The Fine-Grained and Parallel Complexity of Andersen’s Pointer Analysis

ANDERS ALNOR MATHIASEN, Aarhus University, Denmark
ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

Pointer analysis is one of the fundamental problems in static program analysis. Given a set of pointers, the task is to produce a useful over-approximation of the memory locations that each pointer may point-to at runtime. The most common formulation is Andersen’s Pointer Analysis (APA), defined as an inclusion-based set of \( m \) pointer constraints over a set of \( n \) pointers. Scalability is extremely important, as points-to information is a prerequisite to many other components in the static-analysis pipeline. Existing algorithms solve APA in \( O(n^2 \cdot m) \) time, while it has been conjectured that the problem has no truly sub-cubic algorithm, with a proof so far having remained elusive. It is also well-known that APA can be solved in \( O(n^2) \) time under certain sparsity conditions that hold naturally in some settings. Besides these simple bounds, the complexity of the problem has remained poorly understood.

In this work we draw a rich fine-grained and parallel complexity landscape of APA, and present upper and lower bounds. First, we establish an \( O(n^3) \) upper-bound for general APA, improving over \( O(n^2 \cdot m) \) as \( n = O(m) \). Second, we show that even on-demand APA ("may a specific pointer \( a \) point to a specific location \( b \) ?") has an \( \Omega(n^3) \) (combinatorial) lower bound under standard complexity-theoretic hypotheses. This formally establishes the long-conjectured "cubic bottleneck" of APA, and shows that our \( O(n^3) \)-time algorithm is optimal. Third, we show that under mild restrictions, APA is solvable in \( \tilde{O}(n^{\omega}) \) time, where \( \omega < 2.373 \) is the matrix-multiplication exponent. It is believed that \( \omega = 2 + o(1) \), in which case this bound becomes quadratic. Fourth, we show that even under such restrictions, even the on-demand problem has an \( \Omega(n^2) \) lower bound under standard complexity-theoretic hypotheses, and hence our algorithm is optimal when \( \omega = 2 + o(1) \). Fifth, we study the parallelizability of APA and establish lower and upper bounds: (i) in general, the problem is P-complete and hence unlikely parallelizable, whereas (ii) under mild restrictions, the problem is parallelizable. Our theoretical treatment formalizes several insights that can lead to practical improvements in the future.

CCS Concepts: • Software and its engineering → Software verification and validation; • Theory of computation → Theory and algorithms for application domains; Program analysis.

Additional Key Words and Phrases: static pointer analysis, inclusion-based pointer analysis, fine-grained complexity, Dyck reachability

ACM Reference Format:

Authors’ addresses: Anders Alnor Mathiasen, Aarhus University, Aabogade 34, Aarhus, 8200, Denmark, au611509@uni.au.dk; Andreas Pavlogiannis, Aarhus University, Aabogade 34, Aarhus, 8200, Denmark, pavlogiannis@cs.au.dk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.
2475-1421/2021/1-ART1 $15.00
https://doi.org/
Fig. 1. A program where analysis depends on aliasing (left) and the four types of statements in APA (right).

1 INTRODUCTION

Programs execute by allocating memory for storing data and manipulating pointers to that memory. Pointer analysis takes a static view of a program’s heap and asks the question “given a pointer \(a\), what are the memory locations that \(a\) may point-to at program runtime?” Such information is vital to almost all questions addressed by static analyses in general [Ghiya et al. 2001; Hind 2001], hence many static analyzers begin with some form of pointer analysis. In particular, for an analysis to be useful, it needs to be able to determine aliasing, i.e., whether two pointers may be pointing to the same memory location. For example, in the program of Figure 1, the value of \(c\) depends on whether \(a\) and \(b\) are aliases. Naturally, aliasing is decided by (implicitly or explicitly) computing whether the intersection of the points-to space of the two pointers is empty.

As usual in static analyses, points-to information can be modeled at various degrees of precision, which has consequences on the decidability and complexity of the problem. Flow-sensitive formulations, which take into account the order of execution of pointer-manipulation statements, are typically intractable, with results ranging from undecidability [Ramalingam 1994], to PSPACE-completeness [Chakaravarthy 2003] and NP-/co-NP-hardness [Landi and Ryder 1991]. In contrast, flow-insensitive formulations can be viewed as relational approaches that ignore the order of execution, and typically result in more tractable algorithmic problems. Another feature that affects complexity is the level of indirection (i.e., how many nested dereferences can occur in a single statement) [Horwitz 1997], and thus is typically kept small. Flow-insensitive analyses are faster and achieve remarkable precision in practice [Blackshear et al. 2011; Das et al. 2001; Shapiro and Horwitz 1997]. This sweet spot between efficiency and precision has made flow-insensitive analyses dominant over alternatives. Popular approaches in this domain are inclusion-based [Andersen 1994], equality-based [Steensgaard 1996] and unification-based [Das 2000]. We refer to [Smaragdakis and Balatsouras 2015] for an excellent exposition.

Andersen’s pointer analysis. The most commonly used and actively studied formulation is Andersen’s Pointer Analysis (APA) [Andersen 1994]. The input is a set of \(n\) pointers and \(m\) statements of the four types shown in Figure 1. The solution to the analysis is the least fixpoint of a set of inclusion constraints between the points-to sets of the pointers (see Section 2.1 for details). APA has been the subject of a truly huge body of work, ranging from adoptions to diverse programming languages [Jang and Choe 2009; Lyde et al. 2015; Sridharan and Fink 2009], extensions to incorporate various features (e.g., context/flow/field-sensitivity) [Hardekopf and Lin 2011; Hirzel et al. 2004; Pearce et al. 2004; Whaley and Lam 2002] and implementations in various frameworks [Wal 2003; Lhotáš and Hendren 2003; Vallée-Rai et al. 1999], to name a few.

Complexity. The standard statement in the literature with regards to the complexity of APA is that it is cubic. However, the parameter (\(n\) or \(m\), for \(n\) pointers and \(m\) statements) on which this cubic bound is expressed is often left unspecified, leading to a variety of statements. The standard expression is an \(O(m^3)\) bound [Melski and Reps 2000; Möller and Schwartzbach 2018], by reducing
the problem to \( m \) inclusion set constraints [McAllester 1999]. Other works give a more refined bound of \( O(n^2 \cdot m) \) [Kodumal and Aiken 2004; Pearce et al. 2004] which is an improvement over \( O(m^3) \) as \( n = O(m) \). Note that, in general, \( m \) can be as large as \( \Theta(n^2) \), hence both types of statements result in worst-case dependency on \( n \) that is at least quartic, as already observed in [Kodumal and Aiken 2004]. Finally, in most literature, the core algorithm constructs incrementally the closure of a flow graph by introducing edges dynamically. However, the complexity analysis often ignores the cost for inserting edges. As already noted by others [Heintze and McAllester 1997; Sridharan and Fink 2009], the cost of edge insertion needs to be accounted for when analyzing the complexity. Under this consideration, the complexity of all these approaches is at least quartic in \( n \).

The need for a faster algorithm is apparent from the long literature of heuristics [Aiken et al. 1997; Berndl et al. 2003; Dietrich et al. 2015; Fähndrich et al. 1998; Hardekopf and Lin 2007; Heintze and Tardieu 2001b; Pearce et al. 2004; Pek and Madhusudan 2014; Rountev and Chandra 2000; Su et al. 2000; Vedurada and Nandivada 2019; Xu et al. 2009]. Despite all efforts, no algorithmic breakthrough below the cubic bound has been made for over 25 years. In some cases, the complexity of APA can be reduced to quadratic [Sridharan and Fink 2009]. This reduction holds when the instances adhere to certain sparsity conditions, which hold naturally in some settings.

**Exhaustive vs on-demand.** One popular approach to reducing running time lies on the observation that we are typically interested in on-demand variants of the problem. That is, we want to decide whether \( a \) may point to \( b \) for a given pointer \( a \) and memory location \( b \), rather than the exhaustive case that computes the points-to set of every pointer. Under this restriction, many techniques devise analysis algorithms that aim to solve on-demand APA faster [Chatterjee et al. 2018; Heintze and Tardieu 2001a; Lu et al. 2013; Sridharan et al. 2005; Sui and Xue 2016; Vedurada and Nandivada 2019; Zhang et al. 2013; Zheng and Rugina 2008]. On the theoretical side, it is an open question whether on-demand analysis has lower complexity than exhaustive analysis.

**Lower bounds and cubic bottlenecks.** Despite the complete lack of algorithmic improvements for APA for over 25 years, no lower bounds are known. The two basic observations for exhaustive APA are that (i) the output has size \( \Theta(n^2) \), which leads to a trivial similar lower bound for running time, and (ii) the problem is at least as hard as computing the transitive closure of a graph [Sridharan and Fink 2009; Zhang 2020]. For the on-demand case, no lower bound is known. APA is often reduced to a specific framework of set constraints [Heintze 1992; Su et al. 2000], which is computationally equivalent to CFL-Reachability [Melski and Reps 2000]. Set constraints and CFL-Reachability are known to have cubic lower bounds [Heintze and McAllester 1997]. Unfortunately, these lower bounds do not imply any lower bound for APA as the reduction is only one way (i.e., from APA to set constraints). The recurrent encounter of cubic complexity is frequently referred to as the “cubic bottleneck in static analysis”, though the bottleneck is only conjectured for APA, with a proof so far having remained elusive.

**Parallelization.** The demand for high-performance static analyses has lead various researchers to implement parallel solvers of APA [Blaß and Philippsen 2019; Liu et al. 2019; Méndez-Lojo et al. 2012; Méndez-Lojo et al. 2010; Su et al. 2014; Wang et al. 2017]. Despite their practical performance, the parallelizability of APA has remained open on the theoretical level. In contrast, the richer problem of set constraints is known to be non-parallelizable, via its reduction to CFL-Reachability [Melski and Reps 2000] which is known to be P-complete [Reps 1996].
1.1 Our Contributions

In this work, we draw a rich fine-grained complexity landscape of Andersen’s Pointer Analysis, by resolving open questions and improving existing bounds. We refer to Section 3 for a formal presentation of our main results as well as a discussion on their implications to the theory and practice of pointer analysis.

Main contributions. Consider as input an APA instance \((A, S)\) of \(n = |A|\) pointers and \(m = |S|\) statements. Our main contributions are as follows.

(1) We show that Exhaustive APA is solvable in \(O(n^3)\) time, regardless of \(m\). To our knowledge, this gives the sharpest cubic bound on the analysis, as it holds even when \(m = \Theta(n^2)\).

(2) We show that even On-demand APA does not have a (combinatorial) sub-cubic algorithm (i.e., with complexity \(O(n^{3-\epsilon})\) for some fixed \(\epsilon > 0\) under the combinatorial BMM hypothesis. This formally proves the long-conjectured cubic bottleneck for APA.

(3) We consider a bounded version of Exhaustive APA where points-to information is witnessed by bounding the execution of type-4 (Figure 1) statements by a poly-logarithmic bound. We show that bounded Exhaustive APA is solvable in \(\tilde{O}(n^{\omega})\) time, where \(\tilde{O}\) hides poly-logarithmic factors and \(\omega\) is the matrix-multiplication exponent. It is known that \(\omega < 2.373\) [Le Gall 2014], hence our algorithm is sub-cubic.

(4) It is believed that \(\omega = 2 + o(1)\), in which case our previous bound becomes nearly quadratic. We complement this result by showing that even On-demand APA with witnesses that are logarithmically bounded (i.e., a simpler problem than that in Item 3) does not have a sub-quadratic algorithm (i.e., with complexity \(O(n^{2-\epsilon})\) for some fixed \(\epsilon > 0\) under the Orthogonal Vectors hypothesis [Williams 2019]. Hence, our algorithm for Item 3 is optimal when \(\omega = 2 + o(1)\).

