Context-Sensitive Program Analysis as Database Queries

Monica S. Lam John Whaley V. Benjamin Livshits Michael C. Martin
Dzintars Avots Michael Carbin Christopher Unkel

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

{lam,jwhaley,livshits,mcmartin,dra,mcarbin,cunkel}@cs.stanford.edu

ABSTRACT

Program analysis has been increasingly used in software engineering tasks such as auditing programs for security vulnerabilities and finding errors in general. Such tools often require analyses much more sophisticated than those traditionally used in compiler optimizations. In particular, context-sensitive pointer alias information is a prerequisite for any sound and precise analysis that reasons about uses of heap objects in a program. Context-sensitive analysis is challenging because there are over 10^{14} contexts in a typical large program, even after recursive cycles are collapsed. Moreover, pointers cannot be resolved in general without analyzing the entire program.

This paper presents a new framework, based on the concept of deductive databases, for context-sensitive program analysis. In this framework, all program information is stored as relations; data access and analyses are written as Datalog queries. To handle the large number of contexts in a program, the database represents relations with binary decision diagrams (BDDs). The system we have developed, called bddbddb, automatically translates database queries into highly optimized BDD programs.

Our preliminary experiences suggest that a large class of analyses involving heap objects can be described succinctly in Datalog and implemented efficiently with BDDs. To make developing application-specific analyses easy for programmers, we have also created a language called PQL that makes a subset of Datalog queries more intuitive to define. We have used the language to find many security holes in Web applications.

1. INTRODUCTION

While program analysis in the past has been used primarily in compiler optimizations, it is increasingly being used in tools to aid software development. One important use is program auditing. Software security vulnerabilities have caused billions of dollars in damages in recent years. Program auditing tools provide an active means of finding and fixing the errors other than just relying on software testing. Several program auditing tools have been found to be useful in finding generic errors common to many applications [12, 26]. However, programs also contain many non-generic, application-specific errors. For instance, many libraries are available, each of which has its own pitfalls. A GUI library may have resources that the programmer can easily mismanage, such as listener objects that the programmer fails to deregister. A database library may allow attackers to execute arbitrary commands if the application handles user input improperly. Because these errors are specific to individual programs, it is important to empower programmers to write their own analyses, so that they can find bugs specific to their applications. For example, upon finding a bug in a program, they may wish to generalize and look for other code that follows the same erroneous pattern.

Many of these analyses require reasoning about dynamically created objects. For instance, we may wish to know if strings containing user input are eventually used in insecure ways, or if an entry of an allocated resource is removed upon the deallocation of the object. To answer these questions accurately, we must be able to track pointers and references to these objects precisely, across different calling contexts.

1.1 Context-Sensitive Analysis

Context-sensitive pointer alias analysis is especially expensive, because it is not possible to tell in general what a reference or pointer can point to without analyzing the entire program. In addition, what a pointer points to in a particular method invocation depends greatly on the callers currently on the stack. Of course, there can be an unbounded number of calling contexts in any program, due to recursion. A reasonable approximation is to keep track of all calls not involved in recursive cycles. Even then, in real programs, we found that it is not unusual for Java applications to have over 10^{14} contexts. Coping with this explosion of contexts while still producing a context-sensitive pointer analysis result is necessary to answer any of a large number of interesting problems.

We have recently developed a new *cloning-based* technique for context-sensitive analysis, whereby results for all calling contexts are computed and explicitly represented [55]. We represent the exponentially many results by using binary decision diagrams or BDDs [10], which are data structures that can represent data with high levels of redundancy in a compact manner. This approach can be used to develop many other context-sensitive analyses.

This material is based upon work supported by the National Science Foundation under Grant No. 0326227, NSF Graduate Student Fellowships, Stanford Graduate Fellowships, and an Intel student fellowship.

PODS 2005 Washington, D.C. USA

While BDDs can be powerful, they are not easy to use, as shown vividly by our experience with the context-sensitive pointer analysis itself [55]. A direct implementation using BDDs ran out of memory and did not finish at all. After one man-year of effort in optimization, the algorithm was able to process some of the largest open-source Java programs in under 12 minutes. The algorithm itself took several thousand lines of Java code. Getting the algorithm to run efficiently involved applying many optimizations; doing so expanded the code, obscured the algorithm, and introduced subtle errors.

1.2 A Deductive Database Framework

We turned to deductive database technology to create a higher abstraction for our cloning-based approach to context sensitivity. The basic idea is to store the input program and all the information generated for every possible calling context in a database. The full power of a database query language, in this case Datalog [51], can be used to select the information of interest. More importantly, new analyses can also be defined relatively easily in Datalog.

A deductive database with a Datalog interface is well suited to context-sensitive program analysis for several reasons. First, program analysis is often expressed as computing the fixed-point to a set of inference rules. As Datalog allows recursive rule definitions, there is a simple and intuitive correspondence between inference rules and Datalog. Second, this approach provides a uniform interface for retrieving information, whether obtained directly from the source code, previously computed, or to be computed according to given Datalog rules. In program auditing or debugging tools, users are usually interested in specific statements or calling contexts. The database interface makes it easy for users to refer to an individual program fact or a subset of facts. Third, most importantly, because Datalog operates on entire relations at a time, Datalog rule resolution maps well onto BDD operations.

We have developed a deductive database called bddbddb, which stands for BDD-Based Deductive DataBase [55]. The database implementation uses a large number of optimizations drawn not only from the database, compiler, and BDD literature, but from the machine learning field as well.

1.3 Analyses in Datalog

We have developed a large number of program analyses in this framework. We have developed pointer alias analyses for both Java [55] and C [2]. These are the first context-sensitive, *inclusion-based* points-to analyses that have been demonstrated to scale to large programs. A points-to analysis is inclusion-based if two pointers can point to overlapping but not identical sets of targets.

We have used the C pointer analysis in tools for finding two critical security vulnerabilities in C: buffer overruns and format string vulnerabilities [2]. We have used the Java pointer analysis to find critical information such as the targets of virtual method calls, especially in the presence of reflection. We also used it to analyze multithreaded programs to determine the objects that escape the thread in which they are created [55]. Each of these analyses would have taken a long time to develop if not for the bddbddb framework.

To make it easy for programmers to write analyses, we have created a little language called PQL, which, among other things, provides syntactic sugar that allows an important subset of analyses to be specified intuitively. Programmers need not learn the names or meanings of the relations in our database; they can simply write code patterns of interest in Java, and the patterns are automatically translated into Datalog. With this approach, we were able to find dozens of previously unknown security errors in a variety of large, widely used open-source applications relatively easily.

1.4 Background

The concept of formulating data-flow analysis in compilers as a database query was first proposed by Ullman [51]. In 1994, Reps investigated the use of deductive database queries as a means of defining demand-driven interprocedural data-flow analysis [41, 42]. To get a feel for how the landscape has changed over the past decade, it is interesting to compare our project to this seminal work.

The original motivation in Reps's work was to simplify implementation for demand-driven analysis; he observed that by defining the interprocedural analysis in Datalog, magic-set transformations [6, 50] can be used to automatically derive a demand-driven implementation. Managing implementation complexity is also the driving force behind this work. The complexity of implementation we wish to hide is the optimization of BDD operations. In fact, this is even more important as our goal is to help programmers, rather than compiler writers, to write analyses. It turns out that for analyses like pointer aliases, it is necessary to compute all the results exhaustively, rather than in a demand-driven manner, as all pointer updates are potentially inter-related. In contrast, Reps's analysis only handles stack-allocated objects. In our case, the magic-set transformation is found to be useful for some of the lighter weight analyses built on top of pointer analysis.

