Database-Backed Program Analysis for Scalable Error Propagation

Cathrin Weiss
cathrin.weiss@gmail.com

Cindy Rubio-González
University of California, Davis
crubio@ucdavis.edu

Ben Liblit
University of Wisconsin–Madison
liblit@cs.wisc.edu

***Abstract*—Software is rapidly increasing in size and complexity. Static analyses must be designed to scale well if they are to be usable with realistic applications, but prior efforts have often been limited by available memory. We propose a database-backed strategy for large program analysis based on graph algorithms, using a Semantic Web database to manage representations of the program under analysis. Our approach is applicable to a variety of interprocedural finite distributive subset (IFDS) dataflow problems; we focus on error propagation as a motivating example. Our implementation analyzes multi-million-line programs quickly and in just a fraction of the memory required by prior approaches. When memory alone is insufficient, our approach falls back on disk using several hybrid configurations tuned to put all available resources to good use.**

I. Introduction

As computers become more powerful, software applications continue to grow in size. For example, Linux is almost seven times larger now than it was twelve years ago. As a consequence, particular attention is placed on scalability when designing techniques to analyze software. Unfortunately, computer resources are often a limitation when analyzing large code bases. The key scalability limitation is memory.

Memory usage is often correlated with the size of the code base under analysis. Recent papers [1]–[10] describe scalable program analyses that are applied to large code bases. The largest programs analyzed in these papers range in size from 474 KLOC to 8.6 MLOC. Despite the fact that these analyses either have different purposes, or follow different approaches to solving similar problems, it is noteworthy that their memory usage ranges from 808 MB to 20 GB, with a few instances running out of memory in their particular configuration settings. One rarely sees analyses of larger code bases, or analyses of combinations of programs (e.g., analyzing a kernel and user applications in conjunction). Presumably, the main reason is poor scalability.

Doop [11] and SemmleCode (developed as CodeQuest [12], [13]) use database engines to query and analyze program behavior. Databases routinely work with more data than fits in memory, so a database-backed program analysis could potentially break through the memory scalability barrier. However, prior work has not systematically explored the practical consequences of moving analysis from memory-only to memory-plus-disk. For example, Smaragdakis et al. [14] explicitly focus on Doop configurations that do not bring the disk heavily into play.

In this paper, we present a database-backed program analysis technique that makes good use of available memory while allowing very large analyses to fall back on disk. We significantly reduce the amount of memory required, thereby allowing memory-only analyses of larger code bases than was possible in prior work. For problems that are too large for memory, we identify useful memory-plus-disk configurations to optimize use of available resources.

We demonstrate the effectiveness of our technique by implementing a static program analysis that tracks how error codes propagate in C/C++ applications, and finds instances in which unhandled error codes are dropped [15], [16]. Figure 1a shows an example of a dropped error in the Firefox web browser. Function DashArrayToJSVal may return one of two error codes: NS_ERROR_OUT_OF_MEMORY or NS_ERROR_FAILURE. The error code is dropped on line 4 while failing to be stored in the reference parameter error. Ignoring this failure could cause mozDash to remain uninitialized, leading to a confirmed potential security vulnerability.[1] Note that, although the bug fix seems trivial, finding the bug and agreeing on a fix was definitely not an easy task.

As shown in Figure 1a, error-propagation analysis is particularly important because defective error propagation has the potential to cause not only security vulnerabilities, but also silent, unrecoverable data corruption. Furthermore, the systems potentially affected can be massive. Applying error-propagation analysis to such large code bases is an important goal, but one that faces major scalability challenges.

We apply our database-backed analysis to the Linux kernel and the Firefox web browser. Experiments show that we can perform analyses with modest resources while other analysis tools require dedicated, large-memory servers. Our approach is highly configurable, allowing it to work efficiently and scalably across a wide range of data characteristics. Our contributions are as follows:

- We show how to encode error-propagation analysis (Section II) as a graph saturation problem (Section III) tuned for efficient database-backed inference.
- We provide an efficient and scalable implementation of our target analysis using a database-backed graph indexing scheme (Section IV).
- We explore several database configurations representing trade-offs between available memory and time to completion (Section V).

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=779669

```
1 Value CanvasRendering::GetMozDash(Context* cx, ErrorResult& error) {
2   Value mozDash;
3
4   DashArrayToJSVal(CurrentState().dash, cx, &mozDash); // unsaved error
5
6   return mozDash;
7 }
```

(a) Original code with an unsaved error on line 4

```
1 Value CanvasRendering::GetMozDash(Context* cx, ErrorResult& error) {
2   Value mozDash;
3   int temp;
4   temp = DashArrayToJSVal(CurrentState().dash, cx, &mozDash);
5   temp = OK;  // overwritten error
6   return mozDash;
7 }
```

(b) Transformed code with an overwritten error on line 5

Fig. 1. An unsaved error found in the Firefox web browser. Figure 1a is the original code in which function DashArrayToJSVal may return one of two error codes: NS_ERROR_OUT_OF_MEMORY or NS_ERROR_FAILURE. Function GetMozDash fails to propagate the error, leading to a confirmed (and now fixed) potential security vulnerability. Figure 1b is the code after program transformations that convert the unsaved error into an overwritten error.

- We apply our analysis to two large, real-world code bases: the Linux kernel and the Firefox web browser. Our technique dramatically reduces required time and/or memory compared to a prior memory-only approach (Section V).

Section VI discusses related work and Section VII concludes.

II. Background

Incorrect error handling is a longstanding problem in a wide variety of domains. Ideally, whenever a run-time error occurs, software systems should respond accordingly. Unfortunately, that is often not the case. Error-handling code tends to be poorly understood, poorly documented, and poorly tested [17]. Unsurprisingly, then, error-handling code is often buggy. One of the reasons is that handling errors is in general a difficult task. Exceptional conditions must be considered during all phases of software development [18], introducing interprocedural control flow that can be difficult to reason about [19]–[21]. As a result, error-handling code is usually scattered across different functions and files, making software more complex and less reliable.

Modern programming languages such as Java, C++, and C# provide exception-handling mechanisms. Unfortunately, C does not have exceptions, so C programmers emulate exceptions in a variety of ways. The return-code idiom is among the most popular idioms used in large C programs, including operating systems. Errors are represented as simple integer codes, where each integer value represents a different kind of error.[2] These error codes propagate through conventional mechanisms such as variable assignments and function return values. Even C++ programmers sometimes prefer the return-code idiom over language-level exceptions [22]–[25]. Unfortunately, the error-code idiom is highly error-prone.

Several approaches have been proposed to detect or monitor error-propagation patterns at run time, typically during controlled in-house testing with fault-injection to elicit failures [26]–[34]. Other approaches use dataflow analysis to detect bugs in the propagation of errors in system software and user applications [15], [16], [35]. Our case study is particularly inspired by the error-propagation analysis of Rubio-González et al. [16], which tracks errors as they propagate through C/C++ programs.

[2] For example, the Linux kernel defines EIO, EAGAIN, and ENOMEM as integers 5, 11, and 12, respectively. These error codes correspond to "I/O error", "try again", and "out of memory". Error codes are not, in general, standardized across applications: each program or library may define its own idiosyncratic codes.

The error-propagation analysis is an interprocedural, flow- and context-sensitive static program analysis that finds the set of error codes that each variable may contain at each given program point. This information is used to detect a variety of error-handling related bugs in Linux file systems that can potentially cause silent, unrecoverable data corruption [16], [36], [37]. In particular, the analysis detects unhandled errors that are overwritten with a new value (*overwritten error*), go out of scope (*out-of-scope error*), or are returned by a function but not saved by the caller (*unsaved error*). Figure 1a shows an example of an unsaved error.

