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

Pavel Shved[1] Vadim Mutilin[1] Mikhail Mandrykin[2]

[1]Institute for System Programming, RAS

[2]Moscow State University

Sprint Young Researchers’ Colloquium on Software Engineering
12 May 2011

“Heavyweight” Static Program analysis

Static analysis — checking programs against specific properties without executing them (by their source or machine code).
Features:

- \+ all possible inputs are checked
- \+ certain methods can prove the program correct
- − expressiveness of checkable programs is limited

Linux Driver Verification challenge

The aim is to statically verify Linux Kernel device drivers against Kernel core interface.

- \+ no actual equipment is necessary
- \+ driver source code is not too complex for static analysis
- − there is a lot of drivers
- ! static checker should be fast and yield few false positives

Overview of BLAST

BLAST — "Berkeley Lazy Abstraction Software verification Tool". Developed in 2002 by Henzinger, Jhala, and Majumdar. Introduced in their "Lazy Abstraction" paper.

BLAST tries to solve reachability problem: given

- **a source code** of a C program
- **an entry point**, i. e. a name of the "main function"
- **an error location**, i. e. a name of a label

report if there exists a valid path from the entry point to the error location.

Overview of BLAST

BLAST — "Berkeley Lazy Abstraction Software verification Tool". Developed in 2002 by Henzinger, Jhala, and Majumdar. Introduced in their "Lazy Abstraction" paper.

BLAST tries to solve reachability problem: given

- **a source code** of a C program
- **an entry point**, i. e. a name of the "main function"
- **an error location**, i. e. a name of a label

report if there exists a valid path from the entry point to the error location.

Which means it may verify safety properties with inserted assertions:

```
void assert(bool condition)
{ if ( ! condition )
    ERROR: goto ERROR;
}
```

The approach used in BLAST

Keywords: "lazy cartesian predicate abstraction CEGAR of C programs with Craig interpolation":

- **input is a C source code**:
  - no spec is necessary (one `assert()` is enough!)
  - undefined funcitons are treated as pure
- **auto abstraction** — <u>automatically</u> creates an "abstract model" of a program just precise enough to prove inreachability (if it's the case)
- **precise elaboration** of potential errors: feasibility in the "abstract model" is not enough to claim it's an error!
- **counterexample-based refinement**: deduce properties to watch for from the ruled out error traces
- **path-specific analysis**: not just annotates the CFG, but explores all possible <u>paths</u>

BLAST as a tool

BLAST as a tool features:

- **configurable analysis** — supports (limited) modularity
- **utilizes external tools** such as SAT Solvers and Interpolating Provers
- **open-source implementation in OCaml**, an expressive compileable language

The other side of BLAST

In theory it looks good, but in practice (Linux Driver Verification):

- BLAST was slow
- BLAST couldn't parse the Linux Kernel's source code (at all)
- BLAST relied on obsolete tools

The other side of BLAST

In theory it looks good, but in practice (Linux Driver Verification):

- BLAST was slow
- BLAST couldn't parse the Linux Kernel's source code (at all)
- BLAST relied on obsolete tools

Our aim was to overcome these difficulties, retaining all the features.

Input source code

BLAST transforms the program into a set of per-function control-flow automata (or, graphs).

- **parse errors** — ★ updated and patched C frontend (improved from 0% to 95%)
- **overspecification** — too slow if all undefined functions are found
  ★ option to limit function call depth (unsound, but lets us find bugs, at least)

Specifying properties

Safety properties are found in the program or instrumented into it.

Assertions (“if it doesn’t hold, there’s a bug”):

```
void assert(bool condition)
{ if ( ! condition ) ERROR: goto ERROR; }
```

☆ Preconditions (“we assume this is true”):

Specifying properties

Safety properties are found in the program or instrumented into it.

