Precise and Scalable Context-Sensitive Pointer Analysis via Value Flow Graph

Lian Li Cristina Cifuentes Nathan Keynes
Oracle Labs, Brisbane, Australia
{lian.li,cristina.cifuentes,nathan.keynes}@oracle.com

Abstract

In this paper, we propose a novel method for context-sensitive pointer analysis using the value flow graph (VFG) formulation. We achieve context-sensitivity by simultaneously applying function cloning and computing context-free language reachability (CFL-reachability) in a novel way. In contrast to existing clone-based and CFL-based approaches, flow-sensitivity is easily integrated in our approach by using a flow-sensitive VFG where each value flow edge is computed in a flow-sensitive manner. We apply context-sensitivity to both local variables and heap objects and propose a new approximation for heap cloning.

We prove that our approach can achieve context-sensitivity without loss of precision, i.e., it is as precise as inlining all function calls. We develop an efficient algorithm and implement a context-, flow-, and field-sensitive pointer analysis with heap cloning support in LLVM. We evaluate the efficiency and precision of our implementation using standard SPEC CPU2006 benchmarks. Our experimental results show that the analysis is much faster than existing approaches, it scales well to large real-world applications, and it enables more effective compiler optimizations.

Categories and Subject Descriptors F3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages–Program analysis

General Terms Algorithms, Languages, Performance

Keywords context-sensitive analysis, flow-sensitive analysis, demand-driven, function summary, CFL-reachability

1. Introduction

Pointer analysis is a fundamental program analysis that statically computes the possible runtime values for pointer variables. It enables a variety of applications, including bug checking [7, 19], program verification [10], code optimization, and memory management for embedded systems [21, 22]. More precise pointer information directly supports more precise analysis tools, and more aggressive optimizations. However, highly precise pointer analysis techniques have previously been considered to be too slow to be usable in practice.

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.

In pointer analysis, heap memory is typically abstracted as a finite set of abstract allocation sites (memory objects), with potential pointer values represented as the set of memory objects that each pointer may point to at runtime, known as the *points-to set*. The precision of pointer analysis can be improved via two major dimensions: *context-sensitivity* and *flow-sensitivity*. Context-sensitive analysis can be applied to both pointer variables or heap objects (aka heap-cloning [26]). It distinguishes the context of a function invocation and prevents information from being erroneously propagated to different call-sites of the same function, thus greatly improving precision.

In context-sensitive analysis, the context of a function invocation is typically distinguished by its *call path*, which is simply the path from the entry function to the invocation site in the call graph. The analysis is said to be precise if it is performed in such a way that all contexts are differentiated, i.e., the full call path (with cycles on the path being discarded) is used to represent a calling context. As the number of contexts (call paths) in a program can be exponential in the call graph size, precise context-sensitive analyses often suffer from scalability problems.

Existing context-sensitive pointer analyses developed to date are either clone-based [4, 36], summary-based [14, 25, 38, 40, 42], or use a context-free language reachability formulation (CFL-based) [32, 39]. Both clone- and CFL-based approaches achieve context-sensitivity at the expense of lack of flow-sensitivity. Recent summary-based approaches [14, 40] successfully scaled context- and flow-sensitive pointer analysis to large applications by computing compact parameterized function summaries. However, compact function summaries also suggest that some useful information may not be preserved in the summary. Hence the computed results may not be general enough to precisely answer various alias queries. For example, it is difficult to precisely answer the query "do the two pointers alias on a particular call path?", if only points-to sets are preserved without extra context information.

In this paper, we propose a novel method for context-sensitive pointer analysis that computes and preserves precise pointer and context information, while scaling to large applications. At its core, we make use of the value flow graph (VFG) formulation [20] where pointer variables and memory objects are represented as nodes in the graph and edges represent dependencies between them. Context-sensitivity is achieved by simultaneously applying function cloning and computing CFL-reachability in a novel way. We address scalability by developing various effective summary-based optimizations, while at the same time preserving all pointer and context information in the compact VFG representation so they can be easily computed on-demand. In contrast to existing clone-based and CFL-based approaches, flow-sensitivity is easily integrated in our approach by using a flow-sensitive VFG where each value flow edge is computed in a flow-sensitive manner [20, 40]. Last but not

least, we apply context-sensitivity to both local variables and heap objects and propose a new approximation for heap cloning.

In our approach, context-sensitivity is achieved effectively as follows. We apply function cloning at different call-sites where the local value flows of the callee function (i.e., value flows within the function) may be computed differently. This ensures that all local value flows become context-independent so that each function can be analyzed individually. Our approach avoids the creation of redundant clones, as per existing approaches [4, 36], which clone functions based on each distinct call path to the call-site. With our value flow representation and function cloning, we can achieve context-sensitivity by simply applying CFL-reachability to call-sites, where the call-sites of each pair of call/return edges on a value flow path need to match with each other. This is the well known balanced-parentheses problem in CFL-reachability [29] and it can be computed efficiently with various optimizations developed in this paper. In contrast, existing CFL-based approaches apply CFL-reachability to call-sites and heap accesses at the same time. As a result, it is much more expensive to compute CFL-reachability in those approaches and also much more difficult to develop effective optimization techniques for them.

We prove that our approach can achieve context-sensitivity without loss of precision, i.e., it is as precise as inlining all function calls for recursion-free programs. We implement a context, flow- and field-sensitive analysis in LLVM [16] and evaluate its efficiency and precision using standard CPU2006 benchmarks. Our experimental results suggest that the analysis is much faster than the existing Bootstrapping approach [14], and it enables much more effective compiler optimizations in LLVM. To summarize, this paper makes the following contributions:

- We propose a new method for context-sensitive pointer analysis. The method can be easily extended to achieve flow-sensitivity at the same time. We prove that our approach can effectively achieve context-sensitivity without loss of precision.
- We apply context-sensitivity to both pointer variables and heap objects, and propose a new approximation for heap cloning.
- We develop an efficient algorithm, as well as a set of optimization techniques, to compute flow-sensitive and context-sensitive pointer information efficiently. We implement our algorithm in LLVM and show that our analysis can achieve very high precision and scale to large applications.

The rest of the paper is organized as follows: Section 2 introduces the value flow formulation. Section 3 shows how context-sensitivity can be effectively achieved. We present the algorithm in Section 4 and evaluate our implementation in Section 5. Section 6 reviews related work and Section 7 concludes the paper.

2. The Value Flow Formulation

We describe our formulation of pointer analysis using a small language which captures the important properties of the C language. Functions are constant values defined by the expression $F = func(fp_1...fp_n) \Rightarrow (rp_1...rp_m)\mathcal{S}^*$, where F is the function definition, fp_i are formal parameters, and rp_i are formal return parameters. Note that we have generalized function definitions in C to allow functions with multiple returns. Function calls are represented by the CALL statement $C : (x_1...x_m) = f(ap_1...ap_n)$, where C is the label of the call-site, f is the called value, ap_i are actual parameters and x_i are actual returns.

$$
\begin{array}{lll}
\mathcal{P} := & \mathcal{F}* & \\
\mathcal{F} := & F = func(fp_1...fp_n) \Rightarrow (rp_1...rp_m)\mathcal{S}^* & \\
\mathcal{S} := & p_A = \&A & \text{BASE} \\
 & p = q & \text{ASSIGN} \\
 & *p = x & \text{STORE} \\
 & y = *q & \text{LOAD} \\
 & C : (x_1...x_m) = f(ap_1...ap_n) & \text{CALL}
\end{array}
$$

The above set of statements are sufficient for context-sensitive pointer analysis for C. More complicated pointer-manipulating statements can be decomposed into these basic instructions. Nested pointer dereferences are eliminated by introducing auxiliary variables. Allocation of a heap object is modelled by regarding the allocation site as a special memory object.

To support flow-sensitivity, we also need to consider the control flows of the program as in [20]. For clarity, control flow statements are not modeled in our formulation and we explain how to extend our formulation for flow-sensitive analysis in Section 4.1.

2.1 Insensitive Value Flows

In a value flow graph (VFG), values flow along edges between memory objects and pointer variables (represented as nodes). We say that q flows to p, denoted $q \rightarrow p$, if the value of q is assigned to variable p. We say that pointer p points to object A if there exists a value flow path from the object node A to the pointer node p in the VFG, denoted as $A \rightarrow^* p$. The set of objects that p may point to is called the points-to set of p, denoted as $pts(p)$.

