Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code

Kai Wang Aftab Hussain Zhiqiang Zuo Guoqing Xu Ardalan Amiri Sani
University of California, Irvine
{wangk7, aftabh, zzuo2, guoqingx, ardalan}@ics.uci.edu

Abstract

There is more than a decade-long history of using static analysis to find bugs in systems such as Linux. Most of the existing static analyses developed for these systems are simple checkers that find bugs based on pattern matching. Despite the presence of many sophisticated interprocedural analyses, few of them have been employed to improve checkers for systems code due to their complex implementations and poor scalability.

In this paper, we revisit the scalability problem of interprocedural static analysis from a "Big Data" perspective. That is, we turn sophisticated code analysis into *Big Data analytics* and leverage novel data processing techniques to solve this traditional programming language problem. We develop *Graspan*, a disk-based parallel graph system that uses an *edge-pair* centric computation model to compute *dynamic transitive closures* on very large program graphs.

We implement *context-sensitive* pointer/alias and dataflow analyses on Graspan. An evaluation of these analyses on large codebases such as Linux shows that their Graspan implementations scale to millions of lines of code and are much simpler than their original implementations. Moreover, we show that these analyses can be used to augment the existing checkers; these augmented checkers uncovered **132** new NULL pointer bugs and **1308** unnecessary NULL tests in Linux 4.4.0-rc5, PostgreSQL 8.3.9, and Apache httpd 2.2.18.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—program analysis; H.3.4 [*Information Storage and Retrieval*]: Systems and Software

General Terms Language, Measurements, Performance

ASPLOS '17, April 8–12, 2017, Xi'an, China..

http://dx.doi.org/10.1145/3037697.3037744

Keywords Static analysis, graph processing, disk-based systems

1. Introduction

Static analysis has been used to find bugs in systems software for more than a decade now [9, 14, 15, 18, 20, 25, 28, 29, 31, 35, 56, 59, 71, 88, 93]. Based on a set of systems rules, a static checker builds patterns and inspects code statements to perform "pattern matching". If a code region matches one of the patterns, a violation is found and reported. Static checkers have many advantages over recent, more advanced bug detectors based on SAT solvers or symbolic execution [18]: they are simple, easy to implement, and scalable. Furthermore, they produce deterministic and easy-to-understand bug reports compared to, for example, a symbolic execution technique, which often produces non-deterministic bug reports that are difficult to reason about [27].

1.1 Problems

Unfortunately, the existing static checkers use many heuristics when searching for patterns, resulting in missing bugs and/or reporting false warnings. For example, Chou et al. [24] and Palix et al. [59] developed nine checkers to find bugs in the Linux kernel. Most of these checkers generate both false negatives and false positives. For instance, their **Null** checker tries to identify NULL pointer dereference bugs by inspecting only the functions that directly return NULL. However, a NULL value can be generated from the middle of a function and propagated a long way before it is dereferenced at a statement. Such NULL value propagation will be missed entirely by the **Null** checker.

As another example, the **Pnull** checker developed recently by Brown et al. [18] checks whether a pointer dereference such as $a = b{->}\ f$ is *post-dominated* by a NULL test on the pointer such as `if`(b). The heuristic here is that if the developer checks whether b can be NULL after dereferencing b, the dereferencing can potentially be on a NULL pointer. However, in many cases, the dereferencing occurs in one of the many control flow paths and in this particular path the pointer can never be NULL. The developer adds the NULL

Checker	Target Problems	Limitations	Potential Improvement with Interprocedural Analyses
Block	Deadlocks	Focus on "direct" invocations of the blocking functions (Negative)	Use a **pointer/alias analysis** to identify indirect invocations via function pointers of the blocking functions
Null	NULL pointer derefs	Inspect a closure of functions that return NULL explicitly (Negative)	Use a **dataflow analysis** to identify functions where NULL can be propagated to their return variables
Range	Use user data as array index without checks	Only check indices directly from user data (Negative)	Use a **dataflow analysis** to identify indices coming transitively from user data as well
Lock/Intr	Double acquired locks and disabled interrupts not appropriately restored	Identify lock/interrupt objects by var names (Negative)	Use a **pointer/alias analysis** to understand aliasing relationships among lock objects in different lock sites
Free	Use of a freed obj	Identify freed/used objects by var names (Negative)	Use a **pointer/alias analysis** to check if there is aliasing between objects freed and used afterwards
Size	Inconsistent sizes between an allocated obj and the type of the RHS var	Only check alloc sites (Negative)	Use a **pointer/alias analysis** to identify other vars that *point to* the same object with an inconsistent type
Pnull	NULL pointer derefs	Report all derefs post-dominated by NULL tests (Positive)	Use a **dataflow analysis** to filter out cases where the involved pointers *must not* be NULL

Table 1. A subset of checkers used by [18] and [59] to find bugs in the Linux kernel, their target problems, their limitations, the potential ways to improve them using a sophisticated interprocedural analysis; the first six have been used by Chou et al. [24] and Palix et al. [59] to study Linux bugs; the last one was described in a recent paper by Brown et al. [18] to find potential NULL pointer dereferences; positive/negative indicates whether the limitation can result in false positives/negatives.

test simply because the NULL value may flow to the test point from a different control branch.

Our key observation in reducing the number of false positives and negatives reported by these checkers is to leverage *interprocedural analysis*. Among the aforementioned nine checkers, six that check flow properties can be easily improved (*e.g.*, producing fewer false positives and false negatives) using an interprocedural analysis, as shown in Figure 1.

While using interprocedural analyses to improve bug detection appears to be obvious, there seems to be a large gap between the state of the art and the state of the practice. On the one hand, the past decade has seen a large number of sophisticated and powerful analyses developed by program analysis researchers. On the other hand, none of these techniques are widely used to find bugs in systems software.

We believe that the reason is two-fold. First, an interprocedural analysis is often not scalable enough to analyze large codebases such as the Linux kernel. In order for such an analysis to be useful, it often needs to be *context-sensitive*, that is, distinct solutions need to be produced and maintained for different calling contexts (*i.e.*, a chain of call sites representing a runtime call stack). However, the number of calling contexts grows *exponentially* with the size of the program and even a moderate-sized program can have as large as 10^{14} distinct contexts [90], making the analysis both compute- and memory-intensive. Furthermore, most interprocedural analyses are difficult to parallelize, because they frequently involve decision making based on information discovered *dynamically*. Thus, most of the existing implementations of such analyses are entirely sequential.

Second, the sheer implementation complexity scares practitioners away. Much of this complexity stems from optimizing the analysis rather than implementing the base algorithm. For example, in a widely-used Java pointer analysis [78], more than three quarters of the code performs approximations to make sure some results can be returned before a user-given time budget runs out. The base algorithm implementation takes a much smaller portion. This level of tuning complexity simply does not align with the "simplest-working-solution" [46] philosophy of systems builders.

1.2 Insight

Our idea is inspired by the way a graph system enables scalable processing of large graphs. Graph system support pioneered by Pregel [53] provides a "one-stone-two-birds" solution, in which the optimization for scalability is mainly achieved by the (distributed or disk-based) system itself, requiring the developers to only write simple vertex programs using the interfaces provided by the system.

In this paper, we demonstrate a similar "one-stone-two-birds" solution for interprocedural program analysis. Our key observation in this work is that many interprocedural analyses can be formulated as a *graph reachability* problem [64, 65, 72, 78, 96]. Pointer/alias analysis and dataflow analysis are two typical examples. In a pointer/alias analysis, if an object (*e.g.*, created by a malloc) can directly or transitively reach a variable on a directed graph representation of the program, the variable may point to the object. In a dataflow analysis that tracks NULL pointers, similarly, a transitive flow from a NULL value to a variable would make NULL propagate to the variable. Therefore, we turn the programs into graphs and treat the analyses as graph traversal. This approach opens up opportunities to leverage parallel graph processing systems to analyze large programs efficiently.

1.3 Existing Systems

Several graph systems are available today. These systems are either distributed (*e.g.*, GraphLab [52], PowerGraph [32], or GraphX [33]) or single-machine-based (*e.g.*, GraphChi [44], XStream [69], or GridGraph [102]). Since program analysis is intended to assist developers to find bugs in their daily development tasks, their machines are the environments in which we would like our system to run, so that developers can check their code on a regular basis without needing to access a cluster. Hence, disk-based systems naturally become our choice.

