# Calysto: Scalable and Precise Extended Static Checking[*]

Domagoj Babić
Department of Computer Science
University of British Columbia
babic@cs.ubc.ca

Alan J. Hu
Department of Computer Science
University of British Columbia
ajh@cs.ubc.ca

## ABSTRACT

Automatically detecting bugs in programs has been a long-held goal in software engineering. Many techniques exist, trading-off varying levels of automation, thoroughness of coverage of program behavior, precision of analysis, and scalability to large code bases. This paper presents the CALYSTO static checker, which achieves an unprecedented combination of precision and scalability in a completely automatic extended static checker. CALYSTO is interprocedurally path-sensitive, fully context-sensitive, and bit-accurate in modeling data operations — comparable coverage and precision to very expensive formal analyses — yet scales comparably to the leading, less precise, static-analysis-based tool for similar properties. Using CALYSTO, we have discovered dozens of bugs, completely automatically, in hundreds of thousands of lines of production, open-source applications, with a very low rate of false error reports. This paper presents the design decisions, algorithms, and optimizations behind CALYSTO's performance.

## Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]:

## General Terms

Verification

## Keywords

formal verification, static checking, static analysis

## 1. INTRODUCTION

Error removal (verification/testing/debugging) is one of the most time-consuming parts of the software development life cycle. Accordingly, an enormous range of techniques and tools have been developed to support this task.

We can classify these techniques and tools according to the trade-offs they must make along four dimensions:

[*]This work was supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada and a Microsoft Research Graduate Fellowship.

**Coverage** How thoroughly are all possible execution paths and data values covered by the analysis?

**Automation** How much manual effort is required?

**Precision** How precisely does the analysis correspond to the actual software that is executed?

**Scalability** How large of a code base can be analyzed by the technique or tool?

For example, traditional software testing has perfect precision and excellent scalability — because the tests run on the actual code — and can be highly automated as well. Coverage, however, is the weakness, and many corner-case bugs elude traditional testing. Furthermore, as software grows, coverage drops off, or test time increases, exponentially, sparking interest in formal methods. Fully formal verification promises perfect coverage — a proof of correctness considers all possible executions and inputs — but has historically been painfully labor-intensive. Modern, semi-automatic tools (e.g., ESC/Java [19]) are much better, but still require programmer-supplied loop, function, and class invariants, limiting acceptance of these tools in practice.

Model checking [11, 32] brought complete automation to formal verification, but unfortunately, with a very harsh precision/scalability trade-off. At one extreme, some software model checkers (e.g., Spin [23], Java PathFinder [36], CBMC [9]) can be applied directly to the code base, precisely capturing the true behavior of the program code, but with very limited scalability (or with very limited coverage, when used for bug-hunting by only partially exploring the state space). Abstraction-based methods (e.g., SLAM [5], BLAST [21], Java PathFinder, Bandera/Bogor [33], SATABS [10]) improve scalability, but at the cost of precision. As the abstractions are refined, to regain precision, then scalability suffers. The main direction of model-checking research has been to maintain complete automation, thorough coverage, and acceptable precision, while trying to improve scalability.

In contrast, an alternative philosophy towards static program analysis, usually dubbed "static checking", traces back to lint [25]: scalability is paramount, and coverage and precision are sacrificed. The tool doesn't promise to find all bugs, nor does it promise that all bugs reported are real bugs. Ease-of-use is also a high priority, so these tools are typically so automated that the user need not write specifications. Instead, the tools are empirically hand-tuned to search for certain types of common programming errors. Because of the scalability and ease-of-use, these tools have achieved moderate acceptance in practice. Unfortunately, the scalability derives from approximate summarization of the state of the program, losing precision and leading to their major weakness: crying wolf with false error reports ("false positives"). Programmers will often use a static checker, because it's easy, but they will often ignore the results, because too many reports are wrong. Recent research has moved toward more sophisticated analyses, borrowing techniques from formal verification and static analysis, to reduce the false error rate and generalize the types of errors that can be detected (e.g., MC [18], bddbddb [37], Clouseau [20], and Saturn [38]).

*Extended static checking* is a term coined by Detlefs et al. [15] for combining the usage model of static checking (ease-of-use, pretargeting to specific common bugs, no guarantees of coverage or precision) with the machinery of formal verification (generating verification conditions and checking with a theorem prover). The promise is coverage comparable to formal verification, with automation and scalability approaching that of simpler static checkers. In addition, the formal-style analysis is general and principled — the same machinery applies to an enormous variety of errors. Unfortunately, their ESC/Modula-3 [15] and ESC/Java [19] checkers are not fully automatic. To achieve scalability, the formal analyses are *intra*procedural, and the programmer must supply class and method invariants by hand.

This paper presents CALYSTO, our extended static checker. CALYSTO was inspired and influenced by ESC/Java, CBMC, and especially, Saturn. CALYSTO embraces the ESC philosophy of combining the ease-of-use of static checking with the powerful analyses of formal verification. Unlike ESC/Java, though, CALYSTO is fully automatic, performing interprocedural analysis. Also unlike ESC/Java, CALYSTO handles data operations bit-accurately, so effects like overflow are precisely modeled. CALYSTO's complete automation, interprocedural path-sensitivity, and bit-precision resemble the model checker CBMC. Both tools are based on bit-accurate symbolic execution and are fully interprocedurally path-sensitive (i.e., different program paths are accurately and distinctly analyzed, even through procedure calls) and truly context-sensitive (i.e., the analysis of a procedure precisely considers how and from where it was called). CBMC, however, typically can handle only up to a few thousand lines of code, whereas CALYSTO has scaled to real applications of hundreds of thousands of lines of code. The closest work to CALYSTO is the static checker Saturn. Saturn has also demonstrated scalability to hundreds of thousands of lines of real code, while checking similar errors [17]. Saturn, however, is bit-accurate only for the most common integer operators (e.g., addition, subtraction, bitwise operators), and, more importantly, is only *intra*procedurally path-sensitive. Interprocedural analysis is based on automatically computed summaries, which abstract the behavior of procedures by projecting their effects onto small finite-state property automata, thereby losing precision. Similarly, Saturn's context-sensitivity is also only with respect to these abstract states. CALYSTO combines the virtues of CBMC and Saturn, achieving better scalability than anything with comparable precision and coverage, and better precision and coverage than anything with comparable scalability.

Fundamentally, CALYSTO is based on a fully formal analysis, but with some unsound approximations to dodge classical undecidability results in software verification (e.g., loops, recursion, and heap-allocated data structures). Extensive optimizations — algorithm and data structure improvements, abstraction-refinement frameworks, heuristics, careful implementation — are then required to make such an expensive approach scalable in practice. Experimental evaluation shows that the overall system works: on hundreds of thousands of lines of real, open-source applications, CALYSTO identified real bugs, completely automatically, that resulted in developers issuing patches. The false error rate was below 23%. This paper describes how CALYSTO attains such an unprecedented combination of coverage, automation, precision, and scalability.

