Structural Abstraction of Software Verification Conditions *

Domagoj Babić and Alan J. Hu

Department of Computer Science
University of British Columbia

Abstract. Precise software analysis and verification require tracking the exact path along which a statement is executed (path-sensitivity), the different contexts from which a function is called (context-sensitivity), and the bit-accurate operations performed. Previously, verification with such precision has been considered too inefficient to scale to large software. In this paper, we present a novel approach to solving such verification conditions, based on an automatic abstraction-checking-refinement framework that exploits natural abstraction boundaries present in software. Experimental results show that our approach easily scales to over 200,000 lines of real C/C++ code.

1 Introduction

Verification conditions (VCs) are logical formulas, constructed from a system and desired correctness properties, such that the validity of verification conditions corresponds to the correctness of the system. Proving validity of verification conditions is an essential step in software verification, and is the focus of this paper.

In general, proving software VCs requires interprocedural analysis, e.g. of the propagation of data-flow facts. Some properties, like proper nesting of lock-unlock calls, tend to be localized to a single function and are amenable to simpler analysis. Many others, especially pointer-related properties, tend to span through many function calls.

To handle the complexity of interprocedural analysis, the software analysis community has developed a number of increasingly expensive abstractions. For instance, path-insensitive analysis does not track the exact path along which a certain statement is executed, while context-insensitive analysis does not differentiate the contexts from which a function is called. These abstractions work well in optimizing compilers, but are not precise enough for verification purposes. Software verification analysis has to be both path- and context-sensitive (*-sensitive) to keep the number of false errors low.

Precise *-sensitive software verification has two components: (1) we need an analysis that takes a piece of software as input and computes VCs as logical formulas in some logic, and (2) once the VCs are computed, we need to check their validity. This paper proposes a novel approach to checking the validity of *-sensitive VCs.

Our approach is an abstraction-checking-refinement framework that exploits the natural function-level abstraction boundaries present in software. Programmers organize code into functions and use them as abstractions. They tend to ignore the details of

* Research supported by a Microsoft Graduate Fellowship and an NSERC Discovery Grant.

the effects of the function on the caller's context — the easiest invariant to remember is to remember no invariant at all. Analogously, our approach initially treats individual effects of a function call as unconstrained variables and incrementally adds constraints corresponding to the effects of the function call. We demonstrate that such a structural refinement approach works well, even on large general-purpose C/C++ applications.

1.1 Related Work

Interprocedural analysis can have many forms, and is commonly based on some form of summarization. Usually, the more expressive the summaries are, the higher the computational complexity. For instance, if the set of data-flow facts is a finite set, and the data-flow functions distribute over the confluence, interprocedural data-flow analysis can be done in polynomial time [21]. If the summaries are composed of predicates over arbitrary logic the analysis gets more complex, depending on the underlying logic.

If the number of predicates is relatively small, predicate abstraction [14] makes it possible to represent summaries compactly as BDDs [5]. This approach has been effectively used in SLAM [3] and BLAST [15, 16]. Predicate abstraction is very coarse, and hasn't been shown to scale well to large applications for data-intensive properties. Its advantage is that the VCs given to the theorem prover are relatively simple, corresponding to a conjunction of conditions on some path in the program. Saturn [25] handles lock-properties in a similar way — by computing summaries as projections onto a set of predicates, with the difference that it does not abstract VCs before passing them to the theorem prover. In contrast to the above-mentioned approaches, the technique presented in this paper allows summaries to be arbitrary expressions, rather than just projections onto a set of predicates.

Livshits and Lam [20] proposed a path- and context-sensitive points-to analysis and used it for simple security checks. Their summaries represent definition-use chains required for tracking pointers interprocedurally. They demonstrated their analysis on small programs up to 13 thousand lines of code. Whaley and Lam [23] stressed the importance of context-sensitive analysis and proposed a brute force approach to context-sensitive, inclusion-based pointer alias analysis. Their analysis, implemented in the `bddbdd` system, represents input-output relations as BDDs [5]. The BDD-based approach seems to work well for tracking a set of locations, but it is not applicable to verification of assertions because BDDs are known to suffer from exponential blow-up on multiplication, division, and barrel shifters — all frequent operations in software. Both works focused on the software analysis side, while our focus is on proving *-sensitive VCs. We believe that our results could improve the scalability of their approach.

