Static Verification “Under The Hood”: Implementation Details and Improvements of BLAST

Pavel Shved
Institute for System Programming, RAS
shved@ispras.ru

Vadim Mutilin
Institute for System Programming, RAS
mutilin@ispras.ru

Mikhail Mandrykin
Moscow State University
misha.bear.1990@gmail.com

***Abstract*—BLAST is an open-source static verification tool used in checking safety properties of C programs. Given a C program with several assertions, which should not fail at runtime, BLAST statically analyzes the program, and either returns a program execution path that leads to violation of one of the assertions, or proves that no assertion is violated. If BLAST fails to prove inreachability of assertions, it may terminate with error, or loop forever. The framework approach employed in BLAST is counter-example guided abstraction refinement (CEGAR) empowered with lazy abstraction.**

The first record of BLAST dates from 2002. The tool had been constantly improving until July 2008, mostly by its original creators. Beginning in 2009, we continued working on it as a part of Linux Driver Verification project.

In this article we overview the current status of BLAST: outline the algorithms the CEGAR framework approach is implemented on top of, describe the heuristics used and the technical details of the implementation, and list the external components BLAST relies on. Along with this description, we outline and evaluate the improvements we made since its last release by the original BLAST team, and share our view on the further improvement of the tool.

***Index Terms*—Software verification, safety properties, reachability verification, static analysis.**

I. Introduction

BLAST is an acronym of “Berkeley Lazy Abstraction Software verification Tool”. It is a C program verification tool that solves reachability problem. Given a C program, a name of the main function (“entry point”) and a name of a label, it reasons if there exists a program execution path that starts at the entry point and reaches the label specified.

It analyzes the program with CounterExample-Guided Abstraction Refinement approach (for details, see [11]). The way it implements CEGAR is known as “lazy abstraction” [12], a novel approach at that time that aims to retain the part of abstraction that should not change, instead of rebuilding abstraction from scratch after each counterexample analysis. This approach gave the name to BLAST, and the article referenced ([12]) presented BLAST in its “Experimental Results” section; moreover, this article does not contain any reference to another source where BLAST was described or mentioned.

The task in subject is computationally impossible, as the halting problem may be reduced to it. The tool therefore does not guarantee its analysis will terminate. BLAST may terminate in runtime if it detects that the program can’t be analyzed by it, or provide an incorrect result: either “false unsafe” (a program that does not violate the safety property being checked, but is reported as “unsafe”) or “false safe” (an “unsafe” program reported as “safe”). However, if the tool reports an “unsafe”, it also prints the error trace: the path to error location from the entry point, which may be followed through by a human.

To build an open automated system for Linux device drivers described in [13], we needed a verification tool, and that article outlines BLAST as a tool “intended for academic research in software verification”. The experiments demonstrated that its potential may spread beyond mere academic application, serving as a verification engine in an intensive driver checking workflow, as well as provide a ground for research in the domain of static analysis. However, before its potential strength became current a serious work has been done.

A good description of how the algorithms implemented in BLAST work is in [6]. That article contains a step-by-step explanation how a sample program is verified with BLAST, but does not focus on its implementation details.

In this article we describe what BLAST is now, and outline how we improved it since its last official release (version 2.6) by the original team. “Vanilla” algorithms described elsewhere (see [12] and [6]), we list the undocumented but worthwhile improvements the authors of BLAST made in its implementation[1]; we focus, however, on listing our contribution, and evaluate the impact of our improvements in the relevant domains if possible.

A. Algorithms used in BLAST

The algorithms used in BLAST are briefly described as “lazy abstraction CEGAR, with Cartesian predicate abstraction and LA+EUF Craig interpolation as predicate discovery procedure”.

To be more specific, this means:

- **CEGAR** — counterexample-guided abstraction refinement, a process of solving reachability problem by constructing a crude abstraction of all possible paths reached

[1]The authors recommend to always configure BLAST to use these improvements instead of “vanilla” algorithms by specifying “`-craig 2 -predH 7`” in the command line of the tool.