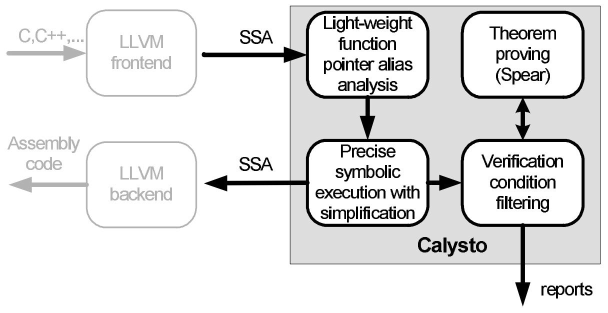
C,C++,...
LLVM frontend
SSA
Light-weight function pointer alias analysis
Theorem proving (Spear)
Assembly code
LLVM backend
SSA
Precise symbolic execution with simplification
Verification condition filtering
Calysto
reports

**Figure 1: High-Level CALYSTO Architecture**

## 2. CALYSTO SYSTEM ARCHITECTURE

The high-level architecture of CALYSTO is shown in Fig. 1. CALYSTO is designed as a compiler pass, in the spirit of Hoare's "verifying compiler" grand challenge [22]: it accepts the compiler's intermediate representation, in static single assignment (SSA) form [14], performs various verification checks, issues bug reports and warnings, and then passes semantically unmodified SSA on to the compiler backend. Designing a static checker in this manner has the obvious advantage of language independence, but also helps to check for errors in the compiler front-end (and any other compiler passes that precede the CALYSTO pass). Supporting a different programming language requires only a different front-end, and, if required, a name demangler to improve the legibility of bug reports. We are using LLVM [28] as our compiler framework, but SSA is standard in modern compilers, so CALYSTO could be retargeted easily to another compiler framework or used as a standalone application.

Internally, the CALYSTO system consists of three stages, supported by an automatic theorem prover, SPEAR. The first stage is a lightweight function pointer alias analysis. It constructs (a sound approximation of) the call graph, including indirect calls through function pointers. We sacrifice some precision, by using a sound, flow-sensitive, but context-insensitive alias analysis, tracking only the function pointers. This pass requires negligible resources, yet is precise enough in practice.

The next stage is symbolic execution [26], which executes the program using symbolic instead of concrete values. CALYSTO symbolically executes functions in the analyzed program, computing symbolic definitions for each modified variable and memory location. These symbolic definitions are used to create *verification conditions* (VCs) — logical formulas that are valid iff some correctness property holds of the program. The symbolic execution machinery allows generating VCs for any assertion at any point in the program. CALYSTO currently supports user-supplied assertions (written as boolean expressions in whatever programming language the compiler front-end is parsing), but in the spirit of static checking, also automatically generates VCs to check that each pointer dereference cannot be NULL. The generated VCs are essentially formal verification conditions: all possible program paths and data values are considered (except for the unsound approximations described in Sec. 3).

The last stage consists of checking and filtering the verification conditions. In principle, the VCs could be sent directly to a theorem prover for checking, and this approach is used in most other tools. We have found, however, that both efficiency and usability can be improved by control and filtering of what VCs are checked. The main efficiency gain is described in Sec. 4.2. To improve usability, if the theorem prover manages to find a falsifying solution (a potential bug), this stage reports the bug and filters away all VCs corresponding to the same property within the same function. For instance, if a pointer that can be NULL is dereferenced at many different places within the same function, and that function can be called in many different contexts, CALYSTO will emit only one bug report per context. This heuristic avoids overloading the programmer with reports that correspond to the same issue. For each falsified VC, CALYSTO dumps a detailed graphical trace; if the falsified VC depends on any global variables, the trace is given all the way from the root of the call graph (the *main* function).

The actual validity-checking of VCs is done by SPEAR, which is a sound and complete, fully automatic theorem prover that supports Boolean logic, bit-vector operations, and bit-accurate arithmetic. Unlike other static checkers, which use general-purpose SAT solvers or theorem provers, SPEAR is custom-designed for the software VCs generated by CALYSTO, optimizing performance.

## 3. DESIGN CHOICES

Our goal was to combine the coverage and precision of formal verification with the scalability of static checking. To achieve this goal, our basic design philosophy was to start with a principled, fully formal, precise analysis, to make as few unsound approximations as possible, and then to focus on improving scalability. This approach helps separate the concerns of the correctness of our analyses from their efficient, practical implementation.

We made three key decisions to make CALYSTO significantly more precise than is typical for static checkers:

- The first decision was to be bit-precise, meaning that we handle machine arithmetic precisely, including all boundary conditions (underflows and overflows) and all standard operators, including multiplication, division, remainder, and shift. This precision incurs a high computational cost, but we believe that the cost is justified. First, boundary conditions themselves are frequent sources of bugs (e.g., [4]). Second, bounded integers are a prerequisite for deciding properties with non-linear operators,[1] and we observed that non-linear operators appear quite frequently in real code.

- Interprocedural path-sensitivity was the second important design decision. Since the number of possible paths typically grows exponentially in code size, this decision is also computationally expensive, but we believe this, too, is justified. For example, a common coding idiom that requires interprocedural path-sensitivity is the handling of erroneous conditions. Applications frequently use long chains of function calls for handling erroneous and exceptional conditions, e.g., checking pre/post-conditions, detecting errors, printing and logging messages, and finally exiting with an appropriate error code. We have seen such sequences that are 5–7 function calls deep. Static checkers that are not interprocedurally path-sensitive can fail to precisely compute the conditions under which the sequence exits, resulting in false positives.

- A consequence of the interprocedural path-sensitivity is the third key decision: CALYSTO is fully, precisely context-sensitive. Context-sensitive analyses differentiate the effects of the state of the program at different call sites where a function is called. In contrast, a context-insensitive analysis can be much more efficient, because a function need be analyzed only once, regardless of how many different places it is called, but this analysis must merge together the states at all possible call sites, thereby losing information and producing false positives. Note that a "context-sensitive" analysis can still be very imprecise, e.g., many software model-checkers abstract the state of the program onto a small set of predicates, and the context-sensitivity is only with respect to these abstract states. CALYSTO goes further, keeping definitions of interprocedural control-flow context, variables, and abstract memory locations to which pointers can point. Achieving such precision is expensive in both time and space.

These decisions greatly increased the computational complexity of our analyses, but enabled the low false positive rate.

We also had to make some design decisions that were unsound (compromising coverage, thereby possibly missing bugs) as well as imprecise (possibly resulting in false positives):

- CALYSTO currently does not support floating-point operations. Floating-point is handled unsoundly by converting floating-point variables and constants to integral ones. In theory, it is straightforward, but tedious, to add bit-accurate floating-point models to the theorem prover. A practically efficient solution, however, will likely require more research. We have not observed any false positives due to this handling of floating-point in any of our benchmarks.