We initially planned to use an existing system to analyze program graphs. We soon realized that a ground-up redesign (*i.e.*, from the programming model to the engine) is needed to build a system for analyzing large programs. The main reason is that the graph workload for interprocedural analyses is significantly different from a regular graph algorithm (such as PageRank) that iteratively performs computations on vertex values on a static graph. An interprocedural analysis, on the contrary, focuses on computing reachability by repeatedly adding *transitive edges*, rather than on updating vertex values. For instance, a pointer analysis needs to add an edge from each allocation vertex to each variable vertex that is transitively reachable from the allocation.

More specifically, many interprocedural analyses are essentially *dynamic reachability* problems in the sense that the addition of a new edge is guided by a constraint on the labels of the existing edges. In a static analysis, the label of an edge often represents the semantics of the edge (*e.g.*, an assignment or a dereference). For two edges $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$, a transitive edge from a to c is added only if the concatenation of l_1 and l_2 forms a string of a (context-free) grammar.

This constraint-guided reachability problem, in general, requires *dynamic transitive closure* (DTC) computation [39, 67, 95], which has a wide range of applications in program analysis and other domains. The DTC computation dictates two important abilities of the graph system. First, at each vertex, all of its incoming and outgoing edges need to be visible to perform label matching and edge addition. In the above example, when b is processed, both $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$ need to be accessed to add the edge from a to c. This requirement immediately excludes edge-centric systems such as XStream [69] from our consideration, because these systems stream in edges in a random order and, thus, this pair of edges may not be simultaneously available.

Second, the system needs to support a large number of edges added dynamically. The added edges can be even more than the original edges in the graph. While vertex-centric systems such as GraphChi [44] support dynamic edge addition, this support is very limited. In the presence of a large number of added edges, it is critical that the system is able to (1) quickly check edge duplicates and (2) appropriately repartition the graph. Unfortunately, GraphChi supports neither of these features.

1.4 Our Contributions

This paper presents Graspan, the first *single machine, disk-based* parallel graph processing system tailored for interprocedural static analyses. Given a program graph and a grammar specification of an analysis, Graspan offers two major performance and scalability benefits: (1) the core computation of the analysis is automatically parallelized and (2) out-of-core support is exploited if the graph is too big to fit in memory. At the heart of Graspan is a parallel *edge-pair* (EP) centric computation model that, in each iteration, loads two partitions of edges into memory and "joins" their edge lists to produce a new edge list. Whenever the size of a partition exceeds a threshold value, its edges are repartitioned. Graspan supports both in-memory (for small programs) and out-of-core (for large programs) computation. Joining of two edge lists is fully parallelized, allowing multiple transitive edges to be simultaneously added.

Graspan provides an intuitive programming model, in which the developer only needs to generate the graph and define the grammar that guides the edge addition, a task orders-of-magnitude easier than coming up with a well-tuned implementation of the analysis that would give trouble to skillful researchers for months.

Recent work shows the effectiveness of backing static analyses with Datalog [17, 90] or Database [89]. While leveraging Datalog makes analysis implementations easier, the existing Datalog engines are designed in *generic* ways, *i.e.*, not considering the characteristics of the program analysis workload. Furthermore, there does not exist any out-of-core Datalog engine that can process very large graphs on a single machine. For example, the Linux kernel program graph has more than 1B edges. The fastest shared memory Datalog engine SociaLite [45] quickly ran out of memory while Graspan processed it in several hours (*cf.* §5.4). While distributed Datalog engines such as Myria [86] and BigDatalog [74] are available, it is unrealistic to require developers to frequently access a cluster in their daily development.

We have implemented fully context-sensitive pointer/alias and dataflow analysis on Graspan. Context-sensitivity is achieved by making aggressive inlining [73]. That is, we clone the body of a function for every single context leading to the function. This approach is feasible only because the out-of-core support in Graspan frees us from worrying about additional memory usage incurred by inlining. We treat the functions in recursions *context insensitively* by merging the functions in each strongly connected component on the call graph into one function without cloning function bodies.

Results We have implemented Graspan in both Java and C++; these implementations are publicly available at https://github.com/Graspan. Graspan can be readily used as a "backend" analysis engine to enhance the existing static

checkers such as BugFinder, PMD, or Coverity. We have performed a thorough evaluation of Graspan on three systems programs including the Linux kernel, the PostgreSQL database, and the Apache httpd server. Our experiments show very promising results: (1) the two Graspan-based analyses scale easily to these systems, which have many millions of function inlines, with several hours processing time, while their traditional implementations crashed in the early stage; (2) in terms of LoC, the Graspan-based implementations of these analyses are an order-of-magnitude simpler than their traditional implementations; (3) using the results of these interprocedural analyses, the static checkers in [59] have uncovered a total of **85** potential bugs and **1308** unnecessary NULL tests.

2. Background

While there are many types of interprocedural analyses, this paper focuses on a pointer/alias analysis and a dataflow analysis, both of which are enablers for all other static analyses. This section discusses necessary background information on how pointer/alias analysis is formulated as graph reachability problems. Following Rep et al.'s interprocedural, finite, distributive, subset (IFDS) framework [65], we have also formulated a fully context-sensitive dataflow analysis as a grammar-guided reachability problem. However, due to space limitations, the discussion of this formulation is omitted.

2.1 Graph Reachability

Pioneered by Reps et al. [65, 72], there is a large body of work on graph reachability based program analyses [16, 42, 61, 80, 91, 92, 97, 99]. The reachability computation is often guided by a context-free grammar due to the *balanced parentheses* property in these analyses. At a high level, let us suppose each edge is labeled either an open parenthesis '(' or a close parenthesis ')'. A vertex is reachable from another vertex if and only if there exists a path between them, the string of labels on which has balanced '(' and ')'.

The parentheses '(' and ')' have different semantics for different analyses. For example, for a C pointer analysis, '(' represents an address-of operation & and ')' represents a dereference *. A pointer variable can point to an object if there is an assignment path between them that has balanced & and *. For instance, a string "&&**" has balanced parentheses while "&**&" does not. This balanced parentheses property can often be captured by a context-free grammar.

2.2 Pointer Analysis

A pointer analysis computes, for each pointer variable, a set of heap objects (represented by allocation sites) that can flow to the variable. This set of objects is referred to as the variable's *points-to* set. Alias information can be derived from this analysis — if the points-to sets of two variables have a non-empty intersection, they may alias.

Our graph formulation of pointer analysis is adapted from a previous formulation in [101]. This section briefly describes this formulation. The analysis we implement is *flow-insensitive* in the sense that we do not consider control flow in the program. Flow sensitivity can be easily added, but it does not contribute much to the analysis precision [37]. A program consists of a set of pointer assignments. Assignments can execute in any order, any number of times.

Pointer Analysis as Graph Reachability For simplicity of presentation, the discussion here focuses on four kinds of three-address statements (which are statements that have at most three operands):

$a = b$	Value assignment	$a = *b$	Load
$*b = a$	Store	$a = \&b$	Address-of

Complicated statements are often broken down into these three-address statements in the compilation process by introducing temporary variables. Our analysis does not distinguish fields in a struct. That is, an expression a-> f is handled in the same way as $*a$, with offset f being ignored. As reported in [101], ignoring offsets only has little influence on the analysis precision, because most fields are of primitive types.

For each function, an *expression graph* – whose vertices represent C expressions and edges represent value flow between expressions — is generated; graphs for different functions are eventually connected to form a whole-program expression graph. Each vertex on the graph represents an expression, and each edge is of three kinds:

- *Dereference edge (D)*: for each dereference $*a$, there is a D-edge from a to $*a$; there is also an edge from an address-of expression $\&a$ to a because a is a dereference of $\&a$.
- *Assignment edge (A)*: for each assignment $a = b$, there is an A-edge from b to a; a and b can be arbitrary expressions.
- *Alloc edge (M)*: for each assignment $a = malloc()$, there is an M-edge from a special Alloc vertex to a.

Figure 1 shows a simple program and its expression graph. Each edge has a label, indicating its type. Solid and dashed edges are original edges in the graph and they are labeled M, A, or D, respectively. Dotted edges are transitive edges[1] added by Graspan into the graph, as discussed shortly.

