Resolving and Applying Constraint Queries on Context-Sensitive Analyses

James Ezick
Department of Computer Science
Cornell University
ezick@cs.cornell.edu

ABSTRACT

A context-sensitive analysis is an analysis in which program elements are assigned sets of properties that depend upon the context in which they occur. For analyses on imperative languages, this often refers to considering the behavior of statements in a called procedure with respect to the call-stack that generated the procedure invocation. Algorithms for performing or approximating these types of analyses make up the core of interprocedural program analysis and are pervasive, having applications in program comprehension, optimization, and verification. However, for many of these applications what is of interest is the solution to the dual problem: given a vertex and a desirable set of properties, what is the set of potential stack-contexts leading to that vertex that results in the desirable property set? Many techniques, such as procedure cloning, have been developed to approximately partition the set of stack-contexts leading to a vertex according to such a condition. This paper introduces a broad generalization of this problem referred to as a constraint query on the analysis. This generalization allows sophisticated constraints to be placed on both the desirable property set as well as the set of interesting stack-contexts. From these constraints, a novel technique based on manipulating regular languages is introduced that efficiently produces a concise representation of the exact set of stack-contexts solving this dual problem subject to the constraints. This technique is applied to a pair of emerging software engineering challenges - resolving program comprehension queries over aggregate collections of properties and statically modifying code to enforce a safety policy decidable by the analysis. Practical examples of both applications are presented along with empirical results.

Categories and Subject Descriptors: F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*program analysis*

General Terms: Languages, Theory

Keywords: Static Analysis, Context-Sensitive Analysis

PASTE'04, June 7-8, 2004, Washington, DC, USA.

1. INTRODUCTION

Context-sensitive analyses are analyses in which program elements are assigned sets of properties that depend upon the context in which they occur. For imperative programs, the most common instances of these analyses are ones in which the effect of statements within a procedure are considered in the context of the procedure's invoking call-stack; its *stack-context*. One popular approach for generating analyses of this type is the so-called *second-order* approach [6]. In this approach, transfer functions are derived for each vertex and procedure that translate a set of data-flow facts that hold before the relevant procedure is considered into a set of facts that hold after the vertex or procedure has been considered. This technique allows a meet over all paths solution to a data-flow analysis problem to be found in such a way that only those paths that respect the natural call and return structure of the program are considered. Given a vertex and a stack-context, it is possible to use the transfer functions generated for the vertex and stack elements to obtain the precise set of properties that hold at the vertex for that stack-context. Analyses that fit this framework have numerous applications to program comprehension, optimization, and verification.

However, for many applications what is of interest is not the set of properties associated with a specific vertex, stack-context pair. Rather, what is desired is the solution to the dual problem: given a vertex and a desirable set of properties, what is the set of potential stack-contexts leading to that vertex that result in the desirable property set? This paper makes three contributions. First, a definition of context-sensitive analyses is provided that separates the means of generating the analysis from the representation of its output. This definition abstracts away distinctions between forward and backward-flow analyses as well as many popular optimizations for demand based analyses. Second, a broad generalization of the dual problem is introduced in the form of the *constraint query* that takes formalized restrictions on the set of stack-contexts and the desired property set. A practical technique based on manipulating regular languages is presented for resolving these queries. This technique produces an automaton that precisely accepts the set of stack-contexts that constitute the solution to the query without requiring the computation of any additional fixed points. Finally, this paper provides empirical data from applying constraint queries to a pair of emerging software engineering problems.

The remainder of this paper is organized as follows: Section 2 introduces a simple example that motivates the utility

```
     Global Integer G

v0:  Procedure A() {          v10: Procedure C() {
v1:    if (...cond_1...) {    v11:   print(G);
s2:      C();                 v12:   G := 1;
s3:      B();                 v13: }
       }
v4:  }                        v14: Procedure main() {
                              v15:   G := 0;
v5:  Procedure B() {          v16:   if (...cond_3...) {
v6:    if (...cond_2...) {    s17:     A();
s7:      A();                       } else {
s8:      C();                 s18:     B();
       }                            }
v9:  }                        v19: }
```

Figure 1: Sample Program with Mutual Recursion

of constraint queries. Section 3 abstracts context-sensitive analyses. The technique for posing and resolving constraint queries is then presented in Section 4. Section 5 provides experience from two practical applications of constraint queries. First, constraint queries are posed on a live variable analysis. These demonstrate how aggregate property constraints can be used to reveal interesting, non-trivial aspects of a program's behavior. A second application that extends recent work on detecting format string vulnerabilities in C is presented next. Here, the outputs from a set of constraint queries become part of a set of code transformations that precisely enforce a safety policy that prevents this vulnerability from being exploited. This is done without requiring any code replication. Finally, Section 6 draws conclusions and discusses some related work.

