Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2020-TR-003

2020-TR-003

Chianina: An Evolving Graph System for Flow- and Context-Sensitive Analyses of Million Lines of C Code

Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Linzhang Wang, Xuandong Li, Guoqing Harry Xu

Technical Report 2020-TR-003

Chianina: An Evolving Graph System for Flow- and Context-Sensitive Analyses of Million Lines of C Code

***Abstract*—Sophisticated static analysis techniques often have complicated implementations, much of which provides logic for *tuning and scaling* rather than *basic analysis functionalities*. This tight coupling of basic algorithms with special treatments for scalability makes an analysis implementation hard to (1) make correct, (2) understand/work with, and (3) reuse for other clients. This paper presents Chianina, a graph system we developed for fully context- and flow-sensitive analysis of large C programs. Chianina overcomes these challenges by allowing the developer to provide only the basic algorithm of an analysis and pushing the tuning/scaling work to the underlying system. Key to the success of Chianina is (1) an *evolving graph formulation* of flow sensitivity and (2) the leverage of *out of core, disk support* to deal with memory blowup resulting from context sensitivity. We implemented two context- and flow-sensitive analyses (including a pointer analysis) on top of Chianina and scaled them to five programs including Linux (17M LoC) on a single commodity PC.**

I. Introduction

Static analysis plays important roles in a wide spectrum of applications, including automated bug discovery, compiler optimization, *etc.*. Static analysis algorithms that distinguish results based on various program properties (such as calling contexts and control flow) are more useful than those that do not. For example, these precise algorithms can uncover more true bugs and report less false warnings. As a result, there is an everlasting interest in the program analysis community to develop techniques that are context-sensitive [1]–[12], field-sensitive [1], [4], [7], [13], [14], flow-sensitive [1], [11], [15]–[18], or path-sensitive [19]–[22].

Although these techniques are superior to their (context, field, flow, or path-) insensitive counterparts, their computation is much more expensive, requiring CPU and memory resources that a single machine may not be able to offer. Given the limited resources available to them, it is hard for them to scale to programs with large codebases such as the Linux kernel. Prior work employs sophisticated treatments that tune the level of sensitivity [23]–[31], explore different forms of sensitivity [32], [33], or pre-process programs with inexpensive techniques [22], [34], in order to find sweatspots between scalability, generality, and usefulness. Despite their commendable efforts, these treatments are specific to the applications they are developed for and very complicated to implement.

This paper is a quest driven by the following question: given an analysis algorithm — *in its simplest form* — can we run it efficiently over large programs without requiring any sophistication from the developer? Achieving this goal possesses a number of advantages: (1) analysis development is significantly simplified — because a developer only writes the basic algorithm without worrying about performance, this enables developers without much training in PL to easily develop and experiment with analyses that used to be accessible only to experienced experts; and (2) porting an existing analysis for different clients is significantly simplified because the analysis implementation contains only the logic necessary to realize the basic functionality, *not* any complex tuning tasks.

Insight and Problem. This paper is inspired by a line of prior work [5], [21], [35], [36] that piggybacks static analysis on large-scale systems — an analysis is implemented by following only a few high-level interfaces while scaling is delegated to the underlying system, which makes it possible for the analysis to run on large programs by enlisting the humongous computing power provided by modern hardware. BDDBDDB [5] and Doop [35] are early examples where an analysis is expressed as a Datalog program, which is executed by a low-level BDD-based Datalog engine for scalability. Graspan [36] is a graph processing system that leverages disk support to scale CFL-reachability computation to large programs that cannot fit into the main memory. This line of work shifts the burden of tuning from developers' shoulders to underlying systems, enabling developers to enjoy both the implementation simplicity and the scalability provided by the underlying system.

Inspired by these techniques, this paper revisits the problem of scaling *context- and flow-sensitive* analyses from a *system perspective* — that is, we aim to develop system support for scaling the *simplest versions* of context- and flow-sensitive algorithms that developers can quickly implement by following interfaces. On the one hand, a context- and flow-sensitive analysis is arguably one of the most expensive analysis techniques because it needs to compute and maintain an analysis solution for each distinct program point under each distinct calling context. On the other hand, it enables *strong update* and produces ultra precise information at each statement.

Problem Formulation. This paper presents a graph system called Chianina, that supports easy development of *any* context- and flow-sensitive analysis (with a monotone transfer function) for C and that is powerful enough to scale the analysis to many millions of lines of code. Chianina makes analysis implementation *simple* and *general* — a variety of flow-sensitive analysis (*e.g.*, analyses of IDE, IFDS, pointer, alias, type, value, *etc.*) can be easily developed with hundreds lines of code. The developer only specifies dataflow facts and transfer functions, *in their basic form without any special treatment*. The work of tuning and scaling (*e.g.*, merging, exploiting similarities, reduction, *etc.*), which used to be tightly coupled with the analysis, now happens under the hood in the system.

To deal with context sensitivity, Chianina uses aggressive cloning — a callee is cloned into each of its callers and cloning is done in a bottom-up fashion from each leaf node on the call graph to the `main` function (if it exists). To handle recursions, functions involved in a strongly connected component are collapsed and treated context insensitively. This appears similar to the approach taken by BDDBDDB [5], [10]. However, Chianina clones both variables (pointers) and objects (pointees) whereas BDDBDDB clones only variables — significant precision loss can result due to lack of object

cloning [3]. Of course, aggressively cloning function bodies can blow up the memory usage and Chianina solves the problem by leveraging out-of-core disk support. To perform cloning, Chianina uses a pre-computed call graph. For the C language, in particular, we use an inclusion-based context-insensitive flow-insensitive pointer analysis with support for function pointers (available in LLVM). Once cloning is done, we have a complete program representation for graph computation.

To deal with flow sensitivity, Chianina formulates a flow-sensitive analysis as a problem of *evolving graph processing* [37]–[42]. An evolving graph contains a set of temporally-related *graph snapshots*, each capturing the set of vertices and edges of the graph at a certain point of time. For example, a social network graph such as Twitter constantly evolves. Analytics tasks such as finding popular users (*i.e.*, PageRank) are often performed on snapshots of the graph periodically and results from these tasks are analyzed to understand the evolution of the graph. Two consecutive snapshots often have large overlap on vertices and edges (*i.e.*, spatial and temporal locality), which can be exploited for efficiency.

This nature of evolving graph processing matches exactly the nature of a flow-sensitive analysis — at each program point, (the most general form of) dataflow facts for variables in the program constitute a graph snapshot; consecutive snapshots, which are captured at consecutive program points, differ only in a small number of vertices and edges due to application of transfer function. Our formulation makes an analysis *immediately amenable to many optimization techniques* (*e.g.*, auto-parallelization, work-balancing, locality, *etc.*) available in the graph system community, tuning and scaling the analysis at a low level without needing any special treatment from the developer. In fact, many of the prior analysis-level treatments (*e.g.*, BDD-based merging) perform essentially the same work as certain system-level optimizations (*e.g.*, locality-aware compression). By pushing the tuning effort down into the system, *every analysis* running atop can enjoy these low-level optimizations, while in the past each analysis only receives a small handful of special treatments tailored for itself.

Summary of Results. To demonstrate scalability and generality, we implemented, on top of Chianina, (1) a fully context- and flow-sensitive pointer/alias analysis and (2) a null-pointer value flow analysis that supports precise context-sensitive heap tracking of null values. We analyzed five large-scale software systems: Linux, Firefox, PostgreSQL, OpenSSL and Httpd. Our results are promising: our analyses completed on the five systems (4 minutes – 20 hours) whereas their conventional counterparts (even without context sensitivity) quickly ran out of memory for large programs. To the best of our knowledge, this is the *first time* that a fully context- and flow-sensitive pointer/alias analysis has been shown to scale to a recent version of Linux with **17 million lines of code**.