Fig. 1. Assumption as a library call

```
void custom_assume(int condition)
{ if (!condition)
    ENDLESS: goto ENDLESS;
}
```

from the entry point and iteratively refining this abstraction by analysis of plausible paths that lead to the error location, until the abstraction contains no such paths—or a reachable error location is found. For more info, see [11];

- **lazy abstraction** — an implementation of CEGAR characterized by refining only those parts of abstraction that should contribute to proving inreachability after a counterexample analysis. Re-evaluation of other parts of abstraction is avoided as much as possible. For more info, see [12];
- **control-flow graph (CFG)** — lazy abstraction assumes that the program is represented as a finite control-flow graph, i.e. a labeled-transition system. The abstraction is then a "reachability tree": a prefix tree of all possible paths in the CFG from the entry point, each node (location) being marked with an abstract state. It's named an "Abstract Reachability Tree", or simply "ART".
- **Cartesian predicate abstraction** — representation of the abstract state of a location as a conjunction of zero or more predicates previously discovered. Unlike Boolean abstraction, Cartesian one restricts usual first-order Boolean logic formulæ to conjunction operator only. For more info, see [2];
- **Craig interpolation** — a procedure of building a Craig interpolant: given two Boolean formulæ with an unsatisfiable conjunction, construct an over-approximation of the first one that uses only terms from the second one, keeping the conjunction of the approximation and the second formula unsatisfiable. As of today, BLAST uses this procedure to construct new predicates for its Cartesian abstraction.

II. Implementation details

A. Generic information

Comprising components written in various languages, BLAST has its core part written in OCaml, and it compiles with OCaml 3.11 version. BLAST runs under Linux.

OCaml abstracts away memory operations, automated processing of which took a considerable time, according to profiling results. By tuning some documented options of OCaml runtime, we decreased memory allocation overhead. Sample programs demonstrated a 20% increase in the amount of locations explored.

B. Program representation

As previously noted, the approach implemented in BLAST requires the whole program to be represented as a finite control-flow graph. It means that all functions should be inlined, and no recursion is allowed. However, BLAST approaches this in a different way; it uses CIL [16] to build per-function control-flow graphs. During the program analysis, BLAST automatically jumps to the proper function unless it is called via a function pointer. This approach provides more flexibility, and allows to implement heuristics that concern function calls. For instance, BLAST is capable to support recursion with restricted depth. We also implemented a similar bounding even for non-recursive calls, as we had noticed that analysis does not need to traverse deeply in the call graph to succeed in finding bugs. In Linux Driver Verification project, this allowed us to use code generation tools that automatically satisfy unresolved external calls to functions specified throughout the whole kernel without a dramatic analysis quality degradation.

The representation in such form ignores loop structure, and unrolls them into a set of conditional jumps. The **goto** operators are respected as well. This allows programs to undergo serious transformations, and still be checkable by BLAST. We needed a functionality to denote assumptions of the form "from now on, a certain condition holds"[2]. Instead of trying to built this into BLAST as a special directive, we devised a library function that solves this problem; it's listed on Figure 1. The function is a valid C as well as it does not confuse BLAST with an unnatural endless loop.

As Linux Kernel sources leveraged the whole power of C language and its GNU extensions, we integrated the latest CIL version to BLAST (1.3.7 instead of 1.3.1), and made several minor improvements to it. Now BLAST is capable to read and process drivers of Linux Kernel of version 2.6.37 with just 2% of modules leading to parse errors during the analysis.

C. Abstract Reachability Tree exploration

As noted above, lazy abstraction approach does not require the abstract reachability tree to present in memory as a separate data structure. Hence, in BLAST the abstraction is stored in a custom data structure as a graph, and the abstract postcondition computation happens at its leaf nodes (also named the "frontier"). If set of possible program states in a leaf is empty, it's not traversed anymore. When a leaf contains a plausible error location, the counterexample analysis begins, and the reachability tree is then cut so that its only leaf is the one specified by the error path analysis. When a leaf is covered by another leaf which was already processed, the analysis stops in favor of that already happening starting from the covering leaf. It is implemented by storing "reached region" that comprises all the locations reached so far.

The order the leaves are processed in is tunable. Presets include depth-first traversal, breadth-first traversal and bounded depth-first search (traverse in depth-first manner up to depth N, add the pending nodes to queue and get the next node from the queue). The default method is BFS; possible reason is that

[2]This is useful to specify preconditions for initial data, which may rule out false positives in certain situations.

it allows to find error locations faster, and the experimental data described in [4] prove that BFS allows faster verification than DFS.