For simplicity, the analysis transforms both out-of-scope and unsaved errors into overwritten errors. Figure 1b shows how to transform the unsaved error of Figure 1a into an overwritten error. The transformation consists of two steps:

1) First, the unsaved error is transformed into an out-of-scope error. This is achieved by introducing a new temporary variable temp on line 3. The variable is assigned the value returned by function DashArrayToJSVal. Because function DashArrayToJSVal may return an error code, the transformed function now contains an out-of-scope error: variable temp goes out of scope while storing an error code.
2) Second, the out-of-scope error is transformed into an overwritten error. For this, an assignment is inserted at the end of the function for each integer local variable. In our example, only one assignment to variable temp is inserted on line 5. Note that OK is a special value introduced by the analysis to represent a non-error value. After this, the transformed function now contains an overwritten error instead of an out-of-scope error: variable temp contains an error that is overwritten with the value OK on line 5.

At the high-level, assignment statements (or overwritten errors) constitute the program points of interest for the analysis. Internally, the analysis can still distinguish between the three kinds of dropped errors, which helps provide more precise diagnostic information. The following paragraphs describe other analysis characteristics.

Exchange variables: The analysis introduces global variables referred to as exchange variables. Exchange variables are used for the purpose of value passing between callers and callees. There is an exchange variable for each function parameter, and function return value. Callers export arguments into the corresponding exchange variables, and callees import these exchange variables into its formal parameters. Similarly,

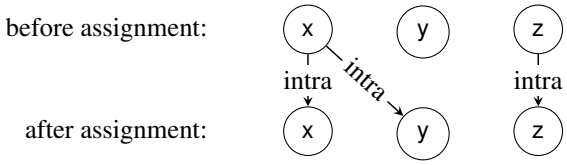


Fig. 2. Example graph fragment corresponding to "y = x;"

callees export their return value into the corresponding exchange variable, and callers import this variable into the receiver. This process makes value passing explicit.

Pointer variables: Each pointer variable p is treated as two locations p and ⋆p, and no pointer is assumed to alias any other. This is neither sound nor complete, but it has been shown to yield useful results in bug finding. Under these conditions, pointer parameters are equivalent to call-by-copy-return parameters. Pointed-to values are copied from the caller to the callee just as for function parameters. Callee values are copied back into the caller. This extra copy-back on return is what distinguishes pointer arguments from non-pointer arguments, because it allows changes made by the callee to become visible to the caller.

Function pointers: Indirect calls are treated as a nondeterministic choice among a conservative over-approximation of all possible callees. Specifically, calls across functions pointers are rewritten as switch statements that choose among possible implementations nondeterministically.

III. Analysis Codification

This Section describes how the error-propagation analysis of Section II can be codified as a graph edge saturation problem. Section IV will describe how this graph problem is mapped into a form suitable for efficient implementation.

A. Variable-Expanded Control Flow Graph

Error-propagation analysis requires determining the set of error codes that a given variable may contain at a given program point. This constitutes an interprocedural dataflow analysis problem. We begin with the interprocedural control-flow graph (CFG). Replace each statement node in this CFG with a vector of nodes: one for each global variable, local variable, or formal function parameter in the program. If x, y, and z are program variables, we might have three nodes representing x, y, and z at some statement s_1, three more representing these variables at a different statement s_2, and so on for each variable at each program location. This makes the graph significantly larger than the original, but allows us to distinguish each variable's value at each program point, thereby supporting flow-sensitive analysis. We call this new, enlarged graph the *variable-expanded CFG*. It can be seen as a specific instance of an *exploded supergraph* as used in the interprocedural finite distributive subset (IFDS) framework of Reps et al. [38].

Suppose statements s_1 and s_2 were statement nodes connected by an intraprocedural control-flow edge in the original CFG. In the variable-expanded CFG, connect variable nodes at s_1 to selected variable nodes at s_2 to reflect possible flows of values as a result of executing s_1. Figure 2 shows an example graph fragment corresponding to the assignment "y = x;". Observe that while y is overwritten with the previous value of x, this assignment has no effect on the values of x or z. Edges reflecting value flow within a single function are labeled "intra" for intraprocedural flow.

At each function call in the original CFG, split the call into a vector of nodes for each variable before the call and a distinct vector of nodes for each variable after the call. Do not directly connect before-call nodes to after-call nodes. Rather, assign each call site a unique identifier. At call site i, create interprocedural call edges labeled "$(_i$" connecting actual arguments in the caller to the corresponding formal arguments at the entry point of the callee. Conversely, create interprocedural return edges labeled "$)_i$" connecting variables at exit points from the callee to variables that may receive returned values in the caller.

Lastly, create a node for each error code. Error codes are immutable, so these nodes need not be replicated at each program location. Connect error code nodes to variable nodes where appropriate to reflect flow of error codes into variables. For example, "y = ENOMEM;" adds an edge from the ENOMEM node to the node representing y after the assignment. Edges from error code nodes can be interprocedural as well, as in "foo(EAGAIN)" or "**return** EIO".

We handle additional program features such as parameter passing in the manner of prior work on error propagation [16]. Refer to Section II for a brief summary.

B. Interprocedurally Valid Paths

The variable-expanded CFG reflects single-step flows of values from one statement to the next. Determining which error codes flow into a given variable at a given program point requires extending these local flows across multiple statements. We are looking for interprocedurally valid paths obeying two criteria:

- Within a function, a valid path must cross a sequence of intraprocedural flow edges ("intra").
- Across a function call, a valid path must enter and exit the callee at the same call site. That is, each call edge "$(_i$" must match a corresponding return edge "$)_i$". Mismatched calls and returns, such as "$(_1\ )_2$", are disallowed as they cannot correspond to any properly-nested function invocation.

Paths corresponding to a complete, normally-terminating execution would begin with some "$(_{\mathsf{main}}$" edge representing the call to the program's main routine, and would end with a corresponding "$)_{\mathsf{main}}$" edge. However, we are interested in partial executions during which error codes are propagating from one part of the program to another. Therefore, our valid paths may begin with a prefix of unmatched return edges, representing returns from calls that were already in progress when the error arose. This is called a *positive flow*: the caller is *receiving* information returned to it by the callee. Likewise, our valid paths may end with a suffix of unmatched call edges, representing calls into functions that are still in progress upon reaching the statement of interest. This is called a *negative flow*: the caller is *giving* information to the callee. If all calls and returns match, then a path is a *matched flow*. Thus, an interprocedurally valid path for a partial execution consists of a positive-flow prefix

followed by a negative-flow suffix, with arbitrary matched flows intermixed at any point along the path.

C. Path Discovery Via Edge Saturation

Discovering interprocedurally valid paths proceeds bottom-up. We begin with flows already present in the variable-expanded CFG. Based on these, we infer progressively longer paths, culminating in interprocedurally valid paths. We codify the inference process as a suite of subgraph patterns given in Figure 3. We saturate the graph using these patterns: whenever any pattern matches any part of the graph, add the corresponding green dashed edge. Repeat this until no new opportunities to add edges can be found. Once that fixed point is reached, the analysis results can be read directly out of the edges labeled "valid".

Edge saturation for interprocedurally valid paths can be formulated in many ways. All would ultimately lead to the same results, but different formulations can vary dramatically in how many intermediate edges they add, and therefore in how well they scale up. We have carefully formulated the patterns in Figure 3 to focus edge creation on known points of interest, avoiding inferring edges that are correct but not useful. These patterns also leverage our database engine's efficient transitive closures (Section IV-D) and fast matching against partial (source, edge, destination) patterns (Section IV-A). Any IFDS problem would use similar patterns, though the specific edge labels and relationships would vary.