Memory objects flow to pointer variables directly via BASE ($p_A = \&A$) instructions, and pointer variables flow to each other either directly, via ASSIGN ($p = q$) instructions, or indirectly, via STORE ($*p = x$) and LOAD ($y = *q$) instructions, as illustrated by the following rules. Without loss of generality, we assume that for each memory object, a unique pointer variable is initialized to take the address of the object and it is accessed via LOAD and STORE instructions to the introduced variable. As such, in a VFG, object nodes are nodes without incoming edges.

$$\text{BASE}\ \frac{p_A = \&A}{A \rightarrow p_A} \qquad \text{ASSIGN}\ \frac{p = q}{q \rightarrow p}$$

$$\text{INDIRECT}\ \frac{\begin{array}{cc} A \rightarrow^* p & A \rightarrow^* q \\ *p = x & y = *q \end{array}}{x \rightarrow y}$$

The BASE and ASSIGN rules describe the direct value flows for BASE and ASSIGN instructions, respectively. The INDIRECT rule handles pointer dereferences via LOAD and STORE instructions. It states that x can flow to variable y indirectly via pointer dereferences, only if x is stored to a memory object A first (via pointer p) and y is loaded from the same memory object afterwards (via pointer q). Note that the above rules are formulated for insensitive analyses. Next we will show how they are extended for context-sensitivity.

2.2 Context-sensitive Value Flows

We distinguish intra- and inter-procedural value flows for the sake of context-sensitivity.

$$\text{CALL}\ \frac{\begin{array}{c} F = func(fp_1...fp_n) \Rightarrow (rp_1...rp_m)\mathcal{S}* \\ F \rightarrow^* f \\ C : (x_1...x_m) = f(ap_1...ap_n) \end{array}}{\wedge_{1 \le i \le n} ap_i \xrightarrow{C_{in}} fp_i \qquad \wedge_{1 \le i \le m} rp_i \xrightarrow{C_{out}} x_i}$$

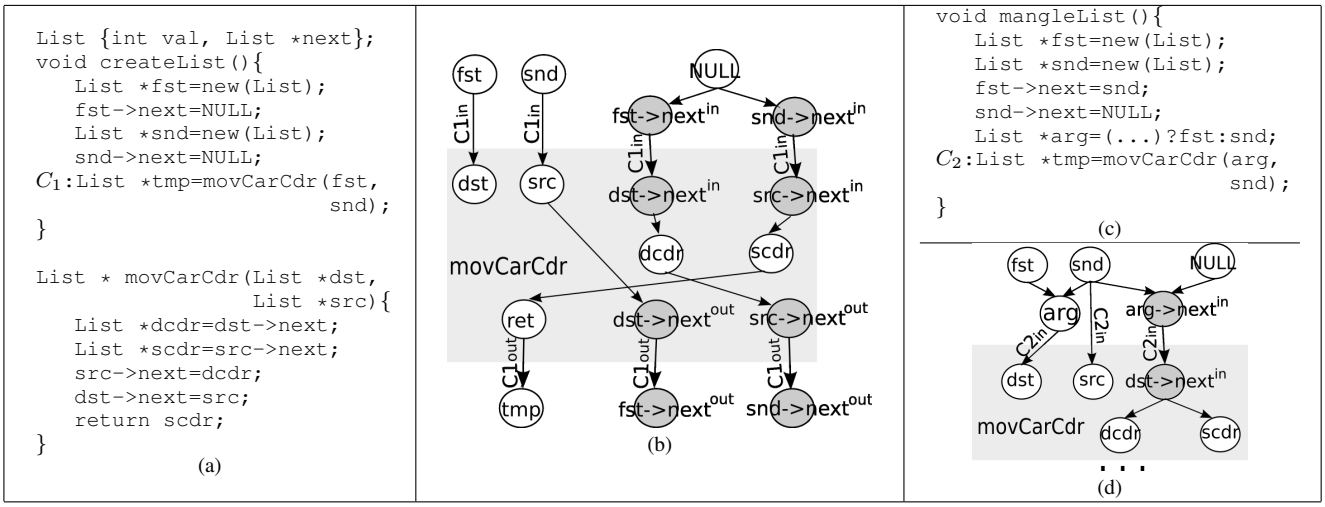

Figure 1. An Example.

The CALL rule models inter-procedural value flows, i.e., value flows introduced via CALL statements. The notation $ap \xrightarrow{C_{in}} fp$ denotes that ap (actual parameter in the caller function) is assigned to variable fp (formal parameter in the callee function) at call-site C, e.g., parameter passing, and the variable fp is said to be a *function input* of the callee function. Similarly, the edge $rp \xrightarrow{C_{out}} out$ represents value flows from the callee function to the caller at C, e.g., function returns, where variable rp is called a *function output* and out is the returned value at call-site C. In our formulation, a function is allowed to have multiple returns. If the callee function creates a heap object that may escape, the created heap object O_H is regarded as a return of the function. A variable p_H will be introduced at its call-site C in the caller function and we have $O_H \xrightarrow{C_{out}} p_H$. Conceptually, for each global variable, a copy is introduced in every function that may access the variable and there exists inter-procedural flow edges between these introduced copies.

Note that inter-procedural value flows can be introduced not only directly via function parameters or return values, but also indirectly via dereferences of those variables. We use a mechanism similar to [5, 11, 38] for inter-procedural indirect value flows. The idea is to introduce auxiliary variables for pointers that may be dereferenced. Thus inter-procedural indirect value flows are represented explicitly as value flows between introduced auxiliary variables and they can be processed in the same fashion as normal function parameter passing or returns.

We introduce auxiliary variables for a function call as follows:

Rule 1. For function input fp where $ap \xrightarrow{C_{in}} fp$ and $pts(ap) \neq \emptyset$, we introduce auxiliary variables ${fp*}^{in}$ (as extra formal parameter) and ${ap*}^{in}$ (as extra actual parameter) where ${ap*}^{in} = *ap$, $*fp = {fp*}^{in}$ and ${ap*}^{in} \xrightarrow{C_{in}} {fp*}^{in}$. In addition, if fp may be stored in the callee function, we introduce ${fp*}^{out}$ and ${ap*}^{out}$ where ${fp*}^{out} = *fp$, $*ap = {ap*}^{out}$, and ${fp*}^{out} \xrightarrow{C_{out}} {ap*}^{out}$. For a heap object O_H where $O_H \xrightarrow{C_{out}} p_H$, we also introduce ${O_H*}^{out} \xrightarrow{C_{out}} {p_H*}^{out}$.

The introduced auxiliary variables are regarded as normal function parameters or return values. For introduced extra formal parameter ${fp*}^{in}$, we may introduce auxiliary variable $fp *^{in} *^{in}$ for its dereference. The number of auxiliary variables need to be bounded for termination, and we will show how to bound the number without loss of precision in Section 2.3. It is only necessary to introduce auxiliary variables for function inputs (escaping objects) that have been dereferenced. Note that extra LOAD or STORE instructions are introduced for the auxiliary variables. The variables ${ap*}^{in}$ and ${fp*}^{in}$ are introduced in such a way that we have $*fp = *ap$ at the entry of the callee function. They represent indirect value flows from the caller function into the callee function. Similarly, after introducing the LOAD and STORE instructions for ${fp*}^{out}$ and ${ap*}^{out}$, we have $*ap = *fp$ at the return site to capture side effects of the callee function.

Let us look at the example in Figure 1(a). Function `movCarCdr` removes the head from one linked list (parameter `src`), and insert it into another linked list (parameter `dst`). Function `createList` calls function `movCarCdr` at call-site C_1 with two single nodes. Figure 1(b) gives the VFG (which is computed flow-sensitively) for the example, where the introduced auxiliary variables are highlighted in grey. By introducing `fst->next`$^{in} \xrightarrow{C_{1_{in}}}$ `dst->next`^{in}, the `NULL` value written to the `next` field of object `fst` can now flow to variables in the callee function `movCarCdr`, e.g., `dcdr`. Similarly, the inter-procedural value flow edge `dst->next`^{out} $\xrightarrow{C_{1_{out}}}$ `fst->next`^{out} enables values written to parameter `dst` in the callee function, i.e., parameter variable `src`, flow to its caller function.

Introduction of auxiliary variables enables us to analyze each function individually: for each function input fp, we assume there exists a symbolic object whose address is taken by fp and dereferences of fp are regarded as values stored to or loaded from the symbolic object. This is a key step to performance and scalability.

2.3 Soundness and Precision

For recursive data structures such as linked lists, the number of auxiliary variables that need to be introduced may be infinite. As shown in Figure 2, we introduce an auxiliary variable `L->next`^{in} as the dereference of formal parameter `L`. Since the introduced auxiliary variable `L->next`^{in} is also dereferenced in the function, we may introduce another auxiliary variable `L->next`^{in}`->next`^{in}. The procedure of introducing auxiliary variables is recursive, and the number of introduced auxiliary variables needs to be bounded to guarantee termination.