Processing a leaf constitutes on determining its region (an over-approximation of all possible program variable states on this path) by incoming edge in the CFG and region in previous location. More on this procedure in Section II-G.

D. Counterexample analysis

When a counterexample—a reachable error location, for which the abstraction contains a non-empty set of program states,—is found in the reachability tree, its analysis starts. A sequence of operations that leads to this location from the root node is fetched from the ART. Then, preconditions of all nodes are taken, and static single assignment (SSA) conversion is applied to the resultant formulæ. The formulæ are stored in a custom OCaml data structure.

Interestingly, a path formula is converted to SSA backwards: i.e. the closer the nodes are to the root of the tree the greater the indexes of their variables are. Therefore, different error paths do not have a common prefix in their path formulæ. An optimization opportunity here would be to reverse the indexes and parallelize ART exploration and path formulæ construction. This might also help with alias analysis (see Section II-I).

For the further analysis, the formulæ are converted from custom format to one of formats suitable for external solvers (special modules take care of that). Due to large size of the formulæ the conversion may take a lot of time. It was the case for SMT solvers format. To overcome this, we focused on this conversion, and made it nearly a thousand times faster, which made the conversion overhead negligible compared to the time to perform an actual formulæ analysis. This result also demonstrates that tight integration with solver's formulæ representation format might not be necessary for a CEGAR-based verification tool.

After proving that the formula is unsatisfiable (hence the counterexample is spurious), predicate discovery procedure starts. These two activities are described in the next two sections.

E. Path feasibility checking

External solvers are to decide if the formula is satisfiable (sat) or unsatisfiable (unsat). If error path formula is unsatisfiable, then the counterexample is spurious, and should be analyzed, and the abstraction should be refined; otherwise, there's an error in the program. For first-order logic in Linear Arithmetic and Uninterpreted Function Symbols theory, formula satisfiability is a computationally hard problem. Thus, careful choice of SAT solver is crucial for building a fast verification tool.

In the BLAST as of 2008 Simplify solver was used; it is a "stack-based" solver[3] (which makes it ideal for analyzing path formulæ concurrently with their generation), but it is a legacy closed-source software with serious licensing limitations. After resolving the performance issues in conversion to SMTlib format (see sec II-D), we turned to experimenting with SMTlib solvers, mainly with CVC3, as its LGPL license fits our aim of building an open toolset for software verification.

We noticed that in BLAST it is possible for SAT solver to report "unknown" instead of "unsat", and these results are indistinguishable for BLAST. Since proving satisfiability of large formulæis hard, and, at the same time, if a formula is satisfiable, it's more likely for the satisfying input to be found really quick, the "unknown" result may server as "unsat" if the solver is tuned properly. We tried CVC3, and discovered that by default it runs in a "honest" mode where no unknown results were possible, and it took CVC3 gigabytes of RAM and several minutes to verify a typical formula appearing in driver source code analysis.

It turned out that the main reason for such a low CVC3 performance on many typical BLAST queries was its use of certain quantifier instantiation heuristics. By default, BLAST path formulæ contain quantified axioms used to model memory with aim to rule out some false unsafes when pointer operations are used. Default settings of CVC3 (used for SMT-LIB benchmark) turned complete quantifier instantiation heuristic, which made it try to instantiate every given axiom with every suitable combination of ground terms occurring in the formula. Also one axiom instance may itself contain new ground terms that can be again used for instantiation. Since the typical BLAST formula contains quantified axioms and a lot of terms, the solver spent much time and memory on the instantiations described above.

We used an option to disable complete instantiation heuristic and put a smaller limit on the number of repeated instantiations. This significantly decreased the number of resulting instances and thus time and memory consumption. Then we also disabled some other heuristics regarding quantifiers. It didn't cause any significant correctness degradation because the axioms rarely helped the solver prove formula unsatisfiability.

As of today, there is no reason to use Simplify anymore, as CVC3, combined with our fixes to integrational components of BLAST, outperforms it.