The CBMC tool [7, 6] verifies C programs, to bounded depth, with bit-accuracy and *-sensitivity. The approach is direct symbolic execution of the C into a SAT instance, unrolling all loops and inlining all function calls, so solving the generated VC is the performance bottleneck for large software. Our results address that bottleneck.

In the domain of programs limited to static memory allocation, Astrée [4] has been successfully applied to verification of mission-critical software systems. Although context-sensitive over the chosen abstract domain(s), Astrée was designed for systems that contain no goto statements, no dynamic memory allocation, no recursive calls, no

recursive data structures, and no pointer arithmetic. Since our focus is verification of assertions in general purpose software, these constraints were not acceptable.

Context-sensitivity is only one component of the problem. Path-sensitivity is the other. The BLAST and SLAM software model checkers enumerate paths one-by-one, hoping that refinement will refute many paths with each added predicate. For each path, the model checker constructs an abstracted theorem prover query, which can correspond to a path that spans through many functions. Such path enumeration during the abstraction-checking-refinement loop seems wasteful — SAT solvers are extremely efficient in path enumeration and refutation of infeasible paths, so we believe that path enumeration should be left to the SAT solver.

Others have realized the importance of letting the theorem prover enumerate the paths as well. For instance, software verification systems like Boogie [18] and ESC/Java [12] do construct a single formula and let the theorem prover enumerate the paths. However, these systems rely on the user to provide interface abstractions, and do not attempt to abstract the formulas before calling the theorem prover.

Our approach to proving *-sensitive VCs merges both SAT-solver-based path enumeration and abstraction, yielding a precise, but practically efficient alternative to previous methods.

2 A Review of Verification Condition Generation

Traditionally, VCs are computed by Dijkstra's weakest precondition transformer [10], as is done for example in ESC/Java [12] and Boogie [18]. A naïve representation of VCs computed by the weakest precondition can be exponential in the size of the code fragment being checked, but this blow-up can be avoided by the introduction of fresh variables to represent intermediate expressions [22, 13, 19]. Here, we give a quick overview of weakest-precondition-based VC computation to illustrate the process, some common problems, and an efficient representation.

Consider the following simple program (modified from [19]):

```
S1: if (x < 0) { y = -2*x - y; }
S2: y = x + y;
S3: assert(0 <= y);
```

The VC can be computed as the weakest liberal precondition $\mathsf{wlp}()$ of a sequential composition of those three statements with respect to true, giving:

$$\begin{aligned}
\mathsf{wlp}(S1;S2;S3,\mathsf{true}) &= \mathsf{wlp}(S1,\mathsf{wlp}(S2,\mathsf{wlp}(S3,\mathsf{true}))) && (1)\\
&= \mathsf{wlp}(S1,\mathsf{wlp}(S2,0 \le y)) && (2)\\
&= \mathsf{wlp}(S1,0 \le x+y) && (3)\\
&= 0 \le \mathit{ITE}(x<0,-(x+y),x+y) && (4)
\end{aligned}$$

where *ITE* is the if-then-else operator. Obviously, continuous application of $\mathsf{wlp}()$ can lead to exponential blowup in the size of the formula. To avoid the blowup, we can perform renaming, which guarantees a single point of definition for each variable (as in Single Static Assignment (SSA) form [9]):

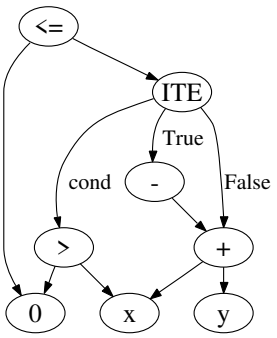

Fig. 1. Graph Representation of the Verification Condition. Non-leaf nodes are labeled with operators; leaf nodes, with variables and constants. Operator nodes are connected to their operands by edges.

```
S1: if (x0 < 0) { y1 = −2∗x0 − y0; } else { y1 = y0; }
S2: y2 = x0 + y1;
S3: assert(0 <= y2);
```