2. A MOTIVATING EXAMPLE

Figure 1 introduces the skeleton of an imperative program exhibiting mutual recursion between two procedures, `A` and `B`. The labels correspond to elements of the vertex set, $\mathcal{V}$, in the standard interprocedural control-flow graph (ICFG) representation of the program. The s-designated labels also refer to elements of the set of call-edges, Σ, by their unique source. For example, s_2 is a call-vertex in procedure `A` that is the source of a call-edge, (s_2, v_{10}), to procedure `C`. Recall that all interprocedural control-flow graphs have a singular `main` procedure whose entry and exit-vertices represent the beginning and end of the program, respectively.

Given the program of Figure 1, consider the following:

What paths in the program to the assignment at v_{12} may lead to the print statement at v_{11} where the call stack at v_{12} begins with successive calls to procedures `A` *and* `B`*?*

Even for such a trivial program, this is a non-trivial code comprehension exercise due to the mutual recursion. In fact, the solution to this query will later be seen to contain an infinite number of paths.

As an extension of the first query, now consider this second exercise:

Modify the code so that when the assignment at vertex v_{12} is reached it does not execute when the print statement at v_{11} may execute in the future and v_{12} has been reached by successive calls to procedures `A` *and* `B`*.*

This is an example of a safety property. Inserting a runtime check to enforce it requires knowledge of both past and possible future events. Enforcement by cloning [2] the procedure `C` is made difficult by the fact that there are only two call sites to `C` and each leads to instances where the assignment statement should be skipped and instances where it should be executed. The next two sections present a framework that efficiently solves both of these exercises.

	Context Transformers		Property Transformers	
v	$[\Gamma(v)](\alpha)$	$[\Gamma(v)](\beta)$	$[\Phi(v)](\alpha)$	$[\Phi(v)](\beta)$
v_0			{p}	{p}
v_1			{p}	{p}
s_2	β	β	{p}	{p}
s_3	α	β	{p}	{p}
v_4			$\emptyset$	{p}
v_5			{p}	{p}
v_6			{p}	{p}
s_7	β	β	{p}	{p}
s_8	α	β	{p}	{p}
v_9			$\emptyset$	{p}
v_{10}			{p}	{p}
v_{11}			{p}	{p}
v_{12}			$\emptyset$	{p}
v_{13}			$\emptyset$	{p}
v_{14}			{p}	{p}
v_{15}			{p}	{p}
v_{16}			{p}	{p}
s_{17}	α	β	{p}	{p}
s_{18}	α	β	{p}	{p}
v_{19}			$\emptyset$	{p}

The s-designated vertices also identify the elements of Σ by call-edge source.

Figure 2: A Context-Sensitive Analysis, $(\{\alpha, \beta\}, \{p\}, \Gamma, \Phi, \alpha)$, over the Sample Program

3. CONTEXT-SENSITIVE ANALYSES

Given vertex set, $\mathcal{V}$, and call-edge set, Σ, it is possible to formally define the notion of a stack-context for an interprocedural control-flow graph.

DEFINITION 1. *A* **stack-context**, *$\bar{\sigma}$, is a finite sequence of the call-edges comprising Σ. A stack-context is a* **valid stack-context** *for $v \in \mathcal{V}$ if and only if $\bar{\sigma} = \epsilon$ and v is a vertex in procedure main or $\bar{\sigma} = \sigma_0 \ldots \sigma_n$, the source of σ_0 is a call-vertex in main, for each $i : 0 \leq i < n$ the target of σ_i is the entry-vertex of the procedure containing the source of σ_{i+1}, and the target of σ_n is the entry-vertex of the procedure containing v.*

Note that the set of valid stack-contexts for a vertex can be infinite. As an example, $s_{17}s_3s_7s_2$ is a one of an infinite number of valid stack-contexts for v_{12}. The path generating this stack-context is also one solution to the first exercise.