We now clarify the meaning and role of each pattern.

Figure 3a, matched-seed: We do not need to know about every variable at every program location; only some of these are actually of interest. Before analysis begins, we mark these nodes of interest with self edges labeled "seed". Trivially a seed self edge is a matched flow of length zero. This zero-length matched flow can then serve as a basis for inferring longer flows using other patterns that follow. Thus, "seed" edges are the seeds from which longer paths grow.

Figure 3b, intra-inferred: Suppose we have already discovered any inferred flow from x to b. Additionally, there is a single step of intraprocedural flow from a to x. Put these two together and we have a slightly longer inferred flow (of the same kind) from a to b. This figure actually represents three patterns, with *inferred* varying over {matched, pos, neg}. In conjunction with Figure 3a, these patterns effectively create a predominantly-backward analysis that begins at seeds and extends paths backward to error codes.

Figure 3c, callee-matched: Suppose we have already discovered any inferred flow from x to y, and that x is a return point from call i. In order to know what reaches y, we need to know what could have happened in that call. So mark the corresponding point just inside the call as a zero-length matched flow to itself. That is not very interesting in its own right, but it serves as a new starting point for further analysis of the body of the called function. Note that the matched flow inferred at a is actually independent of the $)_i$ edge and the x node to which that edge returns. This is an important property: it means that whatever we learn while analyzing the callee will be reusable if some other flow eventually reaches a as well. We never need to search for matched flows to a more than once. This figure actually represents three patterns, with *inferred* varying over {matched, pos, neg}.

Figure 3d, promote-return: If we have found a negative flow that leads up to a return edge, that return edge might eventually match up with a call. This possibility will be explored due to the earlier rule that adds a matched self edge. However, it is also possible that this return edge will never match up with a call, i.e., that this return edge is the final step in a positive flow. Discovering that positive flow is important, because it can eventually become a valid reaching path in conjunction with the negative flow we started out with. So we promote this return edge into a single-step positive flow, and possibly continue working backward from there.

Figure 3e, return-nonneg: Unmatched returns extend positive flows, or turn matched flows into positive flows. This figure actually represents two patterns, with *nonneg* varying over {matched, pos}.

Figure 3f, call-nonpos: Unmatched calls extend negative flows, or turn matched flows into negative flows. However, this should only be done when the negative flow could potentially be part of a valid flow to a seed. This figure actually represents two patterns, with *nonpos* varying over {matched, neg}.

Figure 3g, hoist-justified: Hoist matched flow from callee to caller, but only when we already have some inferred flow after the call to justify our interest. This figure actually represents three patterns, with *inferred* varying over {matched, pos, neg}.

Figure 3h, valid: A valid reaching path is positive flow followed by negative flow, per Section III-B.

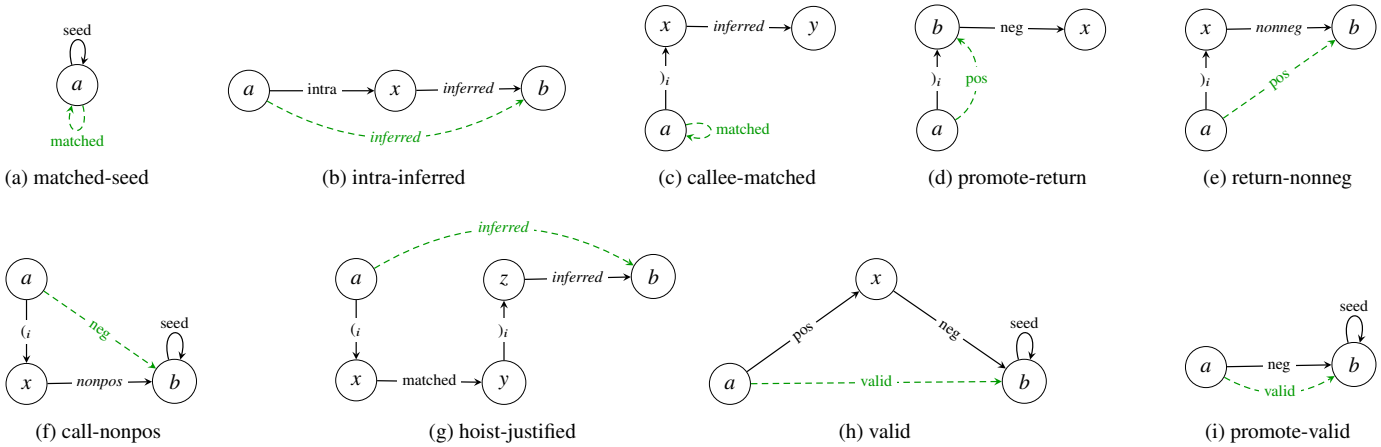
Figure 3i, promote-valid: Alternatively, a valid reaching path can be negative flow only with no preceding positive flow, still ending at a seed of interest. Also inferring a valid reaching path from positive flow only with no preceding negative flow is unnecessary: the b end of such a path will always carry a seed edge from which a zero-length negative flow can be inferred using other rules.

IV. Approach

In theory, Section III reveals the critical elements needed in order to perform error-propagation analysis. In practice, memory and other resource limits present significant scalability challenges. Here we discuss practical considerations that must be addressed to apply the ideas of Section III to large, real-world code bases. Sections IV-A and IV-B describe our disk-based graph indexing scheme. Section IV-C describes how our selected representation is used in the main analysis algorithm. We describe additional supporting analyses and optimizations in Sections IV-D and IV-E, while Section IV-F considers effective use of limited memory and disk for further performance gains.

A. Hexastore: Efficient Management of Large Graphs

Using the graph saturation strategy of Section III requires a suitable data structure to represent the variable-expanded CFG. Unfortunately, this graph (including inferred edges) can easily grow too large to fit in memory, limiting the size of programs that can be analyzed. Databases, however, routinely work with data



(a) matched-seed (b) intra-inferred (c) callee-matched (d) promote-return (e) return-nonneg

(f) call-nonpos (g) hoist-justified (h) valid (i) promote-valid

Fig. 3. Inference patterns for valid-paths analysis. In each pattern, *italics* represent wildcard values to be filled in after matching the pattern, while regular upright text represents specific values that must appear as given. Solid black arrows (⟶) represent edges already present or inferred, while green dashed arrows (⇢) represent new edges to be added when the pattern matches.

too big to fit in memory. Semantic Web research in particular offers *triple stores* optimized for storing and accessing large graph data to and from disk. With a disk-based representation of the variable-expanded CFG, available memory acts merely as a working cache or buffer, not as a limiting upper bound.

Good performance on large graphs depends critically on the indexing scheme used: we must be able to quickly find needed information either on disk or in memory buffers while minimizing I/O and staying within reasonable memory limits. The approach we use, Hexastore [39], indexes graphs six-fold, according to each of the permutations of source, destination, and edge label. For example, Hexastore's (*source*, *edge*, *destination*) index maintains a sorted, disk-based map indexed by source node. Each source node maps to a sorted index of outgoing edge labels, and each of these edge labels maps to a sorted set of destination nodes for the given source node and edge label. Thus with one look-up we can quickly find all edges for a given source node. With a second look-up we can quickly find all destination nodes for a given source node and edge label. Hexastore's other indices allow similar look-ups based on other permuted criteria. In general, given any one or two fixed elements of a (*source*, *edge*, *destination*) tuple, Hexastore can quickly access the related remaining information.