Since each variable has a single point of definition, assignments can be replaced with equivalences (*passive commands* in [19]), and then $\mathsf{wlp}(S1;S2;S3, 0 \leq y_2)$ boils down to:

$$(x_0 < 0 \Rightarrow (y1 \equiv -2x_0 - y_0)) \wedge (x_0 \geq 0 \Rightarrow (y_1 \equiv y_0)) \wedge (y_2 \equiv x_0 + y_1) \wedge (0 \leq y_2)$$

Exponential blowup is avoided at the expense of introduction of fresh variables.

The same VC can be represented in the form of a graph. In particular, we simply represent a logical formula as a directed, acyclic graph, in which non-leaf nodes are labeled with operators, their children are their operands, and the leaves are labeled with variables or constants. A graph representation of a logical formula such that all common subexpression nodes have been merged will be called a maximally-shared graph. Figure 1 depicts a maximally shared graph representation of the computed VC in Eq. 4. The advantage of using maximally-shared graphs for VC representation is that the elimination of common subexpressions is simple, while the graph is still linear in the size of the code fragment.

The work in this paper is to support our static checker CALYSTO, which is being designed to be a general-purpose, bit-precise assertion checker. CALYSTO implements an efficient interprocedural symbolic execution algorithm [1] that converts SSA (computed using the LLVM compiler framework [17]) into function summaries and VCs in the form of acyclic maximally-shared graphs. For each location that a function modifies, CALYSTO computes the resulting expression in terms of the function inputs (including globals). Each such expression is represented as a separate summary expression, which gives fine-grained control during the refinement process (Sec. 3.3). Like other static checkers, CALYSTO makes a few unsound approximations. For example, loops are unrolled a fixed number of times, with the additional assumption that the loop test fails at the loop exit, as is done in ESC/Java [12], Saturn [25], and older versions of Boogie [18]. We could also handle loops soundly by using loop invariants (computed by any

technique), as is done in Boogie. Similarly, CALYSTO handles non-constant array indices by unsoundly replacing them with constant indices. In addition, CALYSTO makes the unsound assumption that pointers passed as function parameters are not aliased, as in [20, 25]. However, CALYSTO's computed VCs are *-sensitive, fully bit-accurate, and support all standard operators (e.g., signed/unsigned division and multiplication on bit-vectors, etc.), except that floating-point arithmetic is not yet implemented.

3 Exploiting Natural Abstraction Boundaries

We begin with an example that provides intuition about how our approach solves *-sensitive VCs. The code used in the example is a simplified and slightly modified piece of code from a real application.[1] To prove an assertion, we need to prove either that the assertion itself is unreachable, or that it always evaluates to true. Through the example, we shall follow a sequence of steps needed to prove the assertion on line 22.

```
1   int global1, global2;
2
3   // If *data<0, returns true and computes *data=abs(*data).
4   bool flip(int *data) {
5       if (*data < 0) {
6           *data = -(*data);
7           return true;
8       }
9       return false;
10  }
11
12  // Assume init is a pure function (no side-effects).
13  int init(int x) {
14      // Some expensive computation...
15  }
16
17  // If global1 is positive and global2 is negative, scales
18  // global1 by abs(global2).
19  void scale() {
20      global2 = init(global1);
21      if (flip(&global2)) {
22          assert(global2 != 0); // Div by zero.
23          global1 /= global2;
24      }
25  }
```

As mentioned earlier, the symbolic execution will compute a graph representing each effect of each function in terms of its parameters (and globals). For example, the function `flip` has two effects: a boolean return value and its effect on the location pointed to by its parameter. At the caller's side, the symbolic execution initially denotes effects of a function call by a placeholder operator node. For example, the return value

[1] The example is modified from our modular arithmetic theorem prover SPEAR.