Assertions ("if it doesn't hold, there's a bug"):

```
void assert(bool condition)
{ if ( ! condition ) ERROR: goto ERROR; }
```

☆ Preconditions ("we assume this is true"):

```
void assume(bool condition)
{ if ( ! condition ) NOT_ERROR: goto NOT_ERROR; }
```

Precise analysis of error trace

Check: is “path formula” (conjunction of local post-conditions along a potential error path) satisfiable?

Precise analysis of error trace

Check: is “path formula” (conjunction of local post-conditions along a potential error path) satisfiable?

Path formula is laid out in an internal format, then converted for an external solver to check (and not once for each path!)

- ☆ **tight optimization**: a better code makes the difference
- ☆ **caching**: reuse already converted parts of the folmula

Precise analysis of error trace

Check: is "path formula" (conjunction of local post-conditions along a potential error path) satisfiable?

Path formula is laid out in an internal format, then converted for an external solver to check (and not once for each path!)

- ☆ **tight optimization**: a better code makes the difference
- ☆ **caching**: reuse already converted parts of the folmula

Result: conversion speedup approached 1000 times; Total: 20% (Amdhal's law).

Using external SAT Solver

BLAST's default was Simplify solver (proprietary, not maintained). We tried CVC3 SMT solver instead. To make it work, we

- ☆ **removed formula simplification** from BLAST code: modern optimized solvers do it much faster
- ☆ **tuned solver for less precision**, as BLAST may accept "unknown" as "UNSAT". Improved memory usage (50M instead of 4G)
- ☆ **patched solver** to improve its speed and compatibility

Now BLAST uses open-source, free CVC3 solver, and the conversion overhead is small.

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

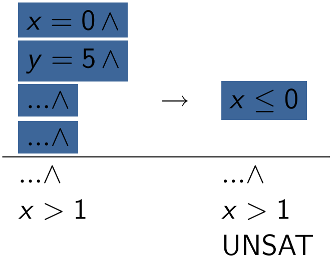
$x = 0 \wedge$
$y = 5 \wedge$
$\ldots \wedge$
$\ldots \wedge$
$\ldots \wedge$
$x > 1$

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```



$x = 0 \wedge$

$y = 5 \wedge$

$\ldots\wedge$ $\rightarrow$ $x \leq 0$

$\ldots\wedge$

$\ldots\wedge$ $\ldots\wedge$

$x > 1$ $x > 1$

UNSAT

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

$$\begin{array}{lcl} x = 0 \wedge & \rightarrow & x \leq 0 \\ \hline y = 5 \wedge & & y = 5 \wedge \\ \ldots\wedge & & \ldots\wedge \\ \ldots\wedge & & \ldots\wedge \\ \ldots\wedge & & \ldots\wedge \\ x > 1 & & x > 1 \\ & & \text{UNSAT} \end{array}$$

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

$$\begin{array}{lcl}
x = 0 \wedge & & \\
y = 5 \wedge & \rightarrow & x \leq 0 \\
\hline
\ldots\wedge & & \ldots\wedge \\
\ldots\wedge & & \ldots\wedge \\
\ldots\wedge & & \ldots\wedge \\
x > 1 & & x > 1 \\
 & & \text{UNSAT}
\end{array}$$

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
   int x=0;
   int y=5;
/* statements */
/* statements */
/* statements */
   if (x>1){
      error();
   }
}
```

$$\begin{array}{lcl} x = 0 \land & & \\ y = 5 \land & \rightarrow & \boxed{x \leq 0} \\ \ldots\land & & \\ \hline \ldots\land & & \ldots\land \\ \ldots\land & & \ldots\land \\ x > 1 & & x > 1 \\ & & \text{UNSAT} \end{array}$$

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

$$\begin{array}{lcl} x = 0 \land & & \\ y = 5 \land & & \\ \ldots\land & \rightarrow & x \leq 0 \\ \ldots\land & & \\ \hline \ldots\land & & \ldots\land \\ x > 1 & & x > 1 \\ & & \text{UNSAT} \end{array}$$

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

$$
\begin{array}{lll}
x = 0 \wedge & & \\
y = 5 \wedge & & \\
\ldots \wedge & \rightarrow & x \leq 0 \\
\ldots \wedge & & \\
\ldots \wedge & & \\
\hline
x > 1 & & x > 1 \\
 & & \text{UNSAT}
\end{array}
$$

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

$x = 0 \wedge$

$y = 5 \wedge$

$\ldots \wedge$

$\ldots \wedge$

$\ldots \wedge$

$x > 1$

UNSAT

Predicate discovery

The path formula undergoes several Craig interpolations, with cut-points between each pair of statements.

Sample program:

```
int main()
{
  int x=0;
  int y=5;
/* statements */
/* statements */
/* statements */
  if (x>1){
    error();
  }
}
```

$$\begin{array}{ccc} x = 0 \land & \rightarrow & x \leq 0 \\ \hline x > 1 & & x > 1 \\ \text{UNSAT} & & \text{UNSAT} \end{array}$$

Predicate discovery

BLAST selects “useful blocks” (think unsatisfiability core) by analyzing the trace with SMT solver calls.

Predicate discovery

BLAST selects “useful blocks” (think unsatisfiability core) by analyzing the trace with SMT solver calls.

- ☆ **requires** $O(\log N)$ **solver calls** instead of $O(N)$, where trace length $N \sim 5000$
- ☆ **caching** makes the conversion overhead negligible
- only “useful” block sets undergo interpolation
- interpolants are split into CNF, and each conjunct is added as an atomic predicate

$$\frac{5000}{\log 5000} \sim 400$$

Miscellaneous

OCaml virtual machine spent 35% of runtime to manage memory.

Miscellaneous

OCaml virtual machine spent 35% of runtime to manage memory.

☆ We tuned VM (via documented options) and achieved 25% performance increase.

Miscellaneous

OCaml virtual machine spent 35% of runtime to manage memory.

☆ We tuned VM (via documented options) and achieved 25% performance increase.

BLAST has an option for a correct pointer aliasing algorithm, which requires a lot of enumeration (slow!).

Miscellaneous

OCaml virtual machine spent 35% of runtime to manage memory.

☆ We tuned VM (via documented options) and achieved 25% performance increase.

BLAST has an option for a correct pointer aliasing algorithm, which requires a lot of enumeration (slow!).

★ We tried to improve the data structures involved, achieving 100x speedup of computing aliases, but that was still not enough to use alias analysis for large programs.

Configurable analysis

BLAST contains limited configurable analysis opportunities. Predicate-based analysis may be supplemented with lattice-based analysis, which has:

- \+ much greater speed
- − high rate of false posistives
- → rule out some infeasible paths with fast lattices, and analyze the rest with slower predicates
- − $stop^{join}$ as coverage checking is hardcoded, leads to missing errors