DEFINITION 2. *A* **context-sensitive analysis** *is an ordered quintuple $(\mathcal{C}, \mathcal{X}, \Gamma, \Phi, \kappa)$, where $\mathcal{C}$ is a finite set of* **contexts**, *$\mathcal{X}$ is a set of* **properties**, *$\Gamma : \Sigma \to (\gamma : \mathcal{C} \to \mathcal{C})$ is a collection of* **context transformers** *that associate to each element of Σ a function, γ, mapping a context to a context, $\Phi : \mathcal{V} \to (\phi : \mathcal{C} \to 2^{\mathcal{X}})$ is a collection of* **property transformers** *that associate to each element of $\mathcal{V}$ a function, ϕ, mapping a context to a subset of the set of properties, and $\kappa \in \mathcal{C}$ is the* **initial context**.

The notion of a *context* introduced by this definition is separate from the previous notion of a *stack-context*. A context in the scope of a context-sensitive analysis corresponds

to a set of assumptions that hold when determining the set of properties associated with a vertex.

A context-sensitive analysis, $(\{\alpha, \beta\}, \{p\}, \Gamma, \Phi, \alpha)$, applicable to the exercises is provided in Figure 2. Intuitively, the property p denotes that the print statement at v_{11} in procedure `C`, may occur in the future. Contexts α and β distinguish between assumptions about whether p holds at the exit of a procedure, with α being the assumption that is does not hold.

For analyses derived as the product of a second-order fixed point computation, the translation to this abstraction is trivial. The transformers are taken from the transfer functions and the contexts are simply a renaming of the property sets that are needed as domains to the various property transformers.

While it is necessary that both Γ and Φ be total functions, it is not strictly necessary that a total function be provided for each element in their respective ranges. Specifically, given $\sigma \in \Sigma$ and $c \in \mathcal{C}$, the analysis need only provide a result for $[\Gamma(\sigma)](c)$ if there is some valid stack-context, $\bar{\sigma}\,\sigma$, for the target of σ such that $\Gamma^*(\bar{\sigma}) = c$. Likewise, given $v \in \mathcal{V}$, $\Phi(v)$ need only be provided for $[\Phi(v)](c)$ if there is some valid stack-context, $\bar{\sigma}$ for v such that $\Gamma^*(\bar{\sigma}) = c$. This relaxation of the totality requirement allows this abstraction to represent demand analyses where transformers are only defined for contexts for which there exists some valid stack-context. However, the constructions of Section 4 will assume that the functions are total, with the assumptions that $\gamma(c) = \kappa$ and $\phi(c) = \emptyset$ if γ or ϕ is not defined on c.

Definition 3. *Given a context-sensitive analysis, the* **stack-context transformer**, $\Gamma^* : \Sigma^* \to \mathcal{C}$, *induced by the analysis maps a stack-context to a context in the analysis,*

$$\Gamma^*(\sigma_0 \ldots \sigma_n) = [\Gamma(\sigma_n) \circ \cdots \circ \Gamma(\sigma_0)](\kappa).$$

Definition 4. *Given a context-sensitive analysis, $v \in \mathcal{V}$, and $\bar{\sigma}$, a stack-context, the* **solution** *for vertex v in stack-context $\bar{\sigma}$ is the subset of properties, $\mathcal{X}$,*

$$\rho(v, \bar{\sigma}) = [\Phi(v)](\Gamma^*(\bar{\sigma})).$$

The transformer does not require the stack-context to be valid for any particular vertex. The solution maps a vertex and a stack-context to the set of properties that hold at that vertex in the given stack-context. The solution is found by first mapping the stack-context down to a context in the analysis and then applying the property transformer associated with the vertex to that context. Recalling the previous example, the solution associated with the vertex v_{12} in the valid stack-context $s_{17}s_3s_7s_2$ is defined as

$$[\Phi(v_{12})][\Gamma(s_2) \circ \Gamma(s_7) \circ \Gamma(s_3) \circ \Gamma(s_{17})](\alpha) = \{p\}.$$

This confirms the observation that the print statement associated with vertex v_{11} could be executed after the execution of the assignment statement associated with vertex v_{12} if that statement was reached from the example stack-context which leads with successive calls to procedures `A` and `B`. This confirms that this stack-context corresponds to one solution to the first exercise.