II. Related Work

Graspan. The work closely related to Chianina is Graspan [36], a disk-based system for scalable dynamic transitive computation. Although Chianina is inspired by the same high-level observation as Graspan, it is impossible to extend Graspan to support arbitrary flow-sensitive analyses without redesigning and re-implementing the system from scratch. The computation model is fundamentally different. Graspan is designed for dynamic transitive closure computation which serves the basis for CFL-reachability based analyses. The simple computation logic for graph reachability does not work for Chianina's complex dataflow semantics.

Evolving Graph Systems. Although we formulate flow-sensitive analysis as an evolving graph processing problem, the nature of the problem differs significantly from that dealt with in the graph system community. None of the existing systems [38], [39], [43], [44] can solve it. System design is heavily dependent on (1) data and (2) computation. On the data side, our program graph differs significantly from the typical data graphs that existing systems target as data graphs do not have any semantics. Our computation model also differs from the computation in existing systems, which is driven solely by the graph algorithm (*e.g.*, pagerank, *etc.*) that is fundamentally different from program analysis workloads. In summary, it is the semantics of the program analysis that makes Chianina unique; none of existing systems are able to perform this type of computation as Chianina does over program graphs.

Flow-Sensitive Analyses. A common optimization of scaling flow-sensitive analysis is to perform a sparse analysis preventing redundant values from being propagated [45], [46]. Hind and Pioli [47] adopted the sparse evaluation graph [48] which eliminates pointer-free statements from the CFG. Hardekopf and Lin [15], [16] proposed to utilize a semi-sparse representation by connecting variable definitions with their uses, allowing dataflow facts to be propagated only to the locations needing the variable. Sui and Xue implemented SVF [49], [50] , which constructs the sparse value-flow graph and performs the pointer analysis in an iterative manner. Other techniques such as [51]–[53] use similar ideas to scale flow-sensitive analysis. Note that all of these techniques are orthogonal to Chianina, which can take any sparse representation as the input graph and perform scalable computation via system support.

In order to accelerate an interprocedural dataflow analysis, a few techniques attempt to parallelize its computation. Rodriguez *et al.* [54] proposed an actor-model-based parallel algorithm for IFDS problems. Garbervetsky et al. [55] developed a distributed worklist algorithm using the actor model to implement a call-graph analysis. Albarghouthi *et al.* [56] parallelize a top-down interprocedural analysis using a MapReduce-like computation model. Several studies [57], [58] attempt to parallelize flow-sensitive pointer analysis. Since they all require large amounts of memory, there is no evidence that these parallel approaches can scale to programs such as the Linux kernel.

Context-Sensitive Analyses. Generally, there are two dominant approaches to context-sensitive interprocedural analysis: the *summary-based approach* and the *cloning-based* approach [59]. The summary-based approach [12], [17], [60], [61] constructs a summary (transfer) function for each procedure, and directly applies the summary to the specific inputs at

the call site invoking the function. Although the summary-based approach is scalable, it does not provide complete alias information for each particular context due to lack of explicit representation of calling contexts. Furthermore, it is difficult to precisely model heap effects. The cloning-based approach [2], [5], [8], [62] provides complete information. However, it requires each procedure to be re-analyzed under each calling context and hence is hard to scale. Demand-driven techniques [7], [9], [63] match call/return edges *on the fly* for context sensitivity. A body of techniques have also been proposed to perform selective context sensitivity [22]–[34], [64], so as to find sweatspots between scalability and precision.

III. Background and Overview

We present Chianina in the context of pointer/alias analysis, which is perhaps the most sophisticated and expensive one in the context- and flow-sensitive analysis family. This section first offers a gentle introduction to the basic algorithm for a context- and flow-sensitive pointer/alias analysis for C. Next, we provide an overview of Chianina, explaining how it works.

A. Background

Alias Analysis as Graph Reachability. A flow-insensitive alias analysis can be easily formulated as a graph-reachability problem. There are a number of existing formulations, of which we use the program expression graph (PEG) [65] based representation as an example to illustrate how Chianina works. Note that this paper makes no contribution on analysis algorithms; any other program representations can be used/implemented on Chianina as well.

A PEG represents a program as a graph where each vertex corresponds to a pointer expression (*e.g.*, a reference variable x, a dereference expression $*x$, or an address-of expression $\&x$). Edges are added based upon the following rules for statements that involve pointer expressions.

Type	***Stmt***	***Edge***	
assignment	$x = y$	$x \xleftarrow{a} y$	(1)
store	$*x = y$	$*x \xleftarrow{a} y$	(2)
load	$x = *y$	$x \xleftarrow{a} *y$	(3)
address-of	$x = \&y$	$x \xleftarrow{a} \&y$	(4)

Each statement that allocates heap memory (*e.g.*, $x = malloc()$) is treated the same way as an address-of statement — we add an edge $x \xleftarrow{a} \&O$ where O represents the abstract memory location (*i.e.*, allocation site). Moreover, *dereference edges (d)* are added (1) from each pointer variable x to $*x$ and (2) from $\&x$ to x.

Based on this graph representation, the alias analysis is formulated as a reachability problem guided by a *context-free language* $\mathcal{L}$ over an alphabet Σ (*i.e.*, the set of $\{a, d\}$ in the context of PEG). Given a PEG whose edges are labelled with elements of Σ, we say a vertex v is $\mathcal{L}$-reachable from another vertex w if there exists a path from v to w on the graph such that the string formed by concatenating edge labels on the path is a member of language $\mathcal{L}$ (*i.e.*, complying with L's grammar). A whole-program alias analysis determines all pairs of such vertices v and w such that w is $\mathcal{L}$-reachable from v, based on the following context-free grammar:

$$Value\ alias \qquad V ::= (M?\ \overline{a})^*\ M?\ (a\ M?)^* \quad (5)$$
$$Memory\ alias \qquad M ::= \overline{d}\ V\ d \quad (6)$$

The non-terminals *V* and *M* represent the *value-alias* and *memory-alias* relations, respectively. Each PEG is a bidirectional graph — for each edge $x \xrightarrow{a} y$ with label x, there exists an inverse edge $y \xrightarrow{\overline{a}} x$ automatically. Two pointer expressions are aliases if they are $\mathcal{V}$- or $\mathcal{M}$-reachable. At the heart of this formulation is finding paths whose edge labels exhibit "balanced-parenthesis" properties (*e.g.*, a and $\overline{a}$): if a pointer value goes from a variable x into a heap location h and later flows to another variable y from a heap location i, the two variables x and y are (pointer) aliases if the two heap locations h and i are (memory) aliases. Given that this formulation is well-known to the SE/PL community, we omit a concrete example here to save space.

Flow-Sensitivity. Flow sensitivity is often achieved using the traditional monotone dataflow analysis framework [66], [67], which consists of the analysis domain, including operations to copy and combine domain elements, and the transfer functions over domain elements with respect to different types of statement in the control flow graph. In the context of a PEG-based alias analysis, the most straightforward way to add flow sensitivity is to model each domain element as a separate PEG and the combination operator as the union of edge sets. Each transfer function with respect to a program statement takes an input PEG that captures the state of the program before the statement, and computes an output PEG by adding and deleting edges according to the semantics of the statement.

Next, a worklist-based algorithm iteratively applies the transfer function for each statement along the control-flow graph (CFG). In our setting, two elements $\mathcal{IN}_s$ and $\mathcal{OUT}_s$ are maintained for each statement s of the CFG, representing the incoming and outgoing PEGs, respectively. Each transfer function s computes a new PEG $\mathcal{OUT}_s$ by adding/deleting edges on $\mathcal{IN}_s$. At each control flow join point where a node s has multiple predecessors $p \in predecessors(s)$, the incoming graph $\mathcal{IN}_s$ of node s is the union of all graphs $\mathcal{OUT}_p$ of its predecessors. The algorithm keeps updating these graphs until seeing the global fixed point [68]. Each transfer function is characterized as addition (*i.e.*, GEN) or deletion (*i.e.*, $KILL$) of a set of edges based on the aforementioned formulation. The GEN set usually denotes the new assignment edge (labeled with a) added due to a statement. The $KILL$ set contains edges that must be deleted due to updated assignments. These deletions enable strong update.

