Checking Memory Safety with Blast*

Dirk Beyer[1], Thomas A. Henzinger[1,2], Ranjit Jhala[3], and Rupak Majumdar[4]

[1] EPFL, Switzerland
[2] University of California, Berkeley
[3] University of California, San Diego
[4] University of California, Los Angeles

Abstract. Blast is an automatic verification tool for checking temporal safety properties of C programs. Given a C program and a temporal safety property, Blast statically proves that either the program satisfies the safety property or the program has an execution trace that exhibits a violation of the property. Blast constructs, explores, and refines abstractions of the program state space based on lazy predicate abstraction and interpolation-based predicate discovery. We show how Blast can be used to statically prove memory safety for C programs. We take a two-step approach. First, we use CCured, a type-based memory safety analyzer, to annotate with run-time checks all program points that cannot be proved memory safe by the type system. Second, we use Blast to remove as many of the run-time checks as possible (by proving that these checks never fail), and to generate for the remaining run-time checks execution traces that witness them fail. Our experience shows that Blast can remove many of the run-time checks added by CCured and provide useful information to the programmer about many of the remaining checks.

1 Introduction

Invalid memory access is a major source of program failures. If a program statement dereferences a pointer that points to an invalid memory cell, the program is either aborted by the operating system or, often worse, the program continues to run with an undefined behavior. To avoid the latter, one can perform checks before every memory access at run time. For some programming languages (e.g., Java) this is done automatically by the compiler/run-time environment. For the language C, neither the compiler nor the run-time environment enforces memory-safety policies. CCured [7, 24] is a program-transformation tool for C which transforms any given C program to a memory-safe version. CCured uses a type-based program analysis to prove as many memory accesses as possible

* This research was supported in part by the NSF grants CCR-0234690, CCR-0225610, and ITR-0326577.

M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 2–18, 2005.

memory safe, and it inserts run-time checks before the remaining memory accesses. The resulting, "cured" C program is memory safe in the sense that it alarms the user if the program was about to execute an unsafe operation. Despite the manyfold advantages of this approach, it has two drawbacks: first, the run-time checks consume additional processor time, and second, the checks give late feedback, just before the program aborts.

We address these two points by combining CCured with a more powerful, path-sensitive program analysis. The additional analysis is performed by the model checker Blast [19]. For each memory access that the type-based analysis of CCured fails to prove safe, we invoke the more precise, more expensive analysis of Blast. There are three possible outcomes. First, Blast may be able to prove that the memory access is safe (even though CCured was not able to prove this). In this case, no run-time check needs to be inserted, thus reducing the overhead in the cured program. Second, Blast may be able to generate an execution trace to an invalid pointer dereference at the considered control location, i.e., an execution trace along which the run-time check inserted by CCured would fail. This may expose a program bug, which can, based on the error trace provided by Blast, then be fixed by the programmer. Third, Blast may time-out attempting to check whether or not a given memory access is always safe. In this case, the run-time check inserted by CCured remains in the cured program. It is important to note that Blast, even though often more powerful than CCured, is not invoked by itself, but only after a type-based pointer analysis fails. This is because where successful, the CCured analysis is more efficient, and it may also succeed in cases that overwhelm the model checker. However, the combination of CCured and Blast guarantees memory-safe programs with less run-time overhead than the use of CCured alone, and it provides useful compile-time feedback about memory-safety violations to the programmer.

Blast performs an abstract reachability analysis to check if a given error location of a C program can be visited during program execution. All paths of the program are checked symbolically and abstractly, by tracking only some relevant facts (called *predicates*) about program variables, instead of the full program state. If a path to the error location is found, the path may be due to the imprecision in the abstraction (a so-called *spurious* counterexample) or it may correspond to a feasible program path (a *genuine* counterexample). In the former case, additional relevant predicates are discovered automatically to remove the spurious error trace. The process is repeated, by tracking an increasing number of predicates, until either a genuine error trace (program bug) is found, or the abstraction is precise enough to prove the absence of error traces. This scheme of counterexample-guided predicate abstraction refinement was first implemented for verifying software by the Slam project [3]. Blast improves on the general scheme in two main ways. First, relevant predicates are discovered locally and independently at each program location as interpolants between the past and the future fragments of a spurious error trace [15]. Second, the discovered new predicates are added and tracked locally only in those parts of an abstract reachability tree where the spurious error trace occurred (*lazy abstraction*) [18]. This

emphasis on parsimonious, nonuniform abstractions renders the analysis scalable beyond 100,000 lines of code [15].

Much recent interest has focused on the addition of run-time checks to improve the memory safety and security of C programs [2, 12, 21], often coupled with a static analysis to reduce the run-time overhead by eliminating dynamic checks [4, 7, 14, 23, 26]. However, to our knowledge, model checking has not been used previously in the elimination of these run-time checks, even though the model checking of software has been a very active area of research in recent years [1, 3, 6, 8, 11, 13, 20, 22] (for more related work on software model checking, see [17]).

2 The Software Model Checker Blast

We illustrate how Blast combines lazy abstraction and interpolation-based, localized predicate discovery on the example shown in Figure 1.

Example Program. The program consists of three functions. Function `altInit` has three formal parameters: `size`, `pval1`, and `pval2`. It allocates and initializes a global array `a`. The size of the allocated array is given by `size`. The array is initialized with an alternating sequence of two values, pointed to by the pointers `pval1` and `pval2`. After the initialization is completed, the last value of the sequence is the value returned to the caller. Function `main` is a test driver for function `altInit`. It reads in an integer number from standard input and ensures that it gets a value greater than zero. Then it calls function `altInit` with the read value as parameter for the size as well as for the two initial values. Finally, the stub function `myscanf` models the behavior of the C library function `scanf`, which reads input values. The stub `myscanf` models arbitrary user input by returning a random integer value.