The approaches in previous works [5, 11, 38] bound the number of auxiliary variables in a similar fashion. The idea is to introduce auxiliary variable for the same object once. For the example in Fig-

```
List {int val, List * next};
void traverse(List * L) {
    List *tmp=L;
    while (tmp) {
        tmp = tmp->next;
        . . .
    }
}
```

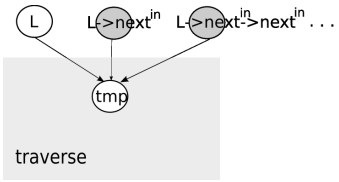

Figure 2. An Example with Possible Infinite Auxiliary Variables.

ure 2, `L->next`^{in}`->next`^{in} will not be introduced if `L->next`^{in} refer to the same memory object as `L`. This approach guarantees soundness. However, precision is sacrificed. One of the contributions in this paper is that we highlight the cause of imprecision and propose a solution to address this problem. With our method, both precision and soundness can be guaranteed as shown in the next subsections.

2.3.1 Introducing Auxiliary Variables

Lemma 1. *In a context-sensitive analysis, introducing auxiliary variables as in Rule 1 is both sound and precise: For all pointer variables in the original program, the computed results are the same as in the transformed original program where all function calls are inlined.*

Proof. By introducing auxiliary variables, we have effectively introduced pair of LOAD and STORE instructions into the original program such that $*X = *Y$. This is both sound and precise if we have $pts(Y) \neq \emptyset$ and $pts(X) = pts(Y)$. For $ap \xrightarrow{C_{in}} fp$, in a context-sensitive analysis, $pts(ap) = pts(fp)$ always holds as only variable ap is assigned to fp at call-site C. Hence introducing $*ap = *fp$ and $*fp = *ap$ at call-site C is both sound and safe. A similar proof is applied to escaping heap objects. □

However, imprecision may be introduced when we try to bound the number of introduced variables by merging introduced auxiliary variables for different arguments/parameters together. Previous work universally adopts a similar idea to bound the number of introduced auxiliary variables: only introduce auxiliary variables for the same object once. If an object is pointed to by two different arguments, i.e., aliased arguments, auxiliary variables are only introduced for one of them and indirect value flows via the other argument will be merged if no other object flows to that argument. This approach is sound but may result in spurious value flows.

In Figure 1(c), we modify our example by calling function `movCarCdr` differently. The two arguments alias. The first argument `arg` points to both objects, `fst` and `snd`, and the second argument is `snd`. As a result, auxiliary variable `arg->next`^{in} are only introduced for the first argument `arg`. The resulting VFG is given in Figure 1(d), where there exists a spurious value flow path `snd` $\rightarrow^*$ `scdr`. This happens because in merging auxiliary variables introduced for different parameters, they assume that their aliases also alias with each other. However, alias relation is not transitive. In the example, `fst` $\rightarrow^*$ `dst` and `dst` aliases with `src`, but there is no alias between `fst` and `src`. Hence `snd`, which is stored into `fst`, cannot flow to `scdr`, which is loaded from `src`. Therefore, we apply the following rule to bound the number of auxiliary variables without loss of precision:

Rule 2. For function input fp_1 where $ap_1 \xrightarrow{C_{in}} fp_1$ and $pts(ap_1) \neq \emptyset$, auxiliary variables will be introduced at call-site C if there exists no function input fp_2 where $ap_2 \xrightarrow{C_{in}} fp_2$ and $pts(ap_2) = pts(ap_1)$. Otherwise, we reuse the auxiliary variables introduced for ap_2 and fp_2.

It may sound expensive as the number of variables introduced can be exponential. The number is bounded to the number of combinations of all objects, i.e., the number of distinct points-to sets. However, if a parameter points to a recursive data structure, very often the parameter and its dereferences (i.e., introduced auxiliary variables) point to the same set of memory objects. As a result, the number of auxiliary variables introduced is small in practice as shown in our experiments over large applications.

Theorem 1. *In a context-sensitive analysis, bounding the number of auxiliary variables using Rule 2 is both sound and precise.*

Proof. In pointer analysis, variables can be safely merged together without loss of precision if and only if they always have the same points-to set. If we have $pts(X) = pts(Y)$, then $pts(*X) = pts(*Y)$ also holds. For $ap_1 \xrightarrow{C_{in}} fp_1$ and $ap_2 \xrightarrow{C_{in}} fp_2$, if $pts(ap_1) = pts(ap_2)$, the auxiliary variables introduced for ap_1 and ap_2 always have the same points-to set and they can be merged together without loss of precision. In a context-sensitive analysis, we also have $pts(fp_1) = pts(ap_1)$ and $pts(fp_2) = pts(ap_2)$ at call-site C, hence we can also merge the auxiliary variables introduced for fp_1 and fp_2 together at C. □

2.3.2 Computing Value Flows

In this section, we show how to extend the INDIRECT rule so that all value flows are computed context-sensitively. Recall that for each function input variable, a symbolic object is introduced whose address is taken by that variable and dereferences of the function input are regarded as LOADs or STOREs to the symbolic object. However, at a call-site, if two input variables of the callee function alias, values stored into one may also flow to values loaded via the other. Hence at a call-site where the actual parameters to function inputs A and B alias, we introduce a new value flow rule:

$$\text{INDIRECT}_{\text{ALIAS}}\ \frac{\{A, B\}_{\text{ALIAS}} \quad A \rightarrow^* p \quad B \rightarrow^* q \quad *p = x \quad y = *q}{x \rightarrow y}$$

The INDIRECT$_{\text{alias}}$ rule guarantees soundness but may lead to spurious value flows. Let us revisit the modified example in Figure 1(c). At call-site C_2, the two parameters `dst` and `src` alias. The introduced auxiliary variable `dst->next`^{in} is regarded as an STORE to `dst`, and `scdr` is loaded from `src`. As a result, there is an spurious value flow edge `dst->next`^{in} $\rightarrow$ `scdr` as well as the spurious value flow path `snd` $\rightarrow^*$ `scdr`.

Hence, we apply the following rule to compute value flows without sacrificing soundness or precision:

Rule 3. Given a STORE $*p = x$ and a LOAD $y = *q$, if x is an introduced auxiliary variable as function input or y is an introduced auxiliary variable as function output, apply the INDIRECT rule. Otherwise, apply the INDIRECT$_{\text{ALIAS}}$ rule.

With the above rule, in the callee function `movCarCdr`, all feasible indirect value flows involve introduced auxiliary variables for parameter `dst` or `src`. As a result, at call-site C_2, the VFG of function `movCarCdr` is computed the same as the one computed at C_1 in Figure 1(b). No spurious value flow is introduced.

Theorem 2. *In a context-sensitive analysis, computing indirect value flows with Rule 3 guarantees soundness and precision.*

Proof. We only prove soundness here and a similar proof can be derived for precision. Given $*p = x$ and $y = *q$, we have $x \rightarrow y$ if there exists A such that $A \rightarrow^* p$ and $A \rightarrow^* q$. The theorem trivially holds if A is in the same function as x and y, or if both p and q alias with the same parameter. Otherwise, assume that $*p = x$ is in $callee$ and $y = *q$ in $callee$. There must exist a function input fp such that we have $A \rightarrow^* fp$ and $fp \rightarrow^* q$. According to Rule 3, there must exist a value flow path from $x \rightarrow^* fp*^{in} \rightarrow^* y$, where $fp*^{in}$ is the introduced auxiliary variable. Similarly, if $*p = x$ is in $callee$ and $y = *q$ in $caller$, there must exist a value flow path $x \rightarrow^* fp*^{out} \rightarrow^* y$. If x and y are in the same function, then we have two inputs $fp1$ and $fp2$ such that $fp1 \rightarrow^* x$, $fp2 \rightarrow^* y$, $A \rightarrow^* fp1$ and $A \rightarrow^* fp2$. Hence $fp1$ and $fp2$ must alias with each other. According to Rule 3, there exists value flow $x \rightarrow^* y$. □

3. Context-Sensitivity

Conceptually, our approach to context-sensitivity involves two steps. We transform the program by cloning functions at different call-sites where their local value flows may be computed differently (Rule 3). With this transformation, local value flows of a function become context-independent. Hence each function can be analyzed individually. In the second step, we apply context-matching to inter-procedural value flow paths. For a feasible value flow path, the call-site of each pair of call and return edges on the path need to match with each other. This ensures that pointer values computed in one call-site cannot erroneously propagate to other call-sites of the same function. The two steps together guarantee context-sensitivity.

In practice, the above two steps are inter-dependent and need to be performed together. We need context-sensitive pointer information to effectively determine when function cloning needs to be applied, which in turn is required to compute such information.

3.1 Function Cloning

We clone functions at call-sites where their local value flows may be computed differently. As such, all local value flows become context-independent. In contrast, existing cloning-based approaches [4, 36] create a function clone at each distinct call path. Our approach avoids explicitly representing the exponentially large number of call paths in a program and only a small number of function clones are created as shown in our experiments in Section 5.