We also removed predicate normalization from BLAST[4], as we supposed that solvers should do it much faster. Our experiments confirmed this.

F. Predicate discovery

Vanilla algorithm to discover predicates with Craig interpolation looks like this. A path formula is cut into conjuncts (basic blocks are cut apart), and at each cut point the conjunction of all terms before and after formula may undergo Craig interpolation. For LA+EUF theory Craig interpolants always exist ([15]), and an interpolating prover is a tool to find them.

However, instead of running an interpolating prover at each cut point, BLAST first determines "useful blocks", a subset of

[3]Allows to push/pop conjuncts of a formula and analyze the conjunction of formulæ currently on stack; this could assist checking several formulæ that share common parts for satisfiability.

[4]Actually, we added an option to turn it off/on.

operations along the trace that contribute to unsatisfiability of the formula: a minimum set of blocks conjunction of which is unsat, while the rest of the formula is satisfiable. There may be several non-overlapping sets of useful blocks for a trace. Essentially, "useful blocks" are close to unsatisfiability core of a path formula, with two differences:

- **granularity** of predicate selection is operator-wise, i. e. a predicate for one assignment or conditional may only participate as a whole, while only a part of it may belong to unsatisfiability core[5];
- **regions may participate** in "useful blocks". A region, by construction, is an over-approximations of the path formula to the location it is assigned to. So, instead of analyzing the trace prior to a certain location, if a conjunction of the region in this location (computed by previous refinement procedures) and the part of the formula past this location is unsat, then we treat the region as if statement, and nominate it as a part of useful block set.

The bits of formula extracted this way may themselves become predicates for the abstraction (as in one of the steps in SLAM tool [1]). However, BLAST goes further, and runs the interpolating prover for each cut point between blocks in each of the "useful block" sets, treating each set as a small error trace. This way it only calls interpolating prover as many times, as there are these useful blocks, and the formulæ for it to handle are much smaller.

To determine these "useful blocks" BLAST joins predicates in path formula one-by-one, beginning from the last, until their conjunction becomes unsat. Then the latest block joined is a useful one. The next useful block is found with the same procedure, but the first useful block found is added to each conjunction. The procedure repeats recursively until the set of blocks found so far and the next useful block alone form an unsatisfiable conjunction.

For stack-based solver, such as Simplify, joining predicates one-by-one is straightforward. For SMT solvers all intermediate conjunctions were to be checked separately. We noticed, however, that conjunction of all blocks from the end of the trace up to i-th one is a monotonous function of i (the more blocks you join, the more likely their conjunction is unsat), and binary search may be applied to find the next useful block. We implemented the binary search for SMT solvers, and we also implemented caching for the predicates converted to SMTlib format. For a complex sample cxausb driver the number of calls to SMT solver was decreased from 32630 to 831, thus reducing the overall verification time of this sample by the factor of 7.

Craig interpolants for each cut point in small "traces" constructed from each of the useful block sets are calculated and added to lists of potential predicates in the locations of the real trace between the first and the last useful blocks, and to the locations in their ART subtrees. The negations of the interpolants are not added at this point, but they will be taken into account during the refinement.

To perform the interpolation, an Apache-licensed CSIsat prover [9] is used, which takes input in "FOCI" format. It sometimes outputs interpolants with real arithmetic (for instance, it may print "$x < 0.1 \cdot y$" instead of "$10x < y$"); in these cases, BLAST ignores its output and finds less predicates hoping that the rest would be enough to prove the safety of a program.

Each predicates is encoded as a Boolean variable, and each region is stored as a BDD (binary decision diagram) over these variables.

G. Abstraction refinement

After an error path was encountered and analyzed, the analysis in the ART subtree of the node corresponding to the "useful" block closest to the root is restarted, and the nodes in it are removed from the queue (also known as "frontier").

To calculate the region of a frontier node, the "abstract postcondition" procedure described in [12] is used. For each of the predicates discovered for this location it is tested if the precondition of the operation along the incoming edge, given the predicate is assumed, is implied by the calculated region in the parent node. To check satisfiability of the formulæ built this way the same SMT solver is used as in the trace analysis.

Since all the data required by such a refinement are local, the exploration of the state space may be made concurrently. We have not implemented this for BLAST, however, a research in this direction yields promising results [14].