Control-Flow Automata. Internally, this program is represented by control-flow automata (CFA), one for each function of the program. A CFA is a directed graph, with locations corresponding to control points of the program (program-counter values), and edges corresponding to program operations. An edge between two locations is labeled by the instruction that executes when control moves from the source to the destination; an instruction is either a *basic block* of assignments, an *assume predicate* corresponding to the condition that must hold for control to go across the edge, a *function call* with call-by-value parameters (Blast also handles call-by-reference, but this is omitted from this exposition for simplicity), or a *return* instruction. Figures 2 and 3 show the control-flow automata for the functions `main` and `altInit`, respectively.

Memory Safety. We wish to prove that our program is memory safe, in particular, that there is no null-pointer dereference. In our example, we focus on one particular pointer dereference in the program: the dereference of the pointer `ptr` at the end of the function `altInit` (on line 19). We wish to prove that along all executions of the program, this pointer dereference is valid, that is, the value of `ptr` is not null. Notice that this property holds for our program: along every

```
#include <stdio.h>
#include <stdlib.h>
int *a;

void myscanf(const char* format, int* arg) {
  *arg = rand();
}

int altInit(int size, int *pval1, int *pval2){
 1:  int i, *ptr;
 2:  a = (int *) malloc(sizeof(int) * size);
 3:  if (a == 0) {
 4:    printf("Memory exhausted.");
 5:    exit(1);
 6:  }
 7:  i = 0;
 8:  while(i < size) {
 9:    i = i + 1;
10:    if (i % 2 == 0) {
11:      ptr = pval1;
12:    } else {
13:      ptr = pval2;
14:    }
15:    a[i] = *ptr;
16:    printf("%d. iteration", i);
17:  }
18:  if (ptr == 0) ERR: ;
19:  return *ptr;
}

int main(int argc, char *argv []){
20:  int *pval = (int *) malloc(sizeof(int));
21:  if (pval == 0) {
22:    printf("Memory exhausted.");
23:    exit(1);
24:  }
25:  *pval = 0;
26:  while(*pval <= 0) {
27:    printf("Give a number greater zero: ");
28:    myscanf("%d", pval);
29:  }
30:  return altInit(*pval, pval, pval);
}
```

Fig. 1. The example C program

execution path to line 19, the pointer `ptr` equals either `pval1` or `pval2`. Moreover, when `altInit` is called from `main`, the actual arguments passed to `pval1` and `pval2` are both `pval` (line 30). We have allocated space for `pval` in `main` (line 20), and we have already checked that the allocation succeeded (the test on line 21 and the code on lines 22–23 ensures that the program exits if `pval` is null). While the actual reason for correctness is simple, the example shows that the analysis to prove safety must be interprocedural and path-sensitive.

We have instrumented the program to check for this property (line 18), by checking whether the pointer `ptr` is null immediately before the dereference. In the next section, we will describe how such instrumentations are inserted automatically by a memory-safety analysis. With the instrumentation, the label `ERR` on line 18 is reached if and only if the pointer `ptr` is null and about to be dereferenced at line 19. In Figure 3 the error location with label `1#22` is depicted by a filled ellipse. We now describe how Blast checks that the label `ERR` (or

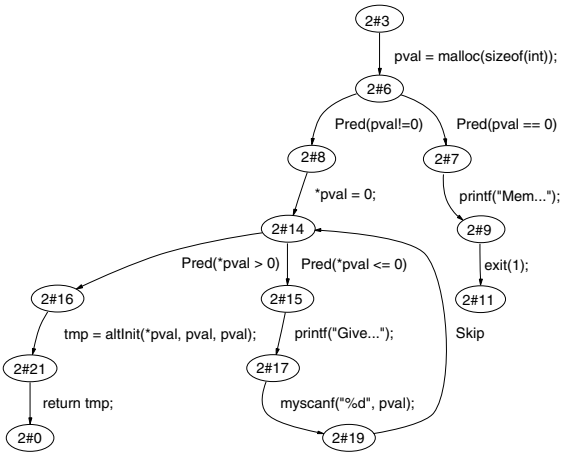

Fig. 2. Control-flow automaton for function `main`

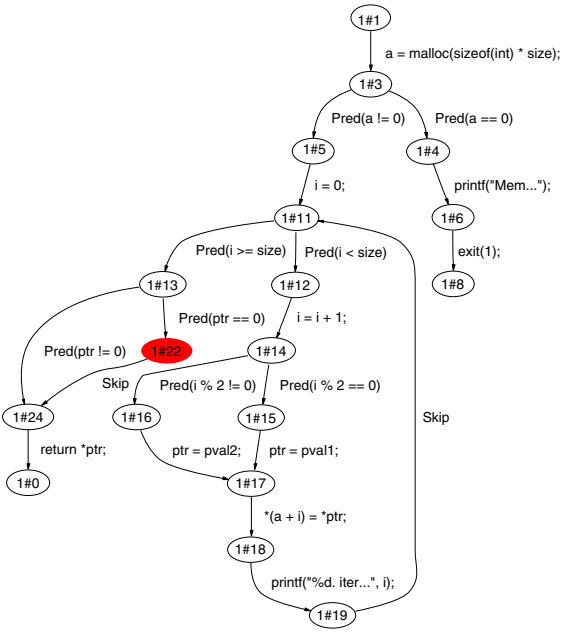

Fig. 3. Control-flow automaton for function `altInit`

equivalently, the location `1#22` of the CFA) is not reached along any execution of the program, and thus proves that the dereference on line 19 never fails.

Abstract Reachability Trees. In order to prove that the label ERR is never reached, Blast constructs an abstract reachability tree (ART). An ART is a labeled tree that represents a portion of the reachable state space of the program. Each node of the ART is labeled with a location of a CFA, the current call stack (a sequence of CFA nodes representing return addresses), and a boolean formula (called the *reachable region*) representing a set of data states. We denote a labeled tree node as $\mathtt{n} : (q, s, \varphi)$, where $\mathtt{n}$ is the tree node, q is the CFA node, s is the call stack, and φ is the reachable region. Each edge of the tree is marked with a basic block, an assume predicate, a function call, or a return. A path in the reachability tree corresponds to a program execution. The reachable region of a node describes an overapproximation of the reachable states of the program assuming execution follows the sequence of operations labeling the path from the root of the tree to the node.