Context-free Grammar The pointer information computation is guided by the following grammar:

Object flow: $OF ::= M\ VF$
Value flow: $VF ::= (A\ MA?)^*$
Memory alias: $MA ::= \overline{D}\ VA\ D$
Value alias: $VA ::= \overline{VF}\ MA?\ VF$

This grammar has four non-terminals *OF*, *VF*, *MA*, and *VA*. For a non-terminal T, a path in the graph is called a

[1] We use term "transitive edges" to refer to the edges dynamically added to represent non-terminals rather than the transitivity of a relation.

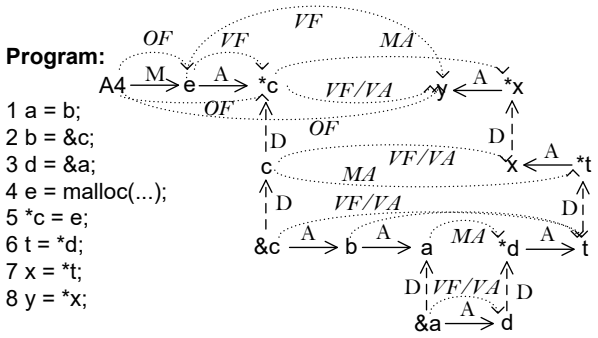

Figure 1. A program and its expression graph: solid, horizontal edges represent assignments (A- and M- edges); dashed, vertical edges represent dereferences (D-edge); dotted, horizontal edges represent transitive edges labeled non-terminals. $A4$ indicates the allocation site at Line 4.

T-path if the sequence of the edge labels on the path is a string that can be reduced to T. In order for a variable v to point to an object o (*i.e.*, a malloc), there must exist an *OF* path in the expression graph from o to v. The definition of OF is straightforward: it must start with an alloc (M) edge, followed by a VF path that propagates the object address to a variable. A VF path is either a sequence of simple assignment (A) edges or a mix of assignments edges and *MA* (memory alias) paths.

There are two kinds of aliasing relationships in C: memory aliasing (MA) and value aliasing (VA). Two lvalue expressions are memory aliases if they may denote the same memory location while they are value aliases if they may evaluate to the same value.

An *MA* path is represented by $\overline{D}\ \ VA\ \ D$. Each edge has an inverse edge with a "bar" label. For example, for each edge $a \xrightarrow{D} b$, the edge $b \xrightarrow{\overline{D}} a$ exists automatically. $\overline{D}$ represents the inverse of a dereference and is essentially equivalent to an address-of. $\overline{D}\ \ VA\ \ D$ represents the fact that if (1) we take the address of a variable a and writes it into a variable b, (2) b is a *value alias of* another variable c, and (3) we perform dereferencing on c, the result is the same as the value in a.

A *VA* path is represented by $\overline{VF}\ \ MA\ \ VF$. This has the meaning that if (1) two variables a and b are memory aliases, and (2) the values of a and b are propagated to two other variables c and d, respectively, through two VF paths, c and d contain the same pointer value. In other words, the path – $c\ \overline{VF}\ a\ MA\ b\ VF\ d$ – induces $c\ VA\ d$.

Note that MA, VA, and VF mutually refer each other. This definition captures the recursive nature of a flow or alias path. In this grammar, $\overline{D}$ and D are the open and close parentheses that need to be balanced.

Example In Figure 1, e points to $A4$, since the M edge between them forms an *OF* path. There is a VF path from $\&a$ to d, which is also a VA path (since VA includes VF). The VA path enables an MA path from a to $*d$ due to the balanced parentheses D and $\overline{D}$. This path then induces two additional VF/VA paths from b to t and from $\&c$ to t, which, in turn, contribute to the forming of the VF/VA path from c to x, making $*c$ and $*x$ memory aliases. Hence, there exists a VF path from e to y, which, together with the M edge at the beginning, forms an OF path from $A4$ to y. This path indicates that y points to $A4$. The dotted edges in Figure 1 show these transitive edges.

Traditional Solution The traditional way to implement this analysis is to maintain a worklist, each element of which is a pair of a newly discovered vertex and a stack simulating a pushdown automaton. The implementation loops over the worklist, iteratively retrieving vertices and processing their edges. The traditional implementation does not add any physical edges into the graph (due to the fear of memory blowup), but instead, it tracks path information using pushdown automata. When a CFL-reachable vertex is detected, the vertex is pushed into the worklist together with the sequence of the labels on the path leading to the vertex. When the vertex is popped off of the list, the information regarding the reachability from the source to the vertex is discarded.

This traditional approach has at least two significant drawbacks. First, it does not scale well when the analysis becomes more sophisticated or the program to be analyzed becomes larger. For example, when the analysis is made *context-sensitive*, the grammar needs to be augmented with the parentheses representing method entries/exists; the checking of the balanced property for these parentheses also needs to performed. Since the number of distinct calling contexts can be very large for real-world programs, naïvely traversing all paths is guaranteed to be not scalable in practice. As a result, various abstractions and tradeoffs [41, 76–78] have been employed, attempting to improve scalability at the cost of precision as well as implementation straightforwardness.

Second, the worklist-based model is notoriously difficult to parallelize, making it hard to fully utilize modern computing resources. Even if multiple traversals can be launched simultaneously, since none of these traversals add transitive edges into the program graph as they are being detected, every traversal performs path discovery completely independently, resulting in a great deal of wasted efforts.

A "Big Data" Perspective Our key insight here is that adding *physical* transitive edges into the program graph makes it possible to devise a Big Data solution to this static analysis problem for two reasons. First, representing transitive edges *explicitly* rather than *implicitly* leads to addition of a great number of edges (*e.g.*, even larger than the number of edges in the original graph). This gives us a large (evolving) dataset to process. Second, the computation only needs to match the labels of consecutive edges with the productions in the grammar and is thus simple enough to be "systemized". Of course, dynamically adding many edges can make the computation quickly exhaust the main memory. However, this should not be a concern, as there are already

many systems [34, 44, 52, 68, 84, 87] built to process very large graphs (*e.g.*, the webgraph for the whole Internet).

3. Graspan's Programming Model

In this section, we describe Graspan's programming model, *i.e.*, the tasks that need to be done by the programmer to use Graspan. There are two main tasks. The first task is to modify a compiler frontend to generate the graph. The second task is to use the Graspan API to specify the grammar. Next, we will elaborate on these two tasks. We will then finish the section by discussing the applicability of Graspan's programming model to interprocedural analyses.

Generating Graph For Graspan to perform an interprocedural analysis, the user first needs to generate the *Graspan graph*, which is a specialized program graph tailored for the analysis, by modifying a compiler frontend. Note that since this task is relatively simple, the developer can generate the Graspan graph in a mechanical way without even thinking about performance and scalability. In this subsection, we briefly discuss how we generate the Graspan graph in the context of the pointer/alias analysis. We finish by generalizing graph generation for other interprocedural analyses.

For the pointer/alias analysis, we generate the Graspan graph by making two modifications to the program expression graph described in §2. These modifications include (1) inclusion of inverse edges and (2) context sensitivity achieved through inlining. For the former, we model inverse edges explicitly. That is, for each edge from a to b labeled X, we create and add to the graph an edge from b to a labeled $\overline{X}$.

For the latter, we perform a bottom-up (*i.e.*, reverse-topological) traversal of the call graph of the program to inline functions. For each function, we make a *clone* of its entire expression graph for each call site that invokes the function. Formal and actual parameters are connected explicitly with edges. The cloning of a graph not only copies the edges and vertices in one function; it does so for *all* edges and vertices in its (direct and transitive) callees.

For recursive functions, we follow the standard treatment [90] – strongly connected components (SCC) are computed and then functions in each SCC are collapsed into one single function, and treated context insensitively. Clearly, the size of the graph grows exponentially as we make clones and the generated graph is often large. However, the out-of-core support in Graspan guarantees that Graspan can analyze even such large graphs effectively. For each copy of a vertex, we generate a unique ID in a way so that we can easily locate the variable its corresponds to and its containing function from the ID. In the Graspan graph, edges carry data (*i.e.*, their labels) but vertices do not. Finally, the graph is dumped to disk in the form of an edge list.

In general, the approach of aggressive inlining provides *complete information* that an analysis intends to uncover. Among all the existing analysis implementations, only Whaley et al. [90] could handle such aggressive inlining but they only clone variables (*not* objects) and have to use a binary decision diagram (BDD) to merge results. In addition, no evidence was shown that their analysis could process the Linux kernel. On the contrary, Graspan processes the exploded kernel graph in a few hours on a single machine.