- Loops create the classical halting problem undecidability result. Accordingly, CALYSTO unsoundly approximates loops by unrolling them once and terminating them with an assumption that the loop test has failed, similarly to ESC/Java [19]. This is a major source of missed bugs, because possible program paths are not analyzed. For example, in the following real code (abstracted from the HYPERSAT benchmark):

```
1      int cnt = 0;
2      bool c2 = false;
3      while ( c1 /* some condition */ ) {
4          if ( c2 ) {
5              cnt++;
6          }
7          c2 = true;
8      }
9      if ( cnt == 0 ) { exit(1); }
10     ...
```

  variable *c2* is false in the first iteration, so the *cnt* counter can be incremented only in the second iteration. If the loop is unrolled only once, line 10 becomes unreachable. Since CALYSTO does not check unreachable code, it can therefore miss bugs. Fortunately, we have seen only a very few false positives due to this handling of loops in preliminary experiments checking programmer-specified assertions, and none at all when checking the automatically generated VCs.

- Recursion creates the same undecidability problem as loops, and we handle it in a similar manner. Like Saturn [38], CALYSTO simply breaks cycles in the call graph, ignoring the recursive call. This practice causes a small number of false positives in practice. For example, in the application Postfix,[2] a safe allocator *xmalloc* tries to allocate memory on the heap and, if successful, returns a valid pointer. If it's unsuccessful, it calls the function *fatal*, which prints an error

[1] Undecidability with (unbounded) integer multiplication and quantifiers follows from Gödel's Incompleteness Theorem. Even without quantification, undecidability follows from the undecidability of Hilbert's Tenth Problem. See, e.g., [31].

[2] Version 2.5.20070614. We could not compile Postfix completely with the LLVM front-end (some functions were not compiled into the binaries), so we do not use Postfix for benchmarking in Sec. 5.

message and exits (doesn't return). The function *fatal*, however, calls *xmalloc* to construct its error message! To prevent possibly infinite recursion, the programmers added a re-entry counter, and if the counter exceeds 2, then *fatal* exits without trying to construct an error message. If our analysis cuts out the recursive call to *fatal*, it cannot infer that *xmalloc* cannot return NULL, producing false positives.

- Pointer arithmetic (and therefore arrays) is known to be undecidable in general [7]. We use a simpler memory model, similar to the "logical memory model" [6], in which $*(ptr + i)$ and $*ptr$ are assumed to refer to the same object, except that our symbolic execution does distinguish those two locations if our expression simplifier (Sec. 4.1) can simplify $i$ to a constant. Such constant offsets are often used for access to structure fields (making CALYSTO field-sensitive as well), so this added precision is important in practice.

Obviously, there are no perfect solutions to the undecidable problems, but these are well-studied issues, and CALYSTO's design is compatible with standard approaches to handling these problems soundly or at least more precisely than we do currently. For example, there is no theoretical obstacle preventing CALYSTO from checking and using user-supplied loop invariants to handle loops soundly, but we chose to make CALYSTO fully automatic. Heuristics based on abstract interpretation [12] could be used to infer loop invariants automatically, as in [30], providing automation and sound handling of loops, but at the risk of introducing more false positives. Similar techniques could be applied to infer function, class, and heap invariants, with the same trade-off of soundness versus precision. We could also make CALYSTO's analysis progressively more precise by generalizing our current approximations to some small, bounded depth, e.g., unrolling loops several times instead of just once, allowing a bounded number of recursive calls, and modeling the first few elements of arrays precisely before lumping the rest of the elements together. The trade-off here would be the greater computational complexity versus the greater precision. In general, the overall goal is to achieve the most useful, practical balance between coverage, automation, precision, and scalability. We based our design decisions on the state-of-the-art, and our belief in the improvements we could make (Sec. 4). As future research changes the balance (e.g., a more precise heuristic for inferring loop invariants, a more efficient theorem prover, etc.), these decisions should be revisited.

## 4. IMPROVING SCALABILITY

As mentioned, bit-precise and *-sensitive (path-, context-, and field-sensitive) analysis is computationally extremely expensive; our first attempts did not scale beyond a couple thousand lines of code. This section describes the novel techniques we developed to make CALYSTO practical. The three subsections correspond to the three main parts of CALYSTO in Fig. 1: the symbolic execution, the verification condition filtering, and the theorem prover.

Three general principles underlie the improvements to the analyses: preserving and exploiting problem structure; using fast, approximate analyses to filter and simplify tasks before applying heavyweight, precise analyses; and caching to reuse previously computed results. Indeed, even the top-level architecture of CALYSTO reflects the filtering idea — the function pointer analysis stage is a lightweight, approximate analysis that simplifies the task of the later, more expensive analyses.

### 4.1 Structure-Preserving Symbolic Execution

*Symbolic Execution.*

The two standard methods for computing verification conditions (or other symbolic representations of programs) are symbolic execution [26] and weakest (liberal) precondition [16]. The first is a forward analysis; the second, backward. We chose forward symbolic execution for several reasons:

- Pointer definitions are known before the pointers are dereferenced. Knowing the pointer definition, our symbolic execution can precisely simulate the effects of pointer reads and writes, by simply executing them symbolically. Doing the equivalent analysis backward is possible, in theory, but would produce excessively complex expressions representing all possible pointer definitions at program points between the def and the use.

- Going forward, one builds more complex expressions from simpler ones, without performing substitutions. Thus, the constructed expressions will not be modified later. Immutability of the constructed expressions allows them to be simplified while being built. For instance, consider a sequence of code like:

```
int x = 1;
...
if (0 < x) y = 0;
...
return y*z;
```

  A backward analysis substitutes the 1 for $x$ after the entire expression $ITE(0 < x, 0, y) \times z$ has already been constructed. To simplify it to zero, the simplifier has to revisit all expressions that contain the modified sub-expression. On the other hand, a forward analysis would construct $0 < 1$ at the **if** statement, which simplifies to `true`, continue with $ITE(\texttt{true}, 0, y)$, which simplifies to 0, and end up with $0 \times z$, which simplifies to 0. This early expression simplification saves a substantial amount of memory.

- Symbolic execution and weakest precondition can both generate expressions exponential in the size of the code (e.g., [13, 29]). Representing the expressions as graphs that share common sub-expressions avoids that blowup [27]. Just as with the early simplification, forward analysis facilitates immediate common sub-expression elimination while the expressions are being built.

Our symbolic execution algorithm processes each function exactly once, proceeding bottom-up in the call graph, computing a symbolic representation of the function's effects. Program functions can have multiple effects, e.g., returning a value, modifying globals, and modifying memory locations reachable through passed-in parameters. For each side-effect, CALYSTO computes a symbolic expression. When a function is called, symbolic expressions for small effects (up to 50 nodes) are inlined immediately, while larger effects are represented in the symbolic expression by place-holding summary operators. The summary operators are expanded during VC-checking only if needed (Sec. 4.2).

*Efficient Gated Single Assignment Form.*

The basic step of symbolic execution is to find the correct symbolic definition of each variable or memory location that is read, and then to update the correct symbolic definition of each variable/location that is written. These operations are performed so many times that their implementation needs to be extremely fast.