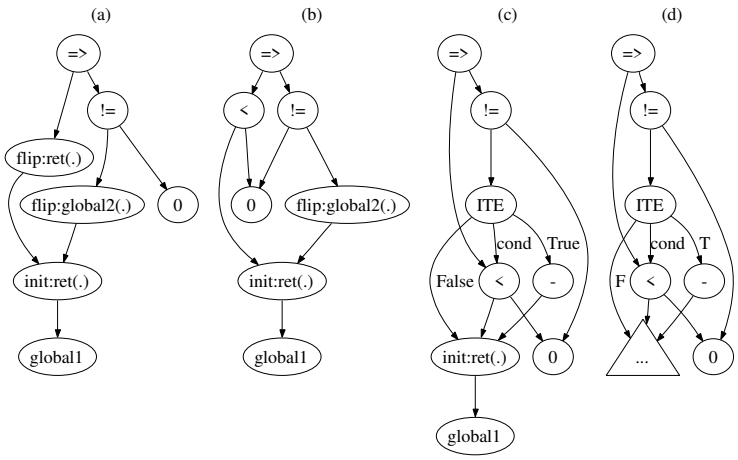

Fig. 2. Sequence of Refinements of the Computed VC. Summary nodes are structurally refined in the following sequence: `flip:ret`, `flip:global2`, and finally `init:ret`. The subgraph obtained by the refinement of `init:ret` is represented by a triangle. For simplicity, these figures do not show pointer references and dereferences.

of a call to `flip` will be an operator node labeled $flip : ret$ whose child is the argument to `flip`.

The VC will be an implication: if line 22 is reachable, then the asserted condition must hold. Let us ponder the structure of the computed VC. The antecedent contains two nested function calls. The consequent is a simple comparison of zero with the effect of `flip` on the global variable. Observe that the expression is written in terms of the initial values of all involved variables, facilitating common subexpression elimination by simple graph rewriting. Graphically, the VC can be represented as a maximally-shared graph (Fig. 2a). Summaries of the individual effects of each called function are at first represented as unconstrained fresh variables. Those nodes will be called summary nodes. Interpretation of a summary node corresponds to replacing the node with a node that represents the summarized expression. Such expansion corresponds to a round of inlining.

To be fully context-sensitive, the obvious approach is to completely inline all calls. Such inlining leads to exponential blow-up even on small applications. We found that aggressive inlining of non-recursive function calls works only on several very small applications, resulting in roughly 50-180X increase in the size of the code.

A better approach is to track the individual effects of a function separately. This fine-grained approach makes it possible to expand only the slice of the called function that is actually in the cone of influence of the verified property. We consider this approach to be the state of the art and shall use it as the base case for comparison with our abstraction-based approach in Sec. 4. Together with the common subexpression elimination, this approach is more scalable, but does not offer satisfactory performance.

The crux of the problem is that interprocedural analysis can't decide when to stop inlining. After only three refinements, *-sensitive analysis would expand computation-

ally expensive `init`, rendering the problem much harder for the decision procedure. However, the VC can be proved to be valid after only two refinements

$$\begin{array}{l} \text{Let } x = \mathit{init} : \mathit{ret}(\mathit{global1}) \\ x < 0 \Rightarrow \mathit{ITE}(x < 0, -x, x) \neq 0 \end{array}$$

which simplifies to true, no matter what `init` returns. Cases like this appear frequently in practice, especially during *-sensitive verification of data-intensive properties, like checking of assertions or global pointer properties.

Our approach gradually refines the maximally-shared graph, until the VC becomes valid, or the decision procedure finds a falsifying assignment that does not depend on any summary nodes. The rest of this section gives the details of our approach.

3.1 Algorithm Overview

The proposed approach follows the general paradigm of automatic, counter-example-guided, abstraction refinement [8], but unlike typical CEGAR approaches, our abstraction and refinement operations are entirely structural, and the refinement works incrementally on abstract counterexamples (rather than concretizing the abstract counterexample, proving it spurious, and then analyzing the proof). Locations modified by a function call (either indirectly through a pointer, or directly via returned values) are initially considered to be unconstrained variables. Those unconstrained variables are incrementally refined until the formula represented by the graph becomes valid, or the falsifying assignment does not depend on any unconstrained variables. In our case, incremental refinement is structural refinement on the maximally-shared graph. The refinement step replaces an unconstrained variable with a subgraph that represents the summary expression and the edges that were pointing to the unconstrained variable are relinked to point to the newly constructed expression. We shall say that refinement *expands* summary nodes.