As discussed in Section 2.3, local value flows of a function may be computed differently given different input values.

Theorem 3. *The local value flows of a function are the same at different call-sites if 1) the set of introduced input variables are identical, 2) aliases between function inputs (represented as alias sets) are the same, and 3) for input variables which are function pointers, their points-to sets are the same.*

Proof. The first two conditions guarantee that local indirect value flows within the function are computed exactly the same, according to the INDIRECT and INDIRECT$_{\text{ALIAS}}$ value flow rules (Rule 3). The third condition ensures that all CALL statements in the function also behave the same, i.e., same value flows introduced via CALL statements. Since local direct value flows introduced via BASE and ASSIGN statements always exist regardless of the call-sites. The local value flows of the function will be computed the same. □

As for our example in Figure 1(a) and Figure 1(c), the function `movCarCdr` is called at two different locations, C_1 in `createList`, and C_2 in `mangleList`. Since the function inputs alias differently at C_1 and C_2, a function clone $\widehat{\texttt{movCarCdr}}$ is created at C_2 where its local value flows are computed with the given alias set $\{$`dst, src, dst->next`$^{\text{in}}\}$. For this example, the value flows of $\widehat{\texttt{movCarCdr}}$ happen to be identical to that of `movCarCdr` in Figure 1(b).

Various summary-based approaches [11, 25, 38, 40] use methods similar to ours to differentiate calling contexts. In contrast to our approach, those approaches compute parameterized summaries for each function, which are then instantiated at their call-sites to create the summaries for their callers. Compact function summaries are key to performance. However, it also suggests that some useful information may not be preserved in the summary and the computed result, may not preserve enough context information to precisely answer queries such as "do the two variables alias on a particular call path?". In addition, it is very difficult to compute a full compact function summary, especially when flow-sensitivity is needed. One solution is to use partial summary functions [38]. We bring this idea one step further by cloning the function instead of generating partial summaries. This enables us to compute precise pointer information efficiently without loss of generality.

3.2 Context Matching

By explicitly representing indirect value flows in the VFG and with our function cloning, context-sensitive pointer analysis can be simplified to a well known balanced-parentheses problem in CFL-reachability [29]. The CFL-reachability formulation is an extended graph reachability problem: graph edges are annotated with labels and two nodes in the graph are connected only if there exists a path such that the concatenation of the labels on the edges of the path is acceptable in a defined context-free language. We apply CFL-reachability to the VFG with a simple language for context matching [29], where the call-site C of each pair of inter-procedural value flow edges, $\xrightarrow{C_{in}}$ and $\xrightarrow{C_{out}}$, need to match with each other.

One advantage of the CFL-reachability formulation is that context (i.e., call path) and pointer information are implicitly encoded in a compact graph representation and they can be computed on-demand. Existing CFL-based context-sensitive pointer analyses [32, 39] define a language using a grammar to model both function calls and memory accesses at the same time. They can be very efficient for some applications where we only need to compute pointer information for a small subset of variables. However, if we need the precise pointer information for all variables, CFL-based pointer analysis is generally more expensive as it has O(L^3N^3) complexity [29], where L is the size of the grammar and N is the size of the graph.

Optimizations. We improve performance by computing CFL-reachability only for feasible inter-procedural value flow paths. In addition, since all local value flows become context-independent with our function cloning, we can develop effective optimizations to further improve its performance. Note that those optimizations are not applicable to existing CFL-based pointer analyses.

- We summarize the local value flow path $x \rightarrow^* y$ as a value flow edge while all other nodes on the path can be discarded. This greatly reduces the number of nodes visited during context matching, without loss of precision. Note that the summarized value flows from function inputs to outputs are effectively transfer functions as in summary-based approaches.
- We represent pointer values with respect to the values of function inputs, i.e., the points-to set of a variable is the set of function inputs it aliases. This greatly improves performance and reduces memory footprint. We compute full points-to sets for function pointers only, while the full points-to sets for other

variables can be computed on-demand as in other demand-driven pointer analyses [32, 41].

- We memoise matching inter-procedural value flow paths. Specifically, for a matching value flow path $ap \xrightarrow{C_{in}} fp \rightarrow^* ret \xrightarrow{C_{out}} out$, a local value flow edge $ap \rightarrow out$ will be introduced (in the caller function) as a summary of the path. This enables further optimization opportunities.

It can be proved that the above two steps together guarantee context-sensitivity: for recursion free programs, the analysis is as precise as inlining all function calls. For recursive programs, precise context-sensitive analysis is undecidable. Similar to [32, 36], we handle recursion by regarding function calls and returns within a recursive cycle as gotos. During the analysis, when recursion is detected, inter-procedural value flows within the recursive cycle are treated as intra-procedural value flows and the value flows of all functions in the cycle are computed together insensitively.

3.3 Heap Cloning

Recall that if a heap object O_H in the callee function escapes, a variable p_H is introduced at its call-site C in the caller function and we have $O_H \xrightarrow{C_{out}} p_H$. With heap cloning, for $O_H \xrightarrow{C_{1_{out}}} p'_H$ and $O_H \xrightarrow{C_{2_{out}}} p''_H$, two distinct heap objects will be introduced as the copy of O_H at the two different call-sites C_1 and C_2, respectively. As a result, heap objects created at different call-sites are distinguished. In contrast, the edge $O_H \xrightarrow{C_{out}} p_H$ is simply regarded as a local value flow edge if heap cloning is not supported. Previous works [17, 26] show that heap cloning can significantly improve precision.

```
char * sm_rpool_allocblock_x(rpool, size){
    p = sm_malloc(sizeof(SM_POOLHDR_T)+size);
    rpool->sm_pool = p;
    return p+sizeof(SM_POOLHDR_T);
}
void * sm_rpool_malloc_x(rpool, size){
O1: if (...) return sm_malloc(...);
O2: if (...) return sm_rpool_allocblock_x(...);
O3: return sm_rpool_allocblock_x(...);
}
HDR * allocheader(...){
C1:   h=sm_rpool_malloc_x(rp, sizeof(HDR));
C2:   ... = sm_rpool_malloc_x(rp,sizeof(HDR));
    ...
}
```

Figure 3. Code snippet from sendmail 8.17.

However, algorithms with heap cloning often suffer from scalability problems as the number of heap objects grows exponentially [17, 31, 32]. Existing analyses trade precision for scalability in various ways. The most common approach is the K-limiting approach [8, 31], which restricts the depth of the call paths to K, and call paths deeper than K are not distinguished. In practice, K is often set to a small number (2 or 3) for scalability. However, many applications, such as sendmail 8.17 in Figure 3, use 5 levels of wrapper functions (some are not shown in Figure 3) to support their customized memory allocation schemes. It is not practical to set K to such a large number. The authors in [17] propose an alternative solution which merges heap objects with equivalent structures to avoid creating too many heap objects. Their approach can distinguish full acyclic call paths and is very fast. However, it is based on a unification-based approximation [33] to merge as many objects as possible. Compared to inclusion-based analysis, unification-based analysis is much less precise.

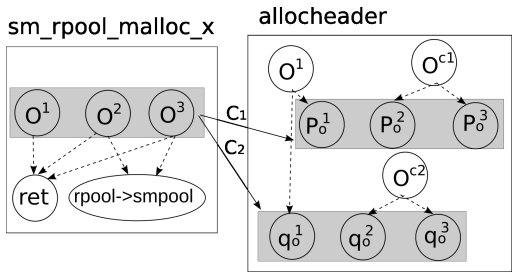

Figure 4. Heap cloning for sendmail 8.17 with one cloned object per call-site. Solid edge represent inter-procedural value flows.

We propose a different tradeoff for heap cloning. Similar to [17], we merge the clones for distinct objects O_{H_1} and O_{H_2} together if they *escape in the exact same way*, i.e., clones of O_{H_1} and O_{H_2} can be merged together if for any flow path $O_{H_1} \rightarrow^* ret$ where ret is a function output, we have $O_{H_2} \rightarrow^* ret$. This is both sound and precise as the merged objects are always pointed to by the same set of pointers. As in our example Figure 3, the three objects O^1, O^2 and O^3 in function `sm_rpool_malloc_x` may escape. As a result, in Figure 4, at C_1, three pointer variables P_{O^1}, P_{O^2} and P_{O^3} are introduced where $O^1 \xrightarrow{C_{1_{out}}} P_{O^1}$, $O^2 \xrightarrow{C_{1_{out}}} P_{O^2}$, and $O^3 \xrightarrow{C_{1_{out}}} P_{O^3}$. Since the two objects O^2 and O^3 escape in the same way, only one clone O^{C_1} is created for them at C_1.