Given a region (set of data states) φ and program operation (basic block or assume predicate) $\mathtt{op}$, let $post(\varphi, \mathtt{op})$ be the set of states reachable from φ by executing the operation $\mathtt{op}$. For a function call $\mathtt{op}$, let $post(\varphi, \mathtt{op})$ be the set of states reachable from φ by assigning the actual parameters to the formal parameters of the called function. For a return instruction $\mathtt{op}$ and variable $\mathtt{x}$, let $post(\varphi, \mathtt{op}, \mathtt{x})$ be the set of states reachable from φ by assigning the return value to $\mathtt{x}$. An ART is *complete* if (1) the root is labeled with the initial states of the program; (2) the tree is closed under postconditions, that is, for every internal node $\mathtt{n} : (q, s, \varphi)$ of the tree with $\varphi \neq \emptyset$,

(2a) if $q \xrightarrow{\mathtt{op}} q'$ is an edge in the CFA of q and $\mathtt{op}$ is a basic block or assume predicate, then there is a successor node $\mathtt{n}' : (q', s, \varphi')$ of $\mathtt{n}$ in the tree such that the edge $(\mathtt{n}, \mathtt{n}')$ is marked with $\mathtt{op}$ and $post(\varphi, \mathtt{op}) \subseteq \varphi'$,

(2b) if $q \xrightarrow{\mathtt{op}} q'$ is a CFA edge and $\mathtt{op}$ is a function call, then there is an $\mathtt{op}$-successor $\mathtt{n}' : (q'', s', \varphi')$ in the tree such that q'' is the initial location of the called function, the call stack s' results from pushing the return location q' together with the left-hand-side variable of the function call onto s, and $post(\varphi, \mathtt{op}) \subseteq \varphi'$,

(2c) if $q \xrightarrow{\mathtt{op}} q'$ is a CFA edge and $\mathtt{op}$ is a return instruction, then there is an $\mathtt{op}$-successor $\mathtt{n}' : (q'', s', \varphi')$ in the tree such that $(q'', \mathtt{x})$ is the top of the call stack s, the new call stack s' results from popping the top of s, and $post(\varphi, \mathtt{op}, \mathtt{x}) \subseteq \varphi'$;

and (3) for every leaf node $\mathtt{n} : (q, s, \varphi)$ of the tree, either q has no outgoing edge in its CFA, or $\varphi = \emptyset$, or there exists an internal tree node $\mathtt{n}' : (q, s, \varphi')$ such that $\varphi \subseteq \varphi'$. In the last case, we say that $\mathtt{n}$ is *covered* by $\mathtt{n}'$, as every program execution from $\mathtt{n}$ is also possible from $\mathtt{n}'$. A complete ART overapproximates the set of reachable states of a program. A complete ART is *safe* with respect to a CFA location q (the *error* location) if for every node $\mathtt{n} : (q, \cdot, \varphi)$ in the tree, we have $\varphi = \emptyset$. A complete ART that is safe for q serves as a certificate (proof) that q cannot be reached by any execution of the program [16].

Figure 4 shows a complete ART for our example program. We omit the call stack for clarity. Each node of the tree is labeled with a CFA node, and the