Algorithm 1 Main abstraction-checking-refinement loop.

```
1: Let F be a node in the maximally-shared graph representing some VC.
2: f = encode(F)
3: while ¬solve(f) do        ▷ solve returns false if a solution (falsifying assignment) is found.
4:     if ¬REFINE(F, current_solution) then
5:         Report solution and exit.
6: Report VALID and exit.
```

An abstract rendition of our algorithm is given in Alg. 1. The checked verification condition is represented by a root F in the maximally-shared graph. The algorithm encodes F on the fly into formula f and passes it to the decision procedure ($\mathit{solve}()$). In our case, F is bit-accurately translated to CNF by the standard Tseitin transform [22], but from the maximally shared graph after common subexpressions have been eliminated. Summary nodes are encoded as unconstrained variables. If the decision procedure proves f valid, we are done. Otherwise, refinement takes F and the table of current

assignments to variables represented by nodes in the support of F, and returns true if the graph was refined, and false otherwise. If the graph was not refined, then all the summary nodes related to the falsifying assignment have been expanded, and the main loop terminates. Otherwise, the abstraction-checking-refinement cycle continues. Since maximally-shared graphs are acyclic, the algorithm necessarily terminates.

The algorithm interacts gracefully with incremental decision procedures — each expansion of a summary node replaces only a single node with the expression represented by the summary node, monotonically increasing the set of constraints.

Our lazy approach to interpretation of function summaries resembles the intuition behind lazy proof explication [11], a technique used to bridge between different theories in a theorem prover. The shared intuition is to abstract away expensive reasoning — expanding a function summary or solving a sub-theory query — as unconstrained variables, and then constrain them lazily, only as needed to refute solutions to the abstracted problem. The specifics of what to abstract and how to refine, of course, are different, since we are solving different problems.

3.2 Checking

Since critical software bugs (e.g. [2]) are often caused by the finite nature of bit-vector arithmetic, it is important to maintain the bit-level behavior of the verified software. CALYSTO computes bit-precise VCs, which are translated to CNF directly — even expensive 64-bit arithmetic operations, like division and remainder, are handled precisely. The bit-vectors are represented with the same bit-width as in the compiled code. In our case that means that integers and pointers are represented with 64 bits.

Path enumeration is completely left to the SAT solver. We found that it is important for the SAT solver to process the variables in an order that roughly corresponds to reverse preorder traversal (all predecessors are visited before the successors). If the opposite traversal is used, the solving phase typically requires 7-10X more time. This supports our conclusion that most of the paths become infeasible close to the VC root node.

3.3 Structural Refinement

The first few iterations of the main loop of Alg. 1 will likely return false counterexamples, since the initial abstraction is usually very crude. So, the refinement algorithm has to identify very quickly a set of summary nodes that are relevant to the found solution.

The algorithm attempts to minimize the number of expanded summaries to avoid expensive computation. Given a falsifying assignment, our refinement scheme searches the graph and selects a single summary node to expand, thereby refining the model. In particular, the algorithm starts traversing the formula from the VC root. During the traversal, the algorithm detects don't-care values — values that are irrelevant to the current solution and can therefore be ignored.[2] To formalize the concept of don't-care values, we define absorptive element as:

[2] The anonymous reviewers noted a connection between our analysis and strictness analysis in functional programming, as well as the work of Wilson and Dill [24]. The commonality is the goal of finding cases in which a value is not used or needed. For example, in an ITE in a

Definition 1 (Absorptive Element). *If there exists an element a for some operator $\star$, such that $\forall x : a \star x = a$, then a is an absorptive element of $\star$, denoted as* $\mathsf{abelem}(\star) = a$.

For instance, $\mathsf{abelem}(\wedge) = \mathsf{false}, \mathsf{abelem}(*) = 0$, and so on.