Note that the PEG representation discussed above describes the *basic analysis algorithm* without any scalability treatments. Naïvely running this algorithm will be unscalable. Chianina provides scalability with graph optimizations and disk support.

B. Chianina Overview

Chianina contains a C-based frontend and a language-independent backend (which can be readily used to analyze

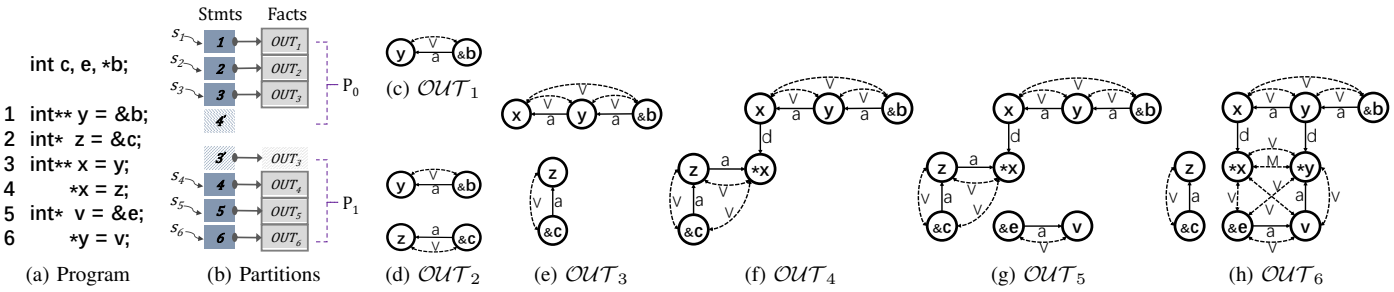


Fig. 1: (a) The example program under analysis. (b) The two partitions: each CFG vertex links to a PEG; CFG edges are stored with their source vertices but not shown in the figure. (c)-(h) PEGs at each program point as iterative computation is performed by the backend graph engine; inverse edges are omitted for simplicity; The "V" and "M" edges represent transitive *value-alias* and *memory-alias* relationships shown earlier in Equations (5) and (6).

programs in other languages although this paper focuses on the C language). The frontend is a Clang-based intraprocedural compiler pass that analyzes each C function to produce a control flow graph (CFG) of the function where each vertex of the CFG (*i.e.*, a statement) contains an PEG representing the dataflow fact at the statement. The initial PEG for each statement just contains edges induced by the statement itself. The backend is a graph engine that performs iterative computation over the CFG to update PEGs associated with each statement. The CFG generation is generic and independent of client analysis, but the graph representing each dataflow fact (contained in each CFG vertex) is *client-specific* and needs to be provided by the developer. For our pointer/alias analysis, each dataflow fact is a PEG, which will grow/shrink as computation is performed by the backend. Note that the developer can also customize the CFG structure generated for each function. For example, our analysis implementation actually generates a sparse def-use graph proposed in [16], which is more efficient than the general CFG. For generality, we will still use term CFG in the rest of the paper to refer to the graph representation.

Cloning for Context Sensitivity. Once the CFG for each function is generated, Chianina relies on a pre-computed call graph (*i.e.*, constructed by LLVM) to perform cloning for context sensitivity. The CFG for each function is cloned and incorporated into that of each of its callers by creating assignment edges to connect vertices representing formal and actual parameters. Cloning of a CFG includes cloning of each PEG contained in each of its vertices. To handle recursion, we first identify the strongly connected components (SCCs) over the pre-computed call graph. Functions in each SCC are collapsed and their CFGs are connected in trivial ways without cloning. Note that although this handling may potentially reduce analysis precision, we find C programs often have very small SCCs (*e.g.*, most SCCs have less than 3 functions) and hence its impact on precision is small.

Clearly, aggressive inlining is extremely space-expensive as the space requirement grows exponentially. For example, for the Linux kernel, there are totally **48.2 million** function inlines, generating a total number of **661.6 million** edges in the final CFG produced. Naïvely running this process would quickly hit the memory wall and crash. We developed an *out-of-core* cloner that does efficient swapping by partitioning the call graph into a set of subgraphs. Cloning is done on each subgraph simultaneously if they do not have dependencies. As cloning is moving up on the call graph, it consumes more memory and we work on fewer subgraphs at the same time. Those finished subgraphs that do not have dependencies with the working subgraphs are swapped out onto disk. Once the cloning is done, the generated global CFG (GCFG), which is stored as a large disk file in the adjacency list format, is already a *fully context-sensitive* representation of the program (except for functions in SCCs).

Evolving Graph Computation. Figure 1a shows an example C program. The dataflow fact associated with each statement, represented as a PEG, is initialized by the frontend compiler pass as a small PEG containing only edges induced by that statement. For space efficiency, only $\mathcal{OUT}$ is maintained explicitly since $\mathcal{IN}$ for a statement can be easily derived by taking a union of $\mathcal{OUT}$ of its predecessors.

As the first step, Chianina divides the GCFG into multiple partitions. Figure 1b shows such an example with two disjoint partitions, containing vertices of the logical ranges [1-3] and [4-6], respectively. For edges that cross partitions, such as the one between statement 3 and 4 in Figure 1a, we create two *mirror vertices* $3'$ and $4'$ and place them respectively into the two partitions. Such edges induce dependencies between partitions. With multiple partitions available on disk, the Chianina scheduler picks a number of partitions at a time and loads them into memory for parallel computation. The number of partitions to load at each time is determined by (1) memory availability and (2) the number of CPU cores. Partitioning and scheduling is detailed in §IV-C.

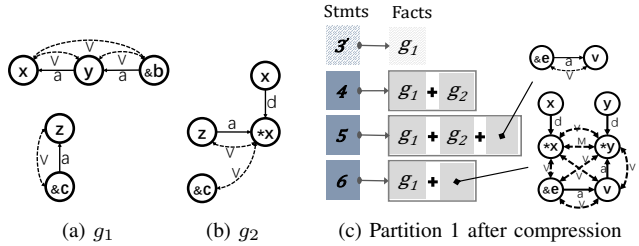
Assuming that both partitions are selected for computation in our example, Chianina loads into memory all CFG edges that belong to P_0 and P_1 and dataflow facts (PEGs) associated with each vertex. The computation engine runs the iterative algorithm over the subgraph represented by the partition in a *Bulk Synchronous Parallel* (BSP) style [69]. For our example, Chianina uses two threads to run the iterative computation over the two partitions. The iterative algorithm, which is the same as the traditional dataflow algorithm, keeps updating PEGs until a fixed point is reached. For example, when the computation reaches the mirror vertex 4 in P_0, it stops because vertex 4 is *not* present in the partition and there is no other path to continue the algorithm.

Before Chianina writes all updated PEGs back to disk for

P_0, it adds statement 4 into the *active list* of P_1 via a message, together with the new PEG for this statement computed in P_0. When the current computation for P_1 finishes, the scheduler identifies that P_1 has an active vertex (meaning an updated PEG for the vertex has been computed from another partition). As a result, it selects P_1 for computation again in the next round. This next round of computation for P_1 is *incremental* — it starts at statement 4 (known as *frontier* in the terminology of graph processing) and only updates subsequent PEGs that are affected by the change. The repetitive process stops until a global fixed point is seen — no partition has any active vertices to process. In our example, the final $\mathcal{OUT}$ PEGs for the statements 1–6 are shown in Figure 1c–1h, respectively.

Alias Computation. There are two choices as to how to compute an alias solution (based on Equation 5 and 6) on each PEG. The first choice is that alias computation is performed on each PEG after the iterative algorithm finishes globally. While the approach simplifies the dataflow transfer function (which only needs to update direct assignment (*i.e.*, a-) edges during iterative computation), we are *not* able to perform *strong update* (*i.e.*, edge deletion) at each update because the pointer/alias information is unknown when transfer functions are applied. The second choice is we compute transitive edges on each PEG *on the fly* as the PEG is updated. This approach enables strong updates because the alias information is available at each update, at a cost of complicating transfer functions — now each transfer function has to additionally take care of addition/deletion of transitive (*i.e.*, V- and M-) edges besides assignment (a-) edges. Due to the importance of strong update in a flow-sensitive analysis, Chianina adopts the second approach, which computes and updates transitive edges on the fly.