(5) We show that APA is P-complete, and hence unlikely parallelizable. On the other hand, we show that bounding APA as in Item 4 is in NC, and hence highly parallelizable.

Technical contributions. Our main theoretical results rely on a number of technical novelties that might be of independent interest.

(1) Virtually all existing algorithms for APA represent the analysis as a flow-graph that captures inclusion constraints between pointers. In contrast, we develop a Dyck-graph representation over the Dyck language of 1 parenthesis type \(D_1\), which allows us to develop new insights for the problem, and establish upper and lower bounds.

(2) We show that Dyck-Reachability over \(D_1\) can be solved in time \(\tilde{O}(n^\omega)\), where \(\omega\) is the matrix-multiplication exponent, by a purely combinatorial reduction of the problem to \(O(\log^2 n)\) matrix-multiplications.

(3) Our lower bounds are based on fine-grained complexity, an emerging field in complexity theory that establishes relationships between problems in P. We believe that this field can have an important role in understanding and optimizing static program analyses. Our work makes some of the first steps in this direction.

2 PRELIMINARIES

In this section we give a formal presentation of Andersen’s pointer analysis and develop some general notation. We also define Dyck graphs and show how reachability relationships in such graphs can be used to represent points-to relationships between pointers. Finally, we present the main theorems of this paper. Given a number \(n \in \mathbb{N}\), we denote by \([n]\) the set \(\{1, 2, \ldots, n\}\).
<table>
<thead>
<tr>
<th>Type</th>
<th>Statement</th>
<th>Inclusion Constraint</th>
<th>Operational Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a = b$</td>
<td>$[b] \subseteq [a]$</td>
<td>$[a] \leftarrow [a] \cup {b}$</td>
</tr>
<tr>
<td>2</td>
<td>$a = &amp; b$</td>
<td>$b \in [a]$</td>
<td>$[a] \leftarrow [a] \cup {b}$</td>
</tr>
<tr>
<td>3</td>
<td>$a = * b$</td>
<td>$\forall c \in [b]: [c] \subseteq [a]$</td>
<td>$[a] \leftarrow [a] \cup {\bigcup_{c \in [b]} [c]}$</td>
</tr>
<tr>
<td>4</td>
<td>$* a = b$</td>
<td>$\forall c \in [a]: [b] \subseteq [c]$</td>
<td>$\forall c \in [a]: [c] \leftarrow [c] \cup {b}$</td>
</tr>
</tbody>
</table>

Table 1. The four types of statements of APA, the inclusion constraints they generate and the associated operational semantics.

### 2.1 Andersen’s Pointer Analysis

We begin with giving the formal definition of Andersen’s pointer analysis, as well as a bounded version of the problem.

**Andersen’s pointer analysis (APA).** An instance of APA is a pair $(A, S)$, where $A$ is a set of $n$ pointers\(^1\) and $S$ is a set of $m$ statements. Each statement has one of the four types shown in Table 1. Conceptually, the pointers may reference memory locations during the runtime of a program which uses the statements to manipulate the pointers.

1. A type 1 statement $a = b$ represents pointer assignment.
2. A type 2 statement $a = \& b$ represents making $a$ point to the location of $b$.
3. A type 3 statement $a = * b$ represents an indirect assignment $a = c$, where $c$ is pointed by $b$.
4. A type 4 statement $* a = b$ represents an indirect assignment $c = b$, where $c$ is pointed by $a$.

As standard practice, more complex statements such as $* a = * b$ have been normalized by introducing slack pointers [Andersen 1994; Horwitz 1997; Møller and Schwartzbach 2018]. We assume wlog that $S$ does not contain the same statement twice, and hence $m = O(n^2)$ \(^2\). Given some $i \in [4]$, we denote by $S_i$ the statements of $S$ of type $i$. Given a pointer $a$, we let $[a] \subseteq A$ be the points-to set of $a$. Typically in pointer analysis, $[a]$ is an over-approximation of the locations that $a$ can point-to during the lifetime of a program. In Andersen’s inclusion-based pointer analysis, the sets $[a]$ are defined as follows. Each statement generates an inclusion constraint between various points-to-sets, as shown in Table 1. The solution to APA is the smallest assignment $\{[a] \rightarrow 2^A\}_{a \in A}$ that satisfies all constraints.

**Exhaustive vs on-demand.** As standard in the literature, we distinguish between the exhaustive and on-demand versions of the problem. In each case, the input is an instance of APA. The Exhaustive APA problem asks to compute the points-to set of every pointer $a \in A$. The On-demand APA problem asks to compute whether $b \in [a]$ for a given pair of pointers $a, b \in A$. Hence, On-demand APA is a simplification of Exhaustive APA where the size of the output is a single bit, as opposed to $\Theta(n^2)$ bits required to output the points-to set of every pointer. The two variants can be viewed as analogues to the all-pairs and single-pair formulations of graph problems (e.g., reachability).

**Operational semantics.** Since the statements in APA come out of programs, it is convenient to consider them as executable instructions and assign simple operational semantics to them. The semantics are over a global store $\cdot : A \rightarrow 2^A$ that maps every pointer to its points-to set, which is initially empty. Executing one statement corresponds to updating the store as shown in Table 1.

---

\(^1\)Although in practice not all variables are pointers, we will use this term liberally for simplicity of presentation.

\(^2\)Note that the analyzed program might indeed contain the same statement twice. Generating the APA instance $(A, S)$ from the program is performed in time linear in the size of the program, after which the input is in the assumed form.
\[ b = \&a, \quad d = \&a \quad 1. \ b = \&a \quad 5. \ d = \star e \quad 1. \ [b] \leftarrow \{a\} \quad 5. \ [d] \leftarrow \{c, a\} \]

\[ c = \&b, \quad d = \&c \quad 2. \ c = \&b \quad 6. \ \star d = c \quad 2. \ [c] \leftarrow \{b\} \quad 6. \ [a] \leftarrow \{b\} \]

\[ \star d = c, \quad d = \star e \quad 3. \ d = \&c \quad 7. \ d = a \quad 3. \ [d] \leftarrow \{c\} \quad 7. \ [d] \leftarrow \{c, a, b\} \]

\[ e = \star d, \quad e = \&f \quad 4. \ e = \star d \quad 8. \ \star d = c \quad 4. \ [e] \leftarrow \{b\} \quad 8. \ [b] \leftarrow \{a, b\} \]

Fig. 2. An instance of APA (left), a witness program for \( b \in [b] \) (middle), and the updates to the store while executing the witness (right). Note that the statement \( e = \&f \) is not used, while \( \star d = c \) is used twice.

This operational view already hints an (albeit inefficient) algorithm for solving APA, namely, by iteratively executing some statement until no execution modifies the store.

**Witnesses.** The operational semantics allow us to define witnesses of points-to relations. Given two pointers \( a, b \in A \), a witness program (or simply, witness) for \( b \in [a] \) is a sequence of statements from \( S \) that results in \( b \in [a] \). See Figure 2 for an illustration.

**Bounded APA.** Motivated by practical applications, we introduce a bounded version of APA that restricts the length of witnesses. Consider two pointers \( a, b \in A \) and a witness program \( P \) for \( b \in [a] \). Given some \( i \in [4] \) and \( j \in \mathbb{N} \), we say that \( P \) is \((i, j)\)-bounded if \( P \) executes at most \( j \) statements of type \( i \). For example, the witness for \( b \in [b] \) in Figure 2 is \((3, 2)\)-bounded but not \((2, 2)\)-bounded. The problem of \((i, j)\)-bounded APA asks for a solution to the inclusion constraints of APA such that for any two pointers \( a, b \) with \( b \notin [a] \), any witness program that results in \( b \in [a] \) executes more than \( j \) statements of type \( i \).

**Remark 1.** The boundedness of \((i, j)\)-bounded APA is only one-way, i.e., for relationships of the form \( b \notin [a] \) and not of the form \( b \in [a] \). In particular, it is allowed to have \( b \in [a] \) even if this is witnessed only by programs that execute statements of type \( i \) more than \( j \) times.

Bounded versions of APA do not necessarily have a unique solution, e.g., if the shortest witness for \( a \) pointing to \( b \) exceeds the bound, we can have \( b \in [a] \) or \( b \notin [a] \). However, any solution suffices as long as (i) every points-to relationship \( b \in [a] \) reported has a witness, and (ii) all points-to relationships that have a bounded witness are reported (wrt the given bound). Similar techniques for witness bounding are used widely in practice in order to speed up static analyses.

### 2.2 Dyck Reachability and Representation of Andersen’s Pointer Analysis

Here we develop some notation on Dyck languages and Dyck reachability, and use it to represent instances of APA as Dyck graphs.

**Dyck languages.** Given a non-negative integer \( k \in \mathbb{N} \), we denote by \( \Sigma_k = \{\varepsilon\} \cup \{\alpha_1, \overline{\alpha}_1\}_{i=1}^k \) a finite alphabet of \( k \) parenthesis types, together with a null element \( \varepsilon \). We denote by \( \mathcal{D}_k \) the Dyck language over \( \Sigma_k \), defined as the language of strings generated by the following context-free grammar \( G_k \):

\[
S \rightarrow S \mid S \alpha_1 \overline{\alpha}_1 \mid \cdots \mid S \alpha_k \overline{\alpha}_k \mid \varepsilon
\]

In words, \( \mathcal{D}_k \) contains all strings where parentheses are properly balanced. In this work we focus on the special case where \( k = 1 \), i.e., we have only one parenthesis type. To capture the relationship between \( \mathcal{D}_1 \) and APA, we will let \( \alpha_1 \) be \( \& \) and \( \overline{\alpha}_1 \) be \( \star \). This relationship will become clearer later in this section.

**Dyck graphs.** A Dyck graph \( G = (V, E) \) is a digraph where edges are labeled with elements of \( \Sigma_1 \), i.e., \( E \subseteq V \times V \times \Sigma_1 \) and edges have the form \( \tau = (a, b, \lambda) \). Often we will only be interested in
the endpoints $a, b$ of an edge, in which case we represent $\tau = (a, b)$, and we will denote by $\lambda(\tau)$ the label of $\tau$. The label $\lambda(P)$ of a path in $G$ is the concatenation of the labels along the edges of $P$. We often represent $P$ from $a$ to $b$ graphically as $a \xrightarrow{\lambda(P)} b$, and, given some $i \in \mathbb{N}$, we write $a \xrightarrow{i\lambda} b$ (resp., $a \xrightarrow{\infty} b$) to denote a path $P: a \xrightarrow{\lambda} b$ with label $i$ consecutive symbols $\&$ (resp., $*$), possibly interleaved with $e$ symbols. We say that $b$ is Dyck-reachable (or $D$-reachable) from $a$ if there exists a path $P: x \xrightarrow{\lambda(P)} y$ with $\lambda(P) \in D_1$. We say that $b$ flows into $a$ via a node $c$ if (i) we have $b \xrightarrow{\&} c$, and (ii) $a$ is $D$-reachable from $c$. The $D_1$-Reachability problem takes as input a Dyck graph and asks to return all pairs of nodes $(b, a)$ such that $a$ is $D$-reachable from $b$.

**Graph representation of APA.** For convenience, we frequently represent instances of APA using Dyck graphs. In particular, given an instance $(A, S)$ of APA, we use a Dyck graph $G = (A, E)$ where $E$ represents all statements in $S \setminus S_4$. In particular, we have the following edges.