If the decision procedure returns a falsifying assignment, each node F in the graph representing the checked VC has some assigned value, which we shall denote as $\mathsf{val}(F)$. If F is an operator $\star$, our algorithm checks $\mathsf{val}(x)$ for each operand x of F. If $\mathsf{val}(x)$ is an absorptive element of $\star$, it is a sufficient explanation of the value of F in the falsifying assignment (the other operand is a don't-care). Hence, it suffices to refine only x. Our refinement procedure is given in Alg. 2. As is usual for graph traversal, visited nodes are marked during traversal to avoid re-visiting nodes; marking is not shown in the pseudocode.

Algorithm 2 Structural Refinement Algorithm. F is a node in the maximally-shared graph, and x and y are its operands. The return value indicates whether a summary has been expanded.

```
 1: function REFINE(graph node F, values assigned to nodes)
 2:     if F is a summary node then
 3:         expand the summary for F; return true
 4:     else if F is a leaf node then
 5:         return false
 6:     else if F ≡ x ⋆ y then
 7:         if val(x) = abelem(⋆) then
 8:             return REFINE(x)
 9:         else if val(y) = abelem(⋆) then
10:             return REFINE(y)
11:         else
12:             return REFINE(x) or REFINE(y)
13:             (The or is lazy: if either call succeeds, the other is skipped.)
14:             (The order is arbitrary. Either x or y can be refined first.)
```

Some operators (like implication and if-then-else) do not have absorptive elements, but allow similar don't-care analysis. Our implementation performs such reductions according to the rules in Alg. 3.

Returning to the example in Fig. 2, in 2a, the checker treats the placeholder nodes as unconstrained variables and finds a falsifying assignment where $flip$: *ret* is true and the != is false. Alg. 3 will derive the refinement in 2b, where a possible falsifying solution gives the *init* : *ret* node a negative value. Next, the algorithm might choose to expand the $flip$: $global2$ node, yielding the refinement in 2c, which is valid. We were able to avoid the expensive expansion of the *init* : *ret* node.

functional programming language, the condition argument is strict because it is always evaluated, whereas the other two arguments are non-strict. In our case, we are refining a falsifying solution, so we have much more don't-care information available, e.g., we know the value of the condition argument, so we know exactly which branch need not be refined.

Algorithm 3 Additions to the Basic Refinement Algorithm

1: **if** $F \equiv (x \Rightarrow y)$ and $\mathsf{val}(x) \equiv \mathsf{false}$ **then** return REFINE(x)
2: **else if** $F \equiv (x \Rightarrow y)$ and $\mathsf{val}(y) \equiv \mathsf{true}$ **then** return REFINE(y)
3: **else if** $F \equiv (x \Rightarrow y)$ return REFINE(x) **or** REFINE(y)
4:
5: **if** $F \equiv ITE(c,x,y)$ and $\mathsf{val}(c) \equiv \mathsf{true}$ return REFINE(c) **or** REFINE(x)
6: **else if** $F \equiv ITE(c,x,y)$ return REFINE(c) **or** REFINE(y) ▷ $\mathsf{val}(c)$ must be true or false.

Unlike other approaches, our approach to refinement does not require a theorem prover. The downside is that our refinement might be less precise and result in more refinement cycles. However, each refinement cycle only adds additional constraints to the decision procedure incrementally, making the solving phase more efficient as well.

4 Experimental Results

To test our approach, we used CALYSTO to generate VCs for six real-world, publicly-available C/C++ applications, ranging in size from 9 to 228 thousand lines of code (KLOC) before preprocessing. The benchmarks are the Dspam spam filter, our boolean satisfiability solver HYPERSAT, the Licq ICQ chat client, the OpenLDAP implementation of the Lightweight Directory Access Protocol, the Wine Windows OS emulator, and the Xchat IRC client. For each program, for each pointer dereference, we generated a VC to check that the pointer is non-NULL (omitting VCs that were solved trivially by our expression simplifier).