Configurable analysis

BLAST contains limited configurable analysis opportunities. Predicate-based analysis may be supplemented with lattice-based analysis, which has:

- + much greater speed
- – high rate of false posistives
- → rule out some infeasible paths with fast lattices, and analyze the rest with slower predicates
- – ~~*stop*^{join} as coverage checking is hardcoded, which makes the checker miss real errors~~
- + ⭐ implemented several *stop* and *merge* operators available as options, and chose *stop*^{sep} and *merge*$^{pred-join}$

Configurable analysis

BLAST contains limited configurable analysis opportunities. Predicate-based analysis may be supplemented with lattice-based analysis, which has:

- \+ much greater speed
- – high rate of false posistives
- → rule out some infeasible paths with fast lattices, and analyze the rest with slower predicates
- – ~~$stop^{join}$ as coverage checking is hardcoded, which makes the checker miss real errors~~
- \+ ★ implemented several *stop* and *merge* operators available as options, and chose $stop^{sep}$ and $merge^{pred-join}$

★ Now lattice-supplemented analysis works 50% slower, but
★ it doesn't elicit errors

Performance evaluation

Linux Kernel 2.6.31.6, `lock()-unlock()` correctness checks, 2160 drivers.
Resource limit per driver: 15 minutes CPU time, 1 Gb virtual memory.

Measurement	Time spent	Failures	Bugs found	Time/mem
Old BLAST	105.8 hours	784	8	446
New BLAST	12.5 hours	368	42	152
Improvement	**8.5** times	**2** times	**5** times	**3** times

The actual speedup is greater than 8.5, because both slower and faster versions had “large” times substituted with 15 minutes (limit).

Conclusion and future work

Results of the work suggest that:

- BLAST was speeded up (more than 8.5 times on lock/unlock checking), and can now find more errors (5 times more)
- BLAST is not obsolete, and is capable to absorb new algorithms
- qualitative alias analysis boost is necessary to use it in real-world programs
- formula conversion overhead is negligible

Conclusion and future work

Results of the work suggest that:

- BLAST was speeded up (more than <u>8.5 times</u> on lock/unlock checking), and can now find more errors (<u>5 times</u> more)