Although this subsection focuses on the generation of pointer analysis graphs, graphs for other analyses can be generated in a similar manner. Here we briefly summarize the steps. First, vertices and edges need to be defined based on a grammar; this step is analysis-specific. Second, if inverse edges are needed in the grammar, they need to be explicitly added. Finally, context sensitivity can be generally achieved by function inlining. The developer can easily control the degree of context sensitivity by using different inlining criteria. For example, we perform *full context sensitivity* and thus our inlining goes from the bottom functions all the way up the top functions of the call graph. But if one wishes to perform only *one-level* context sensitivity, each function only needs to be inlined once.

Specifying Grammar Once the program graph is generated, the user needs to specify a grammar that guides the addition of transitive edges at run time. Unlike any traditional implementation of the analysis, Graspan adds transitive edges (*e.g.*, dotted edges in Figure 1) to the graph in a parallel manner. Specifically, for each production in the grammar, if Graspan finds a path whose edge labels match the RHS terms of the production, a transitive edge is added covering the path and labeled with the LHS of the production.

Since Graspan uses the edge-pair-centric model, it focuses on a pair of edges at a time, which requires each production in the grammar to have no more than two terms on its RHS. In other words, the length of a path Graspan checks at a time must be ≤ 2.

For example, the above mentioned pointer analysis grammar cannot be directly used, because the RHSes of *VF*, *MA*, and *VA* all have more than two terms. This means that to add a new *VF* edge, we may need to check more than two consecutive edges, which does not fit into Graspan's EP-centric model. Fortunately, every context free grammar can be *normalized* into an equivalent grammar with at most two terms on its RHS [65], similar to the Chomsky normal form. After normalization, our pointer analysis grammar becomes:

Object flow:	OF	::=	$M \;\; VF$
Temp:	T_1	::=	$A \mid MA$
Value flow:	VF	::=	$T_1 \mid VF \;\; T_1 \mid \epsilon$
Mem alias:	MA	::=	$T_2 \;\; D$
Temp:	T_2	::=	$\overline{D} \;\; VA$
Value alias:	VA	::=	$T_3 \;\; VF$
Temp:	T_3	::=	$\overline{VF} \;\; MA \mid \overline{VF}$

At the center of Graspan's programming model is an API `addConstraint(Label` *lhs*`, Label` *rhs1*`, Label` *rhs2*`)`, which can be used by the developer to register each production in the grammar. *lhs* represents the LHS non-terminal

while *rhs1* and *rhs2* represent the two RHS terms. If the RHS has only one term, *rhs2* should be NULL.

Graspan Applicability How many interprocedural analyses can be powered by Graspan? First, we note that pointer analysis and dataflow analysis are already representatives of a large number of analysis algorithms that can be formulated as a grammar-guided graph reachability problem. Second, work has been done to establish the convertibility from other types of analysis formulation (*e.g.*, set-constraint [42] and pushdown systems [10, 11, 11]) to context-free language reachability. Analyses under these other formulations can all be parallelized and made scalable by Graspan.

Note that Graspan currently does not support analyses that require constraint solving, such as path-sensitive analysis and symbolic execution. In our future work, we plan to add support for constraint-based analyses by encoding constraints into edge values. Two edges match if a satisfiable solution can be found for the conjunction of the constraints they carry.

4. Graspan Design and Implementation

We implemented Graspan first in Java. Due to performance issues in the JVM (most of which were caused by the GC when copying arrays), we re-implemented Graspan in C++. The Java and C++ versions have an approximate 6K and 4K lines of code, respectively. Graspan can analyze programs written in any languages.

4.1 Preprocessing

Preprocessing partitions the Graspan graph generated for an analysis. The graph is in the edge-list format on disk. Similar to graph sharding in GraphChi [44], partitioning in Graspan is done by first dividing vertices into *logical intervals*. However, unlike GraphChi that groups edges based on their target vertices, one interval in Graspan defines a partition that contains edges whose *source vertices* fall into the interval. Edges are sorted on their source vertex IDs and those that have the same source are stored consecutively and *ordered on their target vertex IDs*. The fact that the outgoing edges for each vertex are sorted enables quick edge addition, as we will discuss shortly. Figure 2(a) shows a simple directed graph. Suppose Graspan splits its vertices into three intervals 0–2, 3–4, and 5–6; Figure 2(b) shows the partition layout.

When a new edge is found during processing, it is always added to the partition to which the source of the edge belongs. Graspan loads two partitions at a time and joins their edge-lists (§4.2), a process we refer to as a *superstep*. Given that only two partitions reside in memory at a given time, the size and hence the total number of partitions are determined automatically by the amount of memory available to Graspan.

Preprocessing also produces three pieces of meta-information: a *degree file* for each partition, which records the (incoming and outgoing) degrees of its vertices, a global *vertex-interval table* (VIT), which specifies vertex intervals, and a *destination distribution map* (DDM) for each partition p that maps, for each other partition q, the percentage of the edges in p that go into q. The DDM is essentially a matrix, each cell containing a percentage.

Graspan uses the degree file to calculate the size of the array to be created to load a partition. Without the degree information, a variable-size data structure (*e.g.*, ArrayList) has to be used, which would incur array resizing and data copying operations. The VIT records the lower and upper-bounds for each interval (*e.g.*, (0, [0, 10000]), (1, [10001, 23451]), *etc.*). Graspan maintains the table because the intervals will be redefined upon repartitioning. The DDM measures the "matching" degree between two partitions and will be used by the Graspan scheduler to determine which two to load.

4.2 Edge-Pair Centric Computation

Graspan supports in-memory and out-of-core computations. For small graphs that can be held in memory, our preprocessing only generates two partitions, both of which are resident in memory. For large graphs with more than two partitions, Graspan uses a scheduling algorithm (discussed shortly) to load two partitions in each superstep, joins their edge lists, updates their edges, and performs repartitioning if necessary. Each superstep itself performs a fixed point computation — newly added edges may give rise to further edges.

The computation is finished when no new edges can be added. The updated edge lists may or may not be written back to disk depending on the next two partitions selected by the scheduler. This iterative process is repeated until a global fixed point is reached, that is, no new edges can be added for any pair of partitions. Since the VIT and the DDM are reasonably small in size, they are kept in memory throughout the processing.

In-Memory Edge Representation The edge list of a vertex v is represented as two arrays of (vertex, label) pairs, as shown in Figure 2(c). The first array (O_v) contains "old" edges that have been inspected before and the second (D_v) contains edges newly added in the current iteration. The goal is to avoid repeatedly matching edge pairs (discussed shortly).

Parallel Edge Addition Algorithm 1 shows a BSP-like algorithm for the parallel EP-centric computation. With two partitions p_1 and p_2 loaded, we first merge them into one single partition with combined edge lists (Line 1 – 2). Initially, for each vertex v, its two arrays O_v and D_v are set to empty list and the original edge list of v, respectively (Line 4 and Line 5). The loop between Line 6 and Line 24 creates a separate thread to process each vertex v and its edge list, computing transitive edges using a fixed-point computation with two main components.

The first component (Line 7 – 14) attempts to match each "old" edge in O_v that goes to vertex u with each "new" edge of u in D_u. The second component (Line 15 – 20) matches each "new" edge in D_v with both "old" and "new" edges in O_u and D_u of vertex u. The idea is that we do not need to match an "old" edge of v with an "old" edge of u, because

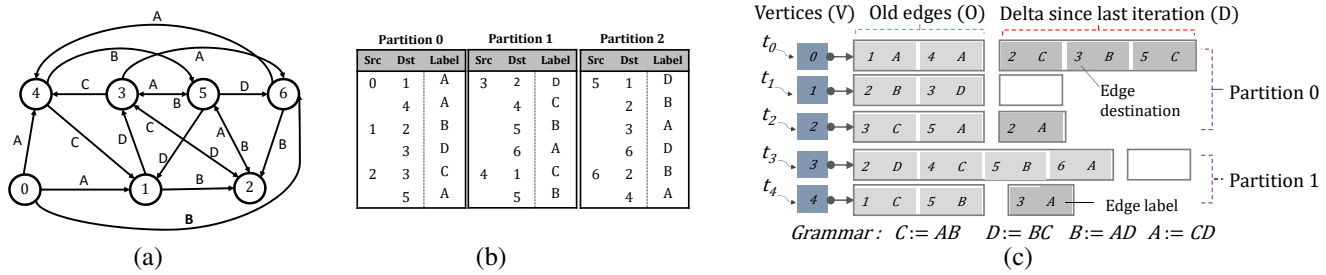

Figure 2. (a) An example graph, (b) its partitions, and (c) the in-memory representation of edge lists.