4. RESOLVING CONSTRAINT QUERIES

This section introduces the notion of a *constraint query* and shows how such queries can be used to solve the exercises posed in Section 2.

Definition 5. *Given a context-sensitive analysis, $(\mathcal{C}, \mathcal{X}, \Gamma, \Phi, \kappa)$, over an ICFG, G, with call-edge alphabet Σ, a* **constraint query** *on the analysis takes an ordered triple, (v, c, Δ), where v is a vertex in G, c is a regular expression over Σ, and Δ is a decidable subset of $2^{\mathcal{X}}$, and returns an automaton, M, that accepts $\bar{\sigma} \in \Sigma^*$ if and only if*

1. *$\bar{\sigma}$ is a valid stack-context for v, and*
2. *$\bar{\sigma}$ is in the language of c, and*
3. *$\rho(v, \bar{\sigma}) \in \Delta$.*

Given this formulation, the set of stack-contexts that solve the first exercise of Section 2 can be seen as the solution to the constraint query: $(v_{12}, (s_7 + s_{17})(s_3 + s_{18})(s_2 + s_3 + s_7 + s_8 + s_{17} + s_{18})^*, \{\{p\}\})$ on the analysis of Figure 2.

The objective of a constraint query is to find the valid stack-context pre-image of ρ given a vertex, v, and a *solution constraint*, Δ. The *stack-context constraint*, c, further restricts that pre-image to some interesting subset. Resolving and applying constraint queries revolves around intersecting regular languages that encode each of the constraints.

4.1 Valid Stack-Contexts

The set of valid stack-contexts for a vertex, v, can be described as a regular language over the set of call-edges, Σ. This is demonstrated by constructing a finite automaton that precisely accepts the set of valid stack-contexts for v.

Theorem 1. *Given v, a vertex in an ICFG, G, the set of valid stack-contexts for v is precisely accepted by a finite automaton, $M_v = (\mathcal{P} \cup \{\epsilon\}, \Sigma, main, \delta, \{P_v\})$, where, $\mathcal{P} \cup \{\epsilon\}$ is the set of procedures of G, $\mathcal{P}$, plus a dead state, ϵ, Σ is the set of call-edges in G, the procedure main in G is the initial state, $\delta(q_s, \sigma) = q_t$ where q_t is the procedure in G containing the target of σ if and only if $q_s \neq \epsilon$ and q_s is the procedure containing the source of σ, otherwise $q_s = \epsilon$, and $\{P_v\}$ is the singleton set of accepting states consisting only of the procedure containing v in G.*

Proof. If $\bar{\sigma} = (\sigma_0 \ldots \sigma_n)$ is a valid stack-context for v then the source of σ_0 is *main*, the initial state of M_v. For each σ_i in $\bar{\sigma}$, $\delta(q_s, \sigma_i) = q_t$ in M_v, where q_s is the procedure containing the source of σ_i and q_t is the procedure containing the target of σ_i since σ_i is an element of a valid stack-context for v. Finally, the target of σ_n must reside in the procedure containing v and this is an accepting state of M_v. Thus, M_v accepts $\bar{\sigma}$.

Conversely, if M_v accepts $\bar{\sigma}$, then there is a sequence of states, $q_0 \ldots q_n$, that accepts $\bar{\sigma}$. Since no accepting sequence contains ϵ, each state in the sequence is a procedure in G. By definition of M_v, q_0 is the procedure *main* in G. For each subsequence q_iq_{i+1} in the accepting sequence $\delta(q_i, \sigma_i) = q_{i+1}$ thus, σ_i is a call-edge whose source is a call-vertex in the procedure q_i and whose target is the entry-vertex of the procedure q_{i+1}. Finally, since q_n is the single accepting state, it must be the procedure containing v. Hence, $\bar{\sigma}$ is a valid stack-context for v. □

Weihl [7] has shown that constructing the call graph (which is required to build M_v in Theorem 1) is PSPACE-hard for recursive programs with function pointers. However, a graph that conservatively over-estimates the set of possible calls can be used. M_v then accepts stack-contexts that are valid with respect to the supplied call graph.

Figure 3 illustrates the finite automaton that accepts L_v, the language of valid stack-contexts for v_{12}.