- BLAST is not obsolete, and is capable to absorb new algorithms
- qualitative alias analysis boost is necessary to use it in real-world programs
- formula conversion overhead is negligible

Future work:

- improve memory constumption
- create a faster aliasing solution

Thank you

http://forge.ispras.ru/projects/ldv/
Download LDV tools
(the new BLAST will be included)

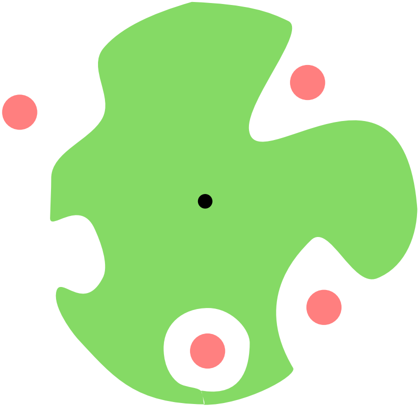
:-)

Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).

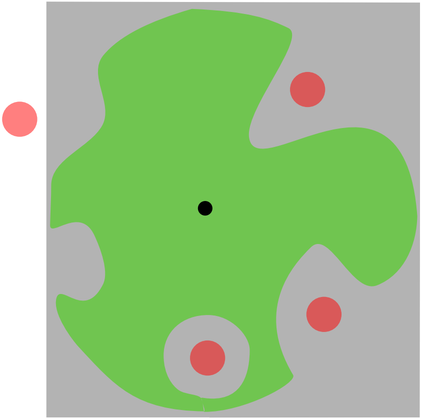
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an “abstraction” of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



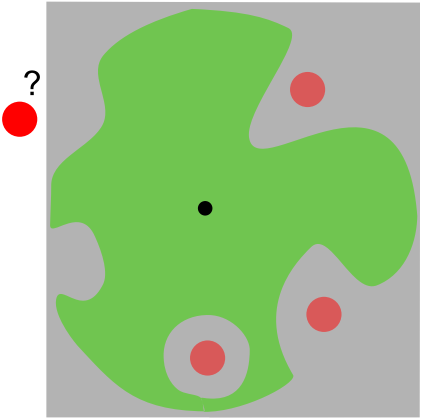
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



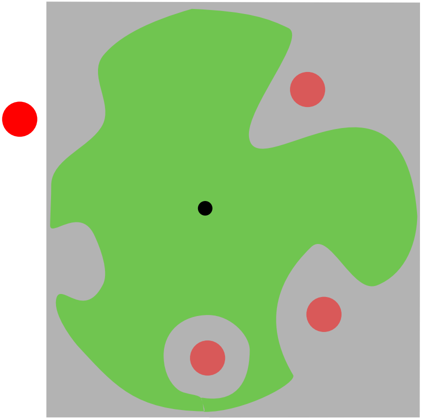
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an “abstraction” of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



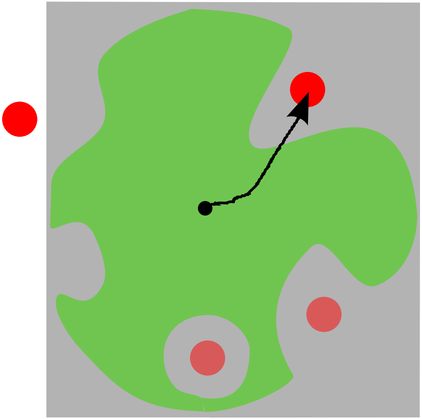
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



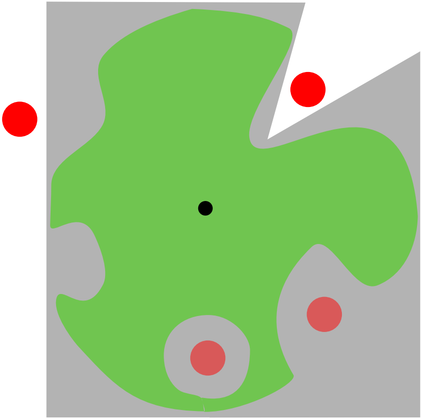
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an “abstraction” of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



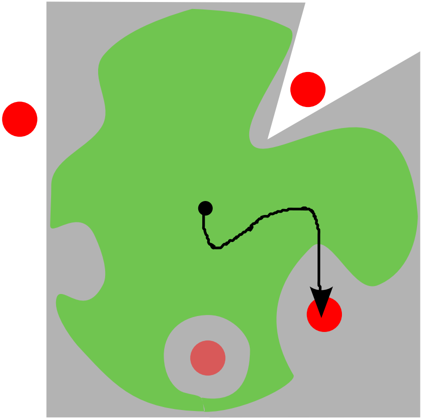
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



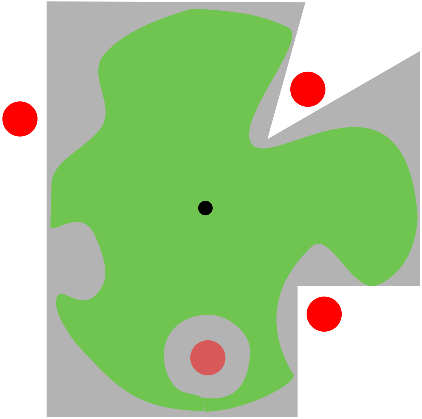
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an “abstraction” of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



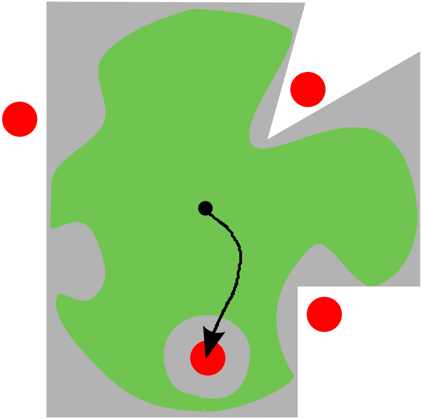
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



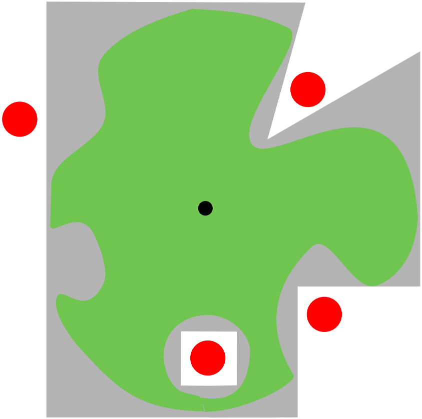
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



"Automatic" abstraction

BLAST's abstraction of a program is

- **predicate-based**: a location's region is predicate over variables, such as $x < 10$ or $(a < 5) \vee (a > 100)$
- **cartesian**: region of a location is a conjunction of small, "atomic" predicates
- **counterexample-based**: a set of predicates is discovered automatically from infeasible error traces

Optimization opportunity: explore independent paths concurrently. We didn't try, but [Lopez,Rybalchenko] achievments sound promising.