Hexastore allows storing indices on disk [40]. Reads from this persistent structure are relatively cheap, while updates require rebuilding the entire persistent structure. Reindexing is relatively quick even for large numbers of edges: about 8 minutes for 100 million edges on a standard desktop machine with an average hard drive. Doing this for every new edge is impractical, but batch updates can work well for long-running analyses. New edges can be held in memory until enough have accumulated to make it worthwhile to pause, rebuild indices, and then continue.

A disk-based representation has other advantages besides reducing memory requirements. The entire analysis is naturally persistent; it can easily be interrupted mid-analysis and resumed later, even on a different machine. Distributed analysis is also straightforward: multiple machines can trivially share the same disk-based representation of the initial variable-exploded CFG, with each machine exploring different paths in parallel. Pooling discovered edges would require cross-machine coordination. However, the idempotent and monotonic nature of edge saturation makes this easier to manage. For example, we always have the option to redundantly discover edges that have already been discovered elsewhere. This lets us reduce or eliminate cross-machine communication in exchange for extra local work.

B. Index Optimizations

It is convenient to refer to graph nodes by unique names. We use names of the form "*function*.*location variable*" to refer to the value of a given *variable* at a particular numbered *location* of a given *function*. Location numbers are derived from a simple intraprocedural numbering of CFG nodes. Local variables' names are systematically renamed to be globally unique.

In practice, these names are interned and mapped into unique ID numbers suitable for use as index offsets. Unfortunately, the sheer number of such unique names would require an unacceptably large string intern table. To our knowledge, this problem has not previously arisen in popular Semantic Web data sets. Relative to those graphs, our data set contains a much larger number of nodes that are much less densely connected: most of our nodes have only one incoming and one outgoing edge.

We optimize the string intern pool and data indices by treating function names as namespaces. Location numbers and variables are tuples in those namespaces. Thus, rather than interning a complete "*function*.*location variable*" string, we instead intern and assign ID numbers to just the function and variable names. We then combine these with the location (already numerical) to form a unique triple of numbers. This triple can then be used as the key to look up the value of a given variable at a given location in a given function in various Hexastore indices.

C. Graph Saturation

The main analysis iterates over the set of all assignments, after applying the program transformations described in Section II. For each of these assignments, we perform two graph saturations

as described in Section III: one determines the new value of the variable after the assignment, while the other determines the previous value of the same variable before the assignment. We refer to this pair of saturations as one *analysis instance*. During each analysis instance we maintain a work list with recently discovered edges so that we only look for new pattern matches where new discoveries were made. After each instance completes, we print its results for diagnostic purposes, then continue to the next assignment. In practice, variables are often overwritten by constants. In this case, saturation to determine the variable's new value completes within one step. Saturation to find the old value commonly takes much longer.

The main analysis operates on a graph representation with implicit identity edges. Implicit flows are materialized as needed to match inference patterns from Figure 3, but are never persistently stored in memory or on disk.[3] New edges inferred during saturation are always fully explicit, and ultimately comprise the bulk of the saturated graph as shown in Table I.

D. Global Variable Bypass

Assignments to global variables are rare: most global variables flow unchanged through nearly all functions except for the few places where one or another is explicitly overwritten. If saturation leads through many function calls with many edges to traverse, we may add a vast number of edges without receiving any new information. However, if we know in advance that certain graph regions do not modify certain global variables, we can bypass those areas entirely during saturation. By avoiding this unnecessary work, we can reduce both time and memory requirements while preserving correctness.

To achieve this, we use several pre-analyses to determine which functions modify which global variables. One intraprocedural analysis determines which global variables may or must be modified directly by each function. A second intraprocedural analysis determines which functions may or must be called directly from within each function.

During graph saturation, our analysis monitors which specific variable corresponds to a given inferred edge. When saturation could traverse into a called function (Figures 3c to 3e), we consider whether the resulting edge corresponds to a global variable that must never be changed during that call. This requires computing an interprocedural transitive closure across the intraprocedural may/must facts recorded earlier. Hexastore's indices are ideally suited to computing transitive closures such as these, using a number of queries linear in the size of the result set [39]. Hexastore also has the advantage of being able to compute this information on demand when needed by the saturation analysis. Intermediate inferences are memoized for possible reuse during later graph saturation iterations.

If we determine that the global variable in question would not be modified during the call, then we bypass the call entirely. We do not descend into the callee as suggested by Figures 3c to 3e. Instead, we bridge the global variable's value across the call as an inferred identity flow. The effect is similar to that of the hoist-justified pattern in Figure 3g, but without spending time and memory to find a matched $x \rightarrow y$ flow across the callee.

[3] This approach is analogous to a video player that decompresses and displays video frames on-the-fly without ever storing the entire decompressed stream.

E. Unreachable Code After Infinite Loops

Linux intentionally contains numerous infinite loops immediately after code that detects fatal errors. Without treating these carefully, our predominantly-backward analysis might discover extra flows of values originating in code that could never actually run. To exclude these, we impose an extra reachability check whenever saturation discovers an assignment from a constant to a variable. The reachability check determines whether the assignment itself can be reached from the entry of the containing function. This check is performed on-demand within our analysis. We check intraprocedurally only: we do not attempt to filter out unreachable code that follows a call to a non-returning function. However, this simple intraprocedural check proves sufficient in practice.

F. Memory and Disk Management

Our analysis tool is very flexible in how it uses memory and disk storage. We can leave the initial graph in memory while keeping inferred edges in memory only up to a certain number. After that we can discard inferred edges (possibly rediscovering them later) or instead add them to persistent disk storage. As described in Section IV-A, writing each discovered edge to disk one-by-one is prohibitively slow. Batched updates, however, are quite practical. Another option is to leave the initial graph persistently on disk and to keep some or all discovered edges in memory. Intuitively, while the analysis can cater to almost arbitrary memory constraints, the more data that can be managed in memory the faster the analysis runs.

V. Experimental Evaluation

We have conducted several experiments to evaluate the usefulness of our database-backed approach. We introduce the benchmarks and hardware configuration used for our experiments in Section V-A. Section V-B discusses correctness with respect to a reference implementation. The remainder of this Section presents our experiments with different analysis configurations and varying memory usage constraints. Our first experimental setup (Section V-C) compares the performance of the all-in-memory configuration of our tool to the in-memory error-propagation analysis of Rubio-González et al. [16]. Our second experiment (Section V-D) demonstrates an optimized configuration strategy for programs whose initial edges fit easily in memory, but whose many inferred edges must periodically be flushed to disk. Lastly (Section V-E), we introduce a complementary configuration for programs with very large initial graphs, but relatively few inferred edges.

A. Benchmarks and Hardware Configuration

We evaluated our implementation on a suite of benchmarks of varying size and complexity. Our "Ext3" and "ReiserFS" benchmarks are parts of the Linux 2.6.38.3 operating system kernel that implement specific file systems. We include these relatively small examples because analysis of error management

TABLE I
SIZES OF PROGRAMS ANALYZED

Program	Lines of code	Assignments	Edges	
			Initial	Inferred
Ext3	175,631	15,059	426,699	4,582,498
ReiserFS	186,334	13,041	438,234	4,527,496
Linux	1,144,394	120,784	3,316,392	1,368,826,792
Linux*	1,144,394	120,783	3,316,391	1,269,389,507
Firefox	3,026,254	68,653	10,636,371	13,045,589

in file system code was the main focus of prior work in this area. Our "Linux" and "Linux*" benchmarks represent a much larger portion of the Linux 2.6.38.3 kernel. These two differ only in that the Linux* benchmark excludes a single assignment that consistently required an unusually long time to analyze. This assignment appears deep within a widely-used memory management module; its pre-assignment value can come from myriad other locations throughout the kernel. As it happens, analysis ultimately showed that this specific assignment does not constitute an error-propagation bug. The incrementally-reduced Linux* benchmark is a practical analysis target if one is willing to skip a single difficult (but ultimately non-buggy) assignment.[4] The full Linux benchmark shows expected performance if everything must be checked without even a single exception. These four Linux-derived benchmarks are written in C. Of course, error-propagation analysis is also important beyond Linux and C. Our "Firefox" benchmark is the complete source code to the open-source web browser. Firefox is written in C++, but uses the return-code idiom instead of exceptions.