Plain SSA form doesn't directly provide the information needed for fast lookup of definitions. At join points (i.e., places where different program paths merge), SSA introduces a $\phi$-"function" for each definition, which denotes somehow choosing the correct definition depending on where the preceding flow of control came from. If the exact definition matters (and it does for verification), the analysis has to track where the flow of control came from, and pick the appropriate definition.

Gated Single Assignment (GSA) form solves this problem (e.g. [35]). GSA extends SSA with a gating function $\gamma$ that replaces the $\phi$-function. A $\gamma$-function in a basic block $B$ can be intuitively understood as an expression that determines which definitions reach $B$ and under which conditions. Our symbolic execution constructs $\gamma$- from $\phi$-functions on the fly.

The challenging aspect of efficiently implementing GSA form is in handling memory locations accessed through pointers. For example, consider:

```
if (c) p = &x;
else p = &y;
*p = 1;
```

It's easy to construct a $\gamma$-function that gives the correct value of $p$ as $ITE(c, \&x, \&y)$, but it's less obvious how to update the new symbolic values for $x$ and $y$ efficiently. CALYSTO maintains partial, conditional expressions, which compactly represent the different definitions that might be live from the different basic blocks. Missing definitions (which represent either infeasible paths or values dependent on the calling context) are represented by a placeholder, which is later substituted with a real definition during structural refinement (Sec. 4.2).

### *Preserving Structure.*

As mentioned earlier, preserving and exploiting problem structure was a key principle behind improving efficiency. Fortunately, the graph-based representation of symbolic expressions that we use to prevent expression-size blow-up also captures and preserves the dataflow structure of the program. All computed expressions are represented as maximally-shared graphs:

DEFINITION 1 (MAXIMALLY-SHARED GRAPH). *Let $G = (N, E, \mathscr{L})$ be a labeled, directed graph, where $N$ is the set of nodes, $E \subseteq N \times N$ is the set of edges, and $\mathscr{L} : N \to O$ is a labeling function from $N$ to some set of operators $O$. Let $|n|$ denote the out-degree of node $n$. For all nodes $n$, the arity of the operator $\mathscr{L}(n)$ must be equal to $|n|$. Furthermore, assume the outgoing edges are ordered, and let the node pointed-to by the i-th edge of a node $n$ be denoted as $child_i(n)$. Two nodes $n_1$ and $n_2$ are defined to be equivalent ($n_1 \triangleq n_2$) iff $|n_1| = |n_2|$, $\mathscr{L}(n_1) = \mathscr{L}(n_2)$, and:*

$$\forall 0 \leq i \leq |n_1| : child_i(n_1) \triangleq child_i(n_2)$$

*Graph $G$ is maximally-shared if $\neg\exists n_1, n_2 \in N : n_1 \neq n_2 \wedge n_1 \triangleq n_2$.*

The following code helps illustrate how maximally-shared graphs preserve the flow of data in the program:

```
int glob; // Global
...
void f(int a) {
    bool flip = false;
    if (a < 0) {
        a = -a;
        flip = true;
    }

    if (flip) {
        assert(a != 0); // A1
        glob /= a;
    }
    return;
}
```

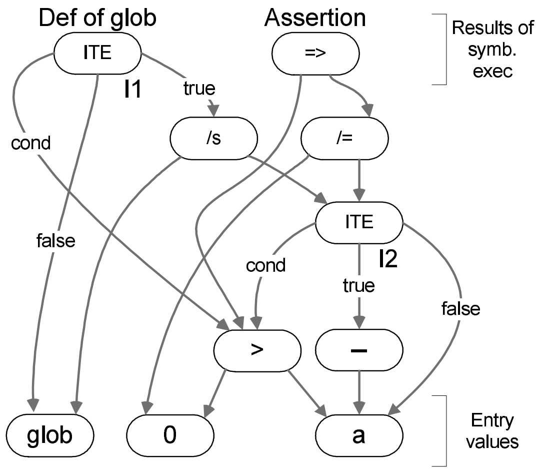

**Figure 2: Example of a Maximally-Shared Graph. The initial values at function entry are shown at the bottom, while the computed symbolic values at function exit are at the top of the graph. Implication is represented as $=>$, disequality as $/=$, signed division as $/s$, unary minus as $-$, while other operators have standard meanings. Given the initial values of *glob* and *a* (bottom), symbolic values of *glob* and the VC corresponding to A1 at function exit are represented by the *ITE* node I1 and the implication node $=>$ (top).**

Function $f$ modifies a global and contains assertion A1. Fig. 2 shows the result of symbolic execution of function $f$: two symbolic expressions, representing the effect on *glob* and the VC corresponding to A1. Note that A1 is unreachable if $a \geq 0$, so the VC is vacuously `true` in that case, hence the implication node.

Maximally-shared graphs provide several benefits:

- Node $n_i$ is reachable from node $n_j$ if and only if $n_j$ is data-dependent on $n_i$. Thus, irrelevant sub-expressions are automatically sliced away. Slicing is crucial for efficient VC checking, since redundant subexpressions and variables significantly slow down theorem provers and confuse their heuristics.

- Common sub-expressions are always shared. Aside from the obvious savings in memory, this knowledge can be exploited to speed up the theorem-proving phase. For example, assume that some function $g$ calls $f$ and after the call checks some property that is either data- or control-dependent on *glob* modified by $f$. Denote the assertion that checks that property A2. Symbolic expressions computed for A1 and A2 share at least $ITE_2$ in Fig. 2. Theorem provers can exploit this sharing to avoid solving the same sub-expressions multiple times (as in [2]).

- The structure of the graph can be exploited for abstraction and refinement. Note that the validity of assertion A1 is independent of the actual value of $a$, even though the implication node is data-flow dependent on $a$. If $a \geq 0$, the VC is vacuously `true`. Otherwise, if $a < 0$, then $ITE_2$ will be equal to $-a$, and the VC is again `true` because $a < 0 \Rightarrow (-a \neq 0)$. So, if we have some complex function $h$ that returns an integer, and we call $f$ with the result of $h$ as a parameter:

```
...
f(h());
...
```

we do not even need to analyze what *h* returns, because whatever it returns, assertion A1 will hold. This is the core idea of structural abstraction in Sec. 4.2.

Our use of maximally-shared graphs is similar to the use of BDDs [8] in bddbddb [37] and, to a lesser degree, in Saturn [38]. Both data structures exploit sharing to (try to) avoid an exponential space blow-up, and in both data structures, the shared structure enables caching and re-using previously computed results. The key difference is that maximally-shared graphs are not canonical representations of functions. The maximally-shared graphs preserve program structure, and are therefore linear in the size of the program (if we don't in-line function effects). Because BDDs are canonical, they provably *must* blow-up exponentially for most functions, including common operations such as multiplication [8]. Maximally-shared graphs are the lightweight, good-enough solution, with the theorem prover as backup; the canonicity of BDDs is heavy-weight overkill for this task.