After predicate for a node is verified, the node should be tested for coverage. For this, it constructs a BDD that denies that the calculated region for this node implies the reached one, and checks it for truthfulness via BDD, each predicate from the Cartesian abstraction being represented as a distinct BDD term. If this crude check fails, a more precise one with use of the SAT solver is performed; it takes into account that predicates share variables, and are not independent from one another. If a node is not covered, its children are added to the frontier, and the reached region for the location is updated.

H. Configurable verification

In BLAST it is possible to use lattice-based data-flow analysis to aid CEGAR. Lattices are known to over-approximate the feasible program states, so they may be used to rule out infeasible paths in combination with usual CEGAR analysis to analyze the rest. BLAST contains several such lattices, and only one of them (SymbolicStore) is a generic-purpose lattice that fits all C programs; it is capable to store information on concrete values of integers and structure fields as well as perform shape analysis [5].

To utilize capabilities of lattice-based data-flow analysis, BLAST extends the structure of node's region beyond the usual conjunction of interpolants. The region in BLAST is a tuple of CEGAR's predicate constraint, and of several lattice elements, the set of lattices being configured by user. If any of tuple elements is $\bot$ (or false, for predicate regions), then

[5] It is especially important for more complex formulæ: when alias analysis (see Section II-I) or other techniques (such as [17]) are used.

the further path exploration is not necessary, since one of the means has proved it infeasible.

As for coverage checking, the lattice-aided verification contained a severe issue: the $stop^{join}$ operator was hardcoded; it made BLAST nominate a single joined region as reached instead of a set of regions. This cuts feasible program paths, since SymbolicStore lattice regions are not a powerset domain [7] (while predicate regions are). Also, $stop^{join}$ made the number of false safes too big for a certain environment model[6], so we implemented $stop^{sep}$ which checks coverage against a set of reached regions, and several versions of $merge$ operator: join at meet-points (merge-join), join at equal predicates (merge-pred-join)[7], and no join (merge-sep). After experiments we chose $stop^{sep}$ with merge-pred-join as our default setting.

As a result, the runtime of BLAST with a SymbolicStore lattice had a 50% increase, but the precision was improved significantly: the amount of true unsafes increased by 20% approximately.

This concept of combining different operators in the exploration of state space of a single program in a configurable way was then developed by one of the authors of BLAST in the other tool, CPAchecker [8]. Our experience demonstrates that while it's not trivial to add more operators the BLAST implementation is loosely-coupled, and it is only lack of syntax and framework sugar what prevents a developer from configuring such operators easily, but the changes one is required to make are not dramatic.

I. Alias analysis

BLAST employs flow-insensitive may-alias analysis for more precise reasoning about pointer assignments. As pointed out in [5], the analysis is "home-brewed", and we'll describe the algorithm here briefly.

The alias analysis starts when the first feasible error location is found. Originally, BLAST performed this costly procedure at the beginning of analysis of a program, but for programs with unreachable error locations, or in cases when all error paths are ruled out by lattice analysis (see Section II-H), but we fixed this.

First, the whole program is analyzed, and the aliasing relation is calculated: if x may point to y at any point of program, then "may-alias(x,y)" is true. The relation is not reflexive: while x may be any identifier, y should refer to a concrete memory location (stack- or statically-allocated memory, or a location with a `malloc()` call). An over-approximation is built by analyzing each assignment (if x is assigned an address of z then x may point to z[8]), and closing it transitively (i. e. if y is assigned to x, and y may-point to z then x may-point to z). This way, an over-approximation of an "ideal" may-aliasing relation is built.

[6]Environment models are "main" functions generated based on templates for Linux device drivers. For more see [13].

[7]Predicate equality was tested via BDDs that stored them.

[8]Pointer operators are ignored at this point, only identifier names are essential here.

Fig. 2. Verification of this program requires more lvalues than it contains

```
void mutex_lock(struct mutex* mtx)
{ assert (*mtx == 0);
  *mtx = 1;           }
void mutex_unlock(struct mutex* mtx)
{ assert (*mtx == 1);
  *mtx = 0;           }
int main()
{ struct mutex* m;
  mutex_lock(m);
  mutex_unlock(m);
}
```

Each expression is encoded as a bit-vector, and the relation is stored in the a BDD. If x aliases q, and the expressions are encoded as vectors X and Q respectively, then true value for bit-vector (X, Q) is inserted into BDD.