Reps's implementation is based on Coral [40], which is a conventional deductive database. A comparison with manually coded analysis revealed that the database approach ran four to six times slower. Our implementation, bddbddb, is a custom BDD-based deductive database designed for our purpose. We found that analyses run on bddbddb are faster even than applications we have carefully hand-tuned for months.

1.5 Organization

The organization of the rest of this paper is as follows. Section 2 motivates user-specified analysis. In Section 3 we show our program database representation and specify an analysis in Datalog. Section 4 explains our approach to context-sensitive analysis. Section 5 shows how Datalog is translated into BDD operations. Section 6 details how bddbddb optimizes Datalog programs. Section 7 presents our experiences using and developing Datalog and bddbddb for program analysis. Section 8 discusses related work, and our conclusions are in Section 9.

2. USER-SPECIFIED ANALYSIS

Static analysis has been shown to be useful for finding errors in programs. Today's large, extensible applications are full of invariants that programmers writing, extending, and maintaining these applications must respect in order for the applications to work properly. Violations of these rules often lead to subtle, hard-to-detect problems that may manifest long after the erroneous code is executed. These programming rules or patterns are either specified in comments or, more likely, not specified at all. Programmers working with large systems are often forced to learn programming invariants the hard way—by violating them. Static analysis can systematically check if the invariants are maintained throughout the program and note all violations as errors.

One of the most important applications for our framework is finding security errors, especially those residing in Web applications. More and more Web-based enterprise applications address crucial business needs, often dealing with sensitive financial and medical data; compromise can cause millions of dollars in damages. A security assessment by the Application Defense Center included more than 250 Web applications from e-commerce, online banking, enterprise collaboration, and supply chain management sites [54]. It concluded that at least 92% of Web applications are

```
query simpleSQLInjection()
    uses
       object HttpServletRequest r;
       object Connection c;
       object String p;
matches {
    p = r.getParameter(_);
    c.execute(p);
}
```

Figure 1: Simple SQL injection query.

vulnerable to some form of hacker attacks. Another survey found that about 75% of all attacks against Web servers target Web-based applications [28].

Most of these vulnerabilities are caused by having unchecked input take control of the application in unexpected ways. We discuss a number of such vulnerabilities in this section to motivate why programmers need to write their own analyses to check for violations of application-specific invariants. We also introduce the PQL language to show how these analyses can be expressed simply, and demonstrate that precise pointer information is necessary to generate high-quality checkers.

2.1 SQL Injection

SQL injection vulnerabilities are ranked as one of the top five external threats to corporate IT systems [52]. They are caused by unchecked user input propagating to a database for execution. A hacker may be able to embed SQL commands into the SQL query the application passes to the database for execution. These unauthorized commands may view, update, or delete records. This type of vulnerability is especially critical in Web applications exposed to a large audience; any vulnerabilities at all mean that database information may be forged or stolen by anyone.

Let us look at a simple, concrete example. Here is a code fragment that may be found in a Java servlet hosting a Web service:

```
String p = request.getParameter("query");
con.execute(p);
```

This code reads a parameter from an HTTP request and passes it directly to a database back-end. By supplying a properly crafted query, a malicious user can gain unauthorized access to data, damage the contents in the database, and in some cases, even execute arbitrary code on the server.

To catch this kind of vulnerability in applications, we wish to ask if there exist some

- object r of type `HttpServletRequest`,
- object c of type `Connection`, and
- object p of type `String`

in some possible run of the program such that the result of invoking `getParameter` on r yields string p, and that string p is eventually used as a parameter to the invocation of `execute` on c. Note that these two events need not happen consecutively; the string p can be passed around as a parameter or stored on the heap before it is eventually used.

The PQL language allows us to describe this pattern simply, as shown in Figure 1. PQL queries are expressed as a pattern of dynamically executed statements. The statements listed in the query form a regular expression (in this case, a simple sequence of two method invocations) and the variables represent parameterized objects. Variables whose values are immaterial are represented by the "don't care" symbol "_". Conceptually, these statements represent

```
query main()
returns object Object source, tainted;
uses object java.sql.Statement stmt;
matches {
    source = req.getParameter();
    tainted := derivedString(source);
    stmt.execute(tainted);
}

query derivedString(object Object x)
returns object Object y;
uses object Object temp;
matches
      y := x
    | { temp.append(x); y := derivedString(temp); }
    | { temp = x.toString(); y := derivedString(temp); }
```

Figure 2: A more complete SQL injection query in PQL.

the smallest piece of code that could produce the behavior we are searching for. We do not care about any statements that may occur between them, nor do we care precisely how the objects are named in the code.

It is easy to see how we can translate this to a static analysis. We are looking for two static statements

```
p1 = r.getParameter("query");
```

and

```
c.execute(p2);
```

such that `p1` and `p2` may point to the same object.

In general, SQL injections are more subtle and require more sophisticated patterns to detect. In particular, the contents of the string are typically processed in some way, often inserted into a preformed query, before being passed to the database for execution.

Figure 2 gives a more complete PQL query for SQL injections. The main query binds the variable `source` to an initial input drawn from an HTTP request, then binds `tainted` to any value reachable with zero or more derivation steps. This is handled via the `derivedString` subquery, a tail recursive loop that tracks derivation through the functions involved in string concatenation. Once a tainted object has been identified, it then searches for its use by the database. This query can similarly be translated into a static analysis.

2.2 Taint-Based Vulnerabilities

SQL injection is but one of many widespread vulnerabilities caused by unchecked input in today's Web-based systems. These vulnerabilities can be generalized as a *tainted-data* problem, which is specified by a set of *sources*, *sinks*, and *derivation methods*. Sources are methods that return data obtained from user input. Sinks are the sensitive methods that should not have tainted data passed in. Derivation methods specify how tainted data propagates from one object to another. By varying the composition of these sets, one can express all the vulnerabilities mentioned below.

Cross-site scripting is an attack on applications that fail to filter or quote HTML metacharacters in user input used in dynamically generated Web pages. Typically, the attacker tricks the victim into visiting a trusted URL containing a cross-site scripting vulnerability. This allows the attacker to embed malicious JavaScript code into the dynamically generated page and execute the script on the machine of any user that views the page [18]. When executed, malicious scripts may hijack the user's account, change the user's settings, steal the user's cookies, or insert unwanted content (such as ads) into the page.

HTTP response splitting is an attack on applications that fail to filter or quote newlines in header information. It enables various other attacks such as Web cache poisoning, cross user defacement, hijacking pages with sensitive user information, and cross-site scripting [31]. The crux of the HTTP response splitting technique is that the attacker may cause two HTTP responses to be generated in response to one maliciously constructed request. For HTTP splitting to be possible, the vulnerable application must include unchecked input as part of a response header sent back to the client.

Path traversal vulnerabilities allow a hacker to access or control files outside of the intended path [24, 38]. They occur when applications use unchecked user input in a path or file name; input normally arrives via URL input parameters, cookies, or HTTP request headers. Often the file in question is part of an ad-hoc database, for instance an image in a theme. In addition to reading or removing sensitive files, the attacker may attempt a denial-of-service attack by causing the application to access a file for which it does not have permissions.