*Expression Simplifier.*

The lack of canonicity of maximally-shared graphs could produce inefficiency, because two functionally equivalent nodes might not be structurally equivalent. In theory, the symbolic execution stage could prevent this problem by calling SPEAR to check for functionally equivalent nodes, but such an approach would be prohibitively expensive, analogous to using BDDs. Instead, we again rely on the principle of a fast, good-enough solution for the common cases, leaving the hard cases for the theorem prover later.

As the symbolic execution builds symbolic expressions, those expressions are immediately simplified with a light-weight expression simplifier. The simplifier is mostly not recursive and looks at only one operator node at the time. There is one exception: simplification of conjunctions. We experimentally found that control-flow context expressions frequently contain sub-expressions that are composed of 2–20 conjuncts, which occasionally contradict each other. Simplifying such contradicting conjuncts saves some memory with minimal cost in runtime.

Interestingly, the expression simplification has little effect on theorem-prover runtimes later, when the VCs are checked. Most simplifications are just constant propagation, and theorem provers are extremely efficient at propagation of such facts. The big win from the expression simplifier is the savings in space, caused by the early pruning of duplicate nodes and infeasible paths.

## 4.2 Structural Abstraction

Structural abstraction is the key scalability breakthrough in the verification condition filtering stage of CALYSTO [3]. It is an automatic abstraction/refinement framework, so it first attempts to solve an abstracted, approximate, easier VC, and then progressively refines the VC as needed. The key difference versus other abstraction/refinement approaches for software verification is that both the abstraction and refinement are entirely based on exploiting the structure available in the program (hence the importance of preserving this structure in the symbolic execution stage). Because they are structural, the abstraction and refinement steps are very fast; unlike many other abstraction/refinement schemes, there is no need for expensive proofs of unsatisfiability.

Recall that the symbolic execution stage builds a maximally-shared graph for each function, and that function calls (larger than 50 nodes) are not inlined, but indicated by a placeholder node. Returning to the example in Sec. 4.1, suppose we have a function that calls *f*, with code like:

```
...
f(h());
assert glob < 1; // A2
```

The symbolic expression for the VC for assertion A2 would (ignoring the inlining of small function effects) simply be a graph comparing a placeholder node to 1. The placeholder node would indicate that it is the effect on the global variable glob, of calling *f* with a parameter that is the result of calling *h*, but this effect is not computed, yet.

These placeholder nodes form an abstraction boundary that naturally corresponds to typical programming style: programmers try to modularize their code to minimize dependencies across function calls. In structural abstraction, when a VC is checked for validity, the theorem prover at first considers the placeholder nodes to be unconstrained variables. If the theorem prover still manages to prove validity, the checked VC is valid no matter what the placeholders actually represent. The assertion is OK. If the VC can fail, however, it might be a false positive, because the unconstrained value might not be a possible effect of the function call. Thus, we need a refinement step to eliminate the false positive.

In structural refinement, one placeholder node is expanded with the maximally-shared graph of the function call it represents. This enlarges the graph for the VC, making it more precise and refining the abstraction. This new VC is again checked for validity, and the abstraction/refinement loop continues, inlining placeholder nodes one-by-one, until either the VC becomes valid or the VC can still fail but every relevant placeholder node has been inlined (in which case a bug has been discovered).

Continuing the example, checking the initial, abstract VC for assertion A2 will fail, because the unconstrained placeholder can take on an arbitrary value, say, 2, that is greater than 1. Thus, structural refinement will expand the placeholder node, and the refined VC for A2 will check that the effect of *f* on glob is less than 1. Graphically, we would add a new comparison node to Fig. 2, comparing ITE node I1 to a constant node 1. The node labeled *a* at the bottom of the figure would be a placeholder node for the return value of *h*. This new VC would be checked by the theorem prover. Depending on what is known about the original value of glob, the new VC might be valid (e.g., if glob were equal to 0 before the call to *f*). If not, the structural refinement would next expand the placeholder node for the call to *h*, and the process would continue.

The choice of which definitions of placeholder nodes to inline is important. CALYSTO uses a don't-care analysis to isolate the reason why the VC still fails and to inline only definitions that are logically related to that reason. For instance, if an AND operator node has two branches, and the value of one branch is false, that branch is a sufficient explanation for the value of the AND node, and it suffices to refine only that branch. The refinement is based on structure and the falsifying assignment returned by the theorem prover, so it is simple and fast.

An additional feature of structural refinement is that the VCs change monotonically: each refinement only adds information. Thus, an incremental theorem prover can re-use all of its work (learned clauses and implications) from solving the VC on one iteration as it re-solves the modified VC on the next iteration. The close cooperation between the analysis stages and the theorem prover provide opportunities for improved efficiency.

## 4.3 Application-Specific Theorem Prover

CALYSTO generates highly complex VCs, so a significant portion of runtime is spent in theorem prover calls. To handle such

VCs, we had to develop a theorem prover for bit-vector (machine) arithmetic, SPEAR. The core of SPEAR is essentially a Boolean satisfiability (SAT) solver, but with layers of added functionality to support the needs of software verification. SPEAR won the bit-vector arithmetic category in the SMT 2007 competition.[3]

SPEAR is designed to work closely with CALYSTO, so it understands the structure of the VCs. For example, SPEAR can use information from the VCs to modify its heuristics, e.g., choosing different orderings of constraints or variables. Also, SPEAR is incremental, so it capitalizes on the incremental queries generated by structural refinement.

In addition to the structure-based techniques, two other factors that significantly improved the overall scalability of CALYSTO are the way in which SPEAR encodes arithmetic operators and the CALYSTO-specific tuning of SPEAR.

### *Gate-Optimal Encoding.*

Programs contain non-linear operators, and to be bit-precise, one must have a theorem prover that supports them. A number of different methods have been developed for linear bit-vector arithmetic, but few of them are applicable to non-linear operators. The usual approach is bit-blasting: Variables are encoded as bit-vectors of suitable size, and operators are replaced by digital circuits corresponding to that operator. In effect, VCs become large digital circuits, which can be converted to conjunctive normal form (CNF) using the Tseitin transform [34] and given to a boolean satisfiability (SAT) solver.

Numerous circuits have been proposed for each standard operation. Choosing the right circuit for CNF encoding is a little-researched but important problem — properly selected circuit can easily make the theorem prover an order of magnitude faster. The heuristic we found most effective is to use gate-optimal circuits, i.e., circuits that have the minimal number of gates. Such circuits tend to generate the fewest variables during Tseitin encoding, which avoids flooding the SAT solver with redundant variables.

### *Automatic Optimization of Parameters.*

Parameterized heuristics abound in automated theorem proving, and manual tuning of the respective parameters is difficult and time-consuming. Typically each class of problems exhibits certain specific characteristics, and parameter settings that work well for one do not necessarily work well for another.

We have used AI techniques to automatically tune the parameters controlling the heuristics used by SPEAR to optimize performance for the kinds of VCs generated by CALYSTO [24]. The approach is stochastic local search: the optimization algorithm performs a simple hill-climbing to find local minima and perturbation to escape local minima. The probability of finding the global minimum grows with longer runtimes.

