Error Checking with Client-Driven Pointer Analysis

Samuel Z. Guyer, Calvin Lin

The University of Texas at Austin, Department of Computer Sciences, Austin, TX 78712, USA

Abstract

This paper presents a new *client-driven* pointer analysis algorithm that automatically adjusts its precision in response to the needs of client analyses. Using five significant error detection problems as clients, we evaluate our algorithm on 18 real C programs. We compare the accuracy and performance of our algorithm against several commonly-used fixed-precision algorithms. We find that the client-driven approach effectively balances cost and precision, often producing results as accurate as fixed-precision algorithms that are many times more costly. Our algorithm works because many client problems only need a small amount of extra precision applied to selected portions of each input program.

1 Introduction

Pointer analysis is critical for effectively analyzing programs written in languages such as C, C++, and Java, which make heavy use of pointers and pointer-based data structures. The goal of pointer analysis is to disambiguate indirect memory references so that subsequent compiler passes have a more accurate view of program behavior. In this sense, pointer analysis is not a stand-alone task: its purpose is to provide pointer information to other *client* analyses.

Existing pointer analysis algorithms differ considerably in their precision. Previous research has generally agreed that more precise algorithms are often significantly more costly to compute, but previous work has disagreed on whether more precise algorithms yield more accurate results and whether these results are worth the additional cost [30,28,19,10,26]. In fact, a recent survey by Hind claims that the choice of pointer analysis algorithm should be dictated by the needs of the client analyses [18].

⋆ This work is supported by NSF grants CCR-0085792, EIA-0303609, ACI-0313263, ACI-9984660, DARPA Contract #F30602-97-1-0150, and an IBM Faculty Partnership Award.

```
p = safe_string_copy("Good");          char * safe_string_copy(char * s)
q = safe_string_copy("Bad");           {
r = safe_string_copy("Ugly");            if (s != 0) return strdup(s);
                                         else return 0;
                                       }
```

Fig. 1. Context-insensitive pointer analysis hurts accuracy, but whether or not that matters depends on the client analysis.

In this paper we present a new *client-driven* pointer analysis algorithm that addresses this viewpoint directly: it automatically adjusts its precision to match the needs of the client. The key idea is to discover where precision is needed by running a fast initial pass of the client. The pointer and client analyses run together in an integrated framework, allowing the client to provide feedback about the quality of the pointer information that it receives. Using these initial results, our algorithm constructs a precision policy customized to the needs of the client and input program. This approach is related to demand-driven analysis [20,17] but solves a different problem: while demand-driven algorithms determine which parts of the analysis need to be computed, client-driven analysis determines which parts need to be computed using more precision.

As an example of how different clients require different amounts of precision, consider a context-insensitive analysis of the string copying routine in Figure 1: the pointer parameter `s` merges information from all the possible input strings and transfers it to the output string. For a client that associates dataflow facts with string buffers, this could severely hurt accuracy—the appropriate action is to treat the routine context-sensitively. However, for a client that is not concerned with strings, the imprecision is irrelevant.

We evaluate our algorithm using five security and error detection problems as clients. These clients are demanding analysis problems that stress the capabilities of the pointer analyzer, but with adequate pointer analysis support they can detect significant and complex program defects. We compare our algorithm against four fixed-precision algorithms on a suite of 18 real C programs. We measure the cost in terms of time and space, and we measure the client's accuracy simply as the number of errors reported: the analysis is conservative, so fewer error reports always indicates fewer false positives.

This paper, which is an extended version of earlier work [16], makes the following contributions. (1) We present a client-driven pointer analysis algorithm that adapts its precision policy to the needs of client analyses. For our five error detection clients, this algorithm effectively discovers where to apply more analysis effort to reduce the number of false positives. (2) We present empirical evidence that different analysis clients benefit from different kinds of precision—flow-sensitivity, context-sensitivity, or both. In most cases only a small part of each input program needs such precision; our algorithm works because it automatically identifies these parts. (3) Our results show that whole-program dataflow analysis is an accurate and efficient tool for error detection when it is given adequate pointer information.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the implementation of our framework, and Section 4 presents our client-driven algorithm. Section 5 describes our experimental methodology. Section 6 presents our results, and we conclude in Section 7.