Algorithm 1: The parallel EP-centric computation.

Input: Partition p_1, Partition p_2

1 Combine the vertices of p_1 and p_2 into V
2 Combine the edge lists of p_1 and p_2 into E
3 **for** *each edge list* $v : (e_1, e_2, \ldots, e_n) \in E$ **do in parallel**
4 Set O_v to ()
5 Set D_v to $(e_1, e_2, \ldots, e_n)$
6 **for** *each vertex* $v : (O_v, D_v)$ *whose* D_v *is NOT empty* **do in parallel**
7 Array $mergeResult \leftarrow ()$
8 Let V_1 be the intersection of the target vertices of O_v and V
9 /*Merge O_v with only D_v of other vertices*/
10 List $listsToMerge \leftarrow \{O_v\}$
11 **foreach** *Vertex* $v' \in V_1$ **do**
12 Add $D_{v'}$ into $listsToMerge$
13 /*Merge the sorted input lists into a new sorted list*/
14 $mergeResult \leftarrow$ MATCHANDMERGESORTEDARRAYS($listsToMerge$)
15 /*Merge D_v with $O_v \cup D_v$ of other vertices*/
16 Let V_2 be the intersection of the target vertices of D_v and V
17 $listsToMerge \leftarrow \{D_v, mergeResult\}$
18 **foreach** *Vertex* $v' \in V_2$ **do**
19 Add $O_{v'}$ and $D_{v'}$ into $listsToMerge$
20 $mergeResult \leftarrow$ MATCHANDMERGESORTEDARRAYS($listsToMerge$)
21 /*Update O_v and D_v*/
22 $listsToMerge \leftarrow \{O_v, D_v\}$
23 $O_v \leftarrow$ MERGESORTEDARRAYS($listsToMerge$)
24 $D_v \leftarrow mergeResult - O_v$

this work has been done in a previous iteration. O_v and D_v are updated at the end of each iteration.

An important question is how to perform edge matching. A straightforward approach is that, for each edge $v \xrightarrow{L_1} u$, we inspect each of u's outgoing edges $u \xrightarrow{L_2} x$, and add an edge $v \xrightarrow{K} x$ if a production $K ::= L_1\ L_2$ exists. However, this simple approach suffers from significant practical limitations. First, before the edge is added into v's list, we need to scan v's outgoing edges one more time to check if the same edge already exists. Checking and avoiding duplicates is very important – duplicates may cause the analysis either not to terminate or to suffer from significant redundancies in both time and space.

Doing a linear scan of the existing edges is expensive – it has an O($|E|^2$) complexity to add edges for each vertex, where $|E|$ is the total number of edges loaded. An alternative is to implement an "offline" checking mechanism that removes duplicates when writing updated partitions to disk. While this approach eliminates the cost of online checks, it may prevent the computation from terminating — if the same edge is repeatedly added, missing the online check would make the loop at Line 6 keep seeing new edges and run indefinitely.

Our algorithm performs *quick edge addition* and *online duplicate checks*. Our key insight is that edge addition can be done *in batch* much more efficiently than *individually*. To illustrate, consider Figure 2(a) where vertex 0 initially has two outgoing edges $0 \rightarrow 1$ and $0 \rightarrow 4$. Adding new edges for vertex 0 is essentially the same as *merging* the (outgoing) edges of vertex 1 and 4 into vertex 0's edge list and then filtering out those that have mismatched labels.

In Algorithm 1, to add edges for vertex v, we first compute set V_1 by intersecting the set of target vertices of the edges in O_v and the set V of all vertices in the loaded partitions (Line 8). V_1 thus contains the vertices whose edge lists need to be merged with v's edge list. If an out-neighbor of v is not in V, we skip it in the current superstep — this vertex will be processed later in a future superstep in which its partition is loaded together with v's partition.

Next, we add O_v into a list $listsToMerge$ together with D_u of each vertex u in V_1 (Line 10 – 12), and merge these lists into a new sorted list (Line 14). Since all input lists are already sorted, function MATCHANDMERGESORTEDARRAYS can be efficiently implemented by repeatedly finding the minimum (using an O($log|V|$) min-heap algorithm [13]) among the elements in a slice of the input lists and copying it into the output array. This whole algorithm has an O($|E|log|V|$) complexity, which is more efficient, both theoretically and empirically, than scanning edges individually (O($|E|^2$)) because $|V|$ is much smaller than $|E|$. Furthermore, edge duplicate checking can be automatically done during the merging — if multiple elements have the same minimum value, only one

is copied into the output array. Label matching is performed before copying — an edge is not copied into the output if it has an inconsistent label.

Line 15 – 20 perform the same logic by computing a new set of vertices V_2, and merging D_v and all the edges (*i.e.*, $O_u \cup D_u$) of each vertex $u \in V_2$. At Line 20, all the new edges to be added to vertex v are in $mergeResult$. Finally, to prepare for the next iteration, O_v and D_v are merged (Line 23) to form the new O_v; D_v is then updated to contain the newly added edges (excluding those that already exist in O_v).

Example Figure 2(c) shows the in-memory edge lists at the end of the first iteration of the loop at Line 6 in Algorithm 1. In the next iteration, thread t_0 would merge O_0 with D_1 and D_4, and D_0 with $O_2 \cup D_2$, $O_3 \cup D_3$, and $O_5 \cup D_5$. O_0 and O_1 (and O_4) do not need to be merged again as this has been done before.

Another advantage of this algorithm is that it runs completely in parallel without needing any synchronization. While the edge list of a vertex may be *read* by different threads, edge addition can only be done by one single thread, that is, the one that processes the vertex.

4.3 Postprocessing

When a superstep is done, the updated edge lists need to be written back to their partition files. In addition, the degree file is updated with the new vertex degree information. The (in-memory) DDM needs to be updated with the new edge distribution information.

Repartitioning If the size of a partition exceeds a threshold (*e.g.*, a parameter), repartitioning occurs. It is easy for Graspan to repartition an oversized partition since the edge lists are sorted. Graspan breaks the original vertex interval into two small intervals, and edges are automatically restructured. The goal is to have the two small vertex intervals to have similar numbers of edges, so that the resulting partitions have similar sizes. The VIT needs to be updated with the new interval information. Repartitioning can also be triggered in the middle of a superstep if too many edges are added in the superstep and the size of the loaded partitions is close to the memory size.

Scheduling When a new superstep starts, two new partitions will be selected by the scheduler to join. Since a partition on which the computation was done in the previous superstep may be chosen again, Graspan delays the writing of a partition back to disk until the new partitions are chosen by the scheduler. If a chosen partition is already in memory, significant amounts of disk I/O can be saved.

We have developed a novel scheduling algorithm that has two objectives: (1) maximize the number of edge pairs that can potentially match and (2) favor the reuse of in-memory partitions. For (1), the scheduler consults the DDM. While the percentage information recorded in the DDM measures the matching opportunities between two partitions, it is an overall measurement that does not reflect the changes. Hence, we add another field to each cell of the DDM that records, for a pair of partitions p and q, the change in the percentage of the edges going from p into q since the last time p and q are both loaded. If p and q have never been loaded together, the change is the same as the full percentage.

Using $\delta(p, q)$ to denote this change, our scheduler selects a pair of partitions that have the largest $\delta(p, q) + \delta(q, p)$ score. If multiple pairs of partitions have similar scores (*e.g.*, in a user-defined range), Graspan picks one that involves an in-memory partition. These delta fields in the DDM also determine the termination of the computation — for p and q whose $\delta(p, q) + \delta(q, p)$ is zero, no computation needs to be done on them. Graspan terminates when the delta field in every single cell of the DDM becomes 0.

Program	Ver	#LoC	#Inlines
Linux kernel	4.4.0-rc5	16M	31.7M
PostgreSQL	8.3.9	700K	290820
Apache httpd	2.2.18	300K	58269

Table 2. Programs analyzed, their versions, numbers of lines of code, and numbers of function inlines.

Reporting Results Graspan provides an API for the user to iterate over edges with a certain label. For example, for the pointer analysis, edges with the OF label indicate a points-to solution, while edges with the MA and VA label represent aliasing variables. Graspan also provides translation APIs that can be used to map vertices and edges back to variables and statements in the program.