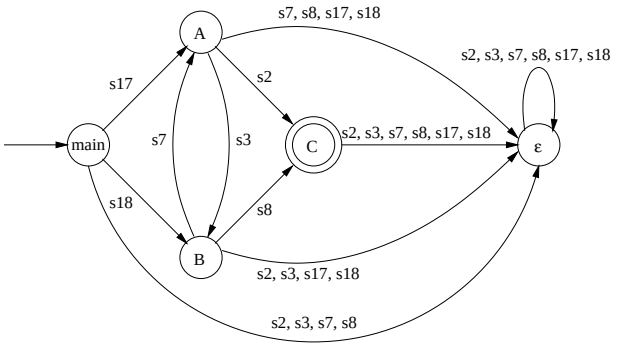

Figure 3: Automaton Accepting L_v, the Language of Valid Stack-Contexts for v_{12}

4.2 Stack-Context Constraints

A stack-context constraint is a regular expression over the set of call-edges, Σ, in an ICFG, G. To simplify the expression of these constraints, $\Pi_P = \pi_0 + \cdots + \pi_n$, where the $\pi_i \in \Sigma$ are the call-edges to the entry-vertex of procedure P. This macro expression captures the notion of any to call to a specified procedure. Similarly, the symbol Ω stands for the regular expression $(\sigma_0 + \cdots + \sigma_n)^* = \Sigma^*$, where $\Sigma = \{\sigma_0, \ldots, \sigma_n\}$. For stack-context constraints, Ω is useful as a wildcard literal standing for an arbitrary sequence of call-edges. Using this notation, the stack-context constraint of the constraint query for solving the first exercise is $L_c = \Pi_{\mathtt{A}}\Pi_{\mathtt{B}}\Omega$.

Regular expressions are powerful enough to express most of the constraints that are of interest for this type of query. They can encode fixed sequences of calls as well as any criteria based on a finite decision tree of calls. The Kleene-star operator allows constraints to include the notion or zero or more calls to a recursive function as well as notions of every n-th invocation of a recursive function. Finally, the closure properties of regular expressions allow them to express the conjunction, disjunction and negation of constraints.

4.3 Solution Constraints

A solution constraint for a constraint query is a decidable subset of the power set of the properties of a context-sensitive analysis. Given such a constraint, there exists a regular language, again over the alphabet of call-edges, Σ, that precisely includes the set of stack-contexts generating an element of the constraint as a solution for a vertex v. This is again demonstrated by constructing a finite automaton that precisely accepts such a language.

THEOREM 2. *Given v, a vertex in an ICFG, G, a context-sensitive analysis, $(\mathcal{C}, \mathcal{X}, \Gamma, \Phi, \kappa)$ over G, and a solution constraint Δ, a decidable subset of $2^{\mathcal{X}}$, then the set of stack-contexts, $\bar{\sigma}$, for v such that $\rho(v, \bar{\sigma}) \in \Delta$, is precisely accepted by a finite automaton, $M_\Delta = (\mathcal{C}, \Sigma, \kappa, \delta, A)$, where $\mathcal{C}$ is a set of contexts in the analysis, Σ is the set of call-edges in G, κ is the initial context of the analysis, $\delta(c_i, \sigma) = [\Gamma(\sigma)](c_i)$, and A is the accepting set of contexts, c, such that $[\Phi(v)](c) \in \Delta$.*

PROOF. Given a stack-context, $\bar{\sigma} = \sigma_0 \ldots \sigma_n$, by definition of M_Δ, $\delta(\delta(\ldots\delta(\kappa, \sigma_0)\ldots, \sigma_{n-1}), \sigma_n) = \Gamma(\sigma_n) \circ \cdots \circ \Gamma(\sigma_0) = \Gamma^*(\bar{\sigma})$. Thus, if $\rho(v, \bar{\sigma}) = [\Phi(v)](\Gamma^*(\bar{\sigma})) \in \Delta$ then M_Δ terminates in state $c = \Gamma^*(\bar{\sigma})$ which is an accepting state. Thus, M_Δ accepts $\bar{\sigma}$.