2 Related Work

Previous work in program analysis, including pointer analysis, has explored ways to reduce the cost of analysis while still producing an accurate result. In this section, we compare our client-driven algorithm with this previous work. We also describe recent related work in error detection, focusing on the role of pointer analysis.

2.1 Precision versus cost of analysis

Iterative flow analysis [25] is an algorithm that adjusts its precision automatically in response to the quality of the results. Plevyak and Chien use this algorithm to determine the concrete types of objects in programs written using the Concurrent Aggregates object-oriented language. When imprecision in the analysis causes a type conflict, the algorithm can perform *function splitting*, which provides context-sensitivity, or *data splitting*, which divides object creation sites so that a single site can generate objects of different types. Brylow and Palsberg use a comparable algorithm to control the level of context-sensitivity for deadline analysis of real-time, interrupt-driven software [3]. The basic mechanism behind both of these approaches is similar to ours, but it differs in important ways. First, since the type of an object cannot change, iterative flow analysis does not include flow-sensitivity. By contrast, our approach supports a larger class of client analyses, known as *type-state* problems [32], which include flow-sensitive problems. More significantly, our algorithm manages the precision of both the client and the pointer analysis, allowing it to detect when pointer aliasing is the cause of information loss.

Demand-driven pointer analysis [17] addresses the cost of pointer analysis by computing just enough information to determine the points-to sets for a specific subset of the program variables. Client-driven pointer analysis is similar in the sense that it is driven by a specific query into the results. However, the two algorithms use this information to manage different aspects of the algorithm. Demand-driven pointer analysis is a fixed-precision analysis that computes only the necessary part of the solution. Client-driven analysis dynamically varies precision but always computes an exhaustive solution. The two ideas are complementary and could be combined to obtain the benefits of both.

Demand interprocedural dataflow analysis [20] also avoids the cost of exhaustive program analysis by focusing on the computation of specific dataflow facts. This algorithm produces precise results in polynomial time for a class of dataflow analyses problems called IFDS—interprocedural, finite, distributive, subset problems. However, this class does not include pointer analysis, particularly when it supports strong updates, which removes the distributive property.

Combined pointer analysis [36] uses different pointer algorithms on different parts of the program. This technique divides the assignments in a program into classes and uses a heuristic to choose different pointer analysis algorithms for the different classes. Zhang et al. evaluate this algorithm by measuring the number of possible objects accessed or modified at pointer dereferences. Instead of using a heuristic, client-driven pointer analysis is guided by feedback: it determines the need for precision dynamically by monitoring the analysis.

A number of previous papers have compared different pointer analysis algorithms, using both direct measurements (sizes of computed points-to sets) and indirect measurements (transitive effects on subsequent analyses). We find that the average points-to set size is not a good measure of the analysis because it treats all pointers as equals. For example, one algorithm might be more accurate than another by reducing the points-to set of a single variable by one pointer. While the overall measure is hardly affected, that one variable could be the critical distinction for the client. We also find that error detection is more demanding than the clients used in previous studies: the transitive benefits of higher precision are more apparent for our clients.

2.2 Pointer analysis for error detection

One of the major challenges in analyzing C programs is to construct a model of the store that is precise enough to support accurate error detection. Previous work has generally settled for a low-cost fixed-policy pointer analysis, which computes minimal store information without overwhelming the cost of error detection analysis [27,2,11]. Unfortunately, this store information often proves inadequate. Experiences with the ESP system [7] illustrate this problem: while its dataflow analysis engine is more powerful and more efficient than ours, the imprecision of its underlying pointer analysis can block program verification. The authors solve this problem by manually cloning procedures in the application in order to mimic context-sensitivity. By contrast, our solution instead automatically identifies these procedures that require context-sensitivity. Our algorithm detects when imprecision in the store model hampers the client, and our algorithm automatically increases precision in the parts of the program where it's needed.