When the path formula is constructed, and an assignment `*y=q` appears in it, alias analysis comes into play. It queries each lvalue (a non-constant expression that denotes a concrete value in the writable memory) encountered in the program if y may-alias its base identifier. For each lvalue x it may alias, the following expression is added to the formula in addition to the usual predicate:

$$((y = x) \to (*x = q)) \wedge ((y \neq x) \to (*x = *x_{old})) \qquad (1)$$

where $*x_{old}$ is the previous instance of expression $*x$ in the SSA form. The expression means "if y *really* points to the same place x does, then the value of $*x$ also becomes q. Otherwise, this assignment does not change the value of $*x$".

The lvalues iterated should not be constrained by those encountered in the program. For instance, consider the following program on Figure 2. To verify it, we need to consider $*m$ as an lvalue, while the program does not contain it. BLAST has a functionality to close the set of lvalues of a program under dereference and taking a field (for structures) operations up to a specified number of dereferences. The depth of such a closure is required to be at least one for the program shown above, but it already prohibitively increase the number of iterations over lvalues, beyond the sensible time limits.

Some may-aliases are also must-aliases. For instance, CIL frontend generates additional variables for assignments to complex lvalues that involve multiple dereferences and field takings. These variables are known to alias only one single variable, and the expressions like (1) generated for them will not contain any disjunctions, and unconditionally assign the value to the must-alias.

We tried to decrease such a number of iterations over lvalues by withdrawing must-aliases from the set of lvalues, then by adding all "const" values to the set of must aliases (to withdraw even more), but we could not make the iterations fast enough. This improved the speed of alias analysis alone by factor of hundreds, but even this wasn't fast enough. Using faster data structures might help, but we think that a qualitative research boost should precede fast verification of pointer-abundant programs.

J. Interaction with user

UI was aimed to satisfy a user that looks at the console output: plain text printing of debug information, analysis work, statistics and reports mingled together. We did not change this much, but tuned the output of error trace and verdicts to fit automatic processing of it. Now the external tools may read the verdict, and the error trace with additional information. This is especially useful when the exploration is cut due to function call depth limit a user specified, because it would be unclear from the trace that the limit was enforced rather than the function is not found.

K. Infrastructure

We added regression tests based on situations that occur during Linux device drivers analysis. They contain both expected and current results, for tracking improvements as well as degradations.

External SAT solvers are connected through a special layer, that allows parallel execution of queries to the external tools. For instance, CVC3 is known to have a lag between it outputs an answer and finished the work (perhaps, due to complicated resource deallocation), but the layer between BLAST and a solver does not make BLAST wait and reap the process. Ditto for interpolating prover.

BLAST contains a lot of dead code. Only a narrow set of options is supported: some configurations do not work at all, and terminate with an exception unconditionally. We did not try to eliminate it; one of the reasons is that it contains surprises: for example, we were going to implement the closure under dereferences we described in Section II-I on our own, but we suddenly found the working code for this commented out.

1) External components summary: Default shipment of BLAST includes:

- **CUDD** package — utilities for binary decision diagrams. Implemented in C, distributed under MIT-like license.
- **CVC3** solver [3] — proves (in)satisfiability of various formulæ. Implemented in C++, licensed under LGPL. Communicates to BLAST via SMTlib competition format.
- **CSIsat** interpolating prover [9] — computes Craig interpolants for LA+EUF. Implemented in OCaml, licensed under Apache. Communicates via FOCI-like interface (which is supported, for instance, by MathSAT [10] interpolating prover as well).
- **CIL** C frontend — converts C program into syntax tree stored as OCaml structures. Implemented in OCaml, licensed under Apache.

L. Known limitations

BLAST does not support assignments of structures as a whole; does not support function calls by pointer (although some dead code on this matter is included); ignores inline assembly; does not provide automatic deduction of properties involving reasoning about lists and other complex pointer-based structures; does not support arrays, treats each array field as a separate identifier, and can not associate `a[i]` and `a[j]` if $i = j$ (i and j being the variables); can not reason about pointer inequalities; does not have a fast aliasing solution; ignores short logic in conditional statements; does not cope with interpolants with real numbers; lacks automatic modularization, and always analyzes the program as a whole.