All tainted-data problems, and many other error patterns, can be expressed easily in PQL. PQL allows programmers to use a familiar Java syntax to track operations applied to objects simply, regardless of how these objects are referred to in the program text. To find matches of such patterns statically requires precise pointer information: we need to know if statements, often dispersed throughout methods and files of the program, may refer to the same run-time object. As we shall show next, we derive static checkers for PQL queries by translate them into Datalog, which is a more general interface to a large class of static analyses.

3. PROGRAM ANALYSIS IN DATALOG

In this section we show how programs and analysis results are represented in our database, and how analyses are specified in Datalog.

3.1 Datalog

A *relation* is a set of n-ary tuples of attribute values. Each relation attribute is associated with a *domain* D of values that the attribute may take on. An expression $R(x_1, \ldots, x_n)$ is true iff the tuple $(x_1, \ldots, x_n)$ is in relation R. Likewise, $\neg R(x_1, \ldots, x_n)$ is true iff the tuple $(x_1, \ldots, x_n)$ is *not* in relation R.

A Datalog program is a set of rules of the form:

$$E_0 : -E_1, \ldots, E_k.$$

where, for any particular variable assignment, expression E_0 (the rule *head*) is inferred to be true iff expressions $E_1, \ldots, E_k$ (the rule *subgoals*) are true. The rule head is of the form $R(x_1, \ldots, x_n)$, while subgoal expressions may be of the form $R(x_1, \ldots, x_n)$ or the form $\neg R(x_1, \ldots, x_n)$. bddbddb allows negation only in *stratifiable* programs [14], those in which the rules can be grouped into strata such that every negated predicate refers only to relations fully computed in previous strata.

3.2 Program Representation

We store all information available in a source program as a set of database relations; such a representation abstracts away the program syntax and makes the information easily accessible. For the sake of simplicity, let us limit our discussion here to the subset of Java bytecodes relevant to pointer analysis.

The domains of the relations of interest include Java bytecodes B, program variables V, methods M, fields F, calling contexts C, heap objects named by their allocation site H, and integers Z. A front end translates the source program into a set of input relations. The ones relevant to this paper are:

- $assign$: $V \times V$. $assign(v1, v2)$ is true iff the program contains the assignment $v_1 = v_2$.
- vP_0: $V \times H$. $vP_0(v, h)$ is true iff the program directly places a reference to heap object h in variable v in an operation such as `s = new String()`.
- $fldld$: $B \times V \times F \times V$. $fldld(b, v_1, f, v_2)$ means that bytecode b executes "$v_1 = v_2.f$".
- $fldst$: $B \times V \times F \times V$. $fldst(b, v_1, f, v_2)$ means that bytecode b executes "$v_1.f = v_2$".
- $actual$: $B \times Z \times V$. $actual(b, z, v)$ means that variable v is zth argument of the method call at bytecode b.
- ret: $B \times V$. $ret(b, v)$, means that variable v is the return result of the method call at bytecode b.

3.3 SQL Injection

For our first example of a program analysis in Datalog, let us look at the simple version of the SQL injection query shown in Figure 1. The expression of this query in Datalog is simple if a previous context-sensitive analysis has already been performed to find the targets of references and virtual method invocations. We will discuss how these results can be found in Section 4, but for now, we assume that the following two relations are available:

- $calls$: $C \times B \times C \times M$. Targets of virtual method invocations. $calls(c_1, b, c_2, m)$ holds if the method call at bytecode b, when executed in context c_1, may invoke context c_2 of method m.
- vP_c: $C \times V \times H$. Context-sensitive points-to results. $vP_c(c, v, h)$ means that variable v in context c may point to heap object h.

Given these relations, the Datalog to rule to find basic SQL injections is simply:

$$\begin{aligned} SQLInjection(h) \quad :- \quad & calls(c_1, b_1, \_, \text{“}\texttt{getParameter}\text{”}), \\ & ret(b_1, v_1), vP_c(c_1, v_1, h), \\ & calls(c_2, b_2, \_, \text{“}\texttt{execute}\text{”}), \\ & actual(b_2, 1, v_2), vP_c(c_2, v_2, h). \end{aligned}$$

This Datalog rule says that an object h is the cause of a SQL injection if:

1. there is a bytecode b_1 in context c_1 that calls `getParameter`,
2. the returned value is store in some variable v_1,
3. v_1 in context c_1 is found to point to object h,
4. similarly, there is a bytecode b_2 in context c_2 that invokes `execute`,
5. some variable v_2 is passed in as the first argument, and
6. v_2 in context c_2 is also found to point to object h.

Note that it is possible that c_1 and c_2 refer to the same context, for example if the two calls occur consecutively in the program text.

One important thing to note about this query is that it is *flow insensitive*. That is, it does not necessarily respect the order of execution of statements within the program. The analysis would report a match even if the call to `getParameter` occurs *after* the call to `execute`.

Comparing this Datalog program with the PQL query in Figure 1 shows that there is a direct correspondence between the two, and

RELATIONS

input	vP_0	($variable$: V, $heap$: H)
input	*fldst*	($bytecode$: B, $base$: V, *field* : F, $source$: V)
input	*fldld*	($bytecode$: B, $dest$: V, *field* : F, $base$: V)
input	$assign$	($dest$: V, $source$: V)
output	vP	($variable$: V, $heap$: H)
output	hP	($base$: H, *field* : F, $target$: H)

RULES

$$vP(v,h) \quad :- \quad vP_0(v,h). \tag{1}$$

$$vP(v_1,h) \quad :- \quad assign(v_1,v_2), vP(v_2,h). \tag{2}$$

$$hP(h_1,f,h_2) \quad :- \quad fldst(\_,v_1,f,v_2), \; vP(v_1,h_1), vP(v_2,h_2). \tag{3}$$

$$vP(v_2,h_2) \quad :- \quad fldld(\_,v_1,f,v_2), \; vP(v_1,h_1), hP(h_1,f,h_2). \tag{4}$$

Figure 3: Datalog program for context-insensitive points-to analysis

that translation from PQL to Datalog is a simple syntactic translation. This example illustrates how important error patterns can be expressed easily once context-sensitive points-to information and virtual method resolution are available.

3.4 Context-Insensitive Points-to Analysis

To illustrate how a more complex algorithm can be expressed in Datalog, let us use a *context-insensitive*, flow-insensitive points-to analysis as our second example. We shall keep the discussion simple by assuming that all the virtual method targets have already been discovered, so parameter passing in a context-insensitive analysis can simply be modeled with assignments between actual and formal parameters.

This pointer analysis produces two relations:

$vP(v,h)$: $V \times H$ is the context-insensitive version of vP_c; it is true if variable v may point to heap object h at any point during program execution.

$hP(h_1,f,h_2)$: $H \times F \times H$ is true if heap object field $h_1.f$ may point to heap object h_2.

The points-to analysis can be expressed simply in 4 lines of Datalog, as shown in Figure 3. Rule 1 incorporates the initial points-to relations into vP. Rule 2 computes the transitive closure over assignments. If variable v_2 can point to object h and v_2 is assigned to v_1, then v_1 can also point to h. Rule 3 models the effect of stores to fields in an object. Given a statement $v_1.f = v_2$, if v_1 can point to h_1 and v_2 can point to h_2, then $h_1.f$ can point to h_2. Rule 4 resolves loads of fields. Given a statement $v_2 = v_1.f$, if v_1 can point to h_1 and $h_1.f$ can point to h_2, then v_2 can point to h_2. Applying the rules repeatedly, until no rule application produces new relation elements, finds all the points-to relations in the source program.