To illustrate, consider statement 6 in Figure 1a where $*y$ points to a *singleton memory location*. A strong update is performed there — the effect of this is to *kill*, from the PEG $\mathcal{OUT}_5$, (1) all direct assignment edges going to $*y$ and expressions that *must alias* $*y$, as well as (2) all transitive edges induced by these assignment edges heretofore. In our example, there exists no direct assignment edge to $*y$, but our must-alias analysis determines that $*y$ and $*x$ must alias. As such, the direct assignment edge $z \xrightarrow{a} *x$ as well as the induced edges $z \overset{V}{\longleftrightarrow} *x$ and $\&c \overset{V}{\longleftrightarrow} *x$ are deleted. Details about strong update and edge deletion can be found in §IV-E.



(a) g_1 (b) g_2 (c) Partition 1 after compression

Fig. 2: Two frequent subgraphs mined over the PEGs in Partition P_1, with frequency ≥ 2 and size ≥ 3: (a) g_1 whose frequency = 4 and size = 7; (b) g_2 whose frequency = 2 and size = 4; (c) concise representation of P_1 based on g_1 and g_2.

Exploiting Locality between Consecutive PEGs. One clear advantage of our evolving graph formulation is that we can exploit similarities between PEGs for increased efficiency. In particular, Chianina extracts *frequent common subgraphs* (FCS) among PEGs and composes each PEG by *assembling existing FCSes* instead of duplicating these common edges and vertices in each PEG. In our example, P_1 consists of 4 PEGs. We invoke an off-the-shelf *itemset miner* Eclat [70] to discover the frequent edge-sets across these PEGs. Figure 2a and 2b depict two frequent subgraphs (g_1 and g_2), mined by using 2 as the frequency threshold and 3 as the size threshold. These two thresholds determine, respectively, the minimum occurrences of a subgraph and the minimum number of edges for the subgraph to be considered as a FCS. Next, Chianina *de-duplicates* PEGs by replacing each instance of g_1 and/or g_2 in each PEG with a reference. As shown in Figure 2c, $\mathcal{OUT}_3$ is now represented as a reference to g_1 and $\mathcal{OUT}_4$ as two references to g_1 and g_2. $\mathcal{OUT}_5$ and $\mathcal{OUT}_6$ are stored as a hybrid set of g_1 and g_2 references together with *residue edges* that do not belong to any FCS. Details of this algorithm is discussed in §IV-D.

Dynamic Edge Pruning. Note that the pre-computed call graph may contain spurious calls due to the imprecision of the (inexpensive) points-to analysis used. To improve analysis precision, Chianina enables dynamic pruning of edges if our client is a pointer or alias analysis. Edge pruning can be easily done by checking the validity for edges connecting actual and formal parameters in the cloned control flow graph. The precise points-to set of the target variable computed by our system is used *on the fly* to determine whether such an edge is spurious. A spurious edge would not be traversed and hence everything reachable from it would not be traversed.

IV. Chianina Design and Implementation

We architect Chianina as an *disk-based, out-of-core* graph system running on a single machine — since static analysis is our application domain, the desired system should run on developers' working machines, providing support for their daily development tasks. This section first discusses how a developer can use Chianina and then its design.

A. Programming Model

Similarly to the monotone framework [66], [67], implementing a client analysis on Chianina requires two major tasks. First, the developer needs to follow a *frontend interface* to provide the graph representation for dataflow facts. Second, the developer implements two functions `combine` and `transfer`, which are used to merge dataflow facts at the control join points and compute dataflow facts at each statement, respectively.

As discussed earlier in §III-B, the frontend is a compiler pass that generates, by default, the CFG for each function, and each vertex of the CFG references another graph representing the dataflow fact at the vertex. The developer needs to create a subclass of class `DataflowFactGraph` to provide her own graph implementation. In the case of pointer/alias analysis, this subclass is PEG. The Chianina frontend also allows the developer to customize the CFG captured — by default, when vertices and edges of the CFG of each function are traversed by

the compiler pass, they get captured *as is*. The developer can define her own `onCFGVertex` and `onCFGEdge` functions to tell the frontend what to do when each CFG edge/vertex is traversed. For our pointer analysis, we reimplemented these two functions to capture a sparse def-use graph.

Algorithm 1: Chianina two-level parallel computation.

```
1  V ← {all vertices in the cloned GCFG}
2  G ← {all initialized dataflow facts}
3  [P_0:⟨F_0, G_0⟩, ..., P_i:⟨F_i, G_i⟩, ...] ← PARTITION(V, G)
4  repeat
5      scheduled ← SCHEDULE()
6      /*Level 1: BSP computation at partition level*/
7      for Partition P_i ∈ scheduled do in parallel
8          ⟨F_i, G_i⟩ ←LOAD(P_i)
9          PROCESSPARTITION (F_i, G_i)
10         COMPRESSFCS(G_i)
11     for Each partition P_i do in parallel
12         if Q_i ≠ ∅ then
13             F_i ← Q_i
14         if P_i ∈ scheduled then /*for loaded partitions*/
15             Write G_i, F_i back to disk
16             Delete P_i from memory
17 until ∀i, F_i = ∅
18 Procedure PROCESSPARTITION(F_i, G_i)
19 changeset ← ∅
20 /*Level 2: Async. dataflow computation at stmt level*/
21 for each CFG vertex k ∈ F_i do in parallel
22     Remove k from F_i
23     IN_k ← COMBINE(k)
24     Temp_k ← TRANSFER(IN_k)
25     if ¬ ISISOMORPHIC(Temp_k, OUT_k) then
26         OUT_k ← Temp_k
27         changeset ← changeset ∪ {k}
28         F_i ← F_i ∪ SUCCESSOR(k) \Mirror
29 /* Process CFG vertices with changed dataflow facts*/
30 foreach CFG vertex k ∈ changeset do
31     foreach s ∈ SUCCESSOR(k) do
32         if s is a mirror vertex then
33             Q_j ← {⟨s, OUT_k⟩} ∪ Q_j, where s ∈ P_j
```

B. Two-level Parallel Computation

Algorithm 1 provides Chianina's iterative computation algorithm. Chianina exploits parallelism at two levels: (1) bulk *synchronous* parallel computation (BSP) at the partition level (Line 7) and (2) *asynchronous* computation at the CFG vertex level (Line 21). The loop between Line 5 and Line 17 describes a typical BSP style computation — partitions scheduled to process are loaded and processed completely in parallel during each superstep (*i.e.*, loop iteration). Each partition $\mathcal{P}_i$ has three data structures: (1) $\mathcal{F}_i$ — the active CFG vertices that form the frontier for the partition, (2) $\mathcal{G}_i$ — the set of dataflow fact graphs, and (3) $\mathcal{Q}_i$ — the message queue. In the beginning, $\mathcal{F}_i$ contains all vertices in the partition (Line 3).

The partition-level BSP computation is done by the loop from Line 7–10. Chianina loads the active vertices in $\mathcal{F}_i$ and the dataflow fact graphs $\mathcal{G}_i$ of each scheduled partition $\mathcal{P}_i$ into memory (Line 8), processes the partition (Line 9), and finds and exploits frequent common subgraphs (Line 10).