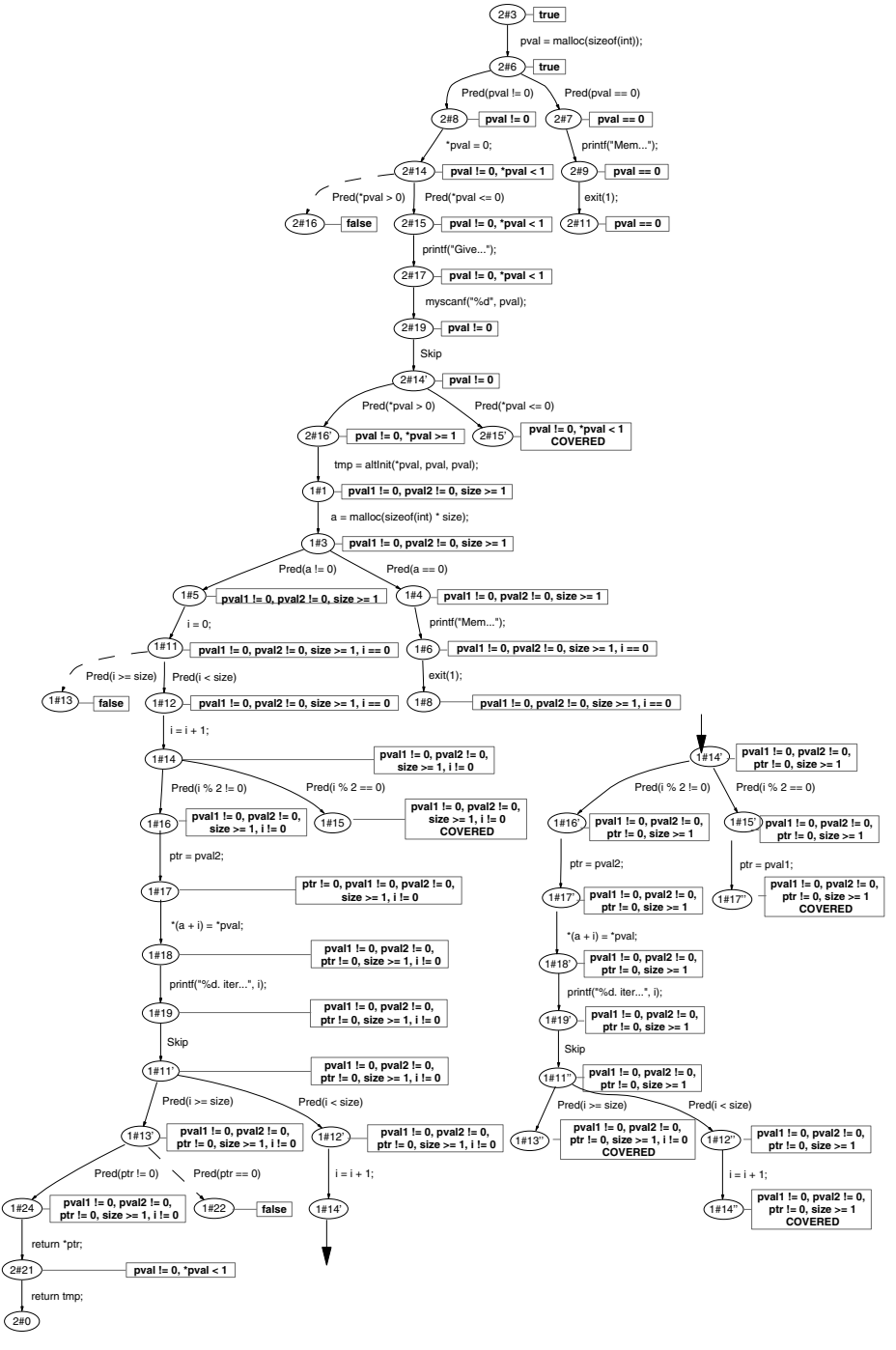

Fig. 4. Complete abstract reachability tree

reachable region is depicted in the associated rectangular box. The reachable region is the conjunction of the list of predicates in each box. Notice that some leaf nodes in the tree are marked "COVERED". Since this ART is safe for the error location `1#22`, this proves that `ERR` cannot be reached in the program. Notice that the reachable region at a node is an overapproximation of the concretely reachable states in terms of some suitably chosen set of predicates. For example, consider the edge $\texttt{1\#16} \xrightarrow{\texttt{ptr=pval2}} \texttt{1\#17}$ in the CFA. Starting from the region

$$\texttt{pval1} \neq 0 \wedge \texttt{pval2} \neq 0 \wedge \texttt{size} \geq 1 \wedge \texttt{i} \neq 0,$$

the set of states that can be reached by the assignment `ptr=pval2` is

$$\texttt{pval1} \neq 0 \wedge \texttt{pval2} \neq 0 \wedge \texttt{size} \geq 1 \wedge \texttt{i} \neq 0 \wedge \texttt{ptr} = \texttt{pval2}.$$

However, the tree maintains an overapproximation of this set of states, namely,

$$\texttt{pval1} \neq 0 \wedge \texttt{pval2} \neq 0 \wedge \texttt{size} \geq 1 \wedge \texttt{i} \neq 0 \wedge \texttt{ptr} \neq 0,$$

which loses the fact that `ptr` now contains the same address as `pval2`. This overapproximation is precise enough to show that the ART is safe for the location `1#22`. Overapproximating is crucial in making the analysis scale, as the cost of the analysis grows rapidly with increased precision. Thus, the safety-verification algorithm must (1) find an abstraction (a mapping of control locations to predicates) which is precise enough to prove the property of interest, yet coarse enough to allow the model checker to succeed, and (2) efficiently explore (i.e., model check) the abstract state space of the program.

Counterexample-Guided Abstraction Refinement. Blast solves these problems in the following way. It starts with a coarse abstraction of the state space and attempts to construct a complete ART with the coarse abstraction. If this complete ART is safe for the error location, then the program is safe. However, the imprecision of the abstraction may result in the analysis finding paths in the ART leading to the error location which are infeasible during the execution of the program. We call such paths spurious counterexamples. In this case, Blast refines the current abstraction by running a counterexample-analysis algorithm that determines whether the path to the error location is genuine (that is, there is a bug) or spurious. The counterexample-analysis algorithm uses an interpolation-based predicate-discovery algorithm which adds predicates locally to rule out spurious counterexamples [15]. For a given abstraction (mapping of control locations to predicates), Blast constructs the ART on-the-fly, stopping and running the counterexample analysis whenever a path to the error location is found in the ART. The refinement procedure refines the abstraction locally, and the search is resumed on the nodes of the ART where the abstraction has been refined. The parts of the ART that have not been affected by the refinement are left intact. This algorithm is called *lazy abstraction* [18]; we now describe how it works on our example.

Constructing the ART. Initially, Blast starts with no predicates, and attempts to construct an ART. The ART construction proceeds by unrolling the

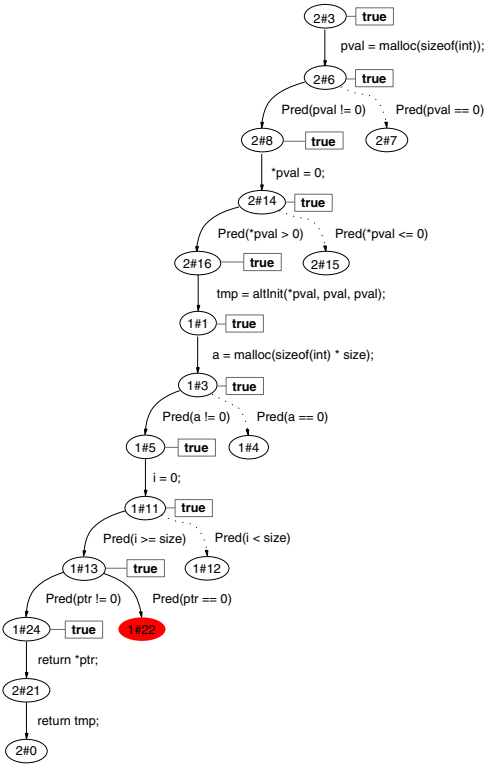

Fig. 5. Abstract reachability tree when the first spurious error path is found

CFAs and keeping track of the reachable region at each CFA node. We start with the initial location of `main`, with the reachable region *true* (which represents an arbitrary initial data state). For a tree node $\mathtt{n} : (q, s, \varphi)$, we construct successor nodes of `n` in the tree for all edges $q \xrightarrow{\mathtt{op}} q'$ in the CFA of q. The successor nodes are labeled with overapproximations of the set of states reachable from (q, s, φ) when the corresponding operations `op` are performed. To handle function calls and returns, BLAST implements a context-free reachability algorithm [25]. For our first iteration, since we do not track any facts (predicates) about variable values, all reachable regions are overapproximated by *true* (that is, the abstraction assumes that every data state is possible). With this abstraction, BLAST finds that the error location may be reachable. Figure 5 shows the ART when BLAST finds the first path to the error location. This ART is not complete, because some nodes have not been processed yet. In the figure, all nodes with incoming dotted edges (e.g., the node `2#7`) have not been processed. However, the incomplete ART already contains an error path from node `2#3` to `1#22` (the error node is depicted as a filled ellipse).