Table I summarizes the size and complexity of our data sets, measured in various ways. The nature of this "complexity" is very different for the Linux and Firefox benchmarks. Firefox has roughly equal numbers of initial and inferred edges, meaning that the initial graph accounts for a significant fraction of Firefox's total memory needs. By contrast, Linux's initial graph is one third the size, but has twice as many assignments, and analysis of these infers one hundred times as many edges. Thus, analyzing Linux puts a much greater burden on efficient management of inferred edges. This also suggests that flows in Linux are significantly longer and more complex than those in Firefox.

Experiments were run on one 2.67 GHz CPU of a 12-way Intel Xeon desktop workstation, with 24 GB of RAM, 8 GB of swap space, and a commodity 7,200 RPM SATA hard drive, running Red Hat Enterprise Linux 6.3. In Tables II, IV and V, "RAM (GB)" columns refer to the peak memory used at any moment during the entire analysis of a given benchmark.

B. Implementation and Correctness

We offer two front ends for converting source code into graph form for analysis. First, we have adapted a CIL-based [41] front end used in prior work by adding the intraprocedural may/must analyses described in Section IV-D. CIL handles C (but not C++), including non-standard C extensions used in the Linux kernel. Second, we have developed an entirely new front end derived from Clang and LLVM [42]–[44]. Custom LLVM passes implement the may/must analyses and graph generation. Clang handles both C and C++, but not some C extensions used by Linux [45]. We use the CIL-based front end for all Linux-derived benchmarks, and the LLVM-based front end for Firefox.

Our goal is to perfectly replicate the analysis results of prior error-propagation work while dramatically reducing memory requirements and thereby improving scalability. To provide a basis for comparison, we reran all experiments from scratch using a reference implementation provided by Rubio-González et al. [16]. This implementation treats error propagation as a path problem over weighted pushdown systems. It uses the WALi weighted pushdown system library [46] version 3.4 to compute meet-over-all-paths solutions before and after each assignment. WALi is a general framework that must be instantiated using data structures selected specifically for the problem at hand. The WALi implementation of error-propagation analysis represents transfer functions (also referred to as weights) using binary decision diagrams (BDDs) [47] as implemented by the BuDDy BDD library [48] version 2.4, carefully hand-tuned for performance on this problem. In the remainder of this Section, we refer to this reference implementation as the "WALi-based analysis" or simply "WALi". We refer to our database-backed approach as the "Hexastore-based analysis" or simply "Hexastore".

The remainder of this Section focuses on evaluating performance and scalability. However, we have also carefully checked the correctness of the analysis results in all reported experiments, taking the WALi-based analysis as a reference. Our Hexastore-based analysis results exactly agree with those of the WALi-based analysis at each of the 338,320 assignments across all five of our benchmarks. The only difference observed between the two implementations consists of 1,331 Firefox assignments that Hexastore reports but WALi does not.

We have manually inspected 824 out of these 1,331 assignments. We found that all involve synthetic assignments introduced at the end of functions to "clear" local variables that are about to go out of scope. Most of these local variables, in turn, are temporary variables introduced by our front end. The additional assignments missed by WALi, then, predominantly arise as side effects of internal representations and are not a significant practical concern. We believe these are actually unreachable code, such as functions never called from the main entry point. Our Hexastore-based analysis works backward, stopping once it reaches a constant such as EIO. Unlike WALi, it does not try to prove that the originating constant assignment is itself reachable. This can cause Hexastore to report results for unreachable assignments that WALi's forward analysis would not report.

C. In-Memory Only

A database-backed approach can potentially fall back on disk if data grows too large for memory. To establish a baseline for comparison, we first experimented with a Hexastore configuration that is strictly memory-based. We held all indices

[4] Outliers such as this problematic assignment stand out easily in our analysis logs. Furthermore, analyses are disk-backed and saved per assignment. Thus, one can easily stop, recognize and exclude an outlier, then resume without it.

TABLE II
PERFORMANCE OF IN-MEMORY ANALYSES. "-" MARKS WALI ANALYSES THAT REQUIRE MORE THAN 24 GB OF RAM PLUS 8 GB OF SWAP SPACE.

	RAM (GB)		Time (minutes)	
Program	WALi	Hexastore	WALi	Hexastore
Ext3	1.08	0.30	0.56	0.66
ReiserFS	1.47	0.30	0.69	0.74
Linux	-	11.00	-	460.68
Linux*	-	9.57	-	423.74
Firefox	-	4.74	-	7.98

TABLE III
COMPARISON OF IN-MEMORY ANALYSIS TIMES FOR LARGE BENCHMARKS. RELATIVE TIMES ARE THE RATIO OF HEXASTORE TO WALI.

	Absolute (minutes)		
Program	WALi	Hexastore	Relative
Linux	966.00	421.69	0.44
Linux*	966.77	387.98	0.40
Firefox	12.75	7.12	0.56

representing the initial graphs in memory. We also held inferred edges in memory, but checked how many we had accumulated by the end of each analysis instance (i.e., after analyzing each assignment). If we had more than two million, then we purged all inferred edges from memory and started fresh for the next assignment. Keeping inferred edges around lets us avoid some redundant work, but managing large data structures adds overhead. Empirically, we found that discarding after two million edges was a good balance.

Unlike WALi, Hexastore was designed with disk in mind and is not necessarily optimized for memory-only performance. We therefore expected that the WALi-based analysis would be significantly faster than a memory-only configuration of the Hexastore-based analysis. Table II shows our results, comparing both time and memory required to complete each analysis. Contrary to our expectations, Hexastore outperformed WALi on all counts. For the smaller Ext3 and ReiserFS benchmarks, Hexastore is nearly as fast as WALi and uses far less memory. We attribute Hexastore's superior performance to several factors. As mentioned in Section IV-D, Hexastore's indices allow it to compute transitive closures very efficiently. These indices are also very compact, with the last level of each stored as a densely-packed, cache-friendly, sorted vector. Less memory also means less time spent allocating, filling, and managing that memory.

Hexastore succeeded on all three large benchmarks where WALi failed outright due to memory exhaustion. For the sake of completeness, we also ran the WALi analyses on a large, dedicated server not available to most developers. WALi required 39 GB of RAM to analyze Firefox. In theory, WALi would require 22 GB to analyze Linux or Linux*. In practice, memory pressure from the operating system and other processes doomed the attempt on our developer-grade workstation: the WALi analysis ran for several hours, swapped heavily, and was eventually killed by the operating system when both RAM and swap were depleted. Even if ample swap space were available, common wisdom holds that database-managed I/O outperforms generic OS swapping [49]. Thus, swapping alone is unlikely to solve the memory scalability barrier.