The tuning process is extremely slow. We therefore tuned on a small set of VCs. The optimization technique also adaptively chooses the number of training instances to use for each parameter setting: while poor settings can be discarded after a few algorithm runs, promising ones are evaluated on more instances.

We then evaluated the automatically-tuned parameter settings on a separate test set of VCs, measuring a 500-fold speedup over the manually optimized version of SPEAR on CALYSTO-computed verification conditions. (All tuning was completed before we ran the benchmarks in Sec. 5.) The speedup made CALYSTO much more practical, and also gave us insights into the relation between characteristics of CALYSTO-generated VCs and search parameters.

[3]For details, see http://www.smtcomp.org/

| Benchmark | LOC (total) | LOC (code) | Modules |
|---|---|---|---|
| bftpd 1.8 | 4532 | 3306 | 1 |
| bftpd 1.9.2 | 4602 | 3368 | 1 |
| HYPERSAT 1.7 | 9123 | 6022 | 1 |
| spin 4.3.0 | 28394 | 20481 | 1 |
| openssh 4.6p1 | 81908 | 45304 | 11 |
| inn 2.4.3 | 122727 | 71102 | 46 |
| ntp 4.2.4p2-RC5 | 185865 | 74230 | 10 |
| ntp 4.2.5p66 | 192019 | 74277 | 9 |
| bind 9.4.1p1 | 393318 | 184204 | 26 |
| openldap 2.4.4a | 374266 | 223595 | 27 |
| TOTAL | 1406754 | 685408 | 133 |

**Table 1: Benchmarks Used for Experiments. The second and third columns show the number of lines of code before and after preprocessing ("LOC(code)" does not include comments, empty lines, and pragma-disabled code). The fourth column gives the number of compilation units produced by LLVM's front-end.**

## 5. EXPERIMENTAL RESULTS

To evaluate CALYSTO, we checked a number of publicly available, real-world applications: the openssh remote access server and client, the inn Usenet system, the ntp network time protocol server and client, the bind DNS system, and the OpenLDAP Lightweight Directory Access Protocol system. Those benchmarks are the largest open-source benchmarks that we could successfully compile with both LLVM's front-end and with Saturn. We also used some smaller applications where we were able to get particularly prompt and precise feedback from the developers: the Bftpd FTP server, the HYPERSAT boolean satisfiability solver, and the Spin explicit-state model-checker. Table 5 lists the benchmarks.

We checked the automatically generated assertions that dereferenced pointers cannot be NULL. This is an excellent property to use to evaluate a static checker because: (i) the property is well-defined and automatic, (ii) pointers are often passed through a long sequence of calls, which necessitates interprocedural analysis, (iii) pointer manipulation in programs depends on both data- and control-flow of the program, exercising all the components of a static checker, and (iv) pointers are dereferenced very frequently in code — the number of produced VCs is probably larger than what would be generated by any other property (proper locking, for instance), and the sheer number of VCs pushes static checkers to their limits.

Initially, we started sending raw reports to developers, but quickly found that developers were very unwilling to separate out real bugs from false positives (inadvertently validating our research goal of minimizing false positives!). We began filtering the reports ourselves, omitting all reports that we could prove infeasible. All remaining reports were sent to the developers. At that point, we ran into an unexpected problem: the developers would either take a very defensive stance, claiming that a particular bug was either irrelevant or very improbable, or would take a very cautious stance, fixing everything in the code just to be safe, without much thought about whether a bug was feasible or not. To be rigorous, we have defined a bug strictly as follows:

- Only a dereference of a pointer which is either uninitialized or NULL is considered a bug.

- There must exist a feasible path from the point where the pointer was initially defined to the point where it was dereferenced. For CALYSTO's evaluation, we also required that if any globals are included in the trace, the trace must be given all the way from the main function (root of the call graph). For Saturn's evaluation, we waived this constraint.

- Every feasible NULL pointer dereference was considered a bug, no matter how improbable or irrelevant it might be.

- Many applications contained pointer checks. Usually, such checks exit if the pointer is NULL and print/log an appropriate message. In this case, the NULL pointer is never dereferenced, so failed pointer checks were not counted as bugs. In other words, only dereferences that would cause a segmentation fault were considered bugs.

- If a pointer *ptr* was guaranteed not to be NULL, then we also assumed that pointers with offset $ptr + i$ can never be NULL either. The likelihood of an integral overflow ($ptr > 0 \wedge ptr + i = 0$) is extremely remote, and the developers do not take such reports seriously. Checking that the offset is within allowed bounds is a different property, not to be confused with NULL-pointer checking.

For all reports that we could not prove to be false, we asked for a feasibility confirmation from the developers. Reports that neither we nor developers could prove to be feasible or infeasible are classified as unknown.

## 5.1 Comparison to CBMC

As mentioned in the introduction, the software model checker CBMC [9] was an inspiration for CALYSTO. CBMC promises a similarly high-level of coverage and precision (bit-accurate, path- and context-sensitive) as CALYSTO. We also evaluated SATABS [10], the successor to CBMC, which adds an automatic abstraction engine.

We used CBMC v2.6 and SATABS v1.9 for our experiments. For compiling larger projects (that require linking), those two tools required the goto-cc compiler. We were able to compile only the smaller benchmarks with these tools: bftpd v1.8 and v1.9.2, HYPERSAT v1.7, and Spin v4.3.0.

We ran CBMC in two modes: one with default settings and the other with the `--unwind=1` option, which unrolls all loops only once. SATABS ran with its default settings. CBMC in default mode ran out of memory on all benchmarks (`std::bad_alloc` exception), without producing any results. CBMC with one loop unrolling terminated with internal assertion failures on the bftdp runs and HYPERSAT, and ran out of memory on Spin. SATABS ran out of memory on bftpd and Spin, and timed out after 15 hours on HYPERSAT. Experiments were performed on a dual Opteron 2.8 GHz machine with 16 GB RAM.

These fully formal tools promise soundness, guaranteeing that no bugs will be missed, and bit-accurate precision, guaranteeing no false positives, either. Unfortunately, due to the lack of scalability, they produced neither. The theory of formal verification has produced deep and valuable insights into program analysis, but directly applying the theory appears not to produce a practically scalable tool.

## 5.2 Comparison to Saturn

Saturn [38], another inspiration for CALYSTO, was designed for scalability from the start. Despite CALYSTO's more precise, more expensive analysis, can it match Saturn's proven scalability?

We are comparing against Saturn v1.1.[4]. We used only Saturn's NULL pointer analysis, which finds possible NULL pointer dereferences. We used the best known parameter settings for each tool. For CALYSTO, we used the default options, with a 10 second timeout per VC and limiting the number of VCs per function to 500, effectively setting the timeout per function to 5000 seconds. Saturn's tutorial recommends using a 60 second timeout per function, so all the experiments presented in tabular form were obtained using the 60sec timeout. When we attempted to use 5000 second timeouts, Saturn produced the same results on Bftpd and Ntp and ran out of 16 GB of memory on all other benchmarks. Experimental results are presented in Table 2.