5. Evaluation

We built our frontend based on LLVM Clang. Our graph generators for the pointer/alias and dataflow analysis have 1.2K and 800 lines of C++ code, respectively. To use Graspan, the pointer/alias analysis has a grammar with 12 productions (*i.e.*, invoking the API function `addConstraint` 12 times) while the dataflow analysis has 2 productions. We first performed the pointer analysis. The dataflow analysis was designed specifically to track NULL value propagation. It was built based on the pointer analysis because it needs to query pointer analysis results when analyzing heap loads and stores.

We used the call graph generated by LLVM to perform inlining. Three large system programs were selected: the Linux kernel, the PostgreSQL database, and the httpd server. Their detailed statistics are reported in Table 2. Linux kernels are not directly compilable with LLVM. Thanks to the LLVM-Linux project [6] that provides kernel patches for LLVM compilation, we were able to build the kernel version 4.4.0-rc5 (the latest version supported by LLVMLinux). Although we spent much effort trying to link as many modules as possible, some modules might still not be appropriately linked at the time of evaluation.

For the other two systems, we picked their latest versions that could be successfully compiled by LLVM. **#Inlines** reports the total number of times functions are inlined – the larger this number, the more calling contexts a program has. For the Linux modules that were not linked appropriately, inlining only occurred inside.

Since our goal is to enable developers to use Graspan on development machines, we ran Graspan on a Dell desktop, with a quad-core 3.2GHZ Intel i5-4570 CPU, 8GB memory, and a 1TB SSD, running Linux 4.2.0. The size of the Java heap given to Graspan was 6GB. 8 threads were used when the EP-centric computation was performed.

Our evaluation focuses on the understanding of the following four research questions:

- Q1: Can the two analyses we implemented find new bugs in large-scale systems? (§5.1)
- Q2: How does Graspan perform in terms of time and space and how does it compare to existing graph systems? (§5.2)
- Q3: How do Graspan-based analysis implementations compare with other analysis implementations in terms of development effort and performance? (§5.3)
- Q4: How does Graspan compare with other backend systems when processing analysis workloads? (§5.4)

Since our analyses have already achieved the highest level of context sensitivity, we did not compare their precision with that of existing analyses. The main goal of this evaluation is to (1) demonstrate the usefulness of these interprocedural analyses through the detection of new bugs, and (2) show the efficiency and scalability of Graspan when performing such expensive analyses that would be extremely difficult to make scalable otherwise.

5.1 Effectiveness of Interprocedural Analyses

To understand the effectiveness of our interprocedural analyses, we re-implemented the seven static checkers listed in Table 1 in Clang. We used these existing checkers as the baseline to understand whether the combination of interprocedural pointer/alias and dataflow analyses are able to improve them in finding new bugs or reducing false positives (as described in Table 1 in §1). Note that our interprocedural analyses are not limited to these checkers; they can be used in a much broader context to find other types of bugs as well (*e.g.*, data races, deadlocks, *etc.*). We would also like to evaluate our analyses on commercial static checkers such as Coverity and GrammaTech. Unfortunately, we could not obtain a license that allows us to publish the comparisons, and hence, we had to develop these checkers from scratch.

We have added a new interprocedural checker UNTest that aims to find unnecessary, over-protective NULL tests – tests on pointers that must have non-NULL values – before dereferencing these pointers. Although these checks are not bugs, they create unnecessary code-level basic blocks that prevent compiler from performing many optimizations such as common sub-expression elimination or copy propagation, leading to performance degradation. Hence, these checks should be removed for compiler to fully optimize the program.

Checker	BL(4.4.0)		GR(4.4.0)		BL(2.6.1)
	RE	FP	RE	FP	RE
Block	0	0	0	0	43
Null	20	20	**+108**	23	98
Free	14	14	+4	4	21
Range	1	1	0	0	11
Lock	15	15	+3	3	5
Size	25	23	+11	11	3
UNTest	N/A	N/A	**+1127**	0	N/A

Table 3. Checkers implemented, their numbers of bugs reported by the baseline checkers (BL), and *new bugs* reported by our Graspan analyses (GR) on top of the BL checkers on the Linux kernel 4.4.0-r5; **RE** shows total numbers of bugs reported while **FP** shows numbers of false positives determined manually; to provide a reference of how bugs evolve over the last decade, we include an additional section **BL(2.6.1)** with numbers of *true* bugs reported by the same checkers in 2011 on the kernel version 2.6.1 from [59]. UNTest is a new *interprocedural* checker we implemented to identify unnecessary NULL tests; '+' means new problems found.

We manually checked *all bug reports* from both the baseline checkers and our analyses (except those reported by UNTest as described shortly) to determine whether a reported bug is a real bug. Since some of these checkers (such as Block, Range, and Lock) are specifically designed for Linux, Table 1 only reports information *w.r.t.* the Linux kernel. For checkers that check generic properties (*i.e.*, Null and UNTest), we have also run them on the two other programs; their results are described later in this section.

For the first six baseline checkers that found many real bugs in older versions of the kernel (used by [59] in 2011 to check Linux 2.6.x and by Chou et al. [24] in 2001 to check Linux 2.4.x), they could find only 2 real bugs in Linux 4.4.0-r5 (with the Size checker). This is not surprising because they were designed to target very simple bug patterns. Given that many static bug checkers have been developed in the past decade (including both commercial and open source), it is likely that most of these simple bugs have been fixed in this (relatively) new version of Linux. For example, the Null checker detected most of the bugs in [59] and [24]. In this current version, while it reported 20 potential bugs, a manual inspection confirmed that all of them were false positives.

Unnecessary NULL Tests We used our interprocedural analyses to identify NULL tests (*i.e.*, `if(p)`) in which the pointers checked *must not* be NULL. We have identified a total of 1127 unnecessary NULL tests in Linux, 149 in PostgreSQL, 32 in httpd. These are over-protective actions in coding, and may result in performance degradation. Because these warnings are too many to inspect manually, we took a sample of 100 warnings and found these tests are truly

```
void*probe_kthread_data(
    task_struct *task){
 void *data = NULL;
 probe_kernel_read(&data);

 /*data will be
  dereferenced after
  return.*/
 return data;
}

long probe_kernel_read
 (void *dst){
 if(...)
   return -EFAULT;
 return
 __probe_kernel_read(dst);
}
```

(a) NULL deref in kernel/kthread.c

```
#define page_private(page)
    ((page)->private)

bool swap_count_continued
 (...){
 head=vmalloc_to_page(...);
 if(page_private(head)
       != ...){
   ...
 }
}
page*vmalloc_to_page(...){
 page *page = NULL;
 if (!pgd_none(*pgd)){
  //...
 }
 return page;
}
```

(b) NULL deref in mm/swapfile.c

Figure 3. Two representative bugs in the Linux kernel 4.4.0-rc5 that were missed by the baseline checkers.

Modules	NULL pointer defs	Unnecessary NULL Tests
arch	0	75
crypto	0	15
init	0	1
kernel	4 (2)	65
mm	3 (0)	84
security	0	78
block	6 (2)	31
fs	19 (3)	84
ipc	0	17
lib	0	39
net	10 (8)	269
sound	15 (5)	83
drivers	25 (3)	286
Total	108 (23)	1127

Table 4. A breakdown of the new Linux bugs found by our analyses; in parentheses are numbers of false positives.

unnecessary. This is the *first time* that unnecessary NULL tests in the Linux kernel are identified and reported.

New Bugs Found Our analyses reported 108 new NULL pointer dereference bugs in Linux, among which 23 are false positives. All of these 85 new bugs involve complicated value propagation logic that cannot be detected by intraprocedural checkers. Figure 3 shows two example bugs.

in Figure 3 (a), function `probe_kthread_data` invokes `probe_kthread_read` to initialize pointer `data`. However, in `probe_kthread_read`, if a certain condition holds, an error code (`-EFAULT`) is returned and the pointer never gets initialized. Function `probe_kthread_data` then returns `data` directly without any check and the pointer gets dereferenced immediately after the function returns to its caller. In Figure 3 (b), `page_private` may dereference a NULL pointer since function `vmalloc_to_page` may return NULL. This bug was missed by the baseline because the origin of the NULL value and the statement that dereferences it are in separate functions. These types of bugs can only be found by interprocedural analyses. In fact, we show these two bugs because they are relatively simple and easy to understand; most of our bugs involve more than 3 functions and more complicated logic.