1. For every type 1 statement $a = b$, we have $b \xrightarrow{e} a$ in $E$.
2. For every type 2 statement $a = \&b$, we have $b \xrightarrow{\&} a$ in $E$.
3. For every type 3 statement $a = \*b$, we have $b \xrightarrow{*} a$ in $E$.

The instance $(A, S)$ is represented as a pair $(G, S_4)$ where $G = (A, E)$ is the Dyck graph and $S_4$ is the type-4 statements of $S$. See Figure 3 for an illustration. The motivation behind this representation comes from the following lemma, which establishes a correspondence between paths in Dyck graphs and APA without statements of type 4.

**Lemma 2.1.** Consider the Dyck graph representation $(G = (E, V), S_4)$ of $(A, S)$, and the modified APA instance $(A, S \setminus S_4)$. For every two pointers $a, b \in A$, we have $b \in \llbracket a \rrbracket$ in $(A, S \setminus S_4)$ iff $b$ flows into $a$ in $G$.

**Resolved Dyck graphs.** Consider an APA instance $(A, S)$ and the corresponding Dyck-graph representation $(G = (A, E), S_4)$. The **resolved Dyck graph** $\overline{G} = (A, \overline{E})$ is a Dyck graph where $\overline{E}$ is the smallest set that satisfies the following conditions.

1. $E \subseteq \overline{E}$.
2. For every statement $\*a = b$, for every node $c$ that flows into $a$ in $\overline{G}$, we have $b \xrightarrow{e} c$.

Intuitively, all type 4 statements have been resolved as $e$-edges in $\overline{G}$. See Figure 3 for an illustration. The following lemma follows directly from Lemma 2.1.

**Lemma 2.2.** For every two pointers $a, b \in A$, we have $b \in \llbracket a \rrbracket$ iff $b$ flows into $a$ in $\overline{G}$.

**Intuition behind the Dyck-graph representation.** Virtually all algorithms for APA in the literature use a flow graph for representing inclusion relationships between pointers. Pointer inclusion occurs when the analysis discovers that, for two pointers $a, b$ we have $[b] \subseteq [a]$, represented via an edge $b \rightarrow a$ in the flow graph.

Our Dyck graph $G$ is a richer structure compared to the standard flow graph. In fact, we obtain the (initial) flow graph if we remove from $G$ edges representing pointer references (labeled with $\&$) and dereferences of type 3 (labeled with $*$). The only information missing from $G$ is statements of type 4. The analysis can be seen as iteratively discovering type 4 statements $\*a = b$ and inserting an edge $b \xrightarrow{e} c$ in $G$. This process terminates with the resolved graph $\overline{G}$. Lemma 2.2 implies that, at that point, all points-to information can be expressed as flows-into relationships in $\overline{G}$.
3 SUMMARY OF MAIN RESULTS

We are now ready to present our main theorems, followed by a discussion on their implications to the theory and practice of pointer analysis. In later sections we develop the proofs.

Cubic upper-bound of APA. It is well stated that APA can be solved in cubic time. However, “cubic” refers to the size of the input, and typically has the form $O(m^3)$ [Melski and Reps 2000; Møller and Schwartzbach 2018] or $O(n^2 \cdot m)$ [Kodumal and Aiken 2004; Pearce et al. 2004], for $n$ pointers and $m$ statements. Note that $m$ can be as large as $\Theta(n^2)$, which yields the bound $O(n^4)$, as already observed in [Kodumal and Aiken 2004]. Our first theorem shows that in fact, the problem is solvable in $O(n^3)$ time regardless of $m$. Although we do not consider this our major result, we are not aware of a proven $O(n^3)$ bound with $n$ being the number of pointers. We also hope that the theorem will provide a future reference for a formal $O(n^3)$ complexity statement for APA.

Theorem 3.1. Exhaustive APA is solvable in $O(n^3)$ time, for any $m$, where $n$ is the number of pointers and $m$ is the number of statements.

Cubic hardness of APA. Given the cubic upper-bound of Theorem 3.1, it is natural to ask whether sub-cubic algorithms exist for the problem, i.e., algorithms with running time $O(n^{3-\epsilon})$, for some fixed $\epsilon > 0$. Indeed, the rich literature of heuristics (e.g., [Aiken et al. 1997; Berndl et al. 2003; Dietrich et al. 2015; Fähndrich et al. 1998; Hardekopf and Lin 2007; Heintze and Tardieu 2001b; Pearce et al. 2004; Pek and Madhusudan 2014; Rountev and Chandra 2000; Su et al. 2000; Vedurada and Nandivada 2019; Xu et al. 2009]) is indicative of the need for such an improvement. On the other hand, no lower-bound has been known. In fine-grained complexity, there is a widespread distinction between combinatorial and algebraic algorithms. The most famous combinatorial lower bound is for Boolean Matrix Multiplication (BMM). The respective hypothesis states that there is no combinatorial $O(n^{3-\epsilon})$ algorithm for multiplying two $n \times n$ Boolean matrices, for any fixed $\epsilon > 0$. The BMM hypothesis has formed the basis for many lower bounds in graph algorithms, verification, and static analysis [Abboud and Vassilevska Williams 2014; Bansal and Williams 2009; Chatterjee et al. 2018, 2016; Williams and Williams 2018].

Given the combinatorial nature of APA, we examine whether combinatorial sub-cubic improvements are possible. First, note that the edge set $\{ (x_i, x_j) \}$ of a digraph can be represented as a set of pointer assignments $\{ (x_j = x_i) \}$. This observation leads us to the following remark.

Remark 2. Even without statements of type 3 and type 4 (i.e., without pointer dereferences), Exhaustive APA is at least as hard as computing all-pairs reachability in a digraph. Since, under the BMM hypothesis, all-pairs reachability does not have a combinatorial sub-cubic algorithm, the same lower-bound follows for Exhaustive APA.
On the other hand, single-pair reachability is solvable in linear-time in the size of the graph, and is thus considerably easier than its all-pairs version. Hence, the relevant question is whether On-demand APA (i.e., given pointers \(a, b\), is it the case that \(b \in [a]\)?) has sub-cubic complexity. We answer this question in negative.

**Theorem 3.2.** On-demand APA has no sub-cubic combinatorial algorithm under the BMM hypothesis.

Note that Theorem 3.2 indeed relates a problem with output size \(\Theta(1)\) (On-demand APA) to a problem with output size \(\Theta(n^2)\) (BMM). The theorem has four implications. First, it establishes formally the long-conjectured “cubic bottleneck” for APA. Second, it shows that the algorithm of Theorem 3.1 is optimal even for On-demand APA, as far as combinatorial algorithms are concerned. Third, it indicates that the hardness of APA does not come from the requirement to produce large-sized outputs (i.e., of size \(\Theta(n^2)\) for the points-to set of each pointer), as sub-cubic complexity is also unlikely for constant-size outputs. Fourth, it shows that all on-demand analyses (e.g., [Chatterjee et al. 2018; Heintze and Tardieu 2001a; Lu et al. 2013; Sridharan et al. 2005; Vedurada and Nandivada 2019; Zhang et al. 2013; Zheng and Rugina 2008]), which attempt to reduce complexity by avoiding the exhaustive computation of all points-to sets, can only provide heuristic improvements without any guarantees.

**Bounded APA.** Given the hardness of APA under Theorem 3.2, we next seek mild restrictions that allow for algorithmic improvements below the cubic bound. Perhaps surprisingly, we show that bounding the number of times statements of type 4 are executed suffices.

**Theorem 3.3.** For all \(j \in \mathbb{N}^+\), \((4, j)\)-bounded Exhaustive APA is solvable in \(\tilde{O}(n^\omega \cdot j)\) time, where \(n\) is the number of pointers and \(\omega\) is the matrix-multiplication exponent. In particular, \((4, \tilde{O}(1))\)-bounded Exhaustive APA is solvable in \(\tilde{O}(n^\omega)\) time.

Here \(\tilde{O}\) hides poly-logarithmic factors (i.e., factors of the form \(\log^c n\), for some constant \(c\)). It is known that \(\omega < 2.373\) [Le Gall 2014], hence the bound is sub-cubic. Besides its theoretical interest, Theorem 3.3 also has practical relevance, as it reduces the problem to a small number of matrix multiplications. First, some sub-cubic algorithms for matrix multiplication, like Strassen’s [Strassen 1969], often lead to observable practical speedups over simpler, cubic algorithms [Huang et al. 2016; Huss-Lederman et al. 1996]. Second, this reduction can take advantage of both highly optimized practical implementations for matrix multiplication [Kaporin 1999; Laderman et al. 1992] and specialized hardware [Dave et al. 2007]. Third, the algorithm behind Theorem 3.3 is an “anytime algorithm”, in the spirit of [Boddy 1991; Chatterjee et al. 2015]. The algorithm computes \((4, j)\)-bounded Exhaustive APA iteratively for increasing values of \(j\). It can be terminated in any iteration \(j\) according to the runtime requirements of the analysis. At that point, the algorithm has executed for at most \(\tilde{O}(n^\omega \cdot j)\) time, and is guaranteed to have computed all points-to relationships as witnessed by \((4, j)\)-bounded programs. Hence, (i) a timeout does not waste analysis time, and (ii) the obtained results provide measurable completeness guarantees.

It is believed that \(\omega = 2 + o(1)\), in which case Theorem 3.3 yields a quadratic bound. Given such an improvement, a natural question is whether sub-quadratic algorithms are possible when we restrict our attention to witnesses that are poly-logarithmically bounded. Clearly this is not possible for Exhaustive APA, as the size of the output can be \(\Theta(n^2)\), but the question becomes interesting in the case of On-demand APA that has output size \(\Theta(1)\). We answer this question in negative.
Theorem 3.4. (All, $\tilde{O}(1)$)-bounded On-demand APA has no sub-quadratic algorithm under the Orthogonal Vectors hypothesis.

Recall that in the (All, $\tilde{O}(1)$)-bounded version of the problem, we restrict our attention to witness programs of poly-logarithmic length (i.e., on all statement types). The above bound holds even when $m = O(n)$. Orthogonal Vectors is a well-studied problem with a long-standing quadratic worst-case upper bound. The corresponding hypothesis states that there is no sub-quadratic algorithm for the problem [Williams 2019]. It is also known that the strong exponential time hypothesis (SETH) implies the Orthogonal Vectors hypothesis [Williams 2005]. Under Theorem 3.4, our algorithm from Theorem 3.3 is optimal when $\omega = 2 + o(1)$.

Finally, to establish Theorem 3.3, we solve $D_1$-Reachability in nearly matrix-multiplication time.

Theorem 3.5. All-pairs $D_1$-Reachability is solvable in $\tilde{O}(n^\omega)$ time, where $n$ is the number of nodes and $\omega$ is the matrix-multiplication exponent.

Observe the different regimes of $D_k$-Reachability for various values of $k$. When $k = 0$, the problem becomes standard graph reachability, which is solvable in $O(n^\omega)$ time [Munro 1971]. When $k \geq 2$, the problem is solvable in $O(n^3)$ time, and this bound is believed to be tight (wrt polynomial improvements) [Heintze and McAllester 1997]. The case of $k = 1$ was recently solved independently in [Bradford 2018]. However, our algorithm behind Theorem 3.5 is more straightforward: it establishes a purely combinatorial reduction of the problem to $O(\log^2 n)$ many transitive-closure operations. From there, it relies on algebraic, fast-matrix multiplication for performing each transitive closure in $O(n^\omega)$ time. In contrast, the algorithm in [Bradford 2018] is considerably longer and relies on intricate algebraic transformations. In addition, our algorithm is a $\log n$-factor faster. We refer to Section 6.1 for a more detailed comparison.