$$
\left.\begin{array}{l}
\langle \texttt{pval}, 1\rangle = \texttt{malloc}_0 \wedge \langle \texttt{pval}, 1\rangle \neq 0 \wedge \\
\langle *(\langle \texttt{pval}, 1\rangle), 1\rangle = 0 \wedge \langle *(\langle \texttt{pval}, 1\rangle), 1\rangle > 0 \wedge
\end{array}\right\} \text{ function } \texttt{main}
$$

$$
\left.\begin{array}{l}
\langle \texttt{size}, 1\rangle = \langle *(\langle \texttt{pval}, 1\rangle), 1\rangle \wedge \\
\langle \texttt{pval1}, 1\rangle = \langle \texttt{pval}, 1\rangle \wedge \\
\langle \texttt{pval2}, 1\rangle = \langle \texttt{pval}, 1\rangle \wedge
\end{array}\right\} \text{ formals assigned actuals}
$$

$$
\left.\begin{array}{l}
\langle \texttt{a}, 1\rangle = \texttt{malloc}_1 \wedge \langle \texttt{a}, 1\rangle \neq 0 \wedge \\
\langle \texttt{i}, 1\rangle = 0 \wedge \langle \texttt{i}, 1\rangle \geq \langle \texttt{size}, 1\rangle \wedge \\
\langle \texttt{ptr}, 1\rangle = 0
\end{array}\right\} \text{ function } \texttt{altInit}
$$

Fig. 6. Trace formula for the error path of Figure 5

Counterexample Analysis. At this point, BLAST invokes the counterexample-analysis algorithm which checks if the error path is feasible in the concrete program (i.e., the program has a bug), or whether it arises because the current abstraction is too coarse. To analyze the error path, BLAST creates a set of constraints (called the *trace formula*) which is satisfiable if and only if the path is feasible in the concrete program. The trace formula is built by transforming the error path to single-assignment form [10] (every variable is assigned a value at most once, which is achieved by introducing new variables) and then generating constraints for each operation along the path. For the error path of the example, the trace formula is given in Figure 6. Note that in this example, each program variable occurs only once at the left-hand-side of an assignment; if, for instance, the program variable `pval` were assigned a value twice along the path, then the result of the first assignment would be denoted by the new variable $\langle \texttt{pval}, 1\rangle$ and the result of the second assignment would be denoted by the new variable $\langle \texttt{pval}, 2\rangle$. The trace formula is unsatisfiable, and hence the error path is not feasible. There are several reasons why this path is not feasible. First, we set $*\texttt{pval}$ to 0 in `main`, and then take the branch where $*\texttt{pval} > 0$. Further, we check in `main` that $*\texttt{pval} > 0$, and pass $*\texttt{pval}$ as the argument `size` to `altInit`. Hence, $\texttt{size} > 0$. Now, we set `i` to 0, and then check that $\texttt{i} \geq \texttt{size}$. This check cannot succeed, because `i` is zero, while `size` is greater than 0. Thus, the path cannot be executed and represents a spurious counterexample.

Predicate Discovery. The predicate-discovery algorithm takes the trace formula and finds new predicates that must be added to the abstraction in order to rule out the spurious counterexample. New predicates are obtained at each location along the spurious error path using an interpolation procedure. For a pair of formulas φ^- and φ^+ such that $\varphi^- \wedge \varphi^+$ is unsatisfiable, a *Craig interpolant* ψ is a formula such that (1) the implication $\varphi^- \Rightarrow \psi$ is valid, (2) the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and (3) ψ only contains symbols that are common to both φ^- and φ^+. Given an appropriate logical theory, such interpolants always exist [9]. BLAST cuts the infeasible path at every location. At each cut point, the part of the trace formula corresponding to the path fragment up to the cut point is φ^-, and the part of the formula corresponding to the path fragment after the cut point is φ^+. Then, the interpolant at the cut point represents a formula over the live program variables that contains the reachable region after the path up

to the cut point is executed (by property (1)), and is sufficient to show that the rest of the path is unfeasible (by property (2)). The live program variables are represented by those new variables which occur both up to and after the cut point (by property (3)).

For example, consider the cut at location `2#16`. For this cut, φ^- is

$$\langle \texttt{pval}, 1\rangle = \texttt{malloc}_0 \wedge \langle \texttt{pval}, 1\rangle \neq 0 \wedge \langle *(\langle \texttt{pval}, 1\rangle), 1\rangle = 0 \wedge \langle *(\langle \texttt{pval}, 1\rangle), 1\rangle > 0,$$

and φ^+ is

$$\langle \texttt{size}, 1\rangle = \langle *(\langle \texttt{pval}, 1\rangle), 1\rangle \wedge \langle \texttt{pval1}, 1\rangle = \langle \texttt{pval}, 1\rangle \wedge \langle \texttt{pval2}, 1\rangle = \langle \texttt{pval}, 1\rangle \wedge$$
$$\langle \texttt{a}, 1\rangle = \texttt{malloc}_1 \wedge \langle \texttt{a}, 1\rangle \neq 0 \wedge \langle \texttt{i}, 1\rangle = 0 \wedge \langle \texttt{i}, 1\rangle \geq \langle \texttt{size}, 1\rangle \wedge \langle \texttt{ptr}, 1\rangle = 0.$$

The only common symbol across the cut is $\langle *(\langle \texttt{pval}, 1\rangle), 1\rangle$, and the interpolant is $\langle *(\texttt{pval}, 1), 1\rangle \geq 1$. Relating the new variable $\langle *(\texttt{pval}, 1), 1\rangle$ back to the program variable $*\texttt{pval}$, this suggests that the fact $*\texttt{pval} \geq 1$ suffices to prove the error path infeasible. This predicate is henceforth tracked at location `2#16`. Similarly, at nodes `1#1`, `1#3`, and `1#5`, Blast discovers that the predicate $\texttt{size} \geq 1$ is useful, and at location `1#11`, the predicates $\texttt{size} \geq 1$ and $\texttt{i} = 0$ are found. After adding these predicates, Blast refines the ART, now tracking the truth or falsehood of the newly found predicates at the locations where they are useful.

