses with BAM required 11% less time, which can be attributed to the decreased number of timeouts.

The variants that kept the value information as is demonstrated greater efficiency than the variants that retained only explicit values. The variants that reduced the call stack showed efficiency similar to the variants that retained the full call stack. The only new timeouts were introduced in the “unsafe” subset. Only for 5 of those, BAM operators used a substantial amount of time (approximately 25 seconds out of given 120 seconds).

While BAM increased CPU time up to 17% for integration set, it also showed an 11% decrease in time for more complex “unknown” driver subset. Most of the new timeouts can be attributed to an unfortunate program space exploration order. Up to 5 changes in driver verdicts and 2 changes in integration set verdicts require more thorough investigation. While 25 driver verdicts *unsafe* were lost, 57 additional driver verdicts were obtained.

VII. CONCLUSION

The paper has considered the SMG analysis and block abstraction memoization technique which are both implemented in the CPAchecker software verification framework. Several variants of the reduce and expand operators have been presented to enable summarization in the SMG analysis using BAM. These operator variants have been implemented within the CPAchecker framework. The analysis with BAM has been evaluated on two sets of programs: a set of smaller programs from SV-COMP and a set of Linux driver modules.

BAM allowed the analysis to obtain up to 57 new verdicts and spend less computational time on complex driver modules, while up to 25 verdicts were lost due to timeout or error. The less precise operator variants did not demonstrate an improvement in efficiency compared to the more precise variants.

Future work includes: 1) advanced value information reduction to improve reduce and cache hit rate, 2) sophisticated graph reduction, 3) elaborate evaluation of referenced objects to isolate smaller subgraphs, and 4) enabling analysis of programs with recursive procedures. The change in verdicts 7 programs across both sets requires further investigation.

Acknowledgment

The author thanks his colleagues, especially A. Vasilyev, for their advice on the paper.

References

[1] D. Beyer, T. A. Henzinger, and G. Theoduloz, "Program analysis with dynamic precision adjustment," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 29–38. pages 1

[2] D. Beyer, S. Gulwani, and D. A. Schmidt, "Combining model checking and data-flow analysis," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham: Springer International Publishing, 2018, pp. 493–540. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_16 pages 1

[3] D. Beyer, "Competition on software verification and witness validation: SV-COMP 2023," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS 13994, S. Sankaranarayanan and N. Sharygina, Eds. Cham: Springer Nature Switzerland, 2023, pp. 495–522. pages 1

[4] M. Dangl, S. Löwe, and P. Wendler, "Cpachecker with support for recursive programs and floating-point arithmetic," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 423–425. pages 1

[5] K. Dudka, P. Peringer, and T. Vojnar, "Byte-precise verification of low-level list manipulation," in *Static Analysis*, ser. Lecture Notes in Computer Science, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, F. Logozzo, and M. Fähndrich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7935, pp. 215–237. [Online]. Available: http://link.springer.com/10.1007/978-3-642-38856-9_13 pages 1, 2

[6] A. A. Vasilyev and V. S. Mutilin, "Predicate extension of symbolic memory graphs for the analysis of memory safety correctness," *Programming and Computer Software*, vol. 46, no. 8, pp. 747–754, Dec. 2020. [Online]. Available: https://link.springer.com/article/10.1134/S0361768820080071 pages 1, 2, 3

[7] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, "Establishing linux driver verification process," in *Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers 7*, A. Pnueli, I. Virbitskaite, and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 165–176. pages 1

[8] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov, "Configurable toolset for static verification of operating systems kernel modules," *Programming and Computer Software*, vol. 41, pp. 49–64, 2015. pages 1

[9] D. Wonisch and H. Wehrheim, "Predicate analysis with block-abstraction memoization," in *Formal Methods and Software Engineering*, T. Aoki and K. Taguchi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 332–347. pages 1, 2

[10] D. Beyer and K. Friedberger, "Domain-independent interprocedural program analysis using block-abstraction memoization," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 50–62. [Online]. Available: https://doi.org/10.1145/3368089.3409718 pages 1, 2, 5

[11] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. Snowbird Utah USA: ACM, Jun. 2001, pp. 54–61. [Online]. Available: https://dl.acm.org/doi/10.1145/379605.379665 pages 2

[12] P. Muller, P. Peringer, and T. Vojnar, "Predator hunting party (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 443–446. pages 2

[13] M. Chalupa, J. Strejček, and M. Vitovská, "Joint forces for memory safety checking revisited," *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 115–133, 2020/04/01. [Online]. Available: https://doi.org/10.1007/s10009-019-00526-2 pages 2

[14] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, p. 752–794, sep 2003. [Online]. Available: https://doi.org/10.1145/876638.876643 pages 2

[15] P. Andrianov, K. Friedberger, M. Mandrykin, V. Mutilin, and A. Volkov, "Cpa-bam-bnb: Block-abstraction memoization and region-based memory models for predicate abstractions," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 355–359. pages 3

[16] D. Beyer and K. Friedberger, "In-place vs. copy-on-write cegar refinement for block summarization with caching," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 197–215. pages 3

[17] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness," *Communications of the ACM*, vol. 58, pp. 44–46, 2015. pages 5

[18] E. Novikov and I. Zakharov, "Towards automated static verification of GNU C programs," in *Perspectives of System Informatics*, ser. Lecture Notes in Computer Science, A. K. Petrenko and A. Voronkov, Eds. Cham: Springer International Publishing, 2018, pp. 402–416. pages 5

[19] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, Feb. 2019. [Online]. Available: https://doi.org/10.1007/s10009-017-0469-y pages 5

[20] O. Petrov, "Evaluation results for SMG with BAM," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10869482 pages 5