Instead of bounding the depth of the call paths, we limit the number of objects to be cloned at each call-site. As in Figure 4, the number of clones is limited to one per call-site. Hence we do not create clones for O^1. This approximation works well in practice as many applications use deeply-nested wrapper functions on top of the C malloc function to create one or two objects. Some functions may potentially create many heap objects. However, very often only a few of them are created at runtime. Hence many of the created objects escape in the exact same way and their clones can be merged together. In addition, if an object escapes in a very complicated way then it is very hard to be analyzed precisely. In that case, heap cloning will not be very helpful in improving analysis precision.

4. The Algorithm

In our algorithm VF-PTRANALYSIS (Algorithm 1), the two interleaving steps for context-sensitivity are performed together. We start our analysis with only direct value flows. Since all local value flows are context-independent, we compute a local VFG for each individual function. During the analysis, new value flows will be introduced and new functions may be cloned. In the end, the algorithm terminates when all local value flows for all functions are computed.

In Line 2, the procedural INITIALIZE is called to initialize all direct value flow edges. The while loop in Lines 3 - 10 performs the analysis. The local VFG of a given function is computed in COMPUTEVFG (Line 5) , where function clones may be created and added to $funcSet$. For each function f, we maintain two sets of memory objects, $f.directSet$ and $f.indSet$ for those objects in f whose direct and indirect value flows need to be computed, respectively.

Next, context matching is effectively applied at every call-site to f by calling the procedure CONTEXTMATCHING (Lines 6 - 10). Heap objects escaping from f will be cloned and new local value flows will be introduced in the caller function for matching inter-procedural value flow paths. As a result, the caller function needs

Algorithm 1 Context-sensitive pointer analysis

```
    Let funcSet be the set of all functions
    Let f.directSet include all objects of f and f.indSet be ∅
    Let st(x), ld(x) be two empty sets for each variable x
 1: procedure VF-PTRANALYSIS
 2:     INITIALIZE()
 3:     while funcSet ≠ ∅ do
 4:         Select f, and remove it from funcSet
 5:         COMPUTEVFG(f)
 6:         for each call-site C where f is a target do
 7:             Let caller be the caller function
 8:             CONTEXTMATCHING(caller, f, C)
 9:         end for
10:     end while
11: end procedure
12: procedure INITIALIZE
13:     for each function f ∈ funcSet do
14:         for each BASE instruction p_A = &A in f do
15:             Add edge A → p_A
16:         end for
17:         for each ASSIGN instruction p = q in f do
18:             Add edge p → q
19:         end for
20:         for each STORE instruction *p = x in f do
21:             st(p) := st(p) ⋃ {x}
22:         end for
23:         for each LOAD instruction x = *p in f do
24:             ld(p) := ld(p) ⋃ {x}
25:         end for
26:         for each call-site C in f do
27:             Add value flow ap --C_in--> fp for parameter passing
28:             Add value flow ret --C_out--> out for the callee's return value
29:         end for
30:     end for
31: end procedure
```

Algorithm 2 Compute Local Value Flows of a Function

```
 1: procedure COMPUTEVFG(f)
 2:     while {f.directSet ⋃ f.indSet} ≠ ∅ do
 3:         TRAVERSEVFG(f)
 4:         Let C be a call-site in f and callee be the callee function
 5:         if input values of callee changed as Theorem 3 then
 6:             CLONEFUNC(f, callee, C)
 7:         end if
 8:         UPDATEVFG(f)
 9:     end while
10: end procedure
11: procedure TRAVERSEVFG(f)
12:     while f.directSet ≠ ∅ do
13:         Select O, and remove it from f.directSet
14:         Let V be the set of variables in f newly reachable from O
15:         for each variable v in V do
16:             pts(v) := pts(v) ⋃ {O}
17:             st(O) := st(O) ⋃ st(v)
18:             ld(O) := ld(O) ⋃ ld(v)
19:             if {st(v) ⋃ ld(v)} ≠ ∅ then
20:                 f.indSet := f.indSet ⋃ {O}
21:             end if
22:         end for
23:     end while
24: end procedure
25: procedure UPDATEVFG(f)
26:     while f.indSet ≠ ∅ do
27:         Select O, and remove it from f.indSet
28:         stset := st(O)
29:         ldset := ld(O)
30:         if O is function input which aliases with another input O_a then
31:             stset := stset ⋃ st(O_a)
32:             ldset := stset ⋃ ld(O_a)
33:         end if
34:         for each variable p ∈ stset do
35:             for each variable q ∈ ldset do
36:                 if indirect value flow exists as in Rule 3 then
                    // Reachability analysis can be applied to compute
                    // value flows flow-sensitively
37:                     ADDEDGE(f, p → q)
38:                 end if
39:             end for
40:         end for
41:     end while
42: end procedure
43: procedure ADDEDGE(f, p → q)
44:     Add p → q to the local VFG of f
45:     f.directSet := f.directSet ⋃ pts(p)
46: end procedure
```

to be re-analyzed. Since the algorithm always introduces new edges into the value flow graph, it is guaranteed to terminate.

4.1 Computing Local Value Flows

In Algorithm 2, the procedure COMPUTEVFG computes local value flows for each function f individually. In Line 3, local value flows of f are computed in procedure TRAVERSEVFG by traversing its local VFG. In Lines 4 - 7, we check at every call-site in f whether input values of the callee function have changed or not, and function cloning is performed accordingly (Line 6). The local VFG of f is then updated by introducing indirect value flows of each object, i.e., flows from variables stored into an object to those loaded from it (Line 8). As a result, the value flows of some memory objects need to be updated. The algorithm stops at a fixed point where the local VFG is complete and no more indirect value flows can be introduced.

In procedure TRAVERSEVFG, only local flow edges in f are followed during the traversal. The points-to sets of all visited pointer variables are updated (Line 16). For efficiently computing indirect value flows, we also keep track of the set of variables stored into an object and the set of variables loaded from it (Lines 17 and 18). In Lines 19 - 21, those objects whose indirect value flows need to be computed are included in $f.indSet$.

The procedure UPDATEVFG updates the local VFG with new indirect flow edges. Indirect value flows are computed according to the rule in Rule 3 (Lines 28 - 40). When the local VFG is updated with new value flow edge $p \rightarrow q$, as shown in ADDEDGE, the value flows of all memory objects that are pointed-to by p need to be updated (Line 45).

Flow-sensitivity. Note that in our approach, flow-sensitivity can be easily achieved by adopting the method proposed in [20] with small extensions. We compute a flow-sensitive local VFG for each function where each value flow is computed flow-sensitively. Direct value flows can be trivially computed in a flow-sensitive manner after translating the program into static single assignment form (SSA). For indirect value flows, each STORE and LOAD instruction in the original program is represented as a distinct node in the VFG and a sparse data-flow analysis is employed to check whether a STORE can reach a LOAD in the control flow graph without being *killed* by another STORE.

It is said that a STORE $*p = x$ to object O can *kill* all previous values stored to O if O represents a singleton address and the dereferenced pointer p points to object O only. This is also referred to as *strong update*. Hence for the sake of flow-sensitivity, we extend the rule in Theorem 3 where *must aliases* and *may aliases* between function inputs are differentiated and we also check whether a function input can be strongly updated in the callee

function or not. To the best of our knowledge, flow-sensitivity is not supported in existing CFL-based or cloning-based pointer analyses.

4.2 Function Cloning

Algorithm 3 Function Cloning

Let $\widehat{X}$ denote the clone of X
1: **procedure** CLONEFUNC($caller$, $callee$, C)
2: Let fp be a function input and $ap \xrightarrow{C_{in}} fp$
3: Let ret be a function output and $ret \xrightarrow{C_{out}} out$
4: **if** $\nexists\ \widehat{callee}$ with the same input values as at C **then**
5: $funcSet := funcSet \bigcup \{\widehat{callee}\}$
6: Duplicate the VFG of $callee$ to that of $\widehat{callee}$
7: **if** auxiliary variables need to be introduced for $\widehat{fp}$ **then**
8: Add $ap{*}^{in} \xrightarrow{C_{in}} \widehat{fp}{*}^{in}$, $fp{*}^{out} \xrightarrow{C_{out}} \widehat{ap}{*}^{out}$ as in Rule 1
9: $caller.indSet := caller.indSet \bigcup pts(ap)$
10: $\widehat{callee}.indSet := \widehat{callee}.indSet \bigcup \{\widehat{fp}\}$
11: **else if** Aliases of $\widehat{fp}$ changed **then**
12: $\widehat{callee}.indSet := \widehat{callee}.indSet \bigcup \{\widehat{fp}\}$
13: **end if**
14: **else**
15: **if** $\exists\ \widehat{fp}{*}^{in}, \widehat{fp}{*}^{out}$ and $\nexists\ fp{*}^{in}, fp{*}^{out}$ **then**
16: Add $ap{*}^{in} \xrightarrow{C_{in}} \widehat{fp}{*}^{in}$, $\widehat{fp}{*}^{out} \xrightarrow{C_{out}} ap{*}^{out}$
17: $caller.indSet := caller.indSet \bigcup pts(ap)$
18: **else if** $\exists\ \widehat{fp} \rightarrow^* \widehat{ret}$ and $\nexists\ fp \rightarrow^* ret$ **then**
19: ADDEDGE($caller, ap \rightarrow out$)
20: **end if**
21: **end if**
22: Remove $ap \xrightarrow{C_{in}} fp$ and add $ap \xrightarrow{C_{in}} \widehat{fp}$
23: Remove $ret \xrightarrow{C_{out}} out$ and add $\widehat{ret} \xrightarrow{C_{out}} out$
24: **end procedure**

Property 1. *All local value flows of function callee are included in its clone* $\widehat{callee}$.