3.5 A Pointer Analysis Example

Let us illustrate this analysis in action using the simple Java program listed in Figure 4. In this code, `getString` returns the string held by a `StringHolder` object's field `f`. It is easy to see that at the end of the program, `p` points to the string "`select name...`", and that the "select" statement is executed.

Applying our context-insensitive algorithm in Figure 3 to this program would, however, incorrectly infer that the code might also

```
 1  class StringHolder {
 2    String f;
 3  }
 4
 5  String getString(StringHolder sh) {
 6    String x = sh.f;
 7    return x;
 8  }
 9
10  StringHolder a = new StringHolder();
11  StringHolder b = new StringHolder();
12
13  a.f = "select name from users where id=12";
14  b.f = "drop table users";
15
16  String p = getString(a);
17  String q = getString(b);
18
19  database.execute(p);
```

Figure 4: Example program for Java pointer analysis

execute the dangerous SQL query "`drop table users`", destroying a table in our database. The variable `x` in `getString` is assigned to the strings held by `a.f` and `b.f` in two different contexts. Incapable of distinguishing between different contexts, the analysis concludes that both strings will be returned for each of the calls to `getString`.

A context-sensitive analysis would distinguish between the different calls to `getString`, correctly determining that `p` may only point to "`select...`". This example illustrates how imprecision due to context insensitivity can potentially lead to many false-positive warnings.

4. CONTEXT-SENSITIVE ANALYSIS

Adding context sensitivity to an analysis greatly increases its complexity. If we consider the calling context for a particular method invocation to be all calls on the call stack, there can be an unbounded number of calling contexts in any program execution due to recursion. Even if we approximate by only keeping track of calls not involved in recursive cycles, the number of possible contexts can be huge. We found that it is not unusual for Java applications to have over 10^{14} contexts, not counting recursive cycles. Thus, explicitly representing all of the contexts seems to be infeasible.

4.1 Summary-Based versus Cloning-Based

The traditional approach to context-sensitive analysis is to use summaries. Summary-based techniques generate a summary for each method in the program that encapsulates the effects of that method. They then apply that summary at each call site that may target that method. There are two problems that render this approach unsuitable for pointer alias analysis. First, as the summaries of points-to analysis tend not to be compact, such approaches fail to scale to large programs [46, 57, 58]. Second, after the summaries have been generated, to associate the results with any particular context requires the information be passed down the call graph. A query such as determining which variable can point to a particular object would require computing the points-to results across all the exponentially many contexts.

Instead, we use a *cloning-based* approach, in which we create a separate copy of each method for every calling context of interest [21, 55]. Algorithmically, this greatly simplifies the analysis; the context-sensitive algorithm is simply the context-insensitive algorithm applied to the cloned program. Although the cloned program becomes exponentially larger, many of the contexts turn out

RELATIONS

input	vP_0	($variable$: V, $heap$: H)
input	*fldst*	($bytecode$: B, $base$: V, *field* : F, $source$: V)
input	*fldld*	($bytecode$: B, $dest$: V, *field* : F, $base$: V)
input	$assign_c$	($dest_c$: C, $dest$: V, src_c : C, src : V)
output	vP_c	($context$: C, $variable$: V, $heap$: H)
output	hP	($base$: H, *field* : F, $target$: H)

RULES

$$vP_c(\_, v, h) \quad :- \quad vP_0(v, h). \tag{5}$$

$$vP_c(c_1, v_1, h) \quad :- \quad assign_c(c_1, v_1, c_2, v_2), vP_c(c_2, v_2, h). \tag{6}$$

$$hP(h_1, f, h_2) \quad :- \quad fldst(\_, v_1, f, v_2), \; vP_c(c, v_1, h_1), vP_c(c, v_2, h_2). \tag{7}$$

$$vP_c(c, v_2, h_2) \quad :- \quad fldld(\_, v_1, f, v_2), \; vP_c(c, v_1, h_1), hP(h_1, f, h_2). \tag{8}$$

Figure 5: Datalog program for context-sensitive points-to analysis

to be similar. For example, parameters to the same method often have the same types or similar aliases. As we shall show below, we can take advantage of the similarities by using the binary decision diagram (BDD) data structure.

4.2 Adding Context Sensitivity to Pointer Alias Analysis

The simple context-insensitive analysis shown in Figure 3 assumes that a call graph has been computed a priori. With the addition of just a couple of Datalog rules, we can modify the algorithm to discover call targets on-the-fly as points-to results are computed [55].

The *context-sensitive* points-to analysis consists of the following steps. First, it uses a context-insensitive points-to analysis to discover the call graph. Then, it clones all the methods in the call graph, one per context of interest, linking each call site to its unique clone. Every clone of a method is given a unique context $c \in C$. This step generates the *calls* relation used in Section 3.3. Finally, it runs the original context-insensitive analysis over the exploded call graph to get context-sensitive results.

As shown in Figure 5, the Datalog rules for this last step are nearly identical to those in the context-insensitive algorithm shown in Figure 3. As before, assignments are used to model parameter passing and returned results between the cloned callers and callees. We replace the $assign$ and vP relations with their context-sensitive counterparts, $assign_c$ and vP_c, by adding a context for each attribute that represents a variable in the program. We then modify the rules that use $assign_c$ and vP_c so that they obey the context number.

Formulating context sensitivity in this way makes it easy to extract the points-to information for individual contexts. It also allows more powerful queries, such as extracting the set of contexts under which a given points-to relation can or cannot occur.

5. FROM DATALOG TO BDDS

In this section, we explain how we translate Datalog first into relational algebra operations, then into boolean functions and finally into BDD operations.

5.1 Relational Algebra

A Datalog query with finite domains and stratified negation can be solved by applying sequences of relational algebra operations corresponding to the Datalog rules iteratively, until a fixed-point solution is reached. We shall illustrate this translation simply by way of an example, since it is relatively well understood.

The set of relational operations used include join, union, project, rename, difference, and select. $R_1 \bowtie R_2$ denotes the *natural join* of relations R_1 and R_2, which returns a new relation where tuples in R_1 have been merged with tuples in R_2 in which corresponding attributes have equal values. $R_1 \cup R_2$ denotes the *union* of relations R_1 and R_2, which returns a new relation that contains the union of the sets of tuples in R_1 and R_2. $\pi_{a_1,\ldots,a_k}(R)$ denotes the *project* operation, which forms a new relation by removing attributes $a_1, \ldots, a_k$ from tuples in R. $\rho_{a \to a'}(R)$ denotes the *rename* operation, which returns a new relation with the attribute a of R renamed to a'. $R_1 - R_2$ denotes the *difference* of relations R_1 and R_2, which contains the tuples that are in R_1 but not in R_2. The *select* operation, denoted as $\sigma_{a=c}(R)$, restricts attribute a to match a constant value c. It is equivalent to performing a natural join with a unary relation consisting of a single tuple with attribute a holding value c.

To illustrate, an application of the rule

$$vP(v_1, h) :- assign(v_1, v_2), vP(v_2, h).$$

corresponds to this sequence of relational algebra operations:

$$\begin{array}{ll} t_1 & = \rho_{variable \to source}(vP); \\ t_2 & = assign \bowtie t_1; \\ t_3 & = \pi_{source}(t_2); \\ t_4 & = \rho_{dest \to variable}(t_3); \\ vP & = vP \cup t_4; \end{array}$$

Note that rename operations are inserted before join, union, or difference operations to ensure that corresponding attributes have the same name, while non-corresponding attributes have different names.