Conversely, if M_Δ accepts $\bar{\sigma}$ then M_Δ terminates in a context, c, such that $[\Phi(v)](c) = [\Phi(v)](\Gamma^*(\bar{\sigma})) = \rho(v, \bar{\sigma}) \in \Delta$. Therefore, $\bar{\sigma}$ satisfies the solution constraint. □

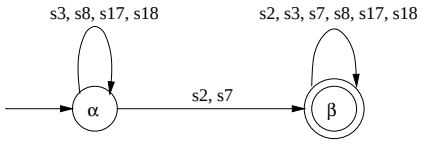

Figure 4: Automaton Accepting L_Δ, the Language of Stack-Contexts, $\bar{\sigma}$, such that $\rho(v_{12}, \bar{\sigma}) \in \Delta$

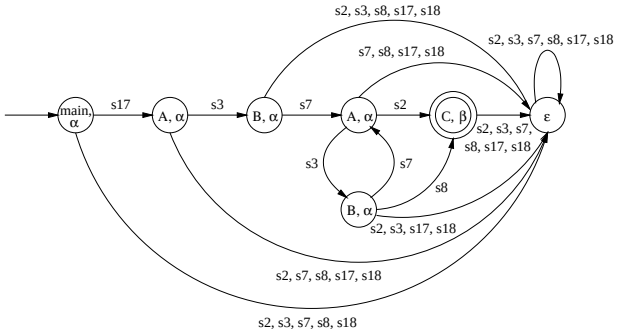

Figure 5: Minimal Automaton Accepting $L = L_v \cap L_c \cap L_\Delta$, the Language of Solutions to the Query

It is not necessary to explicitly generate the set Δ. It is adequate to define a constraint by a function that decides it. Further, any choice of Δ can be made decidable by intersecting it with the finite set of property sets that are mapped to by the the finite number of (vertex, context) pairs.

Figure 4 illustrates the finite automaton that accepts L_Δ, the language of stack-contexts, $\bar{\sigma}$, such that $\rho(v_{12}, \bar{\sigma}) = \{p\}$.

4.4 Resolving a Constraint Query

Given a context-sensitive analysis over an ICFG with call-edge alphabet Σ, let L_v be the language corresponding to the valid stack-context requirement, let L_c be the language corresponding to the stack-context constraint, and let L_Δ be the language corresponding to the solution constraint. Each is a regular language contained in Σ^*. The solution to the constraint query is then the automaton accepting

$$L = L_v \cap L_c \cap L_\Delta.$$

The number of states in the intersection automaton, L, is bounded above by the product of the number of states in the automata accepting L_v, L_c, and L_Δ. However, for most practical applications, it tends to be significantly smaller.

This automaton, which can be minimized, encapsulates the set of solutions to the constraint query. It precisely accepts the set of valid stack-contexts that meet the stack-context constraint as well as the solution constraint and thus satisfy all three criteria of the definition.

Figure 5 illustrates the solution to the example constraint query. This automaton exactly captures the infinite set of stack-contexts, and thus implicitly the set of paths in the program, that are solutions to the first exercise.

This automaton also provides a mechanism for solving the second exercise. To enforce the safety policy, the au-

tomaton resulting from the constraint query is injected as a guard over the assignment statement. When this automaton is reached it is applied to the current dynamic stack-context. If the automaton accepts the stack-context then the assignment is skipped, otherwise it is executed. Thus, enforcement of the safety policy does not require replicating any code.

In essence, the construction of the automaton has extracted the minimal information from the analysis that is necessary to solve the problem and restricted it according to the constraints. Once the automaton has been constructed, the output of the original analysis can be discarded.

5. APPLICATIONS

This section provides experience from two real-world applications of this technique. To increase efficiency, the states and transitions for the automata accepting each of the constraints are restricted to the procedures, contexts, and call-edges that are relevant to accepting computations before the intersection automaton is constructed. The CPU performance data was generated from a Scheme implementation of the various regular language routines executing on a 2.0 GHz Pentium 4 with 1 GB of RAM running Windows XP.

5.1 Comprehension on a Liveness Analysis

Code comprehension refers to the problem of programmers asking questions and getting answers about the behavior of a program. Comprehension is an essential element in both extending legacy source bases as well as ensuring the quality of software. Constraint queries provide a mechanism for exploiting context-sensitive analyses for this purpose.