Saturn's traces were significantly harder to interpret because the tool does not produce the complete trace and because we do not have the same level of familiarity with Saturn as we do with CALYSTO. Interestingly, there is very little overlap between the bugs reported by the two tools. CALYSTO tends to report either violations of C library properties, which Saturn frequently misses (presumably because of incomplete descriptions of C library functions), or very long traces, sometimes spanning through 10–15 functions, which Saturn misses due to lack of interprocedural path-sensitivity. On the other hand, most of bugs that CALYSTO missed were due to unsound handling of loops and to assertion violations (once an assertion is violated, all the code after it becomes unreachable).

After we reported bftpd, Spin, and ntp bugs, the developers immediately fixed all of them in the next release. Thanks to prompt responses from the bftpd and ntp developers, we managed to check the new versions that fixed all the bugs found in the previous version. In the new versions, we found new bugs, which have also been fixed in the meantime.

The most frequent causes of Saturn's false positives were: lack of interprocedural path-sensitivity, incomplete specifications of C library functions (for instance, passing a NULL pointer to *free* function is allowed), and specific code patterns that seemed like inconsistencies to Saturn. Saturn's results on bind are especially interesting. Bind's code is among the highest quality code of all open-source applications we have seen so far — almost every single pointer is checked before dereferencing and complex data structures are checked for consistency before usage. This ubiquitous checking apparently confused Saturn's inconsistency analysis because every single report was provably false. The major sources of CALYSTO's false positives were: missing specifications of external functions, broken cycles in the call graph, and C type unsafety.

The runtimes of both static checkers are comparable. Saturn is faster on some; CALYSTO, on others. Bind was particularly problematic for CALYSTO — it ran out of memory while analyzing 8 and timed out (1 day) on 1 compilation unit. The timeout was caused by a performance bug (failing to re-use certain cached re-

[4]We are comparing against the most recent, most up-to-date version of Saturn available. An earlier version of Saturn reported low false error rates while checking for NULL pointer dereferences, although for a much less stringent notion of what constitutes a true bug [17]: in that paper, inconsistencies in whether a pointer is checked for NULL on different paths were included as real bugs; we are using the stricter definition described earlier. Unfortunately, the developers of Saturn have told us that they believe that the current version of Saturn is no longer as effective at identifying NULL pointer dereferences as the earlier version, that the previous version no longer exists, that they could not re-create it, and that they are not planning to update their NULL analysis to where they have confidence in it once again [1]. Thus, the only apples-to-apples comparison we can make, with the same definition of bugs, the same benchmarks, and the same machines, is with the current version of Saturn, which might not represent Saturn in the best possible light. Fortunately, the central point of the comparison is whether Calysto achieves comparable scalability to Saturn, despite performing analyses that are, by design, more precise and therefore presumably more expensive. Perhaps an earlier version of Saturn would have had precision closer to that of Calysto, but that question is moot.

| Benchmark | LOC (code) | Saturn v1.1 | | | | | CALYSTO v1.5 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Reports | Bugs | Unkn. | FP Rate | Time [s] | Reports | Bugs | Unkn. | FP Rate | Time [s] |
| bftpd 1.8 | 3306 | 3 | 3 | 0 | 0% | 129.51 | 12 | 11 | 0 | 9% | 3.14 |
| bftpd 1.9.2 | 3368 | 3 | 3 | 0 | 0% | 105.17 | 5 | 4 | 0 | 20% | 2.86 |
| HYPERSAT 1.7 | 6022 | 4 | 0 | 0 | 100% | 647.21 | 0 | 0 | 0 | 0% | 14.57 |
| spin 4.3.0 | 20481 | 15 | 6 | 2 | 54% | 2129.04 | 0 | 0 | 0 | 0% | 6858.10 |
| openssh 4.6p1 | 45304 | 14 | 0 | 1 | 100% | 3707.81 | 4 | 1 | 0 | 75% | 8995.64 |
| inn 2.4.3 | 71102 | 288 | $^*$12 | 34 | 96% | 9879.69 | 10 | $^*$6 | 1 | 34% | 1312.33 |
| ntp 4.2.4p2-RC5 | 74230 | 8 | 0 | 0 | 100% | 326.44 | 30 | 26 | 0 | 14% | 558.16 |
| ntp 4.2.5p66 | 74277 | 10 | 0 | 0 | 100% | 319.33 | 13 | 4 | 3 | 56% | 493.39 |
| bind 9.4.1p1 | 184204 | 951 | 0 | 0 | 100% | 14984.72 | 5 | $^*$2 | 3 | 0% | $^\sharp$2436.88 |
| openldap 2.4.4a | 223595 | 163 | $^*$14 | 50 | 88% | 8098.48 | 20 | 15 | 2 | 27% | 200.02 |
| TOTAL | 685408 | 1459 | 38 | 87 | 97% | 40226.40 | 99 | 69 | 9 | 23% | 20875.09 |

**Table 2: NULL Pointer Dereference Checking Results. "LOC (code)" indicates the number of lines of true (after preprocessing) code. "Reports" is the total number of warnings produced on the benchmark. "Bugs" is the number of true bugs found. Starred ($^*$) bug numbers represent our best-effort confirmation when we could not get confirmations from developers. Bug numbers without the star have been confirmed by the developers of the corresponding benchmark. "Unknown" shows the number of reports that could not be proved either feasible or infeasible. "FP Rate" gives the false positive rate, calculated as $1 - \#Bugs/(\#Reports - \#Unknowns)$. "Time" is the total runtime in seconds. The $^\sharp$ indicates that on bind, CALYSTO's runtime does not include instances on which CALYSTO failed to complete — it ran out of memory on 8 compilation units (taking an additional 6263.57 sec), and timed out in one day on one compilation unit. Experiments were on a dual Opteron 2.8 GHz with 16 GB RAM.**

| Benchmark | CALYSTO v1.5 | | |
|---|---|---|---|
| | Total time [s] | SPEAR [s] | Percentage |
| bftpd 1.8 | 3.14 | 1.25 | 39.8% |
| bftpd 1.9.2 | 2.86 | 0.88 | 30.7% |
| HYPERSAT 1.7 | 14.57 | 0.10 | 0.6% |
| spin 4.3.0 | 6858.10 | 473.50 | 6.9% |
| openssh 4.6p1 | 8995.64 | 8167.36 | 90.7% |
| inn 2.4.3 | 1312.33 | 14.77 | 1.1% |
| ntp 4.2.4p2-RC5 | 558.16 | 56.38 | 10.1% |
| ntp 4.2.5p66 | 493.39 | 58.03 | 11.7% |
| bind 9.4.1p1 | $^\sharp$2436.88 | 980.48 | 40.2% |
| openldap 2.4.4a | 200.02 | 181.90 | 90.9% |
| TOTAL | 20875.09 | 9934.65 | 47.5% |