5.2 Boolean Functions

We encode relations as boolean functions over tuples of binary values. Elements in a domain are assigned numeric values consecutively, starting from 0. Thus, a value in a domain with m elements can be represented in $\lceil log_2(m) \rceil$ bits. Suppose each of the attributes of an n-ary relation R is associated with numeric domains $D_1, D_2, \ldots, D_n$, respectively. We can represent R as a boolean function $f : D_1 \times \ldots \times D_n \to \{0, 1\}$ such that $(d_1, \ldots, d_n) \in R$ iff $f(d_1, \ldots, d_n) = 1$, and $(d_1, \ldots, d_n) \notin R$ iff $f(d_1, \ldots, d_n) = 0$.

Let relation R be a set of tuples $\{(1,1), (2,0), (2,1), (3,0), (3,1)\}$ over $D_1 \times D_2$, where $D_1 = \{0, 1, 2, 3\}$ and $D_2 = \{0, 1\}$. The binary encoding for R is function f, displayed in Figure 6, where the first attribute of R is represented by bits b_1 and b_2 and the second attribute by b_3.

For each relational algebra operation, there is a logical operation that produces the same effect when applied to the corresponding binary function represention. Suppose R_1 is represented by function $f_1 : D_1 \times D_2 \to \{0, 1\}$ and R_2 by function $f_2 : D_2 \times D_3 \to \{0, 1\}$. The relation $R_1 \bowtie R_2$ is represented by function $f_3 : D_1 \times D_2 \times D_3 \to \{0, 1\}$, where $f_3(d_1, d_2, d_3) = f_1(d_1, d_2) \wedge f_2(d_2, d_3)$. Similarly, the union maps to the binary $\vee$ operator, and $l - r \equiv l \wedge \neg r$. The project operation can be represented using existential quantification. For example, $\pi_{a_2}(R_1)$ is represented by $f : D_1 \to \{0, 1\}$ where $f(d_1) = \exists d_2 . f_1(d_1, d_2)$.

D_1		D_2	R
b_1	b_2	b_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 6: Binary encoding of a relation.

5.3 Binary Decision Diagrams

Large boolean functions can be represented efficiently using BDDs, which were originally invented for hardware verification to efficiently store a large number of states that share many commonalities [10].

A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes which represent the constants one and zero. This graph represents a boolean function over a set of input decision variables. Each non-terminal node in the DAG is labeled with an input decision variable and has exactly two outgoing edges: a high edge and a low edge. To evaluate the function for a given set of input values, one simply traces a path from the root node to one of the terminal nodes, following the high edge of a node if the corresponding input variable is true, and the low edge if it is false. The terminal node gives the value of the function for that input. Figure 7(a) shows a BDD representation for function f from Figure 6. Each non-terminal node is labeled with the corresponding decision variable, and a solid line indicates a high edge while a dashed line indicates a low edge.

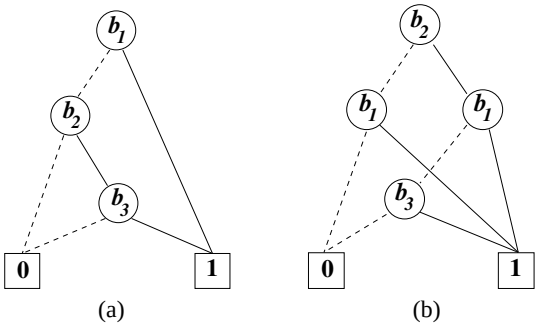

Figure 7: Relation from Figure 6 encoded as a BDD with decision variable order (a) b_1, b_2, b_3 **and (b)** b_2, b_1, b_3**.**

We specifically use a variant of BDDs called *reduced ordered binary decision diagrams*, or ROBDDs [10]. In an *ordered* BDD, the sequence of variables evaluated along any path in the DAG is guaranteed to respect a given total *decision variable order*. The choice of the decision variable order can significantly affect the number of nodes required in a BDD. The BDD in Figure 7(a) uses variable order b_1, b_2, b_3, while the BDD in Figure 7(b) represents the same function, only with variable order b_2, b_1, b_3. Though the change in order only adds one extra node in this example, in the worst case an exponential number of nodes can be added. In addition, ROBDDs are *maximally reduced* meaning common BDD subgraphs are collapsed into a single graph, and the nodes are shared. Therefore, the size of the ROBDD depends on whether there are common boolean subexpressions in the encoded function, rather than the number of entries in the set.

5.4 BDD Operations

The boolean function operations discussed in Section 5.2 are a standard feature of BDD libraries [17]. The $\wedge$ (and), $\vee$ (or), and $-$ (difference) boolean function operations can be applied to two BDDs, producing a BDD of the resulting function. The BDD existential quantification operation `exist` is used to produce a new BDD where nodes corresponding to projected attributes are removed. This operation combines the low and high successors of each removed node by applying an $\vee$ operation. Rename operations are implemented using the BDD `replace` operation, which computes a new BDD where decision variables corresponding to the old attributes have been replaced with decision variables corresponding to the new attribute names.

Natural join operations are frequently followed by project operations to eliminate unnecessary attributes. The BDD relational product operation, or `relprod`, efficiently combines this sequence in a single operation. Similarly, the select and project operations can be combined into a single BDD operation, known as `restrict`.

BDD operations operate on entire relations at a time, rather than one tuple at a time. Thus, the cost of BDD operations depends on the size and shape of the BDD relations, which depends on the encoding, rather than the number of tuples in a relation. Also, due to caching in BDD packages, identical subproblems only have to be computed once.

6. BDDBDDB

bddbddb is a deductive database that accepts Datalog queries and implements relations as BDDs. As described below, the database derives its efficiency from numerous optimizations addressing issues specific to each level of abstraction used: Datalog, relational algebra, and BDDs. It also includes special optimizations to support context-sensitive analyses.

6.1 Optimizing Context-Sensitive Analyses

Our cloning-based approach to context sensitivity creates an exponential number of contexts. Our binary encoding of contexts is designed to allow a compact BDD representation. For each method, clones are assigned context IDs starting from 0. The IDs are chosen such that for each call site, if c_1 and c_2 are the context IDs of matching caller and callee pairs, then the difference between c_1 and c_2 is constant. This numbering scheme allows the BDD representation to take advantage of similarities across related contexts.

6.2 Optimizing Datalog Rules

Stratification. We first remove rules that do not contribute to the output relations. Next, we stratify the rules. We compute dependencies between rules, and assign each strongly connected component in the dependency graph to a separate stratum [14]. The strata are executed in topological order with respect to their dependency relation. Our system requires that programs with negated subgoals are stratifiable; that is, before negation, relations can be fully computed by a previous stratum.

Magic-set transformation. Magic-set transformations can potentially speed up the evaluation of a Datalog program by only generating relation tuples relevant to the output relations [6, 50]. However, they can have the opposite effect when applied to recursive Datalog rules. Since BDDs can efficiently represent and manipulate large relations, removing unnecessary tuples does not necessarily speed up the computation. The transformation can also increase the number of iterations before recursive rules reach closure. In the

case of pointer analysis, the chain of dependencies between pointers can be so long that applying the magic-set transformation to a pointer lookup would be slower than simply computing the global solution and querying that result. Magic-set transformations are applied by default only to strata with no recursive rules, and they have been found to speed up the resolution of some queries significantly.