The large server needed for WALi has faster CPUs (3.07 GHz) than the humble desktop described in Section V-A and used for Table II. Yet Hexastore completes the large analyses dramatically faster, as shown in Table III. Hexastore uses just 40% to 56% as much time as WALi requires for the large benchmarks.[5]

D. Optimized Index Management for Many Inferred Edges

Per Table II, analyzing Linux entirely in memory required 11 GB of RAM. This significantly improves upon WALi, for which 24 GB of RAM plus 8 GB of swap was insufficient. However, we wished to reduce memory consumption even further, to make analysis on common desktop computers feasible.

As we described in Section V-A, the majority of memory needed to analyze Linux comes from the inferred edges. For this experimental setup we therefore maintained the relatively compact initial index in memory as well as some (but not all) inferred edges. We discarded inferred edges between analysis instances once more than two million had accumulated, using the threshold described in Section V-C. Since we needed to allow this threshold to be exceeded during an analysis instance, we introduced a second threshold: the *flush-to-disk threshold*. If the flush-to-disk threshold was exceeded at any point during saturation, all inferred edges indices were immediately flushed to disk, all on-disk Hexastore indices were rebuilt, and the in-memory index representation reset. Once an index was flushed to disk, newly-inferred edges were maintained in memory again until the flush-to-disk threshold was exceeded, and so on. Therefore, inferred edges have to be retrieved from both the in-memory indices, as well as from disk. Disk is slower than memory, of course. It is therefore crucial to choose the flush-to-disk threshold carefully: we must not exceed memory constraints, but also should not waste time rebuilding the persistent indices too often. This setup is particularly beneficial for data sets where individual analysis instances discover too many inferred edges to fit comfortably into memory.

For this setup we expected to see an upper memory consumption bound not to be exceeded, but more analysis time required compared to in-memory only analyses. For Firefox it did not make sense to raise the flush-to-disk threshold above 1,000,000, since no Firefox analysis instance ever infers more than 1,000,000 edges. However, Linux contains 99,437,285 edges for one particular analysis instance. In this case an appropriate flush-to-disk threshold might significantly reduce memory consumption.

[5] All times in Table III were measured using the same large, dedicated server with 3.07 GHz CPUs; all other experimental results in this paper used the slower 2.67 GHz CPUs of our desktop workstation. Thus, the Hexastore analysis times in Table III are consistently slightly smaller than corresponding times in Table II.

TABLE IV
LINUX ANALYSIS PERFORMANCE WHEN FLUSHING EXCESSIVE INFERRED EDGES TO DISK. "∞" MEANS NEVER FLUSH TO DISK.

Program	Flush-to-disk threshold	RAM (GB)	Time (minutes)
Linux	∞	11.00	460.68
Linux	20,000,000	9.57	1,337.25
Linux	10,000,000	10.32	1,961.15
Linux*	∞	9.57	423.74
Linux*	10,000,000	10.32	882.33

Table IV presents the results for this experiment. Analyzing Linux with a flush-to-disk threshold of 20 million edges cuts memory requirements by 13% in exchange for tripling analysis time. Decreasing the flush-to-disk threshold to 10 million does not further reduce peak memory; in fact, memory requirements grow slightly. This lower threshold also increases analysis time by 47%, demonstrating that excessive flushing can be counter-productive. Some memory overhead is required to merge data accessed from disk as well as for recording interprocedural may/must facts computed on demand. This collection of facts may grow fairly large for large numbers of functions and global variables. This is why flushing at 10 million edges does not halve memory consumption relative to flushing at 20 million edges.

We ran Linux* only with a threshold of 10 million edges: no single analysis instance infers more than 15 million edges, so a threshold of 20 million edges would be equivalent to no threshold at all. We found flushing at 10 million edges increased analysis time without reducing memory consumption. In fact, 9.57 GB of memory seems to be a lower bound for both Linux and Linux*. We attribute this to the accumulation of may/must facts that are never discarded from memory.

E. Optimized Index Management for Many Initial Edges

The initial graph is used frequently during analysis, so keeping its indices in memory improves performance significantly. Unfortunately, this may simply be impossible if the initial graph is too large. However, while Hexastore maintains six indices (per Section IV-A), not all are used equally often. Our analysis works predominantly backwards, so the most heavily-used index is that which maps from **d**estination nodes to **e**dge labels to **s**ource nodes: we call this the *DES index*. It may make sense to maintain only the initial graph's DES index in memory and access all other initial-graph indices from disk. This leaves more memory available for storing indices over inferred edges. This can aid in analyzing data sets that have a large initial graph but that infer only a moderate number of edges during each analysis instance. Our Firefox benchmark is one such example.

Table V shows that maintaining only the initial DES index and inferred edges in memory reduces memory consumption by 45% when analyzing Firefox. This is understandable given that the initial edges constitute 43% of Firefox's entire graph data. Maintaining the frequently-accessed DES index in memory allows the analysis to complete within a reasonable time-frame of slightly less than two hours. If all indices were accessed from disk, analysis would take nearly 24 hours. These results are particularly striking considering that the WALi-based approach required over 39 GB of RAM to perform the same analysis. With this configuration we can analyze Firefox on a contemporary laptop computer within reasonable time. For example, at the time of this writing, the best-selling laptop on Amazon.com costs $249 and comes with 4 GB of RAM [50].

TABLE V
FIREFOX ANALYSIS PERFORMANCE FOR DIFFERENT INITIAL INDEX MANAGEMENT STRATEGIES

Strategy	RAM (GB)	Time (minutes)
only initial DES in RAM	2.64	114.58
all initial indices in RAM	4.74	7.98

F. Hexastore Configuration Guidelines

Hexastore is very flexible. It is to be expected that proper tuning is necessary to achieve optimal performance for data sets of this size. Even commercial databases require a fair amount of tuning in order to perform well. To achieve best performance results using our Hexastore-based analysis, we recommend adhering to the following configuration guidelines.

A purely in-memory configuration is a reasonable starting point. We recommend trying this first, especially since this configuration is fastest. Hexastore's more efficient representation means that many problems that did not fit in memory before may now fit with no need for secondary storage.

If memory must be conserved, though, the first priority is to keep the most heavily-used indices in memory. For our predominantly-backward analysis, the top priority is the DES index that maps **d**estinations to **e**dges to **s**ources. Any remaining memory should be devoted first to other indices representing the initial graph, and then to inferred edges, with periodic discarding or flushing to disk as needed to keep within allotted memory bounds. We recommend applying the configurations in the described order to find a data-set-optimal setup.

VI. RELATED WORK

a) Scalable Program Analysis: Modular static program analysis [51]–[54] has been proposed to reduce analysis running time and memory cost. The idea is to analyze parts (e.g., functions) of very large programs separately and then compose the analyses of these program parts to obtain information on the whole program. One advantage of this approach is that the process may be parallelizable, depending on the interactions between the different parts of the program. Another approach is to use sparse analysis techniques to improve scalability [4], [55]–[57]. The key idea is that for a given program analysis, many parts of the program may not be relevant. Excluding irrelevant parts of the program effectively shrinks the analysis problem. Our technique is complementary to these approaches, and could be combined with them to improve scalability even further.

b) Datalog for Program Analysis: The IFDS style of inferring new facts from old is closely related to bottom-up evaluation of Datalog programs. The inference patterns in

Figure 3 could be recast as Datalog inference rules, as can a wide variety of other program analysis problems [11]–[14], [58]–[64].

Doop encodes Java points-to analyses in a dialect of Datalog for evaluation on the commercial LogicBlox database engine [11]. Direct comparison between our approach and Doop is not meaningful, as the two perform different analyses on different source languages. Instead, one might reimplement our approach using LogicBlox's Datalog dialect and engine, or conversely implement a disk-backed, LogicBlox-compatible Datalog engine atop Hexastore.