By construction, Property 1 is enforced when a function is cloned. This is both precise and safe as at a call-site, input values to the callee function may change only if 1) new auxiliary variables need to be introduced, or 2) there are new aliases between function inputs, or 3) there are extra targets for a function pointer. In all three cases, previously computed value flows still exist.

As shown in CLONEFUNC in Algorithm 3, we duplicate all previously computed value flows to the local VFG of the cloned function (Lines 5 and 6). As such, the function clone does not need to be analyzed from scratch. In Line 8, auxiliary variables are introduced as in Rule 1: the auxiliary variables for input variable $fp1$ will be merged into those of $fp2$ if we have $ap1 \xrightarrow{C_{in}} fp1$, $ap2 \xrightarrow{C_{in}} fp2$ and $pts(ap1) = pts(ap2)$. During the analysis, if $pts(ap1) = pts(ap2)$ no longer holds, a copy of the initially merged node will be introduced as the auxiliary variables for $fp1$. This is always safe and precise.

In Lines 9 - 13, the $indSet$ for both the caller function and the function clone are modified so that their local VFGs can be effectively updated. Note that if there already exists a clone of the callee function with same input values, we can reuse the existing clone as the call target instead of creating a new one. The local VFG of the caller function needs to be updated accordingly (Lines 15 - 20). In Lines 22 and 23, we change the target of the call by redirecting all previous inter-procedural value flows to the cloned function. Because of Property 1, this is always safe and precise.

Algorithm 4 Context matching

1: **procedure** CONTEXTMATCHING($caller$, $callee$, C)
2: Let $O \rightarrow^* ret$ be a newly introduced flow path in $callee$ where O is an object that can return and ret is function output
3: **if** O is function input **then**
4: Let $ap \xrightarrow{C_{in}} O \rightarrow^* ret \xrightarrow{C_{out}} out$ be a flow path
5: ADDEDGE($caller, ap \rightarrow out$)
6: $funcSet := funcSet \bigcup \{caller\}$
7: **else if** O is heap object **then**
8: HEAPCLONE($caller$, $callee$, C, $O \rightarrow^* ret$)
9: $funcSet := funcSet \bigcup \{caller\}$
10: **end if**
11: **end procedure**
12: **procedure** HEAPCLONE($caller$, $callee$, C, $O_H \rightarrow^* ret$)
13: **if** $\nexists\ O_H \xrightarrow{C_{out}} p_H$ **then**
// Create new copy for O_H in $caller$
14: Add p_H at C and $O_H \xrightarrow{C_{out}} p_H$
15: Let $\widehat{O_H}$ be the copy for O_H at C, as described in Section 3.3
16: ADDEDGE($caller, \widehat{O_H} \rightarrow p_H$)
17: Add $O_H{*}^{out} \xrightarrow{C_{out}} p_H{*}^{out}$ as described in Rule 1
18: **else if** O_H, O_X were cloned together but escape differently **then**
// Split the merged clones in $caller$
19: Remove edge from the old clone $\widehat{O_X} \rightarrow p_H$
20: Let $\widehat{O_H}$ be the new copy for O_H, as described in Section 3.3
21: ADDEDGE($caller, \widehat{O_H} \rightarrow p_H$)
22: **end if**
23: Let $ret \xrightarrow{C_{out}} out$ be the inter-procedural flow edge
24: ADDEDGE($caller, p_H \rightarrow out$)
25: **end procedure**

4.3 Context Matching

In Algorithm 4, we apply context-matching in such a way that CFL-reachability is computed for matching inter-procedural value flow paths only. The observation is that new matching inter-procedural paths only exist if there exists a new value flow path from function inputs to function outputs (Line 2 in CONTEXTMATCHING). Matching inter-procedural value flow paths are then memoized and summarized as local value flows of the caller function (Lines 4 - 6). In Line 8, the procedure HEAPCLONE is called for heap objects escaping from the callee function. In both cases, new local value flow edges are introduced in the caller function hence the caller function is included in $funcSet$ for further updating (Lines 6 and 9).

In HEAPCLONE, for heap object O_H escaping from f, a heap object $\widehat{O_H}$ will be introduced as its copy in the caller function (Lines 15 and 20). As described in Section 3.3, objects escaping in the same way are cloned together, hence the same object may be used as copies of different objects at the same call-site. Similar to how we handle auxiliary variables, if two objects O_H, O_X, whose clones were merged together, now escape differently, their merged copies need to be separated. As shown in Lines 18 - 22, we split merged clones by introducing a new copy $\widehat{O_H}$ and replacing the flow edge $\widehat{O_X} \rightarrow p_H$ with a new edge $\widehat{O_H} \rightarrow p_H$ (Lines 21 - 23). This amounts to creating a copy of the initially merged node, which is always safe and precise.

5. Implementation and Experimental Results

We have implemented our context- and flow-sensitive points-to analysis in LLVM [16]. The analysis is also field-sensitive [27] in that for objects with struct types, pointers that point to distinct offsets of the same object are not regarded as aliases. In this section we evaluate the efficiency of our implementation in terms of runtime and memory consumption, against standard benchmarks used in the

Benchmark	NC-LOC	Bitcode	# Functions			# Pointers				Points-to Set	
	C/C++	Files	Original	Cloned	Total	Original	With Copies	Auxiliary	Total	Average	Max
sendmail 8.17	115.6K	17.9MB	1,340	201	1,541	103,517	285,935	381,168	667,103	1.25	77
httpd 2.0.63	177.8K	9.1MB	992	1	993	41,966	42,099	29,177	71,276	1.04	8
400.perlbench	126.3K	20.0MB	1,865	185	2,050	177,264	403,876	516,486	920,272	2.10	83
401.bzip2	5.7K	807KB	100	0	100	5,567	5,567	1,955	7,522	1.02	2
403.gcc	234.3K	50.6MB	5,577	1,514	7,091	432,651	975,295	1,610,456	2,585,751	4.14	249
429.mcf	1.6K	578KB	24	2	26	1,261	1,493	1,165	2,658	1.42	8
445.gobmk	157.6K	11.4MB	2,679	238	2,917	81,770	115,779	145,389	261,168	1.02	5
456.hmmer	20.7K	4.8MB	538	27	565	26,398	33,983	20,047	54,030	1.13	5
458.sjeng	10.5K	1.4MB	144	0	14	5,303	5,303	1,023	6,326	1.01	5
462.libquantum	2.6K	610KB	115	0	115	2,095	2,095	1,936	4,031	1.00	2
464.h264ref	36.1K	5.4MB	590	181	771	59,729	217,962	219,491	437,453	1.03	4
471.omnetpp	20.0K	19.6MB	2,887	38	2,925	53,294	60,004	211,706	271,710	1.13	88
473.astar	4.3K	836KB	167	3	170	3,042	3,172	2,910	6,082	1.02	4

Table 1. Summary and Statistics of the benchmark data. The analysis is performed without heap cloning.

points-to analysis literature, as well as its precision, by integration into an existing compiler.

5.1 Benchmarks

Table 1 shows summary information for the benchmarks used in this evaluation. For each benchmark, we list its version number, the number of non-commented lines of C/C++ code (NC-LOC) as reported by the SLOCCount [37] tool, and the size of the bitcode files generated by the LLVM (version 3.1) front-end. The two benchmarks sendmail 8.17 and httpd 2.0.63 were selected in order to compare our analysis with the Bootstrapping approach proposed by Kahlon [14]. Bootstrapping is a state-of-the-art flow- and context-sensitive pointer analysis; these two benchmarks are the largest applications used in Kahlon's experiments. The other benchmarks include all integer benchmarks from CPU2006, except for 483.xalancbmk, which we have not been able to successfully compile with LLVM-3.1.

Our approach to context-sensitivity, together with the various summary-based optimizations, is key to the scalability of our analysis. As shown in Table 1 (Columns 4 - 6), the number of functions that are cloned in our analysis is small. For all benchmarks, the number of function clones is less than or close to 30% of the number of functions in the original program, and it is less than 10% for 10 out of 13 benchmarks.