Parallelizability of APA. Parallelization is an important aspect of the complexity of a problem. In the case of APA, this is further motivated by a multitude of parallel implementations [Blaß and Philippsen 2019; Liu et al. 2019; Mendez-Lojo et al. 2012; Méndez-Lojo et al. 2010; Su et al. 2014; Wang et al. 2017]. The question is thus whether APA is effectively parallelizable. We answer this question in negative.

Theorem 3.6. On-demand APA is P-complete.

Theorem 3.6 implies that all efforts on complete parallelization must stay at the current, heuristic level, while their improvements vanish on hard instances. On the other hand, in spirit similar to Theorem 3.3, we can seek for mild restrictions to the problem that make it parallelizable. Recall that, given some $i \in \mathbb{N}$, the complexity class $NC^i$ contains the problems that are solvable in parallel time $O(\log^i n)$ with polynomially many processors. We show the following theorem.

Theorem 3.7. $(4, \log^i n)$-bounded Exhaustive APA is in $NC^{i+2}$.

Thus, APA with poly-logarithmically many applications of type 4 statements is highly parallelizable. Together, Theorem 3.6 and Theorem 3.7 expose the core hardness in the parallelization of APA as stemming from handling statements of type 4.

In the following sections we present details of the above theorems. To improve readability, in the main paper we present algorithms, examples, and proofs of all theorems. To highlight the main
steps of the proofs, we also present all intermediate lemmas; many lemma proofs, however, are
relegated to a technical report [Mathiasen and Pavlogiannis 2020].

4 A BASELINE ALGORITHM FOR ANDERSEN’S POINTER ANALYSIS

In this section, we present a baseline algorithm for APA. It has similar flavor as other algorithms in
the literature, but allows us to more easily establish Theorem 3.1.

Algorithm 1: AndersenAlgo

Input: An instance \((A, S)\) of APA.
Output: A set \((a, b)\) of all points-to relationships \(b \in [a]\).

\[
\begin{align*}
\text{// Initialization} & \quad \text{// Process inclusion edges} & \quad \text{// Process reference edges} \\
1 \quad \text{Let } W, \text{Done } & \leftarrow \emptyset & \quad \text{Function } \text{ProcessEps}(a, b) : & \quad \text{Function } \text{ProcessRef}(a, b) : \\
2 \quad \text{foreach } a = b & \text{ do } & \quad \text{foreach } (b, c, \epsilon) & \text{ in Done do } & \quad \text{foreach } (b, c, \epsilon) & \text{ in Done do} \\
3 \quad \text{Insert } (b, a, & \text{ in } W & \quad \text{end} & \quad \text{end} \\
4 \quad \text{Insert } (b, a, & \text{ in Done} & & & \\
5 \quad \text{end} \\
6 \quad \text{foreach } a = b & \text{ do} & & & \\
7 \quad \text{Insert } (b, a, & \text{ in } W & & & \\
8 \quad \text{Insert } (b, a, & \text{ in Done} & & & \\
9 \quad \text{end} \\
\text{// Computation} & \quad \text{// Establish new inclusion and} & \quad \text{// Establish new inclusion and} \quad \text{// Establish new inclusion and} \\
10 \quad \text{while } W \neq \emptyset & \quad \text{reference edges} & \quad \text{reference edges} & \quad \text{reference edges} \\
11 \quad \text{Extract } (a, b, t) & \text{ from } W & \quad \text{if } (a, b, t) & \notin \text{ Done then} & \quad \text{if } (a, b, t) & \notin \text{ Done then} \\
12 \quad \text{if } t = \epsilon & \text{ then} & \quad \text{Insert } (a, b, t) & \text{ in } W & \quad \text{Insert } (a, b, t) & \text{ in } W \\
13 \quad \text{else} & \quad \text{end} & \quad \text{end} & \quad \text{end} \\
14 \quad \text{| \text{ProcessEps}(a, b)} & \quad \text{| \text{Establish}(a, c, \&)} & \quad \text{| \text{Establish}(a, c, \&)} & \quad \text{| \text{Establish}(a, c, \&)} \\
15 \quad \text{else} & \quad \text{end} & \quad \text{end} & \quad \text{end} \\
16 \quad \text{| \text{ProcessRef}(a, b)} & \quad \text{| \text{end}} & \quad \text{| \text{end}} & \quad \text{| \text{end}} \\
17 \quad \text{end} & \quad \text{end} & \quad \text{end} & \quad \text{end} \\
18 \quad \text{return } \{(a, b): (b, a, \&) \in \text{ Done}\} & \quad \text{end} & \quad \text{end} & \quad \text{end}
\end{align*}
\]

Algorithm AndersenAlgo. The algorithm performs a form of dynamic transitive closure of a Dyck
graph, in similar spirit to existing algorithms in the literature. The key difference is that instead of
just maintaining inclusion relationships \(a \xrightarrow{\epsilon} b\), the flow graph also explicitly captures points-to
relationships \(a \xrightarrow{\&} b\) in its edges. In each iteration, the algorithm processes a newly inserted edge
\(a \xrightarrow{t} b\), where \(t \in \{\epsilon, \&\}\), and inserts new edges that represent flows-into and inclusion relationships
that are implied by \(a \xrightarrow{t} b\), the current state of the flow graph, and the statements in \(S\). The algorithm
can be seen as an on-the-fly version of the difference-propagation technique [Pearce et al. 2004;
Sridharan and Fink 2009], without the need to store difference-sets explicitly. See Algorithm 1 for a
detailed description.

Proof of Theorem 3.1. The correctness of the algorithm follows by a straightforward induction. Here we argue about the complexity. The initialization clearly runs in time \(O(m) = O(n^2)\). For
every pair of pointers \((a, b)\), the worklist \(W\) can have at most one element of the form \((a, b, \epsilon)\) and
at most one element of the form \((a, b, \&), due to the set Done. Hence, the main loop in Line 10 is
executed at most twice for every pair of pointers \((a, b)\), and thus \(O(n^2)\) times in total. For every
such pair, the loops in Lines 20 and 28 are executed once for each pointer \(c\). Hence, these loops
are executed in $O(n^3)$ time in total. Finally, each of the loops in Lines 31 and 34 is executed once for every pointer $a$. Summing over all statements of type 3 and type 4, which are $O(m)$ many, we obtain that these loops are executed $O(n \cdot m)$ times in total. Since $m = O(n^2)$, we obtain the desired bound $O(n^3)$. □

5 THE CUBIC HARDNESS OF ANDERSEN’S POINTER ANALYSIS

In this section we prove Theorem 3.2, i.e., that even On-demand APA (given pointers $a, b$, is it that $b \in [a]$?) does not have a combinatorial sub-cubic algorithm under the BMM hypothesis.

Exhaustive vs On-Demand APA. Recall the difference between Exhaustive APA and On-demand APA. The former problem asks for the points-to set $[a]$ of every pointer $a$, while the latter focuses on a specific pair of pointers $a, b$, and asks whether $b \in [a]$. Thus, On-demand APA is a special version of Exhaustive APA. The (combinatorial) cubic hardness of Exhaustive APA follows straightforwardly via a reduction from the graph transitive closure, which has the same combinatorial cubic lower bound. Indeed, given a directed graph $H = (V, E)$, we simply create an APA instance $(A, S)$, where $A = \{a_1, a_2 : a \in V\}$ contains two copies of each node in $V$, and $S = S_1 \cup S_2$, where $S_1 = \{a_1 = b_1 : (b, a) \in E\}$ and $S_2 = \{a_1 = &a_2 : a \in V\}$. That is, the type-1 statements directly correspond to the edges in $H$, and the type-2 statements are dummy statements that initialize the points-to sets. It follows immediately that after solving the APA instance $(A, S)$, for every node $a \in V$, the points-to set $[a_1]$ contains all $b_2$ such that $a$ is reachable by $b$ in $H$. Notably, the APA instance does not even make use of type-3 and type-4 statements.

We remark that the reduction does not apply for the on-demand version of APA. In the following we establish the proof of Theorem 3.2. In this direction, we establish a fine-grained reduction [Williams 2019] from the problem of deciding whether a graph contains a triangle to On-demand APA.

Reduction from finding triangles. Consider an undirected graph $H = (V, E)$, of $n'$ nodes, where the task is to determine if $H$ contains a triangle. For notational convenience, we take the node set of $H$ to be the set of integers $[n']$. Hence, the task is to determine if there exist distinct $i, j, k \in [n']$ such that $(i, j), (j, k), (k, i) \in E$. Our reduction constructs four pointers $a_i, b_i, c_i, d_i$ for every node $i' \in [n']$, and uses one additional pointer $s$ such that $s \in [c_i]$ iff $H$ has a triangle.

Intuition. The search for a triangle $(i, j, k)$ of $H$ can be seen as a search for two nodes $i$ and $k$ such that $k$ is both a distance-1 and distance-2 neighbor of $i$. In our reduction, the two pointers $c_i$ and $d_k$ are such that $c_k \xrightarrow{e} d_k$ in the resolved Dyck graph $\overline{G}$ iff $k$ is both distance-1 and distance-2 neighbor of some node $i$. This is achieved in two steps.

(1) In the initial Dyck graph $G$ of the APA instance, $c_k$ flows into $a_i$, by introducing two statements $b_j = &c_k$ and $a_i = b_j$, where $j$ is a neighbor of both $i$ and $k$.
(2) We have a statement $*a_i = d_k$.

We also introduce some additional statements between all $d_i$ and between all $c_i$ such that $c_1$ is D-reachable from $d_i$ in $\overline{G}$ iff there exists some $k$ such that $c_k \xrightarrow{e} d_k$ in $\overline{G}$ (i.e., by the above, $k$ is a distance-1 and distance-2 neighbor of some node $i$). Finally, we have a statement $d_i = &s$, so that $s \in [c_i]$ iff the above condition holds. Figure 4 provides an illustration.

Formal construction. We now proceed with the formal construction. We construct an instance of On-demand APA as follows.

(1) We introduce a distinguished pointer $s$. 

We conclude this section with the proof of Theorem 3.2.
□

The desired result follows.

(i) there exists an

(2) For every node \( i \in [n'] \), we introduce four pointers \( a_i, b_i, c_i, d_i \).

(3) For every \((i, j) \in E \) with \( j < i \), we have (i) \( a_i = b_j \), and (ii) \( b_i = &c_j \).

(4) For every \((i, j) \in E \) with \( j > i \), we have \( *a_i = d_j \).

(5) Finally, we have the following sets of assignments.

\[
\begin{align*}
    d_1 &= &s \\
    d_2 &= &d_1 \\
    & \cdots & \\
    d_n &= &d_{n-1} \\
    \text{and} \\
    c_1 &= &c_2 \\
    c_2 &= &c_3 \\
    & \cdots & \\
    c_{n-1} &= &c_n .
\end{align*}
\]

The on-demand question is whether \( s \in [c_1] \). Observe that the number of pointers of our APA instance is \( O(n') \), and the above construction can be easily carried out in time proportional to the size of \( H \).

**Correctness.** We now establish the correctness of the above construction. The key idea is as follows. Recall our definition of the resolved Dyck graph \( \overline{G} \) from Section 2.2. An edge \( d_k \xrightarrow{c} c_k \) is inserted in \( \overline{G} \) iff \( k \) is both a distance-1- and distance-2 neighbor of some node \( i \). In turn, this implies the existence of a triangle in \( H \) that contains \( i \) and \( k \). We have the following lemma.