Rule application order. Recursive rules within a stratum are evaluated by iteration until closure. We use heuristics to transform recursively defined rules into nested cycles of dependencies. We begin iteration at the entry of the top level loop. Upon encountering the entry of an inner loop, we iterate the inner loop until convergence is reached before continuing with the outer loop. Our heuristics are designed so that cycles that involve fewer rules are iterated before cycles that involve more rules, and rules with fewer subgoals are iterated before rules that have more subgoals.

6.3 Optimizing Relational Algebra Operations

bddbddb next translates the rules into relational algebra operations, introducing temporary relations where necessary. We showed in Section 5.1 how a rule might be translated directly into relational algebra operations. We optimize iterative rule applications with semi-naïve rule evaluation [3] by default. Instead of applying a rule to all the tuples found so far in a relation, we only need to apply it to the ones that are newly found. Because this optimization requires storing the old value of every subgoal, we allow the user to control semi-naïve evaluation on a per-rule basis.

We apply a set of standard compiler optimizations to the relational algebraic operations. They include constant propagation, dead code elimination, common subexpression elimination using global value numbering, hoisting invariants out of loops, copy propagation to eliminate unnecessary temporary relations, and liveness analysis to reclaim storage as soon as relations are no longer live.

6.4 Optimizations of BDD Operations

Relational algebra operations are next replaced with equivalent BDD operations. Pairs of natural joins and project operations are replaced by the single `relprod` operation, and pairs of select and project operations are replaced with the `restrict` operation. There are two important optimizations at the BDD level; the first assigns attributes to decision variables, the second orders these decision variables.

6.4.1 Decision Variable Assignment

As discussed in Section 5.1, rename operations are introduced so that corresponding attributes in a rule are given the same name, while non-corresponding attributes have different names. Similarly, corresponding attributes must be assigned the same decision variables, and non-corresponding ones assigned different ones. Rename operations are translated into `replace` operations. Rename operations can be eliminated altogether if the renamed attribute is assigned the same set of decision variables as the original attribute. We extract equivalence and non-equivalence constraints on variable assignments from the rules. We consider these constraints greedily starting with constraints on the most expensive and frequently performed operations; replace operations need to be introduced only when there is an inconsistency in constraints.

6.4.2 BDD Variable Ordering

As discussed in Section 5.3, the ordering of the decision variables can greatly affect the size of the relations and the execution time of the BDD operations. The difference between a good or bad ordering can mean the termination or non-termination (due to memory exhaustion) of an analysis.

We automatically derive the decision variable ordering with a few techniques. First, the choice of variable orderings is reduced by exploiting the high-level relational semantics [33]. We treat each attribute in a relation as a unit; the bits representing an attribute can either precede, succeed, or interleave with those of another. This greatly reduces the space of the orderings, but the number of possible orderings for a relation is still large, and is given by the sequence of ordered Bell numbers [47], which grows rapidly. The number of orderings for 1 through 8 variables are 1, 3, 13, 75, 541, 4683, 47293, 545835 and 7087261.

Our experience from exhaustively searching for the best variable order in the case of small programs suggests that the best order is non-intuitive. We have therefore formulated the search for good variable orderings as an *active machine learning* problem. As opposed to passive learning, an active learner takes a role in selecting the variable orderings on which execution-time measurements are to be taken. The careful selection of inputs can greatly reduce the number of instances needed to find a good answer, which is critical here because the measurements take a nontrivial amount of time.

Our machine learning algorithm is embedded in the interpretation of Datalog programs in the bddbddb system. When bddbddb encounters a rule application that takes longer than a parameterized amount of time, it initiates a learning episode to find a better variable ordering. By measuring the execution time of a set of carefully selected alternative variable orderings, we can extract their salient features so as to reason about the performance of other, related orders. Because rule applications can be expensive, bddbddb maximizes the effectiveness of each learning episode by actively seeking out those variable orderings whose effects are least known in an effort to quickly increase its knowledge of the search space.

7. EXPERIENCES

We have used Datalog to build a variety of analyses. In addition to a pointer analysis for Java [55], we have also implemented a pointer analysis for C [2]. We used our C pointer analysis to address two of the most prominent security threats in C: buffer overruns and format string vulnerabilities. Building upon the Java analysis, we developed a number of fundamental Java analyses that would have been difficult to create otherwise.

Our experiences with Datalog and bddbddb suggest that they make it much easier to develop context-sensitive program analyses. It is now trivial to combine results of multiple analyses. As each analysis takes tens of lines rather than thousands, it is easier to ensure its correctness. In addition, we found that bddbddb can often execute program analyses faster than a well-tuned hand-coded implementation. Our active learning algorithm even found variable orderings that outperformed those we laboriously optimized by hand or found by exhaustively searching a subset of the space of orderings. Finally, we found that, with the help of PQL, programmers can easily create analyses that address important problems such as finding vulnerabilities in programs.

7.1 C Pointer Alias Analysis and Applications

The semantics of the C language makes a pointer analysis for C much more complex than one for Java. Because C is not type-safe, more opportunities for security vulnerabilities exist, and a pointer analysis must be able to model the effects of all possible program behavior. While Java enforces type safety by limiting user access to memory, C affords the user complete, direct control. A C programmer can take the address of any field within an object, access any portion of memory as a contiguous region of bytes, perform arbi-

trary address arithmetic, and perform arbitrary object casts. Without a sound C pointer analysis, a conservative security tool must assume that every pointer may point to any location in memory.

While our Java analysis consists of 33 rules and 10 relations, our C pointer analysis has 234 rules and 63 relations (176 rules and 21 relations are used for modeling system calls). We originally hand-coded a context-insensitive C pointer analysis directly in terms of BDD operations. We spent a long time optimizing or correcting physical domain assignments on a per-operation basis. We rewrote the algorithm in Datalog and were able to create a context-sensitive analysis easily by just adding context attributes to existing relations. Such a modification would have required rewriting hundreds of lines of low-level BDD operations in the hand-coded analysis.

7.1.1 Buffer Overruns

Buffer overruns are responsible for over 50% of all program vulnerabilities. Programs are exploited by supplying input data that exceeds the length of destination buffers, causing the program to overwrite unintended memory locations. Since the length of input data is unknown in a static analysis, we can only dynamically detect buffer overruns. Static analysis can reduce the dynamic overhead by eliminating unnecessary checks.

The C Range Error Detector (CRED) is a recently developed dynamic bounds checker [43]. Because buffer overflows are typically transported as user input strings, CRED can be run with less overhead by checking only string buffer overflows. CRED operates by implementing an object table to track the base address and extent of allocated objects. It instruments pointer arithmetic, dereferences, and string-manipulation functions such as `memcpy` to associate pointer addresses with the objects to which they refer and verify that out-of-bounds pointers are not dereferenced. Because C is unsafe, all objects need to be entered into the object table because any object can be used as a string.

With the results from our C points-to analysis, we can statically determine which objects will never be used as strings and avoid the overhead of entering those objects into the object table [2]. This optimization significantly reduces the overhead of applications whose strings are used only for inputs. We found that other optimizations are still necessary for applications that operate mostly on strings.

7.1.2 Format String Vulnerability

Another common security threat is the format string vulnerability exploit. A program may be exploited if it passes a user-supplied string as the format string argument to a system function such as `printf`. This string may contain format specifiers that cause the program to write to unintended memory locations. Static detection of user-supplied format strings is another example of the tainted-data problem, which requires the results of a pointer analysis.

To avoid false-positive results, a previous format string vulnerability analysis unsoundly assumed that pointers are unaliased unless proven otherwise with just local information [45]. Because the analysis was unsound, it could not find all potential vulnerabilities.