Treecode [1] is a C implementation of a popular algorithm for performing n-body simulation based on octtrees. A context-sensitive live variable analysis is supplied for this code. This analysis, given a statement and a stack-context, returns the set of program variables that may be used from that point before they are redefined or the program terminates. Properties in this analysis correspond to the liveness of the individual program variables, and the contexts correspond to sets of variables that are possibly live at the end of some procedure.

Figure 6 shows performance information on a selection of constraint queries made over this analysis. The queries are meant to answer meaningful, non-trivial, questions about the live variable sets associated with different points in the code that could be useful for understanding or modifying the behavior of the program. Variables `root` and `bodytab` are the entry points for the octtree and body position-velocity vector data structures, respectively. The code contains a routine, *savestate*, for outputting its state to a file and *Saved* refers to the set of variables that the routine outputs. Locations are provided as a procedure name followed by a source code line number. *Automaton States* refers to the number of states in the minimized automaton accepting each of the constraint languages presented in Section 4. *Time* gives the number of milliseconds required to resolve the constraint query given the output of the analysis.

Notice that the solution constraints take advantage of *aggregate properties* of the analysis. That is, the queries are not limited to the liveness of particular combinations of variables, but also include other decidable properties of the live variable set such as its cardinality and memory footprint. A query such as the one associated with the location in `savestate` could be used to check if there are cases where elements of the live variable set are being omitted by the state saving routine. As the performance data in Figure 6 illustrates, this is a practical technique for resolving actual queries over a context-sensitive analysis for a real program.

5.2 Preventing Format String Vulnerabilities

Format string vulnerabilities are a class of security exploit in C that have arisen from design features in the C standard library coupled with a problematic implementation of variable argument functions. In C, certain library functions, such as `printf`, operate on what are called format strings. These are strings that include format commands using the familiar "`%`" syntax. Recent work [5] has chronicled that when these format string commands are passed malicious arguments in which the number of format string literals exceeds the number of actual arguments, a security vulnerability can result that is significant enough in some cases to cause arbitrary code to be executed.

A context-sensitive analysis is provided for this code that tracks tainted data. The notion of tainted data corresponds to data in the program that can not be trusted not to exploit this vulnerability. Tainted data is usually introduced either through command line arguments or specific read operations. Unlike the live variable analysis of Section 5.1, this analysis is not separable over the data. Rather, the tainted data generated by some statements depends upon other elements of the set of tainted data entering those statements. For example, the left hand side of an assignment statement can become tainted when an element of the right hand side expression is tainted.

Given this analysis, constraint queries are used in a manner similar to the second exercise of Section 2 to enforce a safety policy that requires the program halt if a format string command is about to execute with tainted data. Constraint queries are posed on the analysis for the vertex corresponding to each instance of a format string command. The stack constraint for each query is Ω and the solution constraint is that the tainted set solution contains one or more of the sensitive arguments to the string command.

Each query produces a minimal automaton that accepts only the set of valid stack-contexts that correspond to instances of the command that violate the policy. If no vulnerability exists, then the automaton degenerates to the single state machine that trivially rejects all inputs. In these cases, the automaton can be discarded. If a vulnerability exists that is independent of the calling context, then $L_v \rightarrow L$. This is tested by checking the emptiness of $L_v \cap L_\Delta^c$. In these cases, the automaton can be discarded and the offending command replaced with halt. In all other cases, code to apply the current stack-context is injected to guard the command the first time it executes since the last time the context has changed. If the automaton accepts the stack-context then the program halts. In this way, the safety policy is enforced through a purely static code-transforming analysis without any additional programmer intervention.

Figure 7 provides empirical data from applying this analysis and code transformation to six example codes that have been seeded with tainted data to introduce simulated vulnerabilities. None of the benchmarks contained any naturally occurring context-sensitive vulnerability. Each row represents a unique benchmark. Column *FSC* gives the number of format string commands in the benchmark and *Non-trivial* gives the number of queries for which the resulting automa-