**Lemma 5.1.** We have that \( s \) flows into \( c_1 \) in \( \overline{G} \) iff \( H \) has a triangle.

**Proof.** We prove each direction separately.

\((\Rightarrow)\). Assume that \( H \) has a triangle \((i, j, k)\), with \( i > j > k \). Then we have \( a_i = b_j \) and \( b_j = &c_k \) and thus \( c_k \xrightarrow{k} a_j \) in \( \overline{G} \). In addition, we have \( *a_i = d_k \), and thus \( d_k \xrightarrow{k} c_k \) in \( \overline{G} \). Observe that this creates a path \( s \xrightarrow{i} d_k \xrightarrow{(k-1)} c_k \xrightarrow{(k-2)} c_1 \), which witnesses that \( s \) flows into \( c_1 \) in \( \overline{G} \).

\((\Leftarrow)\). Assume that \( s \) flows into \( c_1 \). Observe that for all \( a_i \), if some node \( x \) flows into \( a_i \) then \( x \) is a \( c \) node. It follows that \( \overline{G} \) is identical to \( G \) with some additional edges from \( d \) nodes to \( c \) nodes. Hence, since \( s \) flows into \( c_1 \), there exists some \( k \in [n'] \) such that \( \overline{G} \) has an edge \( d_k \xrightarrow{k} c_k \). This means that (i) there exists an \( i \) such that \( c_k \) flows into \( a_i \) (thus \( k \) is a distance-2 neighbor of \( i \)), and (ii) there is a statement \( *a_i = d_k \) (thus \( k \) is a distance-1 neighbor of \( i \)). Hence \( H \) has a triangle containing \( i \) and \( k \). The desired result follows.

We conclude this section with the proof of Theorem 3.2.
Theorem 3.3. Due to Lemma 5.1, we have that \( s \) flows into \( c_1 \) iff \( H \) contains a triangle. By Lemma 2.2 we have that \( s \in [c_1] \) iff \( H \) contains a triangle. By [Williams and Williams 2018], triangle detection has no sub-cubic combinatorial algorithm under the combinatorial BMM-hypothesis.

The desired result follows. \( \square \)

6 A SUB-CUBIC ALGORITHM FOR BOUNDED ANDERSEN’S POINTER ANALYSIS

In this section, we first show Theorem 3.3, i.e., that computing points-to relationships when bounding the number of applications of type 4 statements admits a sub-cubic algorithm. To this end, we first prove in Section 6.1 Theorem 3.5, i.e., that \( D_1 \)-Reachability can be solved in nearly matrix-multiplication time. Afterwards, we use this result to prove Theorem 3.3 in Section 6.2.

6.1 A Sub-cubic Algorithm for \( D_1 \)-Reachability

In this section we establish a combinatorial reduction of all-pairs \( D_1 \)-Reachability to \( O(\log^2 n) \) matrix multiplications, establishing that the problem is solvable in nearly matrix-multiplication time. We first set up some helpful notation, and then present the main algorithm. Consider a Dyck graph \( G = (V, E) \).

Path indexing. Consider a path \( P = x_1, \ldots, x_l \). Given some \( i \in [l] \), we denote by \( P[i] = x_i \). Given \( i, j \in [l] \), with \( i \leq j \), we denote by \( P[i : j] = x_i, \ldots, x_j \). For simplicity, we let \( P[j] = P[1 : j] \) and \( P[i :] = P[i : l] \).

Stack heights. Consider a path \( P \) of length \( l \). We denote by \( \#_\&(P) \) (resp., \( \#_*(P) \)) the number of \& (resp., *) symbols that appear in the label \( \lambda(P) \). The stack height of \( P \) is defined as \( SH(P) = \#_\&(P) - \#_*(P) \) (note that we can have \( SH(P) < 0 \)). The maximum stack height of a path is defined as \( MSH(P) = \max_{P'} SH(P') \), where \( P' \) ranges over prefixes of \( P \).

Monotonicity and local maxima. Consider a path \( P \) of length \( l \). We say that \( P \) is monotonically increasing (resp., monotonically decreasing) if for all \( i \) with \( 1 \leq i < l \), we have \( SH(P[i : i]) \leq SH(P[i : i + 1]) \) (resp., \( SH(P[i : i]) \geq SH(P[i : i + 1]) \)). Given some \( i \) with \( 1 \leq i \leq l \), we say that \( P \) has a local maxima in \( i \) if the following conditions hold.

1. Either \( i = 1 \) or \( SH(P[i : i - 1]) < SH(P[i : i]) \).
2. For every \( j > i \) such that \( SH(P[i : i]) < SH(P[i : j]) \), there exists some \( l \) with \( i < l < j \) such that \( SH(P[i : l]) < SH(P[i : j]) \).

Bell-shape-reachability. We call a path \( P \) bell-shaped if it has exactly one local maxima. If \( P \) is bell-shaped, it can be decomposed as \( P : P_1 \circ P_2 \) where \( P_1 \) (resp., \( P_2 \)) is a monotonically increasing (resp., monotonically decreasing) path. Consider two nodes \( x, y \). We say that \( y \) is \( i\&\)-reached (resp., \( i*\)-reached) from \( x \), for some \( i \in \mathbb{N} \), if there is a path \( x \leadsto y \) (resp., \( x \leadsto* y \)). We say that \( y \) is bell-shape-reachable from \( x \) if there exists a bell-shaped path \( x \leadsto y \).

Node distances. Given two nodes \( x, y \), we define the distance \( \delta(x, y) \) from \( x \) to \( y \) as the length of the shortest path \( P : x \leadsto y \) with \( \lambda(P) \in D_1 \), if such a path exists, otherwise \( \delta(x, y) = \infty \). The maxima-distance \( \gamma(x, y) \) is the smallest number of local maxima among all shortest paths \( x \leadsto y \). The following known lemma states that the distances between two reachable nodes is at most quadratic. Note that \( \gamma(x, y) \leq \delta(x, y) \), hence the same bound holds for the maxima-distance.

Lemma 6.1 ([Deleage and Pierre 1986]). For every \( x, y \in V \), if \( \delta(x, y) < \infty \) then \( \delta(x, y) = O(n^2) \).
Routine BellReachAlgo. The main component of our algorithm for $D_1$-Reachability is a routine BellReachAlgo that computes bell-shape-reachability. Given an input Dyck graph $G = (V, E)$, BellReachAlgo computes all pairs of nodes $(x, y)$ such that $y$ is bell-shape-reachable from $x$. The algorithm constructs a sequence of $O(\log L)$ plain (i.e., not Dyck) digraphs $(G_i = (K, R_i))_i$, where $L$ is an upper bound on the distance $\delta(x, y)$ of every pair of nodes $x, y \in V$, given by Lemma 6.1. The node set $K$ is common to all $G_i$ and consists of three copies $x_1, x_2, x_3$ for every node $x \in V$.

Intuitively, the algorithm performs a form of successive doubling on the length of the bell-shaped paths that witness reachability. In iteration $i$, the algorithm performs all-pairs reachability in $G_i$, and using this reachability information, constructs the edge set $R_{i+1}$. In high level, $G_i$ consists of three copies of the graph $G$, where bell-shaped paths of maximum stack height at most $2^i - 1$ are summarized as $\epsilon$-labeled edges in the first and second copy. Paths between the nodes in the first and third copy are used to summarize monotonically increasing and (resp., decreasing) paths in $G$ with labels of the form $2^i \epsilon$ (resp., $2^i \star$). We refer to Algorithm 2 for a detailed description and to Figure 5 for an illustration.

Algorithm 2: BellReachAlgo

```
// Initialization
1. Construct a node set $K = \{x_1, x_2, x_3 : x \in V\}$
2. Construct an edge set $R_1$, initially $R_1 \leftarrow \emptyset$
3. foreach $j \in [2]$
4. Insert $(x_j, y_j) \in R_1$ iff $(x, y, \epsilon) \in E$
5. Insert $(x_j, y_{j+1}) \in R_1$ iff $(x, y, \& \epsilon) \in E$
6. Insert $(y_{j+1}, x_j) \in R_1$ iff $(x, y, \star \epsilon) \in E$
7. end
8. Construct the graph $G_1 = (K, R_1)$
9. Let $L \leftarrow \text{an upper bound on } \delta(x, y) \text{ for all } x, y \in V$

// Computation
10. foreach $i \in [\lceil \log L \rceil]$ do
11. Compute all-pairs reachability in $G_i$
12. Construct an edge set $R_{i+1}$, initially $R_{i+1} \leftarrow \emptyset$
13. foreach $j \in [2]$ do
14. Insert $(x_j, y_i) \in R_{i+1}$ iff $x_1 \rightsquigarrow y_1$ in $G_i$
15. Insert $(x_j, y_{i+1}) \in R_{i+1}$ iff $x_1 \rightsquigarrow y_3$ in $G_i$
16. Insert $(y_{i+1}, x_j) \in R_{i+1}$ iff $y_3 \rightsquigarrow x_1$ in $G_i$
17. end
18. Construct the graph $G_{i+1} = (K, R_{i+1})$
19. end
20. return $R[\lceil \log L \rceil + 1]$
```

Correctness of BellReachAlgo. It is straightforward that for each iteration $i$, if $x_1 \rightsquigarrow y_1$ in $G_i$, then $y$ is $D$-reachable from $x$ in $G$. The following lemma captures the inverse direction restricted to bell-shaped paths, i.e., if $x \rightsquigarrow y$ via a bell-shaped path $P$ in $G$ with $\text{MSH}(P) \leq 2^i - 1$, then $x_1 \rightsquigarrow y_1$ in $G_i$. The key invariants are stated in the following lemma.

**Lemma 6.2.** Consider an execution of the routine BellReachAlgo. For each $i \in [\lceil \log L \rceil]$, the following assertions hold.

1. If $y$ is $D$-reachable from $x$ via a bell-shaped path $P$ in $G$ with $\text{MSH}(P) \leq 2^i - 1$, then $x_1 \rightsquigarrow y_1$ in $G_i$.
2. If $x \rightsquigarrow y$ via a monotonically increasing path in $G$ where the last edge is $\&$-labeled, then $x_1 \rightsquigarrow y_3$ in $G_i$.
3. If $y \rightsquigarrow x$ via a monotonically decreasing path in $G$ where the first edge is $\star$-labeled, then $y_3 \rightsquigarrow x_1$ in $G_i$. 

Algorithm D₁-ReachAlgo. We are now ready to describe our algorithm D₁-ReachAlgo for D₁-Reachability. The algorithm performs ⌈log L⌉ iterations of BellReachAlgo, where L is an upper bound on the distance between any two reachable nodes in G. See Algorithm 3 for a detailed description.

Algorithm 3: D₁-ReachAlgo

Input: A Dyck graph G = (V, E)

Output: A set {(x, y)}x,y∈V such that y is D-reachable from x.

// Initialization
1 Let G₁ = (V, E₁) be a Dyck graph with E₁ = E

// Computation
2 foreach i ∈ [⌈log L⌉ + 1] do
3 Let X = BellReachAlgo on input G_i
4 Let E_{i+1} = E_i ∪ {(x, y, ε): (x, y) ∈ X}
5 Construct the graph G_{i+1} = (V, E_{i+1})
6 end
7 return E_{⌈log L⌉+1}

Correctness of D₁-ReachAlgo. We now establish the correctness of D₁-ReachAlgo. We start with an intuitive description of the correctness, and afterwards we make the argument formal (see Figure 6 for an illustration).