For PostgreSQL and httpd, we detected 33 and 14 new NULL pointer bugs; our manual validation did not find any false positives among them.

Linux Bug Breakdown Table 4 lists the new bugs and NULL tests in Linux into modules. We make two observations on this breakdown. First, the code quality of the Linux kernel has been improved significantly over the past decade. Note that the bugs we found are all complicated bugs detected by our interprocedural analyses; the baseline checkers could not find any (shallow) bug in this version of the kernel. Second, consistent with the observations made in both [24] and [59], drivers is still the directory that contains most (NULL Pointer) bugs. This is not surprising as drivers is still the largest module in the codebase. On the other hand, drivers is also the module of which developers are most cautious (perhaps due to the findings in [24] and [59]), demonstrated by the most unnecessary NULL tests it contains.

5.2 Graspan Performance

Table 5 reports various statistics of Graspan's executions (C++ version). Note that there is a large difference between the initial size and the post-processing size of each graph. For example, in Linux, the number of edges increases 3-5 times after the computation, while for httpd, the Graspan graph for pointer analysis increases more than 100 times. The computation time depends on both program characteristics and analysis type. For example, while the pointer analysis graph for `httpd` has a large number of edges added, its dataflow analysis graph does not change as much and thus Graspan finishes the computation quickly in 11.4 minutes. We found that this is because our dataflow analysis only tracks NULL values and in httpd the distances over which NULL can flow are often short.

We have also attempted to run these graphs *in memory* on the desktop we used and all of them except the dataflow analysis of httpd ran out of memory. While the initial size of each graph is relatively small, when edges are added dynamically, the graph soon becomes very big and Graspan needs to repartition it many times to prevent the computation from running out of memory.

The **Graspan** section of Table 6 reports the breakdown of Graspan's running time into computation and I/O (*i.e.*, disk writes/reads). Clearly, the EP-centric computation dominates the execution. While Graspan needs to perform many disk accesses, the I/O cost is generally low because most disk accesses are sequential accesses. Compared against the Java version of Graspan, its C++ version is 2 – 5 × faster due to (1) the elimination of garbage collection as well as (2) the increased memory packing factor and decreased I/O costs.

Prog	Pointer/Alias Analysis					Dataflow Analysis				
	IS=(E,V)	PS=(E,V)	PT	SS	T	IS=(E,V)	PS=(E,V)	PT	SS	T
Linux	(249.5M,52.9M)	(1.1B,52.9M)	91 secs	27	1.7 hrs	(69.4M, 63.0M)	(211.3M, 63.0M)	65 secs	33	11.9 hrs
PSQL	(25.0M,5.2M)	(862.2M,5.2M)	10 secs	16	6.0 hrs	(34.8M,29.0M)	(56.1M, 29.0M)	35 secs	16	2.4 hrs
httpd	(8.2M, 1.7M)	(904.3M, 1.7M)	3 secs	13	8.4 hrs	(10.0M, 5.3M)	(19.3M, 5.3M)	9 secs	16	11.4 mins

Table 5. Graspan performance: reported are the numbers of vertices and edges before (**IS**) and after (**PS**) being processed by Graspan, Graspan's pre-processing time (**PT**), numbers of supersteps taken (**#SS**), and total running time (**T**).

Analysis	Graspan		ODA [101]	SociaLite [45]
	CT	I/O		
Linux-P	99.7 mins	46.6 secs	OOM	OOM
Linux-D	713.8 mins	4.2 mins	-	OOM
PostgreSQL-P	353.1 mins	4.5 mins	> 1 day	OOM
PostgreSQL-D	143.8 mins	57.1 secs	-	OOM
httpd-P	497.9 mins	7.6 mins	> 1 day	> 1 day
httpd-D	11.3 mins	3.3 secs	-	4 hrs

Table 6. A comparison on the performance of Graspan, on-demand pointer analysis (ODA) [101] implemented in standard ways, as well as SociaLite [45] processing our program graphs in Datalog. The Graspan section shows a breakdown of the running times into computation time (**CT**), I/O time (**I/O**), and garbage collection time (**GC**); P and D represent pointer/alias analysis and dataflow analysis. OOM means out of memory.

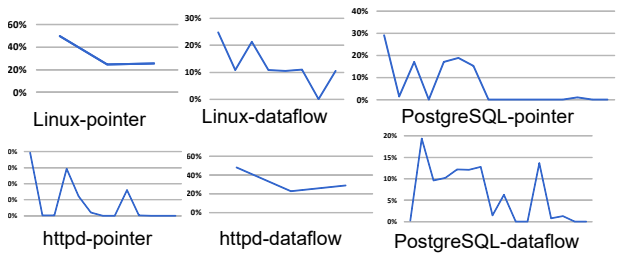

Figure 4. Percentages of added edges across supersteps.

Figure 4 depicts the percentages of added edges across supersteps, measured as the number of added edges divided by the number of edges in each original graph. In general, an extremely large number of edges are added within the first 10 supersteps (*e.g.*, more than 500M for Linux), and as the computation progresses, fewer edges are added.

5.3 Comparisons with Other Analysis Implementations

Data Structure Analysis [47] To understand whether Graspan-based analyses are more scalable and efficient than traditional analysis implementations, we wanted to compare our analyses with existing context-sensitive pointer/alias and dataflow analyses. While we had spent much time looking for publicly available implementations, we could not find anything available except the data-structure analysis (DSA) [47] in LLVM itself. DSA (implemented in 2007) is much more complicated than our pointer/alias analysis implementation — it has more than 10K lines of code while our pointer/alias analysis (*i.e.*, the graph generation part) only has 1.2K lines of code. According to a response from the LLVM mailing list [8], DSA was buggy and removed from LLVM since version 3.3. We tried to use LLVM 3.2 but it could not compile any version of the Linux kernel due to lack of patches.

On-demand Pointer Analysis [101] As no other implementations were available, we implemented the context-sensitive version of Zheng and Rugina's C pointer analysis [101] ourselves. We took the expression graph generated by our frontend and used a worklist-based algorithm to compute transitive closures, following closely the original algorithm described in [101]. The **ODA** section of Table 6 reports its performance. For all but httpd, ODA either ran out of memory or took a very long time (longer than one day) on the same desktop where we ran Graspan. For example, when processing Linux, it ran out of memory in 13 minutes. When we moved it onto a server with 32 2.60GHZ Xeon(R) processors and 32GB memory, it took this implementation 3.5 days to analyze Linux and it consumed 29GB out of the 32GB memory. On the contrary, Graspan finished processing Linux in a few hours with less than 6GB memory on the desktop with a much less powerful CPU.

5.4 Comparisons with Other Backend Engines

Datalog Since Datalog has been used to power static analyses, it is important to understand the pros/cons of using Graspan v.s. a Datalog engine as the analysis backend. While there are many Datalog engines available [7, 45, 74, 86], SociaLite [45] and LogicBlox [7] are designed for shared-memory machines while others [74, 86] are distributed engines running on large clusters. Since a distributed engine is not a choice for code checking in daily development tasks, we focused our comparison against shared-memory engines. LogicBlox is a commercial tool that has been previously used to power the Doop pointer analysis framework [17] for Java. However, it was the same licensing issue that prevented us from publishing comparison results with LogicBlox. Hence, this subsection only compares Graspan with SociaLite, a Datalog engine developed by Stanford that has been demonstrated to outperform other shared-memory engines.

The **SociaLite** section of Table 6 reports SociaLite's performance on the same desktop. SociaLite programs were easy to write — it took us less than 50 LoC to implement either analysis. However, SociaLite clearly could not scale to graphs that cannot fit into memory. For both pointer/alias and dataflow analysis, it ran out of memory for Linux and

PostgreSQL. For httpd, although SociaLite processed the graphs successfully, it was much slower than Graspan.

GraphChi To understand whether other graph systems can efficiently process the same (program analysis) workload, we ran GraphChi — a disk-based graph processing system — because GraphChi is the only available system that supports both out-of-core computation and dynamic edge addition. GraphChi provides an API `add_edge` for the developer to add an edge; it maintains a buffer for newly added edges during computation and uses a threshold to prevent the buffer from growing aggressively. When the size of the buffer exceeds the threshold, the edge adding thread goes to sleep and the function always returns false. The thread periodically wakes up and checks whether the main data processing thread comes to the commit point, at which the edges in the buffer can be flushed out to disk. GraphChi does not check edge duplicates and thus its computation would never terminate on our workloads. We added a naïve support that checks, before an edge is added, whether the same edge exists in the buffer. Note that this support does not solve the entire problem because it only checks the buffer but duplicates may have been flushed to shards. Checking duplicates in shards would require a re-design of the whole system.