Refining the ART. When Blast refines the ART with the new abstraction, it only reconstructs subtrees that are rooted at nodes where new predicates have been added. In the example, a second error path is found; Figure 7 shows the ART when this happens. Notice that this time, the reachable regions are not all *true*; instead they are overapproximations, at each node of the ART, of the reachable data states in terms of the predicates that are tracked at the node. For example, the reachable region at the first occurrence of location `2#14` in the ART is $*\texttt{pval} < 1$ (the negation of the tracked predicate $*\texttt{pval} \geq 1$), because $*\texttt{pval}$ is set to 0 when going from `2#8` to `2#14`, and $*\texttt{pval} < 1$ is the abstraction of $*\texttt{pval} = 0$ in terms of the tracked predicates. This more precise reachable region disallows certain CFA paths from being explored. For example, again at the first occurrence of location `2#14`, the ART has no left successor with location `2#16`, because no data state in the reachable region $*\texttt{pval} < 1$ can take the program branch with the condition $*\texttt{pval} > 0$ (recall that $*\texttt{pval}$ is an integer).

On the second error path, the counterexample analysis discovers the new predicates $\texttt{pval} = 0$, $\texttt{pval2} = 0$, and $\texttt{ptr} = 0$. In the next iteration, Blast finds a third error path, shown in Figure 8, for which it finds the predicate $\texttt{pval1} = 0$.

With these predicates, Blast constructs the complete ART shown in Figure 4. Since this tree is safe for the error location `1#22`, this proves that `ERR` can never be reached by executing the program. Note that some leaf nodes in the tree are covered: as no new states can be reached by exploring states from covered nodes, Blast stops the ART construction at such nodes, and the whole process terminates.