Intuitive argument of correctness. Consider an iteration i, and let x, y ∈ V such that y is D-reachable from x via a path P: x ↔ y. At the end of the iteration, due to the execution of routine BellReachAlgo, all bell-shaped paths u ↔ v in G_i are summarized as ε-labeled edges u ↔ v in G_{i+1}. Hence, P is summarized by a path P' in G_{i+1}, where the bell-shaped sub-paths of P are replaced by ε-edges in P'. How many times do we need to perform this iteration until the whole of P is summarized by a single ε-labeled edge? The key insight is that the number of local maxima in P' is at most half of that in P. Hence, it suffices to compute bell-shape-reachability a number of times that is logarithmic in the maxima-distance γ(x, y). Since γ(x, y) ≤ δ(x, y) and δ(x, y) ≤ L = O(n²) (by Lemma 6.1), ⌈log L⌉ = O(log n) iterations suffice.
We first have two technical lemmas. The first lemma states that all local maxima in $P$ are next to each other, i.e., there are no monotonically decreasing and increasing paths separating them. The following lemma formalizes the following observation: if the beginning of a bell-shaped sub-path of $P$ marks a local maxima for $P'$, then the beginning of the next bell-shaped sub-path of $P$ cannot mark a local maxima for $P'$. This is shown by arguing that the two bell-shaped sub-paths of $P$ are next to each other, i.e., there are no monotonically decreasing and increasing paths separating them.

**Remark 3.** For every index $h$ of $P'$, we have $\text{SH}(P'[: h]) = \text{SH}(P[: f^{-1}(h)])$.

We first have two technical lemmas. The first lemma states that all local maxima in $P'$ appear on the first node of bell-shaped sub-paths of $P$.

**Lemma 6.3.** Assume that $P'$ has a local maxima at some $h$. Then $f^{-1}(h) = j_\ell$ for some $\ell \in [k]$.

The following lemma formalizes the following observation: if the beginning of a bell-shaped sub-path of $P$ marks a local maxima for $P'$, then the beginning of the next bell-shaped sub-path of $P$ cannot mark a local maxima for $P'$. This is shown by arguing that the two bell-shaped sub-paths of $P$ are next to each other, i.e., there are no monotonically decreasing and increasing paths separating them.

**Lemma 6.4.** Assume that $P'$ has a local maxima at some $h$. Then $P_{j_\ell+1} = P_{j_\ell+1} = \epsilon$, where $j_\ell = f^{-1}(h)$.
With Lemma 6.3 and Lemma 6.4, we can now formalize the insight that the maxima-distance between any two nodes halves in each iteration of $D_1$-ReachAlgo. Given some iteration $i$ of the algorithm, we denote by $\gamma_i(x, y)$ the maxima-distance from $x$ to $y$ in the graph $G_i$. We have the following lemma.

**Lemma 6.5.** For each $i \in [\lceil \log L \rceil]$, for any two nodes $x, y \in V$ such that $y$ is reachable from $x$ in $G$, we have that $\gamma_{i+1}(x, y) \leq \gamma_i(x, y)/2$.

Finally, we prove Theorem 3.5, i.e., that all-pairs $D_1$-Reachability is solvable in $\tilde{O}(n^{\omega})$ time.

**Proof of Theorem 3.5.** We first argue about the correctness of $D_1$-ReachAlgo. It follows immediately from the correctness of the routine BellReachAlgo that if $D_1$-ReachAlgo returns that $y$ is $D$-reachable from $x$ then there is a path $y$ is $D$-reachable from $x$ in $G$. Here we focus on the inverse direction, i.e., assume that there is a path $P: x \leadsto y$ in $G$ with $\lambda(P) \in D_1$, and we argue that $D_1$-ReachAlgo returns that $y$ is $D$-reachable from $x$. Recall that $\gamma_i(x, y)$ is the maxima-distance from $x$ to $y$ in the graph $G_i$ constructed by the algorithm in the $i$-th iteration. We have

$$\gamma_i(x, y) = \gamma(x, y) \leq \delta(x, y) \leq L,$$

where the last inequality follows from our choice of $L$ as an upper-bound on $\delta(x, y)$. By Lemma 6.5, we have $\gamma_{i+1}(x, y) = \gamma_i(x, y)/2$ for each $i \in [\lceil \log L \rceil]$, hence after $i = \lceil \log L \rceil$ iterations, we have $\gamma_i(x, y) = 1$. Thus, in the last iteration of the algorithm, $y$ is bell-shape-reachable from $x$, and by the correctness of the routine BellReachAlgo (Lemma 6.2), BellReachAlgo will return that $y$ is $D$-reachable from $x$. Thus $D_1$-ReachAlgo will return that $y$ is $D$-reachable from $x$ in $G$, as desired.

We now turn our attention to the complexity of BellReachAlgo. The algorithm performs $O(\log L)$ invocations to the routine BellReachAlgo. In each invocation, BellReachAlgo performs $O(\log L) = O(\log n)$ transitive closure operations on graphs with $O(n)$ nodes. Using fast BMM [Munro 1971], each transitive closure takes $O(n^{\omega})$ time. The total running time of $D_1$-ReachAlgo is $O(n^{\omega} \cdot \log^2 L) = \tilde{O}(n^{\omega})$, as by Lemma 6.1, we have $L = O(n^2)$.

The desired result follows.

**A comparison note with [Bradford 2018].** A sub-cubic bound for $D_1$-Reachability was recently established independently in [Bradford 2018]. The crux of that algorithm is an elegant algebraic matrix encoding of flat $D_1$-Reachability to $O(\log n)$ AGMY matrix multiplications, each performed in $O(n^{\omega} \cdot \log n)$ time [Alon et al. 1997]. Intuitively, flat $D_1$-Reachability concerns reachability witnessed by sequentially composing bell-shaped paths, and the above reduction gives a $O(n^{\omega} \cdot \log^2 n)$ bound for the problem. A second step solves flat $D_1$-Reachability for $O(\log n)$ iterations, with some special treatment needed in each iteration for ensuring correct AGMY representation. Composing the two steps yields a $O(n^{\omega} \cdot \log^3 n)$ bound for $D_1$-Reachability [Bradford 2018, Theorem 2]. In comparison, the algorithm presented here is a log $n$-factor faster, and relies on a purely combinatorial reduction to $O(\log n^2)$ BMMs, which are then performed in $O(n^{\omega})$ time using algebraic techniques.

**6.2 Bounded Andersen’s Pointer Analysis in Sub-cubic Time**

In the previous section we saw that $D_1$-Reachability can be solved in nearly BMM time, i.e., $\tilde{O}(n^{\omega})$. In this section we show how we can use this result to speed-up bounded Exhaustive APA, towards Theorem 3.3. Recall that, given some $i \in [4]$ and $j \in \mathbb{N}$, the $(i, j)$-bounded APA asks to compute all memory locations $b$ that a pointer $a$ may point to, as witnessed by straight-line programs (under
the operational semantics of Table 1) that use statements of type \(i\) at most \(j\) times. We start with a simple lemma that allows us to consider instances of APA which contain only linearly many statements of type 4.

**Lemma 6.6.** \(\text{Wlog, we have } |S_4| \leq n.\)

**Proof.** Consider any pointer \(a \in A\) such that we have statements \(\{\ast a = b_i\}_i\) in \(S_4\). We introduce a new pointer \(c\), and (i) we insert a new type-4 statement \(\ast a = c\), and (ii) we replace each \(\ast a = b_i\) statement with \(c = b_i\). Performing the above process for each \(a\), we create a new instance \((A', S')\) such that for every \(a, b \in A\), we have \(b \in b_i\) in \((A, S)\) iff the same holds in \((A', S')\). Finally note that \(|A'| \leq 2 \cdot |A|\) and \(|S'| \leq |S| + |A|\), while \(|S'_4| \leq |A'|\) as now every pointer appears in the left-hand side of a type 4 statement at most once.

The desired result follows. \(\square\)

**Algorithm BoundedAPAAlgo.** We now present our algorithm BoundedAPAAlgo which solves \((4, j)\) Exhaustive APA for an instance \((A, S)\) and some given \(j \geq 0\). The algorithm performs \(j + 1\) iterations of \(D_1\)-Reachability on graphs \(G_i = (A, E_i)\), for \(i \in [j + 1]\), where initially \(G_1\) is the Dyck graph in the representation \((G_1, S_4)\) of the APA instance \((A, S)\). In iteration \(i\), the algorithm solves \(D_1\)-Reachability in \(G_i\), and then computes all pointers \(c\) that flows into some pointer \(a\) in \(G_i\) for which there is a statement \(\ast a = b\) in \(S_4\). Then the algorithm resolves the statement by inserting an edge \((b, c, \varepsilon)\) in \(G_{i+1}\). See Algorithm 4 for a detailed description. We conclude this section with the proof of Theorem 3.3.

**Algorithm 4: BoundedAPAAlgo**

```
// Initialization
// \(|S_4| \leq n \text{ wlog}\)
1 Let \((G_1 = (A, E_1), S_4)\) be the Dyck-graph representation of \((A, S)\)
2 Let \(V = \{a_1, a_2 : a \in A\}\) be a node set
3 // Computation
4 foreach \(i \in [j + 1]\) do
5     Solve \(D_1\)-Reachability in \(G_i\) using \(D_1\)-ReachAlgo
6     Let \(Z_i^1 = \{(a_1, b_2) : b = &a\) is a statement in \(S)\)
7     Let \(Z_i^2 = \{(a_2, b_2) : b\) is \(D\)-reachable from \(a\) in \(G_i)\)
8     Solve all-pairs reachability in \(H_i = (V, Z_i^1 \cup Z_i^2)\)
9     Let \(E_{i+1} = E_i\)
10    foreach \(a = b\) in \(S_4\) do
11        foreach \(c \in A\) with \(c_1 \leadsto a_2\) in \(H_i\) do
12            Insert \((b, c, \varepsilon)\) in \(E_{i+1}\)
13        end
14    end
15    Construct the graph \(G_{i+1} = (A, E_{i+1})\)
16 return \(\{a, b : b_1 \leadsto a_2\) in \(H_{i+1}\)\)
```
Proof of Theorem 3.3. The correctness follows directly from the correctness of D₁-ReachAlgo (Theorem 3.5). By induction, at the end of iteration \( i \), BoundedAPAAlgo has solved \((4, i - 1)\)-bounded Exhaustive APA, hence at the end of iteration \( j + 1 \) the algorithm has solved \((4, j)\)-bounded Exhaustive APA.

We now turn our attention to complexity. In each iteration of the main loop in Line 3, we have
\[
\text{which limits the number of applications of type-4 statements. This bounding might miss points-to}
\]

As a Dyck graph (Theorem 3.5). By induction, at the end of iteration 1:
\[
\text{respectively. In particular, we have two edges}
\]

\( x \) and \( y \) are orthogonal as far as the first coordinate is concerned. We achieve
\[
\text{iff there exists an orthogonal pair of vectors in}
\]

\( X, Y \), each containing \( n' \) vectors in \( \{0, 1\}^D \), for some dimension \( D = \omega(\log n') \). The task is to determine if there exists a pair \((x, y) \in (X \times Y)\) that is orthogonal, i.e., for each \( j \in D \), we have \( x[j] \cdot y[j] = 0 \). The respective hypothesis states that the problem cannot be solved in time \( O(n'^{2-\epsilon}) \), for any fixed \( \epsilon > 0 \).