We have developed a static analysis for format string vulnerabities using our C pointer analysis [2]. It is a sound analysis—it guarantees that all potentially tainted format strings will be reported. When applied to a suite of twelve applications, we found that three contain format string vulnerabilities. The analysis reported only fifteen false-positive errors, which were all contained in one application.

7.2 Fundamental Java Analyses

Our simple formulation of context sensitivity lends itself to easy implementation of various fundamental Java analyses. We describe three Java analyses: type analysis, escape analysis, and analysis of Java reflection.

7.2.1 Type Analysis

We used the context-sensitive pointer analysis to find the possible types for each variable. This information is used to discover an accurate call graph for the program, and can also be used to optimize the program by resolving virtual call sites, eliminating dynamic type casts and helping method inlining. More accurate analyses will have fewer possible types for each variable. In our experiments, the percentage of multi-typed variables (variables that point to objects of different types) was between 6–10% when using context-insensitive pointer analysis, but dropped by up to a factor of 20 to less than 1% when using full context sensitivity [55]. This highlights the great improvement in accuracy from using a context-sensitive analysis.

7.2.2 Escape Analysis

Thread escape analysis determines if objects created by one thread may be used by another. The results of the analysis can be used for optimizations such as synchronization elimination and allocating objects in thread-local heaps, as well as for understanding programs and checking for possible race conditions due to missing synchronizations [55, 57].

Previous implementations were extremely complex; there have been numerous publications devoted to escape analysis implementations [57]. By using bddbddb and the results of our context-sensitive pointer analysis, we are able to build an implementation of escape analysis in just 4 lines of Datalog. This analysis was able to find a significant number of thread-local objects and unnecessary synchronization operations.

7.2.3 Resolving Java Reflection

Java's reflection mechanism accepts strings and grants access to the corresponding classes, fields, or methods. This is problematic for static analysis tools. Without a full treatment of reflection, static analysis tools may be unsound due to missing parts of the call graph or writes to object fields. However, resolving reflective accesses is difficult because it depends on the values of the strings passed into the reflective methods. Most static analysis tools treat reflection in an unsound manner or just ignore it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection.

By adding a few Datalog rules to our pointer analysis specification, we can use the current results of the pointer analysis to resolve reflective method invocations, object creations, and field accesses. This allows us to discover the call graph, including reflective calls, on-the-fly and obtain a very accurate and complete call graph. Our experiments indicate that adding support for reflection in the analysis increases the number of methods in an application's call graph by 2%–128%. These extra methods would have otherwise been missed by the analysis.

7.3 Vulnerabilities in Java Web Applications

We have developed a set of analyses to check for the vulnerabilities discussed in Section 2. These analyses are written in PQL, each taking tens of lines of code. We applied the analyses to a set of representative open-source applications: `jboard`, `blojsom`, `snipsnap`, and `blueblog` are Web-based bulletin board and blogging applications; `webgoat` is a J2EE application designed as a test case and teaching tool; and `road2hibernate` is a test program for the popular object persistence library `hibernate`.

We found that every application suffers from one or more vul-

Program	SQL Injection	HTTP Splitting	Cross-Site Scripting	Path Traversal	Total Errors	False Warnings
jboard	0	0	0	0	0	0
blueblog	0	0	1	0	1	0
webgoat	5	0	1	0	6	0
blojsom	0	0	0	2	2	0
personalblog	2	0	0	0	2	0
snipsnap	1	11	0	3	15	12
road2hibernate	1	0	0	0	1	0
pebble	0	0	1	0	1	0
roller	0	0	1	0	1	0
Total	9	11	4	5	29	12

Figure 8: Vulnerabilities found in 9 Web applications (preliminary result summary).

nerabilities we tested, except for the smallest application `jboard`. `snipsnap` is the only application that suffers from the HTTP splitting vulnerability; it has eleven such errors. Path traversal vulnerabilities are found in two applications, whereas potential SQL injection and cross-site scripting errors are located in four applications. In total, our experiment turned up 29 errors.

It is important also to note that our queries raised only 12 false warnings, all within `snipsnap`. This is significant because it is relatively easy to go over all the warnings generated to pick out the true errors. In contrast, if we had substituted our advanced context-sensitive pointer analysis with a more conventional context-insensitive approach, the queries would have generated hundreds of warnings for some of the larger programs and would not have been effective. In addition, not only can we find many errors with little effort, the analysis is sound. We can be assured that there are no more vulnerabilities of the kind we are looking for among the codes we have analyzed.

8. RELATED WORK

Our system builds on many areas of research. We briefly describe some of the most directly related work, organized into several areas.

Program analysis with databases. Ullman first suggested formulating data-flow analysis as database queries [51]. Reps used a deductive database for demand-driven interprocedural data-flow analysis [41, 42].

Finding program errors. Much attention has been given recently to the topic of detecting errors in programs. Penetration testing, which involves attempting to determine input values that exploit holes in an application, is the current typical approach [1, 11, 44]. This approach cannot guarantee that all vulnerabilities will be located. Penetration testers often use "fuzzing" tools, which generate random input, to help locate vulnerabilities.

Others have attempted to locate errors statically; Chess and McGraw provide an overview of these approaches [16]. These tools range from simple lexical analysis to sophisticated program analyses. Of the latter, most practical tools make unsound assumptions regarding pointer aliasing. Intrinsa and Metal, which locate errors in C and C++ programs, and WebSSARI, which finds errors in PHP code, fall into this category [12, 26]. Other approaches, such as those based on type qualifiers [30, 45, 53] may suffer from imprecision or a need for user annotations. Our approach is novel in that it is a sound static analysis with a low false positive rate. Using powerful pointer alias analysis allows us to avoid making unsound assumptions without imprecision producing many false positives.

User-specified program queries. Other systems allow the user to specify program analyses. Metal [26] and SLIC [4] both define state machines with respect to variables. These machines are used to configure a static analysis that searches the program for situations where error transitions can occur. Metal restricts itself to finite state machines, but has more flexible event definitions and limited ability to handle pointers. SLIC machines are capable of counting but cannot exploit pointer information.

Pointer analysis. Pointer alias analysis is a problem with a long history. Here we present only a sampling of the most directly related work. One line of research has focused on scalable pointer analyses; it began with an imprecise but very fast algorithm due to Steensgaard [48]. Recently several more precise algorithms have been shown to scale to large programs [7, 27, 33, 56]. These algorithms are all context-insensitive, except for one due to Das, which includes a single level of context sensitivity [20]. Some attempts at context sensitivity, such as the C pointer analysis due to Fähndrich et al. [22], achieve moderate scalability by sacrificing precision on other dimensions, for example by not distinguishing between different fields of structures. Other lines of work have focused on increasing precision. Many earlier attempts at context sensitivity have not been shown to scale [21, 32, 57, 58]. Our work is similar to Emami et al. in that they also compute context-sensitive points-to results for all different contexts. However, their technique has only been demonstrated to work for programs of less than 3000 lines.

Program analysis with BDDs. BDDs have recently been used in a number of program analyses such as predicate abstraction [5], shape analysis [36, 59], class analyses of object-oriented programs [8] and, in particular, points-to analysis [7, 60, 61]. These analyses use BDD libraries directly, without additional abstraction to ease implementation. Jedd is a Java language extension that provides a relational algebra abstraction over BDDs [34].