Columns 7 - 10 give the number of pointer variables in the original program, as well as the total number of pointer variables after the analysis, where copies of original program variables and auxiliary variables have been introduced. In our implementation, we introduce a copy for each global variable that is accessed in a function (including those indirectly accessed in its callees). As a result, for some benchmarks, e.g., sendmail 8.17, 400.perlbench, 403.gcc, and 464.h264ref, the number of pointer variables more than doubled after introducing these copies (Column 8). Column 9 and 10 present the number of introduced auxiliary variables, and the total number of pointers after the analysis. For 9 out of 13 benchmarks, the number of pointer variables increases by 20% to 100% after the analysis. For the four benchmarks listed above, the total number of pointer variables after the analysis becomes 6 to 8 times larger. This may sound expensive. However, it enables us to analyze each function individually. In all those benchmarks, the average number of pointers in each function is much smaller than the program size hence our analysis can be efficiently performed. The last two columns present the average and largest size of the points-to set, respectively. Since we compactly represent pointer values with respect to function inputs, the average size of the points-to set is very small. This is key to scalability and also explains the small memory footprint of our analysis as shown in Table 3.

5.2 Runtime and Memory Consumption Evaluation

Benchmark	Bootstrapping [14]		Our Approach	
	Time	Mem	Time	Mem
httpd 2.0.63	161s	161MB	2.4s	195MB
sendmail 8.17	939s	939MB	137s	1,346MB

Table 2. Comparison against Bootstrapping [14]. The results of Bootstrapping are directly taken from the paper, where Kahlon conducted his experiments on a slower machine (Intel Pentium4 3.2GHz with 2GB of memory). This is the closest comparison we can make.

We conducted our evaluation on an Intel XeonE5432 3.0GHz processor with 24GB of memory. Table 2 compares our analysis against Bootstrapping. Both analyses are context- and flow-sensitive. Since no precision data is reported in Kahlon's paper, we can only compare the performance. In this experiment, we disabled heap cloning in our analysis and restricted our memory usage to 2GB for a fair comparison. For the two benchmarks, on a slightly faster machine, our analysis is 60× and 7× faster, respectively. It does require more memory to preserve all value flow information computed during the analysis, so that we can precisely answer various alias queries from different client applications.

Table 3 shows the results of our analysis, with different heap cloning strategies, as described in Section 3.3. We evaluate four different heap cloning strategies; namely, no-cloning, 2-level cloning, 2-limiting cloning, and full cloning. The 2-level cloning strategy limits the length of call paths to two, where the call-site and the object creation site, together, are used to distinguish a heap object. In 2-limiting cloning, the number of objects created at each call-site is restricted to two. We use a simple heuristic to choose objects to be cloned: objects that directly return are chosen first, followed by those that escape to input parameters and returns. We clone objects that escape to global variables last.

Recall that we clone two objects together if they escape in the same way. Hence in Table 3, we present the total number of clones (#Clones), together with the number of clones that have been merged (#Merges). We also show the runtime and memory consumption under each strategy. Some benchmarks are processed too quickly for memory consumption to be precisely measured. Hence, for those benchmarks, the memory consumption is not given.

As shown in Table 3, for 10 out 13 benchmarks, full heap cloning support can be achieved with relatively small performance loss. Merging cloned objects is very effective for the two benchmarks, 445.gobmk and 462.libquantum, where 25,827 out of 38,643 cloned objects, and 3,909 out of 4,226 cloned ob-

Benchmark	No-Cloning		2-Level Cloning		2-Limiting Cloning		Full-Cloning	
	#Objs	Time:Mem	#Clones: #Merges	Time:Mem	#Clones: #Merges	Time:Mem	#Clones: #Merges	Time:Mem
sendmail 8.17	14	137s:1.35GB	14,033:4,644	1,523s:1.94GB	27,506:6,983	374s:1.73GB	–:–	–:–
httpd 2.0.63	4	2.4s:195MB	2:1	2.4s:195MB	2:1	2.4s:195MB	2:1	2.4s:195MB
400.perlbench	8	522s:2.61GB	9,332:494	981s:3.13GB	11,068:719	859s:3.27GB	–:–	–:–
401.bzip2	5	68ms:–	24:0	73ms:–	30:0	90ms:-	44:0	85ms:–
403.gcc	9	1,097s:6.3GB	–:–	–:–	–:–	–:–	–:–	–:–
429.mcf	4	25ms:–	6:0	24ms:–	6:0	24ms:-	6:0	27ms:–
445.gobmk	22	120s:696MB	1,795:153	131s:754MB	3,623:420	136s:837MB	38,643:25,827	150s:966MB
456.hmmer	45	585ms:–	1,058:74	618ms:–	1,375:139	600ms:–	5,605:1,951	1.1s:–
458.sjeng	10	111ms:–	38:0	113ms:–	33:0	112ms:–	38:0	120ms:–
462.libquantum	19	59ms:-	480:163	79ms:–	480:183	74ms:–	4,226:3,909	141ms:–
464.h264ref	169	4.2s:327MB	2,009:0	4.7s:345MB	994:0	4.3s:330MB	20,324:1,260	20.3s:455MB
471.omnetpp	211	21.1s:1.09GB	9,850:1,345	103s:1.20GB	11,580:1,803	115s:1.20GB	14,003:2,064	136s:1.27GB
473.astar	32	101ms:–	207:10	119ms:–	199:11	103ms:–	390:126	129ms:–

Table 3. Runtime (minutes:seconds) and memory performance of our analysis using different heap cloning strategies. #Clones is the number of objects that have been cloned, and #Merges is the number of merged object clones.

jects, can be merged together under the full-cloning strategy, respectively. However, for the three benchmarks sendmail 8.17, 400.perlbench, and 403.gcc, it becomes expensive to compute value flows for each cloned heap object and the analysis cannot finish in 4 hours when full cloning is enabled. In 403.gcc, there are several very large functions and any heap cloning strategy will result in tens of thousands of heap objects being created in those functions. As a result, it can only be analyzed without heap cloning support.

Next we compare the two different heap cloning approximations for the two benchmarks sendmail 8.17 and 400.perlbench. For sendmail 8.17, 2-level cloning is 4× slower than 2-limiting cloning, while the number of clones being introduced is actually much smaller. The reason is that with 2-level cloning, the average points-to set size increases to 3.22, compared to 1.97 in 2-limiting cloning. Hence more value flows need to be computed. It seems that for this benchmark, the analysis result is more precise with 2-level cloning: there are 147 functions being cloned, compared to 193 in 2-limiting cloning, suggesting less aliases. However, for 400.perlbench, there are 153 function clones with 2-level cloning, compared to 95 with 2-limiting cloning. In general, it is difficult to tell which approximation is more precise but the 2-limiting approximation scales better.

5.3 Precision Evaluation

Benchmark	Number of LICMs		
	BasicAA	DSA	Ours
400.perlbench	3,567	461	4,273
401.bzip2	899	108	3,023
403.gcc	18,750	1,243	24,320
429.mcf	165	4	221
445.gobmk	8,863	845	10,754
456.hmmer	5,868	429	10,013
458.sjeng	960	416	1,124
462.libquantum	1,090	232	1,156
464.h264ref	13,938	1,749	37,574
471.omnetpp	1,936	880	2,008
473.astar	904	542	962

Table 4. Number of instructions hoisted out of loop with different alias analysis, the larger the better. Our analysis is performed without heap cloning.

We evaluate the precision of our analysis using a classic compiler optimization as the client application: loop invariant code motion (LICM). Table 4 compares our analysis with two alias analyses in LLVM, the default alias analysis (BasicAA), and the data structure alias analysis (DSA) [17]. BasicAA is a simple intra-procedural alias analysis, and DSA is a context-sensitive, unification based alias analysis with full heap cloning support. As shown in the table, with our analysis, the optimization becomes much more effective: for 401.bzip2, the number of instructions being hoisted out of loops is 3,023, compared to 899 with BasicAA, and 108 with DSA.

For this optimization, heap cloning makes little differences in our experiments. This is because pointers referring to the same heap object are conservatively regarded as may aliases, since one heap object may potentially represent multiple objects created at runtime. However, the optimization requires precise must alias information to be effective and there is little difference whether we distinguish distinct heap objects or not. This also explains why BasicAA also outperforms DSA. BasicAA implements quite a few effective checks to compute must and may alias information by examining the local use-def chain of an instruction. While in DSA, such information is computed based on a graph node representation with all aliased pointers being unified. As a result, only a small number of must aliases can be identified. Overall, our analysis can significantly improve the effectiveness of this optimization.

6. Related Work

There are many variations of pointer analysis that make different trade-offs between precision and run time, including a number of analyses that are both context- and flow-sensitive.