Impact of bounding type-4 statements. Theorem 3.3 targets \((4, j)\)-bounded Exhaustive APA which limits the number of applications of type-4 statements. This bounding might miss points-to relationships created by repeatedly nested aliasing. In practice, the level of indirection is typically small, and thus we expect the above algorithm to be relatively complete. Given the algorithmic benefits of this approach, an experimental evaluation of its precision is interesting future work.

7 THE QUADRATIC HARDNESS OF BOUNDED ANDERSEN’S POINTER ANALYSIS

In this section we continue to study bounded APA and prove Theorem 3.4, i.e., if we restrict our attention to points-to relationships as witnessed by programs of length \( \tilde{O}(1) \), even the on-demand problem has a quadratic (conditional) lower bound. Our reduction is from the problem of Orthogonal Vectors [Williams 2019].

Orthogonal Vectors (OV). The input to the problem is two sets \( X, Y \), each containing \( n' \) vectors in \( \{0, 1\}^D \), for some dimension \( D = \omega(\log n') \). The task is to determine if there exists a pair \((x, y) \in (X \times Y)\) that is orthogonal, i.e., for each \( j \in D \), we have \( x[j] \cdot y[j] = 0 \). The respective hypothesis states that the problem cannot be solved in time \( O(n'^{2-\epsilon}) \), for any fixed \( \epsilon > 0 \).

Reduction from OV. Consider an instance \( X, Y \) of OV, and assume wlog that \( D \) is even. Here we show how to construct a On-demand APA instance with two distinguished pointers \( s \) and \( t \) such that \( s \in [t] \) iff there exists an orthogonal pair of vectors in \( X \times Y \).

Intuition. We start with a high-level intuition, while Figure 7 provides an illustration. Consider the first two vectors \( x_1 \in Y \) and \( y_1 \in Y \), and focus on the first coordinate. Recall the representation of APA as a Dyck graph \( G \) and a set of type-4 statements \( S_4 \), and let \( \overline{G} \) be the resolved Dyck graph. We introduce two pointers \( a_1^t \) and \( u_1^t \), with the goal that \( a_1^t \) flows into \( u_1^t \) in the resolved Dyck graph \( \overline{G} \) iff \( x_1[1] \cdot y_1[1] = 0 \), i.e., \( x_1 \) and \( y_1 \) are orthogonal as far as the first coordinate is concerned. We achieve this by introducing a distinguished node \( z \), and having two edges \( a_1^t \xrightarrow{k} z \) and \( z \xrightarrow{\epsilon} u_1^t \). Moreover, if \( x_1[1] = 0 \), we also create a path \( P_1^x : a_1^t \xleftarrow{\epsilon} z \). Similarly, if \( y_1[1] = 0 \), we also create a path \( P_1^y : z \xleftarrow{\epsilon} u_1^t \) (i.e., \( P_1^y \) is simply an \( \epsilon \)-labeled edge). Observe that \( a_1^t \) flows into \( u_1^t \) iff \( x_1[1] \cdot y_1[1] = 0 \).

If \( x_1[1] \cdot y_1[1] = 1 \) then nothing happens and the process stops here. Otherwise, \( x_1 \) and \( y_1 \) are potentially orthogonal, so we proceed with the second coordinate. We create a node \( v_1^t \) and a type-4 statement \( *u_1^t = v_1^t \); since \( a_1^t \) flows into \( u_1^t \), the resolved graph \( \overline{G} \) also has an edge \( v_1^t \xrightarrow{\epsilon} u_1^t \). The contents of \( x_1 \) and \( y_1 \) on the second coordinate are encoded via paths to new pointers \( a_2^t \) and \( u_2^t \), respectively. In particular, we have two edges \( a_1^t \xrightarrow{\epsilon} a_2^t \) and \( u_2^t \xrightarrow{k} v_1^t \). Moreover, if \( x_1[2] = 0 \), we...
also create a path $P_{y1}^2$: $a_1^1 \xrightarrow{\gamma} a_2^1$ (i.e., $P_{y1}^2$ is simply an $\epsilon$-labeled edge). Similarly, if $y_1[2] = 0$, we also create a path $P_{y2}^0$: $u_1^2 \xrightarrow{\gamma} v_1^1$. Observe that $u_1^1$ flows into $a_1^1$ iff $v_1^1 \xrightarrow{\epsilon} a_1^1$ (which is established by the previous step, as $x_1[1] \cdot y_1[1] = 0$) and also $x_1[2] \cdot y_1[2] = 0$, thereby establishing that $x_1$ and $y_1$ appear orthogonal on the first two coordinates. In that case, we have another pointer $b_1^1$ and type-4 statement $u_2^1 \xrightarrow{\gamma} u_1^1$. From here on, the process repeats as before, with $b_1^1$ playing the role of pointer $z$ initially, and the contents of $x_1[3]$ captured in paths from $a_1^1 \xrightarrow{\gamma} b_1^1$, and the contents of $y_1[3]$ captured in paths $u_2^1 \xrightarrow{\gamma} u_1^1$. In the end, we have that $u_1^D$ is D-reachable from $b_1^1$ iff $x_1[j] \cdot y_1[j] = 0$ for all $j \in [D]$.

Finally, to capture all potential pairs of vectors $x_i, y_i \in X \times Y$, we connect the pointer $z$ to all pointers $a_i^j$ and $u_i^j$, as above. To complete the reduction, we make $s \xrightarrow{\gamma} b_1^1$ and $u_2^1 \xrightarrow{\gamma} t$, and thus $s$ flows into $t$ via the D-reachable path $u_1^D \xrightarrow{\gamma} b_1^1$ iff $x_i$ and $y_i$ are orthogonal.

**Formal construction** We now proceed with the formal construction, as follows. First, we introduce a pointer $z$. For every vector $x^i \in X$, we introduce pointers $a_1^i, \ldots, a_d^i$ and $b_2^i, b_3^i, \ldots, b_d^i$.

1. We have $z = &a_1^i$. If $x^i[1] = 0$, we also introduce a new pointer $a_2^i$ and two assignments $z = &a_1^i$ and $a_2^i = &a_1^i$.
2. For every even $j \in [D]$, we have $a_1^i = b_1^j$ and $a_1^i = &a_1^i$. If $x^i[j] = 0$, we also have $a_1^i = a_1^i$.
3. For every odd $j \in [D]$ with $j > 1$, we have $b_1^j = &a_1^i$. If $x^i[j] = 0$, we also introduce a new pointer $a_1^i$ and two assignments $b_1^j = &a_1^i$ and $a_2^i = &a_1^i$.

For every vector $y^i \in Y$, we introduce pointers $u_1^i, \ldots, u_d^i$ and $v_1^i, v_2^i, \ldots, v_d^i$.

1. We have $u_1^i = &z$. If $y^i[1] = 0$, we also have $u_1^i = z$.
2. For every odd $j \in [D]$, we have $u_1^i = &v_1^i$ and $u_1^i = &v_1^i$. If $y^i[j] = 0$, we also have $u_1^i = u_1^i$. 

Fig. 7. Reduction from OV with vector sets $A = \{x_1, x_2\}$ and $B = \{y_1, y_2\}$ to (All, $O(\log n)$)-bounded APA on the pair $s \in [t]$. 

For every vector $x^i \in X$, we introduce pointers $a_1^i, \ldots, a_d^i$ and $b_2^i, b_3^i, \ldots, b_d^i$.

(1) We have $z = &a_1^i$. If $x^i[1] = 0$, we also introduce a new pointer $a_2^i$ and two assignments $z = &a_1^i$ and $a_2^i = &a_1^i$.

(2) For every even $j \in [D]$, we have $a_1^i = b_1^j$ and $a_1^i = &a_1^i$. If $x^i[j] = 0$, we also have $a_1^i = a_1^i$.

(3) For every odd $j \in [D]$ with $j > 1$, we have $b_1^j = &a_1^i$. If $x^i[j] = 0$, we also introduce a new pointer $a_1^i$ and two assignments $b_1^j = &a_1^i$ and $a_2^i = &a_1^i$.

For every vector $y^i \in Y$, we introduce pointers $u_1^i, \ldots, u_d^i$ and $v_1^i, v_2^i, \ldots, v_d^i$.

(1) We have $u_1^i = &z$. If $y^i[1] = 0$, we also have $u_1^i = z$.

(2) For every odd $j \in [D]$, we have $u_1^i = &v_1^i$ and $u_1^i = &v_1^i$. If $y^i[j] = 0$, we also have $u_1^i = u_1^i$. 

We conclude this section with the proof of Theorem 3.4.

Finally, we introduce two pointers $s$ and $t$. For every $i \in [n']$, we have $b_D' = s$ and $t = u_D'$. The on-demand question is whether $s \in [t]$. Observe that we have used $n = O(n' \cdot D)$ pointers, and the above construction can be easily carried out in $O(n)$ time.

**Correctness.** We now establish the correctness of the above construction. The key idea is as follows. Recall our definition of the Dyck-graph representation $(G = (A, E), S_4)$ of the APA instance, and the resolved Dyck graph $\overline{G}$ (see Section 2.2). The resolved graph $\overline{G}$ is constructed from $G$ by iteratively (i) finding three pointers $a, b, c$ such that $a$ flows into $b$ and we have a type 4 statement $\ast b = c$, and (ii) inserting an edge $c \rightarrow a$ in $G$. The above construction guarantees that, for two integers $i_1, i_2 \in [n']$, the following hold by induction on $j \in [D]$.

1. If $j$ is odd, we have $v_{ij}^{k} \rightarrow a_{ij}^{k}$ iff $\sum_{j' \leq j} x_i^{k} [j'] \cdot y_i^{k} [j'] = 0$.
2. If $j$ is even, we have $b_{ij}^{k} \rightarrow u_{ij}^{k}$ iff $\sum_{j' \leq j} x_i^{k} [j'] \cdot y_i^{k} [j'] = 0$.

Once such an $\epsilon$-labeled edge is inserted for some $j$, it creates a path that leads to a flows-into relationship that leads to inserting the next $\epsilon$-labeled edge for $j+1$ iff $x_i^{k} [j+1] \cdot y_i^{k} [j+1] = 0$. Note that $s$ flows into $t$ iff there exist $i_1, i_2 \in [n']$ such that $b_{i_1}^{D} \rightarrow u_{i_2}^{D}$, which, by the above, holds iff $x_i^{k}$ and $y_i^{k}$ are orthogonal. Finally, since $D = \Theta(\log n)$, the witness program for $s \in [t]$ has length $O(1)$.

The above idea is formally captured in the following two lemmas.

**Lemma 7.1.** If there exist $i_1, i_2 \in [n']$ such that $x_i^{k}$ and $y_i^{k}$ are orthogonal, then $s$ flows into $t$ in $\overline{G}$. Moreover, there exists a witness program $P$ of length $O(\log n)$ that results in $s \in [t]$.

**Lemma 7.2.** If $s$ flows into $t$ in $\overline{G}$, there exist $i_1, i_2 \in [n']$ such that $x_i^{k}$ and $y_i^{k}$ are orthogonal.

We conclude this section with the proof of Theorem 3.4.

**Proof of Theorem 3.4.** Lemma 7.1 and Lemma 7.2, together with Lemma 2.2, state the correctness of the reduction. Note that the APA instance we constructed has $n = O(n' \cdot D)$ pointers and $m = O(n' \cdot D)$ assignments, hence it is a sparse instance. Moreover the time for the construction is $O(n' \cdot D)$. Assume that there exists some fixed $\epsilon > 0$ such that On-demand APA can be solved in $O(n^{2-\epsilon})$ time. Then we have a solution for the OV instance in time $O((n' \cdot D)^{2-\epsilon})$ time, which violates the Orthogonal-Vectors hypothesis.