Optimizing Datalog. Liu and Stoller described a method of efficiently implementing Datalog, however, they optimized with respect to a metric that does not apply when using BDDs [35]. There has been much research on optimizing Datalog evaluation strategies; for example, semi-naïve evaluation [3], bottom-up evaluation [13, 37, 50], top-down with tabling [15, 49], the role of rule ordering in computing fixed-points [39], etc. We use an evaluation strategy geared towards the peculiarities of the BDDs—for example, to maximize cache locality, we iterate around inner loops first. Other work transforms Datalog programs to reduce the amount of work necessary to compute a solution. For example, the magic-set transformation is a general algorithm for rewriting logical rules to reduce the number of irrelevant facts generated [6].

Logic programming with BDDs. Iwaihara et al. described a technique for using BDDs for logic programming [29]. They compared two different relation encodings, including the one we use. The Toupie system translates logic programming queries into an implementation based on decision diagrams [19]. Crocopat is a tool for relational computation that is used for structural analysis of software systems [9]. Like bddbddb, they use BDDs to represent relations.

Machine learning BDD variable orders. Recently, Grumberg et al. have tackled the BDD variable order problem with machine learning [25]. Their framework is inherently different from ours, as they lack the higher level notion of domains and, instead, seek to order individual BDD variables. Our search space is therefore smaller and we can search more effectively. Also, we evaluate variable orderings based on running times rather than BDD sizes; smaller BDDs do not necessarily mean faster BDD operations. bddbddb employs active learning to maximize the value of each trial run [23].

9. CONCLUSIONS

This paper describes a deductive database framework that greatly simplifies the development of context-sensitive program analyses. This framework allows users to express a whole-program analysis succinctly with a small number of Datalog rules that operate on a cloned call graph. The deductive database implementation hides the complexity in managing the exponentially many calling contexts in a program. Efficiency is achieved through a large number of techniques, which include using BDDs to represent relations compactly, database query optimizations, compiler optimizations to remove redundant operations, a custom technique to optimize BDD decision variable assignment, as well as the use of active machine learning to find a suitable decision variable ordering.

We have used this framework to develop a large number of analyses: C and Java pointer alias analyses, tools for finding buffer overruns and format strings in C programs, fundamental Java analyses including type inference, escape, and reflection analyses. On top of Datalog, we have implemented a little language called PQL that allows Java programmers to express error patterns intuitively. We have used it to find numerous vulnerabilities in Java web applications.

10. REFERENCES

[1] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *Proceedings of the IEEE Symposium on Security and Privacy*, 3(1):84–87, 2005.

[2] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Using C pointer analysis to improve software security. In *Proceedings of the 27th International Conference on Software Engineering*, May 2005.

[3] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query optimization. *Journal of Logic Programming*, 4(3):259–262, Sept. 1987.

[4] T. Ball and S. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, Jan. 2002.

[5] T. Ball and S. K. Rajamani. A symbolic model checker for boolean programs. In *Proceedings of the SPIN 2000 Workshop on Model Checking of Software*, pages 113–130, Aug. 2000.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, June 2003.

[8] F. Besson and T. Jensen. Modular class analysis with Datalog. In *Proceedings of the 10th International Static Analysis Symposium*, pages 19–36, June 2003.

[9] D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, pages 216–225, Nov. 2003.

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

[11] B. Buege, R. Layman, and A. Taylor. *Hacking Exposed: J2EE and Java: Developing Secure Applications with Java Technology*. McGraw-Hill/Osborne, 2002.

[12] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience (SPE)*, 30:775–802, 2000.

[13] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., 1990.

[14] A. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[15] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[16] B. Chess and G. McGraw. Static analysis for security. *Proceedings of the IEEE Symposium on Security and Privacy*, 2(6):76–79, 2004.

[17] H. Cohen. BuDDy - a Binary Decision Diagram package. `http://buddy.sourceforge.net/`.

[18] S. Cook. A Web developer's guide to cross-site scripting. `http://www.giac.org/practical/GSEC/Steve_Cook_GSEC.pdf`, 2003.

[19] M.-M. Corsini, K. Musumbu, A. Rauzy, and B. L. Charlier. Efficient bottom-up abstract interpretation of prolog by means of constraint solving over symbolic finite domains. In *PLILP '93: Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 75–91, 1993.

[20] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Static Analysis Symposium*, pages 260–278, July 2001.

[21] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[22] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, June 2000.

[23] R. Greiner, A. J. Grove, and D. Roth. Learning cost-sensitive active classifiers. *Artif. Intell.*, 139(2):137–174, 2002.

[24] J. Grossman. WASC activities and U.S. Web application security trends. `http://www.whitehatsec.com/presentations/WASC_WASF_1.02.pdf`, 2004.

[25] O. Grumberg, S. Livne, and S. Markovitch. Learning to order BDD variables in verification. *Journal of Artificial Intelligence Research*, 18:83–116, 2003.

[26] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[27] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.

[28] G. Hulme. New software may improve application security. `http://www.informationweek.com/story/IWK20010209S0003`, 2001.

[29] M. Iwaihara and Y. Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 467–474, 1995.

[30] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 2004 Usenix Security Conference*, pages 119–134, 2004.

[31] A. Klein. Divide and conquer: HTTP response splitting, Web cache poisoning attacks, and related topics. `http://www.sanctuminc.com/pdf/Whitepaper_HTTPResponse.pdf`, 2004.

[32] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[33] O. Lhoták and L. Hendren. Scaling Java points-to analysis using

Spark. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 153–169, April 2003.

[34] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 158–169, 2004.

[35] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183, 2003.

[36] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In *Proceedings of the 9th International Static Analysis Symposium*, pages 196–212, Sept. 2002.

[37] J. F. Naughton and R. Ramakrishnan. Bottom-up evaluation of logic programs. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 640–700, 1991.

[38] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. `http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf`, 2004.

[39] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 359–371, 1990.

[40] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 167–176, May 1993.

[41] T. Reps. Demand interprocedural program analysis using logic databases. *Applications of Logic Databases*, pages 163–196, 1994.

[42] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the Fifth International Conference on Compiler Construction*, pages 389–403, Apr. 1994.

[43] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[44] J. Scambray and M. Shema. *Web Applications (Hacking Exposed)*. Addison-Wesley Professional, 2002.

[45] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 Usenix Security Conference*, pages 201–220, 2001.

[46] M. Sharir and A. Pnueli. Two approaches to interprocedural data-flow analyis. In S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[47] N. J. A. Sloane. The on-line encyclopedia of integer sequences: A000670. `http://www.research.att.com/as/sequences`, 2003.

[48] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.

[49] H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings on Third international conference on logic programming*, pages 84–98, 1986.

[50] J. D. Ullman. Bottom-up beats top-down for Datalog. In *PODS '89: Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, pages 140–149, 1989.

[51] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., volume II edition, 1989.

[52] M. Vernon. Top five threats. ComputerWeekly.com (http://www.computerweekly.com/Article129980.htm), Apr. 2004.

[53] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, 2000.

[54] WebCohort, Inc. Only 10% of Web applications are secured against common hacking techniques. `http://www.imperva.com/company/news/2004-feb-02.html`, 2004.

[55] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[56] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, Sept. 2002.

[57] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference of Object Oriented Programming: Systems, Languages, and Applications*, pages 187–206, Nov. 1999.

[58] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[59] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *Proceedings of the 9th International Static Analysis Symposium*, pages 69–84, Sept. 2002.

[60] J. Zhu. Symbolic pointer analysis. In *Proceedings of the International Conference in Computer-Aided Design*, pages 150–157, Nov. 2002.

[61] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157, 2004.