Function PROCESSPARTITION describes the logic of processing of each partition that exploits parallelism at the (second) CFG-vertex level. Chianina iterates, in parallel, over the active CFG vertices in $\mathcal{F}_i$, applying the two user-defined functions COMBINE and TRANSFER on each vertex. The alias computation logic is done in TRANSFER. If the resulting PEG $Temp_k$ is *not isomorphic* to the previously computed $\mathcal{OUT}_k$ (Line 25), we record k into *changeset* and add k's CFG successors into the frontier set $\mathcal{F}_i$. It is clear that this parallel loop performs *asynchronous* computation — whenever a new active vertex is detected, it is added into $\mathcal{F}_i$ and *immediately* processed by a thread without any synchronization. Locks (omitted here) are used to guarantee data race freedom — no vertex will be processed simultaneously by multiple threads.

Asynchronous computation performs faster updates than synchronous computation at the cost of increased scheduling complexity. At the vertex level, since all CFG vertices of a partition are already in memory, asynchronous parallelism is a better fit as long as we can guarantee the data race freedom and atomicity of the transfer function execution for each vertex. However, at the partition level, our scheduler determines which partitions to load and run based on a set of already complex criteria, and hence, using BSP-style parallelism significantly simplifies our scheduler design.

Finally, the loop at Line 30 iterates over all CFG vertices whose dataflow facts have changed to find *mirror vertices* such as statement 4 in Figure 1a. In particular, we find the partition $\mathcal{P}_j$ that contains each mirror vertex s and puts its dataflow fact graph $\mathcal{OUT}_k$ into its message queue $\mathcal{Q}_j$ (Line 33). Later, when all scheduled partitions are done with their processing (Line 11), the synchronization phase starts (Line 11 – Line 16), updating each partition $\mathcal{P}_i$'s active vertex set $\mathcal{F}_i$ with the messages in $\mathcal{Q}_i$ (received from the processing of other partitions). At the end of each superstep, the updated $\mathcal{G}_i$ and $\mathcal{F}_i$ are written back to disk and removed from memory (Line 14) if partition $\mathcal{P}_i$ is currently in memory.

C. Partitioning and Scheduling

Partitioning. Chianina uses the vertex-centric edge-cut strategy [69] for effective partitioning, which assigns CFG vertices to partitions and cuts certain edges across partitions. Specifically, vertices of the global control flow graph are firstly divided into disjoint sets. A partition is then created by assigning all the edges whose source or destination vertex belongs to this set. There often exist edges of the form $x \rightarrow y$ that cross two partitions P_1 and P_2 (*e.g.*, $x \in P_1$ and $y \in P_2$). Chianina creates *mirror* vertices x' and y', and places the edges $x \rightarrow y'$ and $x' \rightarrow y$ into P_1 and P_2, respectively.

For each partition, its space is consumed by its CFG edges as well as dataflow fact graphs associated with its vertices (including mirror vertices). Dataflow fact graphs are maintained in a separate storage space from CFG edges. As a result of this partitioning scheme, for any vertex (except for mirrors) within a partition, Chianina can apply the transfer function

on it by accessing and updating its incoming and outgoing dataflow facts. For each vertex whose successor is a mirror vertex, when its associated dataflow fact is updated, the mirror vertex is marked as active. A message containing the vertex ID and its updated dataflow fact graph is sent to its containing partition, as shown in Line 33 in Algorithm 1.

Note that how to split GCFG nodes into disjoint sets determines the effectiveness of partitioning, which has further impact on the overall performance. Traditional graph partitioning schemes [71] aim to minimize the number of cuts across partitions, with the goal to save communication costs. However, those schemes do not consider the unique characteristics of our (flow-sensitive analysis) workload. For example, the computation performed by a flow-sensitive analysis follows the structure of the CFG. It is well-known in the program analysis community that the convergence speed of an iterative analysis is significantly affected by the order in which CFG vertices are visited [72]. Intuitively, desirable performance can be achieved if all predecessors of a CFG vertex have been processed before the vertex itself, because the transfer function can just use the latest updates from its predecessors.

Based on the insight, we propose a *balanced, topology-based* partitioning mechanism. Given the number of partitions (specified by the user as a parameter) and the total number of vertices in the GCFG, we first calculate the average number of vertices for each partition. Next, the partitioner traverses the GCFG in a *topological order* (a.k.a. reverse post-order of DFS traversal), starting from each entry vertex of the GCFG. The traversal continues until the number of vertices visited matches (roughly) the average number. Once a partition is generated, we repeat the same process by using another unvisited vertex as the root. Eventually, all partitions are produced with balanced sets of vertices that also follow the traversal order.

This algorithm works well for CFGs without cycles. To deal with cycles (induced by loops), we compute strongly connected components (SCCs for brevity) over the GCFG. The nodes within a SCC are connected to each other. As a result, the control flow graph with cycles turns out to be an acyclic graph with SCCs. The above algorithm can then be conducted over the acyclic graph to produce balanced partitions.

Scheduling. Similarly to the partitioning scheme, the scheduler also needs to take into account topology when deciding which partitions to load and process. Due to dependencies induced by inter-partition edges (say $x \rightarrow y$), one major goal of the scheduler is to schedule the processing of the partition containing x before that of the partition containing y, so that communication costs can be reduced and the algorithm can converge quickly. To this end, we devise a *priority queue based* scheduling mechanism. We assign each partition a priority, which is a function of (1) the number of its active vertices (*i.e.*, the size of $\mathcal{F}_i$) and (2) whether or not the partition is currently in memory. The more active vertices a partition has, the more updates can be generated during computation. Furthermore, if a partition is already in memory, processing it again in the next superstep can save the large cost of a memory-disk round trip, leading to increased efficiency.

Our scheduler selects a number N of partitions with the highest priority. The value of N is determined by (1) the amount of memory each partition is estimated to consume, (2) the total amount of available memory, and (3) the number of CPU cores. Our goal is to fully utilize the memory and CPU resources without creating extra stress.

D. FCS-based De-duplication

Although Chianina divides the input into smaller partitions to avoid running out of memory, partitions are still space-consuming especially because each CFG vertex carries a dataflow fact graph. Dataflow fact graphs exhibit both *temporal* and *spatial* locality — graphs belonging to connected CFG vertices are processed contiguously and have large overlap.

To exploit such overlaps, we propose a frequent-itemset-based approach to find frequent common subgraphs and perform de-duplication by maintaining only one instance for each FCS and replacing other instances with references. De-duplication (Line 10 in Algorithm 1) is conducted before writing dataflow facts back to disk. In particular, our algorithm models each dataflow fact graph (*e.g.*, PEG) as an itemset where each item is an edge. The graph miner discovers frequent itemsets, each of which occurs at least N times (*i.e.*, N is a threshold) among dataflow fact graphs in the same partition.

Once these FCSes are mined, we check each dataflow fact graph and see if it contains any FCSes. If it does, we replace each instance of each FCS with a reference, as illustrated in Figure 2c. Given multiple FCSes, there may exist multiple ways to conduct the placement. Given that the benefits of de-duplication are determined primarily by an FCSes' frequency and size. The higher these numbers are, the more benefit can be reaped. As such, Chianina assigns each FCS mined a priority score, computed as the product of its frequency and size. A greedy algorithm is then used to apply candidate FCSes in the descending order of their priority.

In Chianina, we leverage an off-the-shelf frequent itemset mining tool Eclat [70] to uncover FCSes. Although leveraging these FCSes significantly reduces the size of dataflow facts, it inevitably introduces overhead. With the growth in both the number and size of dataflow facts, the mining cost is non-trivial — it can take several minutes to run each mining task for large partitions in our experiments. To reduce the overhead, we can focus only on very frequent and/or very large FCSes by raising the mining thresholds. Moreover, we randomly sample the dataflow fact graphs in each partition, selecting no more than 10K graphs as our mining dataset. These two approaches collectively bring the overhead down to an acceptable percentage (*i.e.*, less than 5%).