**Table 3: CALYSTO Total Runtime Split. The SPEAR column shows the time spent in the theorem prover, with the next column showing the percentage of the total runtime.**

| Benchmark | Saturn v1.1 | | |
|---|---|---|---|
| | Total time [s] | Minisat [s] | Percentage |
| bftpd 1.8 | 129.51 | 17 | 13.1% |
| bftpd 1.9.2 | 105.17 | 8 | 7.6% |
| HYPERSAT 1.7 | 647.21 | 232 | 35.8% |
| spin 4.3.0 | 2129.04 | 457 | 21.4% |
| openssh 4.6p1 | 3707.81 | 988 | 26.6% |
| inn 2.4.3 | 9879.69 | 2104 | 21.2% |
| ntp 4.2.4p2-RC5 | 326.44 | 82 | 25.1% |
| ntp 4.2.5p66 | 319.33 | 80 | 25.0% |
| bind 9.4.1p1 | 14984.72 | 1141 | 7.6% |
| openldap 2.4.4a | 8098.48 | 949 | 11.7% |
| TOTAL | 40226.40 | 6058 | 15.0% |

**Table 4: Saturn Total Runtime Split. Saturn's theorem prover is the SAT solver Minisat.**

sults) in CALYSTO's interprocedural analysis.

We also analyzed how much time the two checkers spend in theorem-prover calls. Results are in Tables 3 and 4. CALYSTO spends almost 50% of its time in theorem prover calls, even with a small timeout (10 s) and a fast bit-vector arithmetic prover. The amount of time spent in the theorem prover calls is unsurprising, given how difficult the computed VCs are. Despite using a slower theorem prover [24], Saturn spends a much smaller fraction of its time in theorem prover calls. As mentioned earlier, Saturn performs expensive simplification using BDDs during static analysis, whereas we use a fast and incomplete expression simplifier and maximally-shared graphs. Also, because of Saturn's approximate interprocedural analysis, Saturn's VCs are likely much simpler. The different tool design shows up in the different time proportions, but both tools end up being usably fast.

## 6. CONCLUSION AND FUTURE WORK

We have presented CALYSTO, an extended static checker that provides an unprecedented combination of precision and scalability. Among fully automatic tools, CALYSTO is more scalable than anything with comparable coverage and precision, and offers better coverage and precision than anything with comparable scalability. This paper summarizes the key ideas to achieves these results.

Obvious lines of future work are more precise handling of pointer arithmetic, loops, and recursion. There are promising theoretical results in these areas, which we would like to explore. We also believe there is considerable room to further improve the performance of SPEAR, based on additional exploitation of problem structure. The days of fully automatic, thorough, and highly precise static checking of multi-million-line code bases are near.

## Acknowledgments

We would like to thank the LLVM Group, particularly Chris Lattner and Reid Spencer, for their help with LLVM. We thank Gerard J. Holzmann (Spin), Jesse Smith (Bftpd), and Harlan Stenn (ISC, Ntp) for providing us with prompt feedback both on bug reports and CALYSTO in general. Zvonimir Rakamarić has provided invaluable feedback on many ideas that ended up in CALYSTO and SPEAR. Finally, we would like to thank Alex Aiken, Isil Dillig, Tom Dillig, and Peter Hawkins for helpful and timely answers on using Saturn properly and the status of the Saturn project.

## 7. REFERENCES

[1] Saturn-discuss mailing list archives, August 2007. https://mailman.stanford.edu/pipermail/saturn-discuss/.

[2] D. Babić and A. J. Hu. Exploiting Shared Structure in Software Verification Conditions. *Third International Haifa Verification Conference, HVC 2007,* LNCS vol 4899, pp. 169–184. Springer, 2008.

[3] D. Babić and A. J. Hu. Structural Abstraction of Software Verification Conditions. *Computer Aided Verification (CAV),* LNCS vol 4590, pp. 371–383. Springer, 2007.

[4] D. Babić and M. Musuvathi. Modular Arithmetic Decision Procedure. Technical Report TR-2005-114, Microsoft Research Redmond, 2005.

[5] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. *Programming Language Design and Implementation (PLDI),* pp. 203–213. ACM Press, 2001.

[6] T. Ball and S. K. Rajamani. The Slam project: debugging system software via static analysis. *Principles of Prog Languages (POPL),* pp. 1–3. ACM Press, 2002.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma. What 's decidable about arrays? *Verification, Model Checking, and Abstract Interpretation: (VMCAI),* volume 3855 of *LNCS*, pp. 427–442. Springer, 2006.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004),* LNCS vol 2988, pp. 168–176. Springer, 2004.

[10] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Form. Methods Syst. Des.*, 25(2-3):105–127, 2004.

[11] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Workshop on Logics of Programs*, pp. 52–71, May 1981. Published 1982 as LNCS Vol 131.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *POPL*, pp. 238–252, 1977.

[13] D. W. Currie, A. J. Hu, S. Rajan, and M. Fujita. Automatic formal verification of DSP software. *37th Design Automation Conference*, pp. 130–135. ACM/IEEE, 2000.

[14] 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.

[15] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report SRC-RR-159, Digital Equipment Corporation, Systems Research Center, 1998. Now available from HP Labs.

[16] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer, 1990.

[17] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. *SIGPLAN Not.*, 42(6):435–445, 2007.

[18] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. *Operating Systems Design and Implementation (OSDI),* 2000.

[19] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. *Programming Language Design and Implementation (PLDI),* pp. 234–245. ACM Press, 2002.

[20] D. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. *28th Intl Conf on Software Engineering*, pp. 252–261, 2006.

[21] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. *POPL'2002: Principles of Programming Languages*, pp. 58–70. ACM Press, 2002.

[22] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[23] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[24] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu. Boosting Verification by Automatic Tuning of Decision Procedures. *Formal Methods in Computed Aided Design (FMCAD),* pp. 27–34. IEEE Computer Society Press, 2007.

[25] S. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.

[26] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[27] A. Kölbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Intl J of Parallel Programming*, 33(6):645–666, 2005.

[28] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *CGO '04: Intl Symp on Code Generation and Optimization*, pp. 75–88. IEEE Computer Society, 2004.

[29] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[30] K. R. M. Leino and F. Logozzo. Loop invariants on demand. *APLAS*, volume 3780 of *LNCS*, pp. 119–134. Springer, 2005.

[31] Y. V. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, 1993.

[32] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. *5th Intl Symp on Programming*, pp. 337–351. Springer, 1981. LNCS Vol 137.

[33] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular model checking framework. *European Software Engineering Conf./Symp. Foundations of Software Engineering (ESEC/FSE)*, pp. 267–276. ACM Press, 2003.

[34] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970,* pp. 466–483. Springer, 1983.

[35] P. Tu and D. A. Padua. Efficient Building and Placing of Gating Functions. *Programming Language Design and Implementation (PLDI)*, pp. 47–55, 1995.

[36] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[37] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *Programming Language Design and Implementation (PLDI),* pp. 131–144. ACM Press, 2004.

[38] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *Principles of Programming Languages (POPL),* pp. 351–363. ACM Press, 2005.