The desired result follows. ⊓⊔

### 8 THE PARALLELIZABILITY OF ANDERSEN’S POINTER ANALYSIS

In this section we address the parallelizability of APA, and show the following results. In Section 8.1 we prove Theorem 3.6, which shows that APA is not parallelizable under standard hypotheses in complexity theory. In Section 8.2 we prove Theorem 3.7, which shows that bounded APA is efficiently parallelizable as long as we focus on poly-logarithmically (i.e., $O(\log^c n)$, for some constant $c$) many applications of type 4 statements.
8.1 Andersen’s Pointer Analysis is not Parallelizable

In this section we prove Theorem 3.6, i.e., that On-demand APA is P-complete. This implies that the problem is unlikely to be parallelizable. Our reduction is from Monotone CVP.

The problem Monotone CVP. The input to the problem is a circuit represented as a sequence of assignments \((A_1, \ldots, A_n)\), such that for all \(i \in [n']\), \(A_i\) has one of the following types, where \(j < k < i\).

\[
A_i = 0 \quad \quad A_i = 1 \quad \quad A_i = A_j \land A_k \quad \quad A_i = A_j \lor A_k
\]

The condition \(j < k < i\) ensures acyclicity. The assignments \(A_i = 0\) and \(A_i = 1\) are the inputs, whereas all other assignments are the gates. The task is to compute whether \(A_n'\) evaluates to 1 under the standard Boolean algebra interpretation of the operators \(\land\) and \(\lor\). It is known that Monotone CVP is P-complete even when every assignment has fan-out 2 (except \(A_n'\) and inputs \(A_i\)) [Greenlaw et al. 1995].

Reduction from Monotone CVP. Consider an instance \((A_1, \ldots, A_n)\) of Monotone CVP, and we construct an instance \((A, S)\) of On-demand APA. An illustration is given in Figure 8.

1. For every input \(A_i\), we introduce two pointers \(x_i, z_i \in A\). We have a statement \(x_i = z_i \in S\) iff \(A_i = 1\).
2. For every gate \(A_j = A_j \land A_k\), with \(j < k < i\), we introduce (i) three pointers \(x_i, y_i, z_i\), (ii) \(i\) pointers \(x_i^1, \ldots, x_i^i\), and (iii) \(i - 1\) pointers \(y_i^1, \ldots, y_i^{i-1}\). We have the following statements in \(S\).

\[
\begin{align*}
&x_i^1 = \&x_i \quad x_i^2 = \&x_i^1 \quad \cdots \quad x_i^i = \&x_i^{i-1} \quad z_j = \&x_i^i \\
&y_i^{i-1} = \*y_k \quad y_i^{i-2} = \*y_i^{i-1} \quad \cdots \quad y_i^1 = \*y_i^2 \quad y_i = \*y_i^1
\end{align*}
\]

For every gate \(A_j = A_j \lor A_k\), with \(j < k < i\), we introduce (i) three pointers \(x_i, y_i, z_i\), (ii) \(i\) pointers \(x_i^1, \ldots, x_i^i\) and \(i\) pointers \(x_i^1, \ldots, x_i^j\), and (iii) \(i - 1\) pointers \(y_i^1, \ldots, y_i^{i-1}\) and \(i - 1\) pointers...
\[ \hat{y}_i \] \ldots, \hat{y}_i^{-1}. We have the following statements in \( S \).
\[
\begin{align*}
x_i^1 &= &x_i^1 & &\cdots & &x_i^j &= &x_i^j & &z_j &= &x_i^j & &\text{and} \\
\hat{x}_i^1 &= &x_i^1 & &\cdots & &\hat{x}_i^j &= &x_i^j & &z_k &= &x_i^j & &\text{and} \\
y_i^{-1} &= &y_j & &\cdots & &\hat{y}_i^{-1} &= &x_i^{-1} & &y_i &= &x_i^{-1} & &\text{and} \\
\end{align*}
\]

(3) For every gate \( A_i \), we add a statement \( *y_i = z_i \).
(4) Finally, we introduce a special pointer \( s \), and a statement \( z_n' = &s \). The on-demand question is whether \( s \in [x_n'] \).

Note that the size of our APA instance is \( \Theta(n^2) \) i.e., the construction leads to a quadratic blow-up. Nevertheless, it is easy to carry out the above construction in logarithmic space, as required for a log-space reduction.

**Correctness.** We now establish the correctness of the reduction. We first present an intuitive insight and then the formal steps of the proof.

**Intuitive argument of correctness.** Let \((G = (A, E), S_i)\) be the Dyck-graph representation of the constructed APA instance \((A, S)\), and let \( \overline{G} \) be the resolved Dyck graph (see Section 2.2). Recall that the resolved graph \( \overline{G} \) is constructed from \( G \) by iteratively (i) finding three pointers \( a, b, c \) such that \( a \) flows into \( b \) and we have a type 4 statement \( a \rightarrow b = c \), and (ii) inserting an edge \( c \rightarrow a \) in \( G \).

The correctness of the construction relies on the following invariant. For every \( i \) such that \( A_i \) is a gate with inputs \( A_j, A_k \), we have that \( x_i \) is D-reachable from \( z_i \) if (i) \( z_j \rightarrow x_j \) and \( z_k \rightarrow x_k \) (if \( A_i \) is an AND gate), or (ii) \( z_j \rightarrow x_j \) or \( z_k \rightarrow x_k \) (if \( A_i \) is an OR gate). Observe that in \( \overline{G} \) we have potentially other nodes \( u \) that are D-reachable from \( z_i \). For example, in Figure 8, we have \( z_4 \rightarrow x_4^1 \) in \( \overline{G} \), where \( x_4^1 \) is the second node in the path \( x_4 \rightarrow x_2 \) (not explicitly shown in the figure). This occurs because we have a statement \( *y_4 = z_4 \), and \( x_4^1 \) flows into \( y_4 \). The key insight is then that the invariant holds despite such “unwanted” edges. Given the invariant, the correctness proof follows by an induction on the depth of the circuit.

**Formal Correctness.** We now make the above insights formal. We start with a technical lemma, which shows that if \( x_i \) is D-reachable from \( z_i \) in \( \overline{G} \), then, in fact, \( z_i \rightarrow x_i \), as \( x_i \) flows into \( y_i \) and we have a statement \( *y_i = z_i \).

**Lemma 8.1.** For all \( i \in [n'] \), \( x_i \) is D-reachable from \( z_i \) in \( \overline{G} \) iff \( x_i \) flows into \( y_i \) in \( \overline{G} \).

The following lemma establishes our main invariant, which relates the encoding of the output of a gate in \( \overline{G} \) to the encoding of its inputs.

**Lemma 8.2.** Consider a gate \( A_i \). We have that \( x_i \) is D-reachable from \( z_i \) in \( \overline{G} \) iff

(1) \( A_i \) is an AND gate \( A_i = A_j \land A_k \), and \( x_j \) is D-reachable from \( z_j \) and \( x_k \) is D-reachable from \( z_k \), or
(2) \( A_i \) is an OR gate \( A_i = A_j \lor A_k \), and \( x_j \) is D-reachable from \( z_j \) or \( x_k \) is D-reachable from \( z_k \).

The following lemma establishes the correctness of the construction. Its proof follows by an induction on the depth of the circuit, and using Lemma 8.2 on the respective gate.
Lemma 8.3. We have that \( x_{n'} \) is D-reachable from \( z_{n'} \) iff \( A_{n'} \) evaluates to 1.

We conclude this section with the proof of Theorem 3.6.

Proof of Theorem 3.6. Membership in P is known (e.g., Theorem 3.1), so we need to argue that the problem is P-hard. Observe that \( s \) flows into \( x_{n'} \) iff \( x_{n'} \) is D-reachable from \( z_{n'} \). By Lemma 8.3, we have that \( s \) flows into \( x_{n'} \) iff \( A_{n'} \) evaluates to 1. By Lemma 2.2, we have that \( s \in [x_{n'}] \) iff \( A_{n'} \) evaluates to 1. The desired result follows.

8.2 Bounded Andersen’s Pointer Analysis is Parallelizable

Finally, in this section we develop an algorithm for solving bounded Exhaustive APA, and thus prove Theorem 3.7.

Parallel bounded APA. The algorithm is a parallelization of BoundedAPAAlgo (Algorithm 4) for sequential bounded APA. The parallel algorithm performs the iterations of the main loop of Line 3 sequentially, while the body of the loop is run in parallel. We outline the steps of the parallelization.

1. In Line 4, BoundedAPAAlgo invokes the routine D1-ReachAlgo for computing D1-Reachability (Algorithm 3). Recall, by Lemma 6.1, that the distance between two reachable nodes is \( O(n^2) \), and hence the same bound holds for the maximum stack height of the shortest path that witnesses reachability between such nodes. It follows that D1-Reachability can be reduced to standard graph reachability on a graph with \( O(n^3) \) nodes of the form \((u, i)\), where \( u \) is a node of the Dyck graph and \( i \in [O(n^2)] \) encodes the stack height. Finally, graph reachability is solved in parallel [Papadimitriou 1993].

2. In Line 7, BoundedAPAAlgo constructs another graph \( H \) and solves all-pairs reachability on \( H \). Now, we perform the construction of \( H \) in parallel, while all-pairs reachability in \( H \) is also computed in parallel, as in the previous item.

3. Finally, we execute the two nested loops in Line 9 and Line 10 in parallel, by using one processor per triplet of pointers \((a, b, c)\).

The correctness of the parallelization follows directly from the correctness of the sequential version (Theorem 3.3). We conclude with the complexity analysis, which establishes Theorem 3.7.

Proof of Theorem 3.7. In each iteration, the parallel running time is the time required to compute the transitive closure of a graph of \( O(n^3) \) nodes. Since graph reachability is in NC² [Papadimitriou 1993], each iteration takes \( O(\log^2 n) \) time. Hence, executing the main loop sequentially for \( j \) iterations yields \( O(j \cdot \log^2 n) \) time. In particular, for \( j = \log^2 n \) iterations, the algorithm solves \( O(4, \log^4 n) \)-bounded Exhaustive APA in \( O(\log^{4+2} n) \) parallel time, hence the problem is in NC\(^{4+2}\).

The desired result follows.

9 Conclusion

Andersen’s Pointer Analysis is a standard approach to static, flow-insensitive pointer analysis. Despite its long history and practical importance, the complexity of the analysis had remained elusive. In this work, we have drawn a rich fine-grained and parallel complexity landscape based on various aspects of the problem. We have shown that even deciding whether a single pointer may point to a specific heap location is unlikely to have sub-cubic complexity, and additionally, the problem is not parallelizable. These results strongly characterize the hardness of the problem. On the positive side, we have presented a bounded version of the problem that becomes solvable in
nearly matrix-multiplication time, and have established a conditional quadratic lower bound for the bounded version of the problem.

Our positive results build some stable ground for further practical improvements for Andersen’s pointer analysis. We expect that our solution to the bounded version of the problem, which essentially reduces to a small number of standard transitive closure operations, can provide the basis for faster practical approaches: graph transitive closure solvers have been heavily optimized over the years in both software and hardware, while the full parallelizability of the bounded problem opens itself up to more efficient multithreaded implementations. As our focus in this work has been on characterizing the tractability landscape of the problem, we have left the practical realizations of our results for interesting future work.

ACKNOWLEDGMENTS

We wish to thank Phillip G. Bradford for bringing to our attention his result on $D_1$-Reachability and for helpful comments in comparing his algorithm to ours, as well as anonymous reviewers for their constructive feedback in an earlier version of this manuscript.

REFERENCES