E. Strong Update and Edge Deletion

As stated earlier, the dataflow transfer function `transfer` needs to be provided by the developer. For pointer/alias analysis, the `transfer` function not only applies the logic of *GEN* and *KILL*, but also discovers transitive edges on each PEG to compute an alias solution. The logic of *GEN* is straightforward — Rule 1–4 in §III-A clearly describes how new edges should

be added. The algorithm of computing the alias solution from a PEG is based on CFL-reachability [7], [73] (shown in Equation 5 and 6) and well-known to the community [65]. Hence, we do not include this algorithm in the paper. The logic of *KILL* (*i.e.*, edge deletion) involves strong update, which is crucial for achieving high precision of flow-sensitive analysis [74]–[76]. Since this logic is much trickier than that for edge addition, here we focus on the discussion of edge deletion.

Condition for Strong Update. Strong update can be enabled on pointer expression x such that x is guaranteed to refer to a single memory location (*i.e.*, *singleton*) throughout the execution. We follow [74] to identify our singleton set. The detailed algorithm is known and omitted from the paper to save space. Informally, a local or global variable is singleton except for the following cases: (1) dynamically allocated variables, where one abstract variable may correspond to multiple memory locations during execution; (2) local variables of recursive procedures (either directly or transitively recursive), where each variable may have multiple instances on the stack; and (3) array variables where usually only one element is updated.

Finding Edges to Delete. When such an expression (*e.g.*, $*p = v$) is defined, strong update may be performed because the value contained in the location l pointed-to by p changes. This removes the value-aliasing (Equation 5) between $*p$ and any pointer variables that previously receive their values from the location. On the PEG, two kinds of edges need be deleted: (1) all (direct and transitive) edges *going into* any pointer expressions referring to l, and (2) all (direct and transitive) edges *coming out of* any pointer expressions referring to l.

For (1), there are four sub-cases: (1.1) direct assignment edges going to expression $*p$, added due to a previous statement such as $*p = x$ — such a relationship no longer holds; (1.2) direct assignment edges going to expression $*q$ such that p and q *must alias*. p and q must alias if they both have only one and the same memory location o in their points-to set and o is a singleton memory location. We compute the *must-alias* information during our alias solution computation (*i.e.*, part of the transfer function); (1.3) transitive (V- or M-) edges going to expression $*p$ — these edges represent aliasing relationships between the old value inside $*p$ and another pointer expression and thus need to be deleted; and (1.4) transitive (V- or M-) edges going to expression $*q$ such that p and q *must alias*; these edges need to be deleted for similar reasons.

Note that we not only need to remove edges going into $*p$, but also need to remove edges *coming out* of $*p$. For example, for a direct edge coming out of $*p$ due to a previous statement $v = *p$, since $*p$ now contains a different value, v is not longer related to $*p$. Similarly, four sub-cases exist in (2). We need to remove (2.1) direct assignment edges coming out of expression $*p$, (2.2) direct assignment edges coming out of expression $*q$ such that p and q *must alias*, (2.3) transitive (V- or M-) edges coming out of expression $*p$, and (2.4) transitive (V- or M-) edges coming out of $*q$ such that p and q *must alias*.

V. Evaluation

Our evaluation focuses on the following three questions:

- Q1: How does Chianina perform? How does it compare to other analysis implementations? (§V-A)
- Q2: How effective are our de-duplication, partitioning, and scheduling? (§V-B)
- Q3: Is the extra precision gained from context- and flow-sensitivity useful in practice? (§V-C)

TABLE I: Static characteristics of subject programs

Subject	Version	#LoC	#Inlines	#V-CFG	#E-CFG	Description
Linux	5.2	17.5M	48.2M	440.2M	661.6M	OS
Firefox	67.0	7.9M	22.1M	282.8M	503.6M	web browser
PostgreSQL	12.2	1.0M	5.4M	39.3M	80.4M	database
OpenSSL	1.1.1	519K	3.3M	38.9M	91.0M	protocol
Httpd	2.4.39	196K	293K	2.6M	3.8M	web server

We selected five large software systems including the Linux kernel, Firefox, PostgreSQL, OpenSSL, and Apache Httpd server as our analysis subjects. We implemented two context- and flow-sensitive analyses on top of Chianina: a pointer/alias analysis discussed in the paper as an example as well as a null value flow analysis with context-sensitive heap tracking. The null-value analysis is conducted together with the pointer/alias analysis — because pointer information is needed to track flows into/out of the heap, this analysis implements its dataflow fact graph by augmenting the PEG representation from the pointer/alias analysis with additional types of vertices representing null or non-null values. These two pieces of information (pointer and value) are computed together by the transfer function *on the fly*, while past implementations of value-flow analysis often require pointer information *a priori*.

The Chianina-based implementations for the two analyses have **553** and **708** lines of C++ code, most of which are on the implementation of CFL-reachability and strong update. In contrast, a context-, flow-insensitive pointer analysis [74] (that supports strong update) has **2499** lines of C++ code, while the staged context-insensitive, flow-sensitive analysis for C [15] has **10,649** lines.

As discussed earlier, context sensitivity is achieved by aggressive function cloning. Table I reports the static characteristics of each subject including its version information, the number of lines of code excluding whitespace and comments (#LoC), the number of functions inlined (#Inlines), the numbers of CFG vertices (#V-CFG) and edges (#E-CFG) in the global control flow graph after cloning, and the type description.

All the experiments were conducted on a *commodity desktop* with an Intel Xeon W-2123 4-Core CPU, 16GB memory, and 1T SSD, running Ubuntu 16.04. This resource profile is consistent with that of developers' working machines in the real world.

A. Chianina Performance

Table II reports, for the two client analyses, a variety of performance statistics including numbers of partitions generated, numbers of iterations (supersteps) needed for convergence, numbers of PEGs generated, total numbers of vertices and edges in all PEGs, and total computation times. The three PEG-related columns report the information of PEGs finally produced by Chianina.

The numbers of partitions for large programs such as Linux and Firefox are greater than 100. Clearly, it would not have

TABLE II: Chianina performance: columns shown are numbers of partitions (#Part.), numbers of iterations needed to converge (#Ite.), total numbers of PEGs (#PEGs), total numbers of vertices (#V-PEGs) and edges (#E-PEGs) in PEGs contained in the GCFG, and analysis times used (Time), respectively.

	Context- and flow-sensitive alias analysis						NULL value flow analysis with alias tracking					
Subject	#Part.	#Ite.	#PEGs	#V-PEGs	#E-PEGs	Time	#Part.	#Ite.	#PEGs	#V-PEGs	#E-PEGs	Time
Linux	287	337	440.2M	5.9B	125.9B	20.7hrs	289	354	440.2M	6.1B	126.1B	22.1hrs
Firefox	149	181	282.8M	3.4B	83.4B	11.1hrs	149	189	282.8M	3.8B	83.8B	12.3hrs
PostgreSQL	34	43	39.3M	481.8M	13.7B	1.3hrs	42	45	39.3M	513.2M	13.7B	1.5hrs
OpenSSL	10	17	38.9M	354.0M	4.5B	40.9mins	11	19	38.9M	368.1M	4.5B	43.6mins
Httpd	1	1	2.6M	37.6M	585.1M	4.7mins	1	1	2.6M	41.2M	588.7M	5.0mins

been possible to scale the analysis to such large programs without our disk support. Overall, it took the two analyses 20.7 and 22.1 hours to process the entire Linux kernel in a context- and flow-sensitive fashion. These analyses converged much faster for smaller programs such as Httpd (in a few minutes), which can be successfully analyzed as a single partition.

Performance Breakdown. To better understand the performance, we further broke down each analysis execution into four phases – preprocessing (*i.e.*, partitioning), disk I/O (*i.e.*, reading/writing partitions), (in-memory) BSP computation, and FCS de-duplication – and measured the time spent on each phase. On average, these four phases (in the above order) take 3.76%, 6.52%, 86.78%, and 2.94% of the total running time. The in-memory BSP computation dominates the execution. This is expected because each iteration updates hundreds of millions of PEGs, each of which can have thousands of edges. This observation suggests that more CPU resources (*e.g.*, cores, GPUs, or cluster) should be enlisted to bring the cost down.