CALYSTO has a simple, non-recursive expression simplifier that runs during symbolic execution. The simplifier rules are numerous, but straightforward. We noticed that performing constant propagation during the simplification reduces the memory footprint, but does not drastically speed-up the solving phase because our modular arithmetic theorem prover SPEAR (like many others) performs aggressive constant propagation on its own. Other, slightly more complex rules, like $ITE(c,x,\neg c \wedge y) \equiv ITE(c,x,y)$ do speed up the solving phase, but not drastically.

As the basis for comparison, we also solved the VCs using eager expansion of sliced summaries (described in Sec. 3). The approaches were tested under equal conditions: the same simplification and common subexpression elimination were applied to both approaches after every summary expansion, before calling the SAT solver. The same SAT solver was used for both the base case and our approach.

Table 1 and Fig. 3 summarize the results. In a large majority of cases, the structural abstraction approach is superior to the eager approach, which suffers 81% more timeouts, and 75% longer runtime. There are some cases, however, where the eager approach performs significantly better. Analyzing those cases, we found that occasionally our simplifier can simplify some expanded summaries to trivial constants, which in turn can make the VC trivially easy to solve. For example, the most frequent case we have seen is when an expanded summary, which is an antecedent in an implication, trivially simplifies to false, rendering the whole implication true. A priori, there is no way to know whether or not an expanded summary will drastically simplify the VC (akin to the classic debate of eager vs. lazy constraint propagation in SMT solvers). Our experimental results, though, show that for solving software VCs, laziness wins.

Benchmark	KLOC	#VCs	Base Approach		with Struct. Abs./Ref.	
			Time (sec)	Timeouts	Time (sec)	Timeouts
Dspam v3.6.5	37	8003	4451	12	3758	10
HyperSAT v1.7	9	427	32602	108	27025	81
Licq v1.3.4	20	5165	24103	50	4072	4
OpenLDAP v2.3.30	228	4935	738	0	572	0
Wine v0.9.27	126	8831	2598	0	2145	0
Xchat v2.6.8	76	24045	18914	13	10024	6
Total	496	55583	83406	183	47596	101

Table 1. The first column gives the name and version of the benchmark, KLOC is the number of source code lines (in thousands) before preprocessing, and #VCs is the number of checked VCs. The next four columns give the total run time in seconds (including timeouts) and the number of timeouts, for the base approach and for our new structural abstraction and refinement method. The timeout limit was 300 seconds. Experiments were performed on a dual-processor AMD X2 4600+ machine with 2 GB RAM, running Linux 2.6.15.

Overall, our structural abstraction-checking-refinement approach is able to quickly verify *-sensitive, bit-accurate VCs from large software. Moving forward, we believe there is additional structure to be exploited, and that is the direction of future work.

5 Discussion

We believe there are two main reasons behind the success of our approach: efficient path enumeration and exploiting natural abstraction boundaries in software.

In our experience, path enumeration dominates the cost of software verification, even when the loops are abstracted away. In the worst case, any path-sensitive analysis has to analyze all the paths. We use our SAT-solver-based prover SPEAR to enumerate paths efficiently, but the order in which any SAT solver processes constraints is important. Most solvers add variables to the decision queue in the order in which the variables are found in the clauses. So, by starting the SAT solver from a root of the VC graph and letting it enumerate paths, we explore paths in a breadth-first manner. In analyzing several open-source applications, we realized that most paths in an average program are infeasible, and this breadth-first path exploration prunes more paths more quickly. Contemporary software model checkers do not exploit this fact, but rather rely on depth-first search, performing numerous calls to the theorem prover through a counterexample-driven refinement process, just to refute a path that is most likely to be infeasible anyway. Similarly, the lazy summary expansion prunes obviously infeasible paths before function summaries are expanded and analyzed, decreasing the number of paths that need to be explored even further.