Query Constraints			Automaton States				CPU
Location	Context	Solution	M_c	M_v	M_Δ	M_L	Time (ms)
`maketree`:45	Ω	$\lambda S.[root \notin S]?$	1	6	1	6	80
`makecell`:109	Ω	$\lambda S.[root \in S]?$	1	8	5	8	120
`savestate`:328	$\Pi_{\texttt{output}}\Pi_{\texttt{savestate}}$	$\lambda S.[Saved \subset S]?$	5	6	1	1	90
`cputime`:40	$\Omega\Pi_{\texttt{stepsystem}}\Omega$	$\lambda S.[\lvert S\rvert > 35]?$	2	9	9	8	281
`gravsum`:212	$\Omega\Pi_{\texttt{stepsystem}}\Omega$	$\lambda S.[size(S) > 640\ bytes]?$	2	9	4	9	251
`allocate`:27	$\Omega\Pi_{\texttt{makecell}}\Omega$	$\lambda S.[root \notin S \vee bodytab \notin S]?$	2	15	1	9	441

Treecode: Lines = 2711, $|\mathcal{P}| = 67$, $|\Sigma| = 225$, $|\mathcal{C}| = 315$, and $|\mathcal{X}| = 953$

Figure 6: Examples of Constraint Queries for Program Comprehension

Benchmark						Query Set Output			CPU
Program	Lines	$\lvert\mathcal{P}\rvert$	$\lvert\Sigma\rvert$	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{X}\rvert$	FSC	Non-Trivial	Average	Time (s)
`postfix`	134	9	27	35	102	7	1	4	0.1
`compress`	1421	23	39	24	235	15	2	4	0.2
`art`	1270	24	46	52	318	29	2	4.5	0.4
`treecode`	2711	67	225	492	953	46	2	6	2.2
`bzip`	4652	67	208	109	715	71	3	13	2.6
`go`	29246	385	2063	957	4530	11	2	11	7.9

Figure 7: Format String Vulnerability Safety Policy Benchmarks

ton is required in the code to enforce the policy. *Average* gives the average number of states in the non-trivial automata. *Time* gives the time required to generate, resolve, and test for triviality all of the constraint queries necessary to enforce the policy given the output of the analysis.

6. CONCLUSION AND RELATED WORK

This paper has demonstrated how the problem of resolving constraint queries over a context-sensitive analysis can be reduced to the tractable problem of intersecting finite automata. Constraint queries are a broad generalization of the dual problem for context-sensitive analyses that has numerous software engineering applications. In Section 5.1, a selection of interesting program comprehension queries were recast as constraint queries and resolved over an example program. Section 5.2 demonstrated how a safety policy could be enforced through a collection of constraint queries. That section also demonstrated how the automaton generated by each query could then be integrated into a program and be used to decide the safety property in a way the fully exploits the precision of the context-sensitive analysis. This is an improvement on previous static techniques that required the set of call-sites to be strictly partitioned and could only distinguish the potential for aberrant behavior to some bounded stack-depth. Empirical data was provided for both applications that demonstrated that this is a practical approach to addressing these problems.

The notion of tainted data on which the analysis of Section 5.2 is based is borrowed from previous work [5] that detects format string vulnerabilities by using a type system that subtypes data as either tainted or untainted. Recent work [3] has demonstrated how interprocedural analyses could be recast as weighted pushdown automata. As as consequence of this reduction they demonstrate how it is possible to extract from an analysis a regular expression that captures the set of stack-contexts for a program element such that the element has a particular property in those contexts. They alluded to the possibility of using this functionality to solve problems similar to those discussed in this paper. Prior work [4] demonstrated how security policies could be enforced through *security automata*. In that model, program events, such as accessing a file, trigger a transition in a statically constructed state machine. When an event causes a transition to a terminal state the execution of the program is halted thus preventing a violation of the policy. This differs from the approach presented here in that the automata execute concurrently with the program. In the example of Section 5.2, the automata are injected into the code and then used to decide stack-contexts on demand. Thus, each automaton can be minimized and tuned to fit the precise instance where it is to be injected.

7. ACKNOWLEDGMENTS

Thanks to Keshav Pingali, Radu Rugina, and Andrew Myers for many valuable conversations. Thanks to GrammaTech, Inc. for making their software available and for being supportive of this research.

This work was supported by DARPA contract DAAH01-02-C-R120 and NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-0121401.

8. REFERENCES

[1] J. E. Barnes. A modified tree code: Don't laugh, it runs. *Journal of Computational Physics 87, 161*, 1990.

[2] M. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.

[3] T. W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *10th International Symposium on Static Analysis*, pages 189–213, June 2003.

[4] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[5] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, pages 201–220, 2001.

[6] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[7] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of POPL '80: 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 1980.