Existing Analyses. The goal of this comparison is to understand if our context- and flow-sensitive alias analysis is more scalable and efficient than state-of-the-art analysis implementations. However, the difficulty here is that we could not find any available implementation of the same analysis for C. Kahlon reports his context- and flow-sensitive pointer analysis in [11], but its implementation is not available. Hardekopf *et al.* [15], [16] and Lhotak and Chung [74] have both implemented variations of flow-sensitive but context-insensitive pointer analysis for C. Although their implementations are available online, they were developed a long time ago for deprecated versions of LLVM, which are incompatible with the subject programs and our operating system. Doop [35] is a Datalog-based context-sensitive analysis framework, but it only supports Java and does not have a C frontend. The only available tool we can directly run is SVF [49], which also performs flow-sensitive but context-insensitive pointer analysis.

Since no existing implementation for both context- and flow-sensitive pointer analysis was readily available for comparison, we implemented by ourselves the staged flow-sensitive pointer/alias analysis, by faithfully following the algorithm described in [15]. The original analysis in [15] does not consider context sensitivity, and hence, we added context sensitivity to our implementation by letting the analysis take as input the cloned GCFG, which is automatically context sensitive. We compared Chianina with this version in a fully context-sensitive, flow-sensitive manner. This reference implementation failed to analyze most programs except for Httpd in our benchmark set — it ran out of memory quickly in a few seconds. This is not surprising since holding the GCFG for large programs requires a huge amount of memory. For Httpd, the reference implementation (which is single-threaded) takes more than 20 minutes and is much slower than Chianina.

TABLE III: Performance comparison for context-insensitive, flow-sensitive pointer analysis; OOM indicates out-of-memory; - indicates runtime error.

	Linux	Firefox	PostgreSQL	OpenSSL	Httpd
Reference[15]	OOM	OOM	14.7mins	OOM	34.7s
SVF[49]	-	OOM	56.1s	OOM	8.3s
Chianina	1.9hrs	4.2hrs	3.9mins	25.7mins	11.5s

Next, we disabled context sensitivity in Chianina, enabling *direct* comparisons between Chianina, SVF, and the reference implementation of [15]. In this setting, all the three tools ran context-insensitive, flow-sensitive pointer analysis. To disable context sensitivity in Chianina, instead of cloning functions, we connected formal and actual parameters in trivial ways so that the generated CFG is context-insensitive. Table III reports the analysis times the three tools took to analyze the five programs. Without context sensitivity, the reference implementation still failed to analyze Linux, FireFox and OpenSSL due to out-of-memory errors. SVF ran out of memory for Firefox and OpenSSL, and crashed on Linux. For Httpd and PostgreSQL, all the tools successfully analyzed them. Chianina outperformed [15] thanks to parallel processing. For PostgreSQL, however, SVF achieved better performance than Chianina. This is easy to understand — many optimizations Chianina performs for scalability purposes (*e.g.*, preprocessing, scheduling, disk I/O, and FCS de-duplication) take time to run; if scalability is not a concern, these optimizations would only add overhead.

B. De-duplication, Partitioning and Scheduling

Fig. 3: Percentages in numbers of PEG edges, numbers of iterations (*i.e.*, supersteps) needed, and total time spent for Chianina + FCS, using Chianina - FCS as the baseline (100%).

To understand the performance impact of FCS de-duplication, we compared two versions of Chianina, one with de-duplication

enabled (Chianina + FCS) and another without (Chianina - FCS). We ran these two versions under the same configuration and inputs, and collected the relevant execution statistics. Figure 3 depicts the numbers of PEG edges, numbers of iterations needed for convergence, and total time spent for Chianina + FCS, as a fraction of those of Chianina - FCS (*i.e.*, the baseline). Note that since Httpd is a small program with only one single partition, we excluded it from the set for the FCS evaluation. As shown, de-duplication significantly improved all of these aspects. For example, the overall time is reduced by more than 30% on average when FCS de-duplication is enabled.

To understand the efficacy of our partitioning and scheduling algorithms, we collected the statistics in a similar manner by running two versions of Chianina, one with our partitioning and scheduling algorithm (Chianina + PS) and a second that uses default algorithms (Chianina - PS) — in particular, in the second version, we partitioned the GCFG using the min-cut algorithm [69] and scheduled random partitions (with active vertices) for processing in each superstep. We use Chianina - PS as the baseline and report the statistics for Chianina + PS as a fraction in relation to the baseline in Figure 4. The statistics considered include numbers of iterations needed for convergence, and time spent. As shown, our partitioning and scheduling algorithms are effective — they significantly improve the efficiency in all these aspects. For example, total running time is reduced by more than 40% by employing our intelligent CFG-structure-aware partitioning and scheduling.

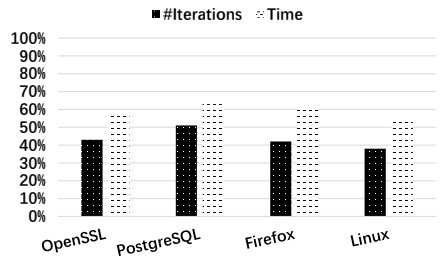


Fig. 4: Percentages in numbers of iterations and total time for Chianina + PS using Chianina - PS as baseline (100%); Httpd is not considered here as it has one single partition.

C. Usefulness of Gained Precision

To understand the accuracy of our context- and flow-sensitive alias analysis, we first examined each pointer dereference expression in *load* and *store* statements of the program, and measured *the sizes of their alias sets* — the smaller the better. For comparison, we considered three alias analyses – our context- and flow-sensitive analysis (CF), a context-insensitive, flow-sensitive analysis (F), and a context-sensitive, flow-insensitive analysis (C). Table IV reports the average sizes of alias sets for each analysis. Clearly, our flow- and context-sensitive analysis has the highest precision. The context-sensitive and flow-insensitive analysis (C) has the largest number (*i.e.*, lowest precision). This observation demonstrates that flow-sensitivity is more important than context-sensitivity for large C programs because analysis precision loses significantly if strong update is disabled.

TABLE IV: Sizes of alias sets of pointer expressions involved in *load* and *store* statements under three different pointer/alias analyses – our context-sensitive and flow-sensitive (CF), context-insensitive and flow-sensitive (F), context-sensitive and flow-insensitive (C).

	Load			Store		
Subject	CF	F	C	CF	F	C
Linux	0.24	0.54	7.20	0.31	0.95	5.54
Firefox	0.29	0.70	5.08	0.14	1.51	3.80
PostgreSQL	0.44	1.54	14.0	1.11	1.57	18.1
OpenSSL	0.84	4.06	13.22	0.08	0.25	0.73
Httpd	0.38	1.46	11.73	1.97	1.97	10.82

Although the size of alias set is a good precision metric, it does not show the usefulness of the increased precision. To answer the question "why is the extra precision needed", we implemented four static checkers: (1) a dataflow-based null pointer dereference checker, (2) a use-after-free checker, (3) a double-free checker, and (4) a belief analysis based null pointer dereference checker. The first three checkers were commonly used in the program analysis community [17], [36] and the last checker was used in the classical bug study done by Engler *et al.* [77], [78]. Note that the original versions of these checkers do not use any pointer information; they only use heuristics. To understand the effectiveness of our flow-sensitive alias analysis, we augmented these checkers with alias information provided by three analyses — our *context-* and *flow-sensitive* analysis (CF), context-sensitive, flow-insensitive analysis (C), and context-insensitive and flow-sensitive analysis (F). We next compared the numbers of warnings generated by these four checkers when augmented with each of these three pieces of alias information. Note that all the three analyses are *sound*, and hence using them would *not* make the checkers miss real bugs (*i.e.*, increasing the number of false negatives). As such, the fewer warnings generates, the better (because more false positives are pruned).

TABLE V: Checkers implemented including the dataflow analysis-based null pointer dereference (DF-Null), use-after-free (DF-UAF), double free (DF-DF) and the belief analysis-based null pointer dereference (BA-Null), their numbers of bugs reported by the baseline checkers augmented with our context- and flow-insensitive analysis (base+CF), context-sensitive and flow-insensitive (base+C), context-insensitive and flow-sensitive (base+F) on Linux kernel 5.2.