SemmleCode uses another Datalog dialect to encode a variety of program queries. These are translated into SQL and executed on any relational database [12], [13]. However, prior Semantic Web research suggests that triple stores outperform relational databases for certain graph traversal tasks, such as transitive closure. This is due to the use of highly-optimized indices that significantly reduce the number of expensive joins [39], [65].

Reps [66] described how to use the "magic-sets" transformation for logic programs [67]–[69] to convert exhaustive dataflow analyses into demand-driven analogues. This method alone is not sufficient to produce our inference patterns from a naïve, exhaustive formulation. However, this and other optimizations from the logic-programming and deductive-database communities could certainly simplify parts of that process, making it easier to formulate new analyses for extreme scalability.

c) Large Graph Data and the Semantic Web: The Semantic Web community actively encourages exploring new ways to manage and reason over large graph data. One example is the Billion Triple Challenge held annually in conjunction with the International Semantic Web Conference (ISWC) [70]. Participants are provided with a data set containing a Semantic Web graph with one billion edges and try to come up with new applications working with this data. Another Semantic Web trend is to put Semantic Web technologies into application context, thereby providing new problem-solving methods to other areas. For example, the Workshop on Semantic Web Enabled Software Engineering (SWESE) was created to improve software development activities by applying Semantic Web technologies. Semantic Web inference naturally requires computing transitive closures of relations. Keivanloo and Rilling [71] exploited inference engines to improve source code clone detection, but did not focus on single-instance scalability.

Motivated by the need for web-scale data management, we have seen some progress on graph-optimized index structures and database systems [39], [65], [72], [73]. However, in recent years research activity has shifted from improving single-instance Semantic Web data management to leveraging cloud services and offloading scalability concerns to distributed storage and compute systems [74], [75]. These developments bode well for long-term future scalability of database-backed program analyses: while we have already reduced single-machine resource needs, distributing program analysis into the cloud will allow even greater leaps of scalability.

VII. Conclusion

As software grows in size and complexity, main memory imposes restrictions in the scalability of program analyses. Fortunately, the database community smashed through the memory barrier long ago; a modern Semantic Web database can represent billions of relations (edges) among entities (nodes) using a tunable mix of main memory and secondary disk storage. This suggests a way for static program analyses to break through the memory barrier as well. We have presented a reformulation of error-propagation analysis as a graph saturation problem in the style of Reps et al.'s IFDS framework. Our inference patterns are carefully designed to create a demand-driven, predominantly-backward analysis that avoids wasted effort wherever possible. To manage the large graphs needed for this analysis, we have adapted Hexastore, a Semantic Web database. Hexastore is especially well-suited to efficiently storing, querying, and transitively-closing millions or even billions of related entities.

Hexastore can be configured to use a flexible mixture of memory and disk, allowing us to solve error-propagation analyses in a fraction of the time and/or memory required by prior work. Even when using no disk at all, our Hexastore-based analyses of large programs finish 44% to 60% faster and use 50% to 88% less memory. Incorporating disk storage allows further memory savings in exchange for longer analysis times. Our database-backed strategy, configured to use both memory and disk, can analyze over one million lines of Linux* in seven hours and under 10 GB of RAM: a very reasonable allocation for overnight processing on any decently-equipped developer workstation. Using prior approaches, Firefox's three million lines of code required twelve minutes, 39 GB of RAM, and a dedicated server to handle the workload. Our database-backed approach requires something closer to a laptop: it completes in eight minutes and less than 5 GB of RAM, or can even be restricted to just 2.6 GB of RAM if one has two hours to spare.

Using the configuration guidelines in Section V-F, developers equipped with our Hexastore-based, database-backed analysis engine can continue to scale important program analyses up to meet the demands of large software today and into the future.

Acknowledgment

We thank the anonymous reviewers for their invaluable feedback. This research was supported in part by NSF grants CCF-0953478, -1217582, -1318489, -1420866, and -1423237; DoE grant DE-SC0008699; DARPA subcontract R18682; and gifts from Oracle and Mozilla. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

The first and third authors are deeply grateful to the organizers of Dagstuhl Seminar 07491, without whose kind invitations we likely would have never met, married, or written this paper.

References

[1] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 290–299.

[2] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *PLDI*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 270–280.

[3] K. Chen and D. Wagner, "Large-scale analysis of format string vulnerabilities in Debian Linux," in *PLAS*, M. W. Hicks, Ed. ACM, 2007, pp. 75–84.

[4] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and implementation of sparse global analyses for C-like languages," in *PLDI*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 229–238.

[5] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *SIGSOFT FSE*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 343–353.

[6] C. Cifuentes, N. Keynes, L. Li, N. Hawes, M. Valdiviezo, A. Browne, J. Zimmermann, A. Craik, D. Teoh, and C. Hoermann, "Static deep error checking in large system applications using Parfait," in *SIGSOFT FSE*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 432–435.

[7] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *POPL*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 226–238.

[8] O. Lhoták and K.-C. A. Chung, "Points-to analysis with efficient strong updates," in *POPL*, T. Ball and M. Sagiv, Eds. ACM, 2011, pp. 3–16.

[9] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code," in *CGO*, A. Moshovos, J. G. Steffan, K. M. Hazelwood, and D. R. Kaeli, Eds. ACM, 2010, pp. 218–229.

[10] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *CGO*. IEEE, 2011, pp. 289–298.

[11] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA*, S. Arora and G. T. Leavens, Eds. ACM, 2009, pp. 243–262.

[12] E. Hajiyev, M. Verbaere, O. de Moor, and K. D. Volder, "CodeQuest: Querying source code with DataLog," in *OOPSLA Companion*, R. E. Johnson and R. P. Gabriel, Eds. ACM, 2005, pp. 102–103.

[13] E. Hajiyev, M. Verbaere, and O. de Moor, "*CodeQuest*: Scalable source code queries with Datalog," in *ECOOP*, ser. Lecture Notes in Computer Science, D. Thomas, Ed., vol. 4067. Springer, 2006, pp. 2–27.

[14] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *POPL*, T. Ball and M. Sagiv, Eds. ACM, 2011, pp. 17–30.

[15] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: Error handling is occasionally correct," in *FAST*, M. Baker and E. Riedel, Eds. USENIX, 2008, pp. 207–222.

[16] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *PLDI*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 270–280.

[17] F. Cristian, "Exception handling," in *Dependability of Resilient Computers*, 1989, pp. 68–97.

[18] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *ICSE*, 2000, pp. 418–427.

[19] R. P. L. Buse and W. Weimer, "Automatic documentation inference for exceptions," in *ISSTA*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 273–282.

[20] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *In Object-Oriented Programming, 11th European Conference (ECOOP)*. Springer-Verlag, 1997, pp. 85–103.

[21] M. P. Robillard and G. C. Murphy, "Regaining control of exception handling," University of British Columbia, Vancouver, BC, Canada, Tech. Rep., 1999.

[22] GCC team, "GCC coding conventions," Sep. 2013. [Online]. Available: http://gcc.gnu.org/codingconventions.html

[23] LLVM Project, "LLVM coding standards," Oct. 2013. [Online]. Available: http://llvm.org/docs/CodingStandards.html

[24] P. Terdiman, "Exceptions: just say no," Mar. 2008. [Online]. Available: http://www.codercorner.com/blog/?p=33

[25] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landray, "Google C++ style guide," Sep. 2013, revision 3.274. [Online]. Available: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

[26] G. Candea, M. Delgado, M. Chen, and A. Fox, "Automatic failure-path inference: A generic introspection technique for Internet applications," in *Proceedings of the The Third IEEE Workshop on Internet Applications (WIAPP '03)*. San Jose, California: IEEE, Jun. 2003, pp. 132–141.