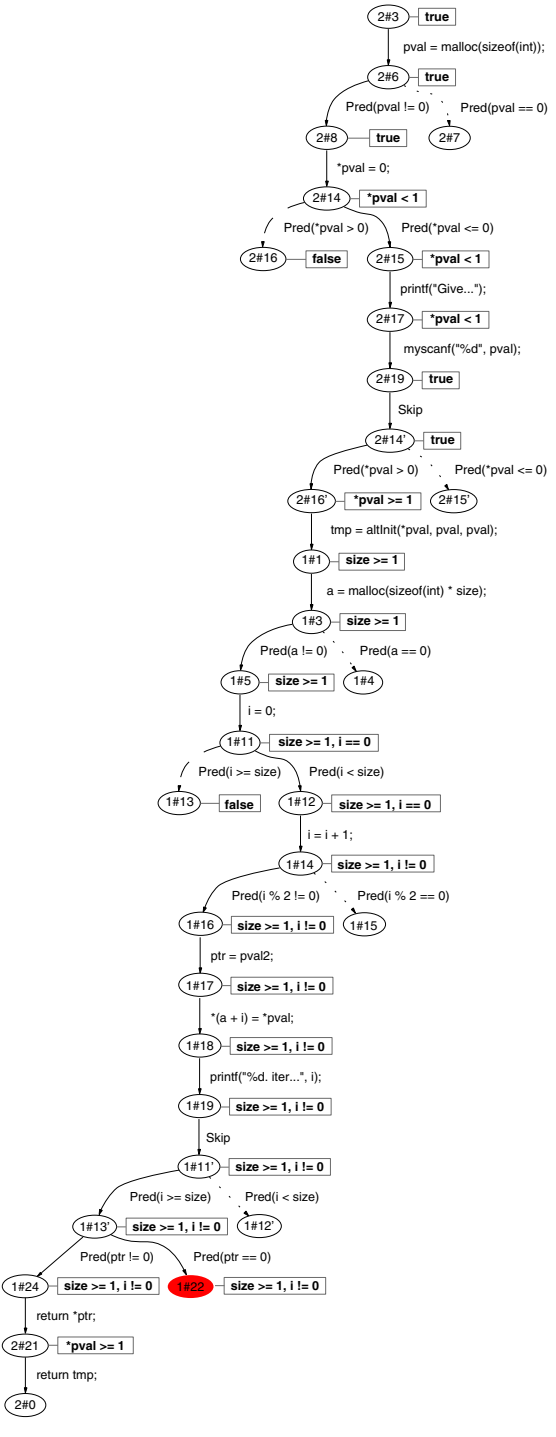

Fig. 7. Abstract reachability tree when the second spurious error path is found

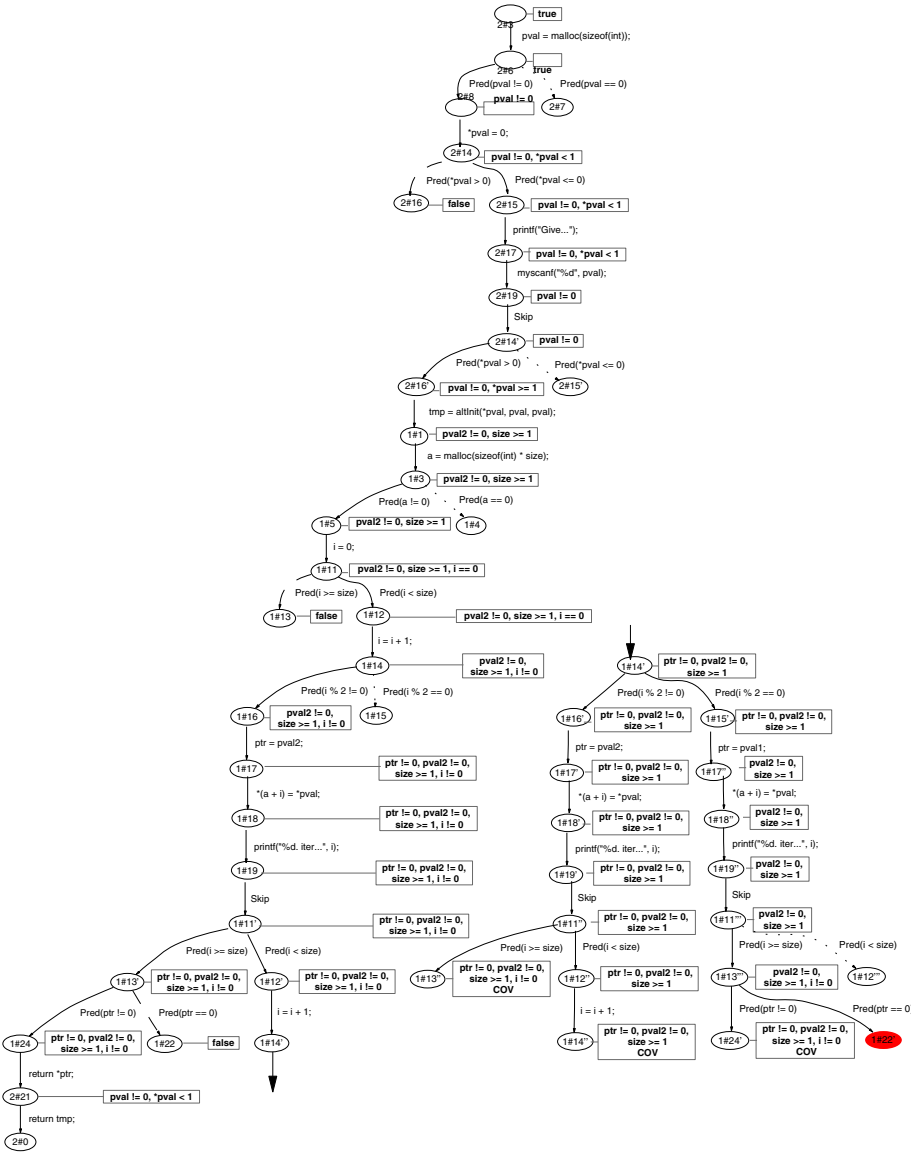

Fig. 8. Abstract reachability tree when the third spurious error path is found

3 Checking Memory Safety

A program is *memory safe* if it only accesses memory addresses within the bounds of the objects it has allocated or to which it has been granted access. Memory safety is a fundamental correctness requirement for most applications. We consider one particular aspect of memory safety: null-pointer dereferencing. Pointers

in C programs can be null (i.e., not pointing to a valid address), or point to an allocated object. Dereferencing a null pointer can cause an arbitrary value to be read, or the program to crash with a segmentation fault.

The absence of null-pointer dereferences is a safety property. In principle, we can annotate every dereference operation in the program with a check that the dereferenced pointer is not null, and run Blast on the annotated program to verify that no such check fails. However, this strategy does not scale well. First, many accesses can be proved memory safe using an inexpensive type-based approach, and using an expensive analysis like Blast is overkill. Second, each annotation should be checked independently, so that the abstractions required to prove each annotation do not interfere and result in a large state space. Therefore, we use CCured [7, 24], a type-based memory-safety analysis, to classify the pointers according to usage and annotate the program with run-time checks. CCured analyzes C programs with respect to a sound type system which ensures that well-typed programs are memory safe. When the type system cannot prove that a pointer variable is always used safely, CCured inserts run-time checks in the program which monitor correct pointer usage at execution time. In particular, each dereference of a potentially unsafe (i.e., not proved safe by the type system) pointer is annotated with a check that the pointer is non-null. The run-time checks abort the program safely, instead of running into undefined configurations. However, each run-time check constitutes overhead at execution time, and CCured implements many optimizations that remove redundant run-time checks based on simple data-flow analyses. Typically, the CCured optimizations remove over 50% of the run-time checks inserted by the type system, and the optimized programs run within a factor of two of their original execution time. We wish to check how many of the remaining run-time checks can be removed by the more sophisticated analysis implemented in Blast.

Specifically, for each potentially unsafe pointer dereference `*p` in the program, CCured introduces a call `__CHECK_NULL(p)` which checks that the pointer `p` is non-null. The function `__CHECK_NULL` terminates the program if its argument is null, and simply returns if the argument is non-null. Thus, if the actual argument `p` at a call site is non-null along all execution paths, then this function call can be removed without affecting the behavior of the program. To check if a call to `__CHECK_NULL` can be removed from the program, Blast does the following. First, it replaces the call to `__CHECK_NULL` with a call to `__BLAST__CHECK_NULL` with the same argument, where `__BLAST__CHECK_NULL` is the following function:

```
void __BLAST__CHECK_NULL(void *p) {
  if (!p) { __BLAST_ERROR: ; }
}
```

Second, Blast checks if the location labeled with `__BLAST_ERROR` is reachable. Both steps are performed independently for each call to `__CHECK_NULL` in the program body. Each call of Blast has three possible outcomes.

The first outcome is that BLAST reports that the label `__BLAST_ERROR` is not reachable. In this case, the function call can be removed, since the corresponding check will not fail at run time.

The second possible outcome is that BLAST produces an error trace that gives a program execution in which `__BLAST__CHECK_NULL` is called with a null argument, which indicates a situation where the run-time check fails. In this case, the check must remain in the program to terminate the program safely should the check fail. This may also indicate a program error, in which case the feedback provided by BLAST (the error trace) provides useful information for fixing the bug. We often encountered error traces of the form that the programmer forgot to check the return value of `malloc`: if the memory allocation fails, then the next dereference of the pointer is unsafe. BLAST assumes that `malloc` may return a null pointer and discovers the problem. However, not every error trace found by BLAST necessarily indicates a program error, because BLAST makes conservative assumptions about library functions.