The proposed abstraction-refinement approach exploits the natural abstraction boundaries in software that correspond closely to the programmer's mental model. To manage complexity, programmers tend to organize code into structural units (functions) and use them as abstractions — whatever a function returns, the rest of the code should work. This common programming tactic, which reduces the mental load on the programmer, inspired our approach. To the extent that programmers use functions effec-

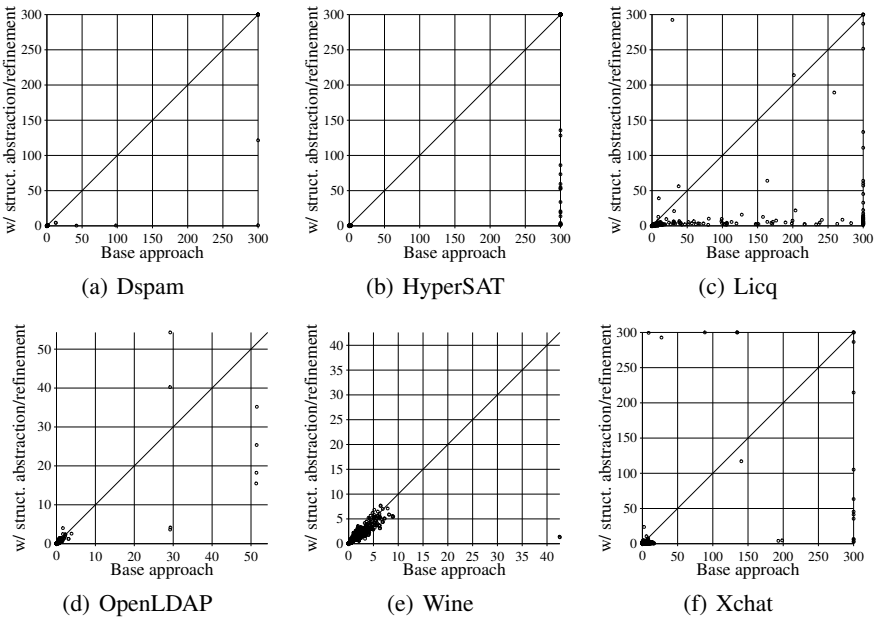

Fig. 3. Results presented as scatter plots. Timeouts are plotted at 300secs.

tively as abstractions, our refinement algorithm can avoid expanding the details that were abstracted away.

Our simple abstraction does not rely on the standard abstraction domains. Hence, the contributions of this paper can be applied to a wide range of software analysis tools that require interprocedural analysis and a decision procedure. If further complexity reduction is needed, we believe our approach should be compatible with classical abstract domains and other abstraction techniques.

References

1. D. Babić and A. Hu. Fast Symbolic Execution for Static Checking. Submitted for publication.
2. D. Babić and M. Musuvathi. Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Research Redmond, 2005.
3. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. *Programming Language Design and Implementation*, pp. 203–213, 2001.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné;, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. *Programming Language Design and Implementation*, pp. 196–207, 2003.
5. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2988, pp. 168–176, 2004.
7. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. *Design Automation Conference*, pp. 368–371, 2003.

8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Computer Aided Verification*, LNCS 1855, pp. 154–169, 2000.
9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans Programming Languages and Systems*, 13(4):451–490, October 1991.
10. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.
11. C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. *Computer Aided Verification* LNCS 2725, pp. 355–367, 2003.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *Programming Language Design and Implementation*, pp. 234–245, 2002.
13. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *Principles of Programming Languages*, pp. 193–205, 2001.
14. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *Computer Aided Verification*, LNCS 1254, pp. 72–83, 1997.
15. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. *Principles of Programming Languages*, pp. 58–70, 2002.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. *Principles of Programming Languages*, pp. 232–244, 2004.
17. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.
18. K. R. M. Leino and P. Müller. A verification methodology for model fields. *European Symposium on Programming*, LNCS 3924, pp. 115–130, 2006.
19. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
20. V. B. Livshits and M. S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. *European Software Engineering Conference/International Symposium on Foundations of Software Engineering*, pp. 317–326, 2003.
21. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *Principles of Programming Languages*, pp. 49–61, 1995.
22. G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pp. 466–483. Springer, 1983.
23. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *Programming Language Design and Implementation*, pp. 131–144, 2004.
24. C. Wilson and D. L. Dill. Reliable verification using symbolic simulation with scalar values. In *37th Design Automation Conference*, pages 124–129. ACM/IEEE, 2000.
25. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *Principles of Programming Languages*, pp. 351–363, 2005.