[27] C. A. Flanagan and M. Burrows, "System and method for dynamically detecting unchecked error condition values in computer programs," United States Patent #6,378,081 B1, Apr. 2002.

[28] T. Goradia, "Dynamic impact analysis: A cost-effective technique to enforce error-propagation," in *ISSTA*, 1993, pp. 171–181.

[29] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *DSN*. IEEE Computer Society, 2001, pp. 161–172.

[30] ——, "Propane: an environment for examining the propagation of errors in software," in *ISSTA*, 2002, pp. 81–85.

[31] ——, "Epic: Profiling the propagation and effect of data errors in software," *IEEE Trans. Computers*, vol. 53, no. 5, pp. 512–530, 2004.

[32] A. Jhumka, M. Hiller, and N. Suri, "Assessing inter-modular error propagation in distributed software," in *SRDS*. IEEE Computer Society, 2001, pp. 152–161.

[33] A. Johansson and N. Suri, "Error propagation profiling of operating systems," in *DSN*. IEEE Computer Society, 2005, pp. 86–95.

[34] K. G. Shin and T.-H. Lin, "Modeling and measurement of error propagation in a multimodule computing system," *IEEE Trans. Computers*, vol. 37, no. 9, pp. 1053–1066, 1988.

[35] M. W. Bigrigg and J. J. Vos, "The set-check-use methodology for detecting error propagation failures in I/O routines," in *Workshop on Dependability Benchmarking*, Washington, DC, Jun. 2002.

[36] C. Rubio-González and B. Liblit, "Expect the unexpected: error code mismatches between documentation and the real world," in *PASTE*, S. Lerner and A. Rountev, Eds. ACM, 2010, pp. 73–80.

[37] ——, "Defective error/pointer interactions in the Linux kernel," in *ISSTA*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 111–121.

[38] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, R. K. Cytron and P. Lee, Eds. ACM Press, 1995, pp. 49–61.

[39] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for Semantic Web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[40] C. Weiss and A. Bernstein, "On-disk storage techniques for Semantic Web data - Are B-trees always the optimal solution?" in *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, Oct. 2009, pp. 49–64.

[41] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *CC*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 213–228.

[42] "clang: a C language family frontend for LLVM," Nov. 2013. [Online]. Available: http://clang.llvm.org/

[43] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE Computer Society, 2004, pp. 75–88.

[44] C. Rubio-González, "Finding error-propagation bugs in large software systems using static analysis," Ph.D. dissertation, University of Wisconsin–Madison, Aug. 2012.

[45] J.-S. Möller, B. Webster, and M. Charlebois, "LLVMLinux," Feb. 2014. [Online]. Available: http://llvm.linuxfoundation.org/

[46] N. Kidd, T. Reps, and A. Lal, "WALi: A C++ library for weighted pushdown systems," Aug. 2012. [Online]. Available: http://www.cs.wisc.edu/wpis/wpds/download.php

[47] R. E. Bryant, "Binary decision diagrams and beyond: enabling technologies for formal verification," in *ICCAD*, R. L. Rudell, Ed. IEEE Computer Society, 1995, pp. 236–243.

[48] J. Lind-Nielsen and H. Cohen, "BuDDy: A BDD package," Apr. 2013. [Online]. Available: http://sourceforge.net/projects/buddy

[49] M. Stonebraker and A. Kumar, "Operating system support for data management," *IEEE Database Eng. Bull.*, vol. 9, no. 3, pp. 43–50, 1986.

[50] "Amazon best sellers: Best sellers in laptop computers," Jan. 2015. [Online]. Available: http://www.amazon.com/Best-Sellers-Electronics-Laptop-Computers/zgbs/electronics/565108/

[51] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw., Pract. Exper.*, vol. 30, no. 7, pp. 775–802, 2000.

[52] P. Cousot and R. Cousot, "Modular static program analysis," in *CC*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 159–178.

[53] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the Saturn project," in *PASTE*, M. Das and D. Grossman, Eds. ACM, 2007, pp. 43–48.

[54] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *PLDI*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 567–577.

[55] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *POPL*, D. S. Wise, Ed. ACM Press, 1991, pp. 55–66.
[56] G. Ramalingam, "On sparse evaluation representations," in *SAS*, ser. Lecture Notes in Computer Science, P. V. Hentenryck, Ed., vol. 1302. Springer, 1997, pp. 1–15.
[57] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of C code," in *NDSS*. The Internet Society, 2004.
[58] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *PLDI*, W. Pugh and C. Chambers, Eds. ACM, 2004, pp. 131–144.
[59] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *PODS*, C. Li, Ed. ACM, 2005, pp. 1–12.
[60] Y. Smaragdakis and M. Bravenboer, "Using Datalog for fast and easy program analysis," in *Datalog*, ser. Lecture Notes in Computer Science, O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, Eds., vol. 6702. Springer, 2010, pp. 245–251.
[61] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *PLDI*, H.-J. Boehm and C. Flanagan, Eds. ACM, 2013, pp. 423–434.
[62] T. W. Reps, "Demand interprocedural program analysis using logic databases," in *Workshop on Programming with Logic Databases (Book), ILPS*, ser. The Kluwer International Series in Engineering and Computer Science 296, R. Ramakrishnan, Ed. Kluwer, 1993, pp. 163–196.
[63] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using Datalog with binary decision diagrams for program analysis," in *APLAS*, ser. Lecture Notes in Computer Science, K. Yi, Ed., vol. 3780. Springer, 2005, pp. 97–118.
[64] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *ICSE*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 391–400.
[65] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.
[66] T. W. Reps, "Solving demand versions of interprocedural analysis problems," in *CC*, ser. Lecture Notes in Computer Science, P. Fritzson, Ed., vol. 786. Springer, 1994, pp. 389–403.
[67] J. Rohmer, R. Lescoeur, and J.-M. Kerisit, "The Alexander method - a technique for the processing of recursive axioms in deductive databases," *New Generation Comput.*, vol. 4, no. 3, pp. 273–285, 1986.
[68] C. Beeri and R. Ramakrishnan, "On the power of magic," *J. Log. Program.*, vol. 10, no. 3&4, pp. 255–299, 1991.
[69] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs," in *PODS*, A. Silberschatz, Ed. ACM, 1986, pp. 1–15.
[70] H. Alani, L. Kagal, A. Fokoue, P. T. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. F. Noy, C. Welty, and K. Janowicz, Eds., *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings*, ser. Lecture Notes in Computer Science, vol. 8218. Springer, 2013.
[71] I. Keivanloo and J. Rilling, "Clone detection meets semantic web-based transitive closure computation," in *Proc. RAISE*. IEEE, 2012, pp. 12–16.
[72] O. Erling, "Virtuoso, a hybrid RDBMS/graph column store," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 3–8, 2012.
[73] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, Eds. ACM, 2007, pp. 411–422.
[74] J. Urbani, J. Maassen, and H. E. Bal, "Massive semantic web data compression with MapReduce," in *HPDC*, S. Hariri and K. Keahey, Eds. ACM, 2010, pp. 795–802.
[75] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal, "OWL reasoning with WebPIE: Calculating the closure of 100 billion triples," in *ESWC (1)*, ser. Lecture Notes in Computer Science, L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds., vol. 6088. Springer, 2010, pp. 213–227.