Precise pointer analysis. Earlier work on flow- and context-sensitive (FSCS) pointer analysis has shown to scale up to thousands of LOC [5, 11, 15, 38]. Great progress has been made recently. Zhu [42] proposes a FSCS pointer analysis using symbolic summary functions. The approach scales to 200 KLOC of C code. In [14], a summary-based approach is also used to scale FSCS pointer analysis. To further improve the performance, the author bootstraps the more precise and costly FSCS pointer analysis by using cheaper and less precise analyses to partition programs into small sections that can be analyzed independently. The algorithm scales up to 128 KLOC of C code. The bootstrapping idea is further extended in [40], where the authors partition the program variables into different points-to levels using an insensitive analysis. FSCS pointer analysis is then applied by processing variables level by level, from the highest to the lowest. The analyzed results of higher level variables can be used in computing the context- and flow-

sensitive pointer information for variables in the lower level. This is the first FSCS analysis that scales to large applications with millions of LOC. The analysis is not fully field-sensitive as struct field accesses are differentiated by scalarization, which cannot be applied to structs whose addresses escape. More importantly, the above approaches do not support heap cloning, which is very important for analysis precision and often a large overhead to performance [17, 26]. To the best of our knowledge, our analysis is the first FSCS analysis that supports heap cloning and scales to large applications.

Context-sensitivity. Many pointer analysis algorithms focus on context-sensitivity only [4, 17, 25, 32, 36, 39, 43]. The approaches in [4, 31] trade off precision for scalability using a *k-limited* representation which restricts the length of call paths to a small fixed number k. This leads to a different yet interesting research question: how to effectively distinguish contexts without representing the full call path? For object-oriented languages, some smart heuristics (e.g., *object sensitivity* [24] and *type sensitivity* [31]) have been developed to effectively differentiate contexts. It is reported that with those heuristics, the loss of precision is insignificant. Alternatively, as in [17, 25, 32, 36, 43], the context can be precisely represented using the full call path (with cycles on the path being discarded). This guarantees precision, making scalability the main challenge.

Most precise pointer analyses developed to date [25, 40, 43] are summary-based. They develop different algorithms to generate function summaries and context-sensitivity is efficiently achieved by applying these function summaries at their corresponding call-sites. On the other hand, the authors in [4, 36] use a clone-based approach which differentiates calling contexts at distinct call paths. Binary-decision diagrams (BDDs) are used to compactly represent the exponentially large number of contexts. The algorithms in [23, 30, 32, 39] formulate context-sensitive pointer analysis as a CFL-reachability problem, where a context-free language is defined to simultaneously model heap accesses and function calls. The authors develop a refinement-based algorithm for scalability: imprecise results with approximation are computed first, which can be refined on queries for more precise results. Our approach to context-sensitivity differs from previous work in that we apply function cloning and CFL-reachability analysis together for context-sensitivity, and compute various summaries for efficiency.

The use of BDDs [1] has been adopted in many pointer analysis algorithms, especially the context-sensitive ones [4, 36, 40, 43]. It attempts to solve the problem of the large amount of data in context-sensitive analysis by representing redundancy efficiently. Our approach to this problem is to use context abstraction for equivalent contexts on distinct call paths and we represent pointer information with respect to the inputs of the context.

Heap-cloning. A number of context-sensitive pointer analyses use a context-sensitive heap abstraction [4, 17, 25, 32, 35, 39]. This abstraction is also referred to as object-sensitivity for object-oriented languages [31]. Those approaches trade precision for scalability in different ways: Lattner et al. [17] use a unification based approximation, the authors in [32, 39] employ a refinement-based approach, Sui et al. [35] use a demand-drive approach based on alias queries from compiler optimizations, and the most common method is to limit the depth of the call paths as in [4, 31]. In this work, we propose an alternative approximation to efficiently support heap cloning by limiting the number of objects that can be cloned at a call-site.

Flow-sensitivity. Flow-sensitive pointer analysis has attracted much attention recently [12, 13, 18, 20]. One of the key insights is to use a sparse representation, such as SSA, so that the def-use information of pointers can be efficiently propagated on the CFG. For example, the approach adopted in this paper [20] directly represents pointer def-use information as flow edges so that flow-sensitive pointer analysis can be performed efficiently.

Value flow. Value flow analysis computes which program variables hold which values of interest and it has been studied in many different areas, including compiler optimizations [2, 3], error detection [6, 34], software validation [9], and symbolic evaluation [28]. Existing value flow analyses rely on external points-to analysis to handle memory dependencies with aliases. Our approach can be used in existing value flow analyses to make them more effective.

7. Conclusion

In this paper, we present a novel method for context-sensitive analysis using the value flow graph formulation that scales well to large applications. We highlight the cause of imprecision in previous approaches and propose a solution to address this problem. We prove that our method is as precise as inlining all function calls and we show that unlike existing clone-based or CFL-based approaches, flow-sensitivity can be easily integrated in our approach. By applying context sensitivity to both, local variables and heap objects, we propose a new approximation for heap cloning.

We develop an efficient context-, flow-, and field-sensitive pointer analysis and implement it in a production compiler: LLVM. Experimental results using the SPEC CPU benchmarks and other real-world code-bases show that our analysis is much faster than existing approaches, and that its precision can significantly improve the effectiveness of existing compiler optimizations such as loop invariant code motion.

References

[1] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 103–114. ACM, 2003.

[2] R. Bodik. *Path-sensitive, value-flow optimizations of programs*. PhD thesis, University of Pittsburgh, 1999.

[3] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 237–251. ACM, 1998.

[4] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 243–262. ACM, 2009.

[5] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 133–146. ACM, 1999.

[6] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 480–491. ACM, 2007.

[7] C. Cifuentes, N. Keynes, L. Li, N. Hawes, M. Valdiviezo, A. Browne, J. Zimmermann, A. Craik, D. Teoh, and C. Hoermann. Static deep error checking in large system applications using parfait. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 432–435, New York, NY, USA, 2011. ACM.

[8] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond K-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 230–241. ACM, 1994.

[9] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 12–22. ACM, 2004.

[10] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, 2008.

[11] 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*, PLDI '94, pages 242–256. ACM, 1994.

[12] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238. ACM, 2009.

[13] B. C. Hardekopf. *Pointer analysis: building a foundation for effective program analysis*. PhD thesis, University of Texas at Austin, 2009.

[14] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 249–259. ACM, 2008.

[15] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248. ACM, 1992.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[17] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 278–289. ACM, 2007.

[18] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16. ACM, 2011.

[19] L. Li, C. Cifuentes, and N. Keynes. Practical and effective symbolic analysis for buffer overflow detection. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 317–326. ACM, 2010.

[20] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 343–353. ACM, 2011.

[21] L. Li, H. Feng, and J. Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Transactions on Architecture Code Optimization*, 6(3):9:1–9:17, Oct. 2009.

[22] L. Li, J. Xue, and J. Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Transactions on Embedded Computing Systems*, 10(2):28:1–28:42, Jan. 2011.

[23] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In R. Jhala and K. Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 61–81. Springer Berlin Heidelberg, 2013.

[24] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transaction on Software Engineering Methodolology*, 14(1):1–41, Jan. 2005.

[25] E. M. Nystrom, H. S. Kim, and W. M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS'04*, pages 165–180, 2004.

[26] E. M. Nystrom, H. S. Kim, and W. M. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 43–48. ACM, 2004.

[27] D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 30(1), Nov. 2007.

[28] 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*, POPL '77, pages 104–118. ACM, 1977.

[29] T. Reps. Program analysis via graph reachability. In *Proceedings of the 1997 international symposium on Logic programming*, ILPS '97, pages 5–19. MIT Press, 1997.

[30] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, New York, NY, USA, 2012. ACM.

[31] 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*, POPL '11, pages 17–30. ACM, 2011.

[32] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 387–400. ACM, 2006.

[33] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41. ACM, 1996.

[34] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 254–264, New York, NY, USA, 2012. ACM.

[35] Y. Sui, L. Yue, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 2013 International Symposium on Code Generation and Optmization*, CGO '13, New York, NY, USA, 2013. ACM.

[36] 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*, PLDI '04, pages 131–144. ACM, 2004.

[37] D. A. Wheeler. SLOC Count User Guide. `http://www.dwheeler.com/sloccount/`. Last accessed: 11 Nov 2012.

[38] 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*, PLDI '95, pages 1–12. ACM, 1995.

[39] 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*, Genoa, pages 98–122. Springer-Verlag, 2009.

[40] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 218–229. ACM, 2010.

[41] 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*, POPL '08, pages 197–208. ACM, 2008.

[42] J. Zhu. Towards scalable flow and context sensitive pointer analysis. In *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, pages 831–836. ACM, 2005.

[43] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 145–157. ACM, 2004.