III. Evaluation

To evaluate our improvements, we compared how the latest BLAST version form the original developers and our version performs on Linux device drivers from `media/` folder of 2.6.37 kernel, and with a simple rule that checks if the mutex locking is correct. Each launch of BLAST was limited with 15 minutes of CPU time and 1 Gb of memory. To make the older BLAST work with our newest tools, we merged several integrational fixes to it, and we had to merge the latest CIL frontend as well, as the default frontend in the older BLAST can process zero drivers. The results of the comparison are in the Table I.

The results demonstrate that the new version of BLAST is capable to find three times more errors[9] (and the newer version found all the five errors found by the older one), and total speed was improved by the factor of 5, given that the precision of BLAST has increased (see Section II-H). The number of drivers that were reported as neither safe nor unsafe is two times less than those of the original version. With newer version, only 30 drivers exceeded resource limits (only two of them timing out), while the older version ran out of allowed resources in 52 cases, and it means that 22 out of 52 the most complex samples were successfully verified under the same constraints.

In the `media` folder the new frontend has eliminated all the parsing errors; however, a more exuberant evaluation of the newer BLAST demonstrates that as much as 1.1% drivers are still not parsed by BLAST in 2.6.37 kernel. Compared to the original BLAST that can process zero Linux kernel drivers without special patches applied to their source code, this is quite an improvement.

IV. Conclusion

Having started from BLAST 2.6 of 2008, we implemented a lot of fixes to BLAST, which improved its productivity on industrial code base (Linux device drivers) by a factor of more than five (as the evaluation in Section III demonstrates), making it, at the same time, more precise and capable to find more errors, as well as more tolerant to the C code it

[9]The correctness rule was intentionally weak, so most of these errors are not kernel bugs, but they are valid if approached as mere assertion violations in C programs.

TABLE I
EVALUATION OF THE ORIGINAL AND THE CURRENT VERSIONS OF BLAST

BLAST version	Total	Failures	SAFE	UNSAFE	Total time	Timed out	Memory limit	Other failures
Original	389	110	274	5	11.5 hours	36	16	58
Current	389	57	317	15	2.1 hours	2	28	27

parses. The precision improvement is not just ad-hoc, caused by optimized resource consumption: the algorithms themselves were improved as well.

During our experiments, we succeeded in utilizing a generic SMT solver, and demonstrated that formulæ conversion from an internal verification tool's format to SMTlib competition format for programs as large as Linux device drivers takes negligible time compared to other activities.

We learned also that BLAST is extensible enough to implement more powerful verification algorithms, albeit it is not a straightforward task for a developer. Thus, the weaknesses of BLAST may be overcome, and it's too early for BLAST to be considered obsolete.

REFERENCES

[1] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4 false alarms. In *Conference on Formal Methods in Computer Aided Design, FMCAD 2010, Lugano, CH*, 2010.

[2] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. *Proc. TACAS*, page 268–283, 2001.

[3] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[4] D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 25–32, nov. 2009.

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

[6] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, 2007.

[7] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 504–518, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. Technical report, School of Computing Science, Simon Fraser University, 2009.

[9] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In *CAV*, pages 304–308, 2008.

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4smt solver. In *CAV*, pages 299–303, 2008.

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

[12] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[13] Alexey Khoroshilov, Vadim Mutilin, Vladislav Shcherbina, Oleg Strikov, Sergei Vinogradov, and Vladimir Zakharov. How to cook an automated system for Linux driver verification. In *2nd Spring Young Researchers' Colloquium on Software Engineering*, volume 2 of *SYRCoSE 2008*, pages 11–14, 2008.

[14] Nuno P. Lopes and Andrey Rybalchenko. Distributed and predictable software model checking. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 340–355, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] K.L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.

[16] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

[17] Pavel Shved. On reasoning about finite sets in software model checking. In *4th Spring Young Researchers' Colloquium on Software Engineering*, SYRCoSE 2010, pages 17–26, 2010.