There is a third possible outcome, namely, that BLAST fails to declare whether the considered run-time check is superfluous or necessary, due to time or space limitations. In this case, we say that BLAST *fails*, and we will provide the failure rate for the experiments below. If BLAST fails on a run-time check, then the check must of course remain in the program. Notice that by changing each call to `__CHECK_NULL` separately, BLAST checks if a run-time check is necessary independently from all other checks. These checks can be run in parallel and often lead to different program abstractions.

We ran our method on several examples. The first seven programs are from the Olden v1.0 benchmark suite [5]. We included the programs for the Bitonic Sort algorithm (bisort), the Electromagnetic Problem in Three Dimensions (em3d), the Power Pricing problem (power), the Tree Add example (treeadd), the Traveling Salesman problem (tsp), the Perimeters algorithm (perimeter), and the Minimum Spanning Tree problem (mst). Finally, we processed the scheduler for Unix systems fcron, version 2.9.5, and the Lisp interpreter (li) from the Spec95 benchmark suite. We ran BLAST on each run-time check inserted by CCURED separately, and fixed a time-out of 200 s for each check; that is, a run of the model checker is stopped after 200 s with *failure*, and the studied run-time check is conservatively declared necessary.

Table 1 presents the results of our experiments. The first column lists the program name, the second and third columns give the number of lines of the original program ("LOC orig.") and of the instrumented program after preprocessing and CCURED instrumentation ("LOC cured"). The three columns of "run-time checks" lists the number of run-time checks inserted by the CCURED type system (column "inserted"), the number of remaining checks after the CCURED optimizer removes redundant checks (column "optim."), and finally the number of remaining checks after BLAST is used to remove run-time checks (column "BLAST"). The column "proved safe by BLAST" is the difference between the "optim." and "BLAST" columns: it shows the number of checks remaining after the CCURED optimizer which BLAST proves will never fail.

Table 1. Verification of run-time checks

Program	LOC		run-time checks			proved safe	potential
	orig.	cured	inserted	optim.	Blast	by Blast	errors found
bisort	684	2,510	51	21	6	15	6
em3d	561	2,831	33	20	9	11	9
power	763	2,891	149	24	24	0	24
power-fixed	763	2,901	149	24	24	12	12
treeadd	370	2,246	11	7	6	1	6
tsp	565	2,560	93	59	44	15	4
perimeter	395	2,292	49	18	8	10	5
mst	582	2,932	54	34	19	15	18
fcron 2.9.5	11,994	38,080	877	455	222	233	74
li	6,343	39,289	1,715	915	361	554	11

The remaining checks, which cannot be removed by Blast, fall into two categories. First, the column "potential errors found" lists the number of checks for which Blast found an error trace leading to a violation of the run-time check; those are potential bugs and the error traces give useful information to the programmer. For example, we took the program with the most potential errors found, namely power, and analyzed its error traces. In many of them, a call to `malloc` occurs without a check whether there is enough memory available. So we inserted after each call to `malloc` a null-pointer check to ensure that the program execution does not proceed in such a case. Analyzing the fixed program (with null-pointer checks inserted after each `malloc`), we can remove 12 more run-time checks. To give an example of the performance of Blast, in the case of power-fixed, the cured program was checked in 15.6 s of processor time on a 3 GHz Linux machine.

Second, the difference between the columns "Blast" and "potential errors found" gives the number of run-time checks on which the model checker fails (times out) without an answer. The number of these failures is not shown explicitly in the table; it is zero for the first five programs. Since Blast gives no information about these checks, they must remain in the program.

Acknowledgments. We thank George Necula and Matt Harren for help with CCured.

References

1. T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proc. CAV*, LNCS 3114, pages 484–487. Springer, 2004.
2. T.M. Austin, S.E. Breach, and G.S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. PLDI*, pages 290–301. ACM, 1994.
3. T. Ball and S.K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

4. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proc. PLDI*, pages 321–333. ACM, 2000.
5. M.C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed Memory Machines*. PhD thesis, Princeton University, 1996.
6. S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Engineering*, 30:388–402, 2004.
7. J. Condit, M. Harren, S. McPeak, G.C. Necula, and W. Weimer. CCured in the real world. In *Proc. PLDI*, pages 232–244. ACM, 2003.
8. J.C. Corbett, M.B. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. ICSE*, pages 439–448. ACM, 2000.
9. W. Craig. Linear reasoning. *J. Symbolic Logic*, 22:250–268, 1957.
10. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadek. Efficiently computing static single-assignment form and the program dependence graph. *ACM Trans. Programming Languages and Systems*, 13:451–490, 1991.
11. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL*, pages 174–186. ACM, 1997.
12. R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. USENIX*, pages 125–136, 1992.
13. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2:72–84, 2000.
14. F. Henglein. Global tagging optimization by type inference. In *Proc. LISP and Functional Programming*, pages 205–215. ACM, 1992.
15. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
16. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. CAV*, LNCS 2404, pages 526–538. Springer, 2002.
17. T.A. Henzinger, R. Jhala, R. Majumdar, and M.A.A. Sanvido. Extreme model checking. In *International Symposium on Verification: Theory and Practice*, LNCS 2772, pages 332–358. Springer, 2003.
18. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
19. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proc. SPIN*, LNCS 2648, pages 235–239. Springer, 2003.
20. G.J. Holzmann. *The* Spin *Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
21. S. Kaufer, R. Lopez, and S. Pratap. Saber-C: An interpreter-based programming environment for the C language. In *Proc. USENIX*, pages 161–171, 1988.
22. M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. OSDI*. USENIX, 2002.
23. G.C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proc. LICS*, pages 93–104. IEEE, 1998.
24. G.C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. POPL*, pages 128–139. ACM, 2002.
25. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL*, pages 49–61. ACM, 1995.
26. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proc. POPL*, pages 132–143. ACM, 1977.