Checker	DF-Null	DF-UAF	DF-DF	BA-Null	total
base+CF	196	648	193	620	1657
base+C	217	1146	212	724	2299
base+F	211	805	200	663	1879

Table V reports these numbers — a large number of false warnings are pruned by enabling context and flow sensitivity. Similarly to the observation made earlier, flow sensitivity seems more important than context sensitivity as well in pruning false warnings. We sampled 100 pruned warnings for manual validation and confirmed that they are indeed false warnings.

VI. Conclusion

This paper presents Chianina, a novel evolving graph system for scalable context- and flow-sensitive analysis for C code.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, pp. 259–269.

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

[3] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: Is it worth it?" in *Proceedings of the 15th International Conference on Compiler Construction*, ser. CC'06, 2006, pp. 47–64.

[4] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," ser. PLDI '07. ACM, 2007, pp. 278–289.

[5] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. ACM, 2004, pp. 131–144.

[6] 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 *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL'15, pp. 83–95.

[7] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. ACM, 2006, pp. 387–400.

[8] G. Xu, A. Rountev, and M. Sridharan, "Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Springer-Verlag, 2009, pp. 98–122.

[9] D. Yan, G. Xu, and A. Rountev, "Demand-driven context-sensitive alias analysis for java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, pp. 155–165.

[10] 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," ser. PODS '05. ACM, 2005, pp. 1–12.

[11] V. Kahlon, "Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. ACM, 2008, pp. 249–259.

[12] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for c programs," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. ACM, 1995, pp. 1–12.

[13] Y. Su, D. Ye, and J. Xue, "Parallel pointer analysis with cfl-reachability," in *2014 43rd International Conference on Parallel Processing*, Sept 2014, pp. 451–460.

[14] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. ACM, 2005, pp. 59–76.

[15] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. IEEE Computer Society, 2011, pp. 289–298.

[16] ——, "Semi-sparse flow-sensitive pointer analysis," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. ACM, 2009, pp. 226–238.

[17] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. ACM, 1995, pp. 49–61.

[18] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up slicing," in *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '94. ACM, 1994, pp. 11–20.

[19] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. ACM, 2007, pp. 43–48.

[20] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. ACM, 2008, pp. 270–280.

[21] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, H. Xu, L. Wang, and X. Li, "Grapple: A graph system for static finite-state property checking of large-scale systems code," in *Proceedings of the Fourteenth European Conference on Computer Systems*, ser. EuroSys '19, 2019.

[22] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, pp. 693–706.

[23] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. ACM, 2011, pp. 17–30.

[24] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. ACM, 2010, pp. 305–315.

[25] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 485–495.

[26] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Scalability-first pointer analysis with self-tuning context-sensitivity," ser. ESEC/FSE 2018. ACM, 2018, pp. 129–140.

[27] ——, "Precision-guided context sensitivity for pointer analysis," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 141:1–141:29, Oct. 2018.

[28] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 239–248.

[29] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 148:1–148:29, Oct. 2019.

[30] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[31] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.

[32] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, pp. 1–41, Jan. 2005.

[33] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. ACM, 2013, pp. 423–434.

[34] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis, "Set-based pre-processing for points-to analysis," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. ACM, 2013, pp. 253–270.

[35] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. ACM, 2009, pp. 243–262.

[36] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017.

[37] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations," ser. ASPLOS '17. ACM, 2017, pp. 237–251.

[38] ——, "Synergistic analysis of evolving graphs," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 32:1–32:27, Oct. 2016.

[39] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '16. ACM, 2016, pp. 5:1–5:6.

[40] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb., pp. 249–263.

[41] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013, pp. 439–455.

[42] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal, "Pagerank on an evolving graph," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. ACM, 2012, pp. 24–32.

[43] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," ser. EuroSys '14. ACM, 2014, pp. 1:1–1:14.

[44] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *Trans. Storage*, vol. 11, no. 3, pp. 14:1–14:34, Jul. 2015.

[45] J. H. Reif and H. R. Lewis, "Symbolic evaluation and the global value graph," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. ACM, 1977, pp. 104–118.

[46] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90. ACM, 1990, pp. 296–310.

[47] M. Hind and A. Pioli, "Assessing the effects of flow-sensitivity on pointer alias analyses," in *Proceedings of the 5th International Symposium on Static Analysis*, ser. SAS '98, 1998, pp. 57–81.

[48] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," ser. POPL '91. ACM, 1991, pp. 55–66.

[49] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[50] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[51] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. ACM, 1998, pp. 97–105.

[52] T. B. Tok, S. Z. Guyer, and C. Lin, "Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers," in *Proceedings of the 15th International Conference on Compiler Construction*, ser. CC'06, 2006, pp. 17–31.

[53] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and implementation of sparse global analyses for c-like languages," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 12. ACM, 2012, p. 229238.

[54] J. Rodriguez and O. Lhoták, "Actor-based parallel dataflow analysis," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, ser. CC'11/ETAPS'11, 2011, pp. 179–197.

[55] D. Garbervetsky, E. Zoppi, and B. Livshits, "Toward full elasticity in distributed static analysis: The case of callgraph analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. ACM, 2017, pp. 442–453.

[56] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, "Parallelizing top-down interprocedural analyses," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 217–228.

[57] V. Nagaraj and R. Govindarajan, "Parallel flow-sensitive pointer analysis by graph-rewriting," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 13. IEEE Press, 2013, p. 1928.

[58] J. Zhao, M. G. Burke, and V. Sarkar, "Parallel sparse flow-sensitive points-to analysis," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. ACM, 2018, p. 5970.

[59] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York, NY: New York Univ. Comput. Sci. Dept., 1978. [Online]. Available: https://cds.cern.ch/record/120118

[60] B. R. Murphy and M. S. Lam, "Program analysis with partial transfer functions," in *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '00. ACM, 1999, pp. 94–103.

[61] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comput. Sci.*, vol. 167, no. 1-2, pp. 131–170, Oct. 1996.

[62] G. Xu and A. Rountev, "Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, pp. 225–236.

[63] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, vol. 56, 2016, pp. 22:1–22:26.

[64] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective context-sensitivity guided by impact pre-analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 14, 2014, p. 475484.

[65] X. Zheng and R. Rugina, "Demand-driven alias analysis for c," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. ACM, 2008, pp. 197–208.

[66] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. ACM, 1973, pp. 194–206.

[67] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Inf.*, vol. 7, no. 3, pp. 305–317, Sep. 1977.

[68] ——, "Global data flow analysis and iterative algorithms," *J. ACM*, vol. 23, no. 1, pp. 158–171, Jan. 1976.

[69] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. ACM, 2010, pp. 135–146.

[70] C. Borgelt, "Find frequent item sets with the eclat algorithm," http://www.borgelt.net/doc/eclat/eclat.html, 2017.

[71] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*. Cham: Springer International Publishing, 2016, pp. 117–158.

[72] K. D. Cooper, T. J. Harvey, and K. Kennedy, "Iterative data-flow analysis, revisited," Tech. Rep., 2004.

[73] J. Kodumal and A. Aiken, "The set constraint/cfl reachability connection in practice," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. ACM, 2004, pp. 207–218.

[74] O. Lhoták and K.-C. A. Chung, "Points-to analysis with efficient strong updates," ser. POPL '11. ACM, 2011, pp. 3–16.

[75] A. De and D. D'Souza, "Scalable flow-sensitive pointer analysis for java with strong updates," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Springer-Verlag, 2012, pp. 665–687.

[76] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," ser. FSE 2016. ACM, 2016, pp. 460–473.

[77] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, ser. OSDI'00, 2000.

[78] F. Brown, A. Nötzli, and D. Engler, "How to build static checking systems using orders of magnitude less code," ser. ASPLOS '16. ACM, 2016, pp. 143–157.