We ran GraphChi on the same desktop to process the Linux dataflow graph. GraphChi ran into assertion failures in 133 seconds with around 65M edges added. This is primarily because GraphChi was not designed for the program analysis workload that needs to add an extremely large number of edges (with many duplicates) dynamically.

6. Related Work

Static Bug Finding Static analysis has been used extensively in the systems community to detect bugs [1, 9, 14, 15, 18, 20, 25, 26, 28–31, 35, 48, 49, 56, 59, 70, 71, 88, 93] and security vulnerabilities [19, 21, 40]. Engler et al. [29] use a set of nine checkers to empirically study bugs in OS kernels. Palix et al. [59] implemented the same checkers using Coccinelle [58]. Commercial static checkers [2–5] are also available for finding bugs and security problems. Most of these checkers are based on pattern matching. Despite their commendable bug finding efforts, false positive and negatives are inherent with these checkers.

Interprocedural analyses such as pointer and dataflow analysis can significantly improve the effectiveness of the checkers, but their implementations are often not scalable. There exists a body of work that makes program analysis declarative [17, 90] — analysis designers specify rules in Datalog and these rules are automatically translated into analysis implementations. However, the existing Datalog engines perform generic table joining and do not support disk-based computation on a single machine. While declarative analyses reduce the development effort, they still suffer from scalability issues. For example, although the pointer analysis from Whaley et al. [90] can scale to reasonably large Java programs (*e.g.*, using BDD), it only clones pointer variables, not objects. Furthermore, there is no evidence that they can perform fully context-sensitive analyses on codebases as large as the Linux kernel on a commodity PC.

Grammar-guided Reachability There is a large body of work that can be formulated as a context-free language (CFL) reachability problem [94]. CFL-recognition was first studied by Yannakakis [94] for Datalog query evaluation. Work by Reps *et al.* [38, 62, 64–66] proposes to model realizable paths using a context-free language that treats method calls and returns as pairs of balanced parentheses. CFL-reachability can be used to formulate a variety of static analyses, such as polymorphic flow analysis [61], shape analysis [63], points-to and alias analysis [16, 78, 79, 79, 80, 91, 92, 97, 99, 101], and information flow analysis [51]. The works in [42, 43, 54] study the connection between CFL-reachability and set-constraints, show the similarity between the two problems, and provide implementation strategies for problems that can be formulated in this manner. Kodumal *et al.* [43] extend set constraints to express analyses involving one context-free and any number of regular reachability properties. CFL-reachability has also been investigated in the context of recursive state machines [11], streaming XML [10], and pushdown languages [12]. Recent work uses CFL-reachability to formulate pointer and alias analysis [16, 78, 79, 79, 80, 91, 92, 97–99, 101]. and specification inference [16].

Graph Systems State-of-the-art graph systems include disk-based systems [36, 44, 50, 69, 84, 87, 100, 102], shared-memory systems [57, 75], as well as distributed systems [22, 23, 32, 33, 52, 53, 55, 60, 68, 81–83, 85]. Graspan is designed specifically for the transitive closure computation workload with two features that the existing systems do not have: repartitioning and quick edge duplicate checking.

7. Conclusion

Graspan is the *first attempt* to turn sophisticated code analysis into scalable Big Data analytics, opening up a new direction for scaling various sophisticated static program analyses (*e.g.*, symbolic execution, theorem proving, *etc.*) to large systems.

Acknowledgments

We especially thank John Thorpe and Sungsoo Son, two undergraduate students who have contributed significantly to Graspan's C++ implementation. We would like to thank the anonymous reviewers for their valuable and thorough comments. The initial idea was inspired by a conversation in 2015 Guoqing Xu had with Qirun Zhang, who also provided useful feedback for us to prepare for the camera ready version. This material is based upon work supported by the National Science Foundation under grant CNS-1321179, CCF-1409829, CNS-1613023, CNS-1617513, and CNS-1617481, and by the Office of Naval Research under grant N00014-14-1-0549 and N00014-16-1-2913.

References

[1] The findbugs Java static checker. http://findbugs.sourceforge.net/, 2015.

[2] The Coverity code checker. http://www.coverity.com/, 2016.

[3] The GrammaTech CodeSonar static checker, 2016.

[4] The HP Fortify static checker, 2016.

[5] The KlocWork static checker, 2016.

[6] The LLVMLinux project. http://llvm.linuxfoundation.org/, 2016.

[7] The LogicBlox Datalog engine. http://www.logicblox.com/, 2016.

[8] Personal communication with John Criswell, 2016.

[9] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *PASTE*, pages 43–48, 2007.

[10] R. Alur. Marrying words and trees. In *PODS*, pages 233–242, 2007.

[11] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[12] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

[13] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29(10):996–1000, 1986.

[14] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.

[15] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[16] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

[17] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

[18] F. Brown, A. Notzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *ASPLOS*, pages 143–157, 2016.

[19] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE S&P*, pages 325–338, 2008.

[20] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[21] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[22] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, pages 215–226, 2014.

[23] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[24] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[25] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

[26] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, pages 12–22, 2004.

[27] D. Engler. Making finite verification of raw C code easier than writing a test case. In *RV*. Invited talk.

[28] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–1, 2000.

[29] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[30] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

[31] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, 1999.

[32] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[34] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[35] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[36] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[37] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[38] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *FSE*, pages 104–115, 1995.

[39] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(2-3):273–281, 1986.

[40] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security*, pages 9–9, 2004.

[41] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434, 2013.

[42] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.

[43] J. Kodumal and A. Aiken. Regularly annotated set constraints. In *PLDI*, pages 331–341, 2007.

[44] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[45] M. S. Lam, S. Guo, and J. Seo. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

[46] B. W. Lampson. Hints for computer system design. In *SOSP*, pages 33–48, 1983.

[47] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

[48] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 20–20, 2004.

[49] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, pages 306–315, 2005.

[50] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, , and U. Kang. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData*, pages 159–164, 2014.

[51] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *PASTE*, pages 50–56, 2008.

[52] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[53] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and G. Inc. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[54] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[55] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[56] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[57] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[58] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*, pages 247–260, 2008.

[59] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. In *ASPLOS*, pages 305–318, 2011.

[60] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *SoCC*, pages 195–208, 2015.

[61] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[62] T. Reps. Solving demand versions of interprocedural analysis problems. In *CC*, pages 389–403, 1994.

[63] T. Reps. Shape analysis as a generalized path problem. In *PEPM*, pages 1–11, 1995.

[64] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[65] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[66] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *FSE*, pages 11–20, 1994.

[67] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, pages 184–191, 2004.

[68] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[69] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[70] 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*, pages 270–280, 2009.

[71] C. Rubio-González and B. Liblit. Defective error/pointer interactions in the linux kernel. In *ISSTA*, pages 111–121, 2011.

[72] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

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

[74] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD*, pages 1135–1149, 2016.

[75] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[76] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

[77] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *PLDI*, pages 485–495, 2014.

[78] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[79] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

[80] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95, 2015.

[81] K. Vora, R. Gupta, and G. Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, 2016.

[82] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*, 2017.

[83] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *OOPSLA*, pages 861–878, 2014.

[84] K. Vora, G. Xu, and R. Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*, pages 507–522, 2016.

[85] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[86] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[87] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015.

[88] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, pages 260–275, 2013.

[89] C. Weiss, C. Rubio-González, and B. Liblit. Database-backed program analysis for scalable error propagation. In *ICSE*, pages 586–597, 2015.

[90] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[91] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[92] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

[93] J. Yang, C. Sar, and D. Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *OSDI*, pages 10–10, 2006.

[94] M. Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242, 1990.

[95] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Inf.*, 30(4):369–384, 1993.

[96] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *PLDI*, pages 91–103, 1999.

[97] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, pages 435–446, 2013.

[98] Q. Zhang and Z. Su. Context-sensitive data dependence analysis via linear conjunctive language reachability. In *POPL*, pages 344–358, 2017.

[99] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for C. In *OOPSLA*, pages 829–845, 2014.

[100] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

[101] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 2008.

[102] X. Zhu, W. Han, and W. Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, pages 375–386, 2015.