More recent work has focused on developing a pointer analysis specifically for error detection [24], using the format string vulnerability as a basis for evaluation. The internal representation used for this analysis, called *IPSSA*, is very similar to our interprocedural factored def-use chains (see Section 3). Nevertheless, the approach differs from ours in several ways. First, the analysis algorithm is fixed precision: the authors improve performance by making an *a priori* decision about which parts of the application programs need more precision (the so-called *hot locations*). Second, it uses unsound assumptions to reduce the number of false positives. Our algorithm is sound (within the limitations described later), which makes it applicable to other clients, such as optimization, which cannot tolerate false negatives. Furthermore, sound analysis allows us to validate programs as bug-free. Finally, we show later in this paper that detecting format string vulnerabilities is not a problem that requires very precise analysis: almost perfect results are produced by context-insensitive analysis.

3 Analysis Framework

Our analysis framework is part of the Broadway compiler system, which supports high-level analysis and optimization of C programs [15,13]. In this section we describe the details of this framework, including the overall architecture, the representation of pointer information, and the analysis algorithm. Our framework has two enabling features that are critical to our client-driven analysis algorithm. First, it solves both the pointer and client analysis problems simultaneously, which allows it to monitor and control their interaction. Second, it allows precision to be specified at a fine grain: context-sensitivity can be controlled on a per-procedure basis, and flow-sensitivity can be controlled on a per-memory-location basis. The client-driven algorithm supplies the ***precision policy*** that decides which procedures to make context-sensitive and which memory locations to make flow-sensitive.

Table 1 summarizes the design dimensions of our analysis framework. It brings together several commonly-used algorithms for pointer and dataflow analysis, each with its own cost and precision characteristics [1,5,9,12,22]. The main contribution of this paper is not a new addition to these algorithms, but rather a mechanism for combining different algorithms during analysis and a policy for choosing which algorithm to use on which parts of the input programs.

We use a lightweight annotation language to specify the client analysis problems [14]. The error checking clients that we present in Section 5 are all designed to detect improper or unsafe use of system library calls in application programs. The language is designed to extend compiler support to software libraries; it is not used to describe the application programs themselves. The language allows us to concisely summarize the pointer behavior of library routines, and it provides a way to define new library-specific dataflow analysis problems.

Feature	Setting
Representation	Points-to sets using storage shape graph
Flow-sensitivity	Configurable—on a per-object basis
Context-sensitivity	Configurable—on a per-procedure basis
Assignments	Uni-directional (subset-based)
Strong updates	Yes–when applicable (see discussion)
Flow dependences	Factored use/def chains
Struct/union fields	Optional—turned on by default
Program scope	Whole-program, interprocedural
Heap object naming	By allocation site (see discussion)
Pointer arithmetic	Limited to pointers within an object
Arrays	All elements represented by a single node

Table 1
Specific features of our pointer analysis framework.

The remainder of this section describes our analysis framework, which provides the underlying analysis mechanisms. The framework consists of the following major components:

- **Program representation**: a traditional intermediate representation consisting of simple statements organized into a control-flow graph.
- **Memory representation**: a storage shape graph in which the vertices represent memory locations (variables and heap objects), and the edges represent points-to relationships.
- **Analysis algorithm**: an iterative dataflow analysis algorithm that simultaneously computes pointer information and solves client dataflow analysis problems.

3.1 Program representation

Our internal program representation supports the analysis framework in three ways: (1) it represents C code in a canonical form that consists of sequences of simple operations organized into a control-flow graph, (2) it enables whole-program analysis by bringing together all the procedures in a program (even across multiple source files) and organizing them into a call graph, (3) it provides per-procedure context sensitivity through procedure cloning.

Our compiler accepts as input a set of C source files, which it processes in several ways in preparation for analysis. In particular, it dismantles the code into a medium-level intermediate representation. This IR consists of simple assignment statements, similar to three-address instructions, organized into basic blocks, which are in turn organized into a control-flow graph. This representation preserves some of the high-level constructs of C, such as `struct` types and `union` types, and array indexing.

Our system implements per-procedure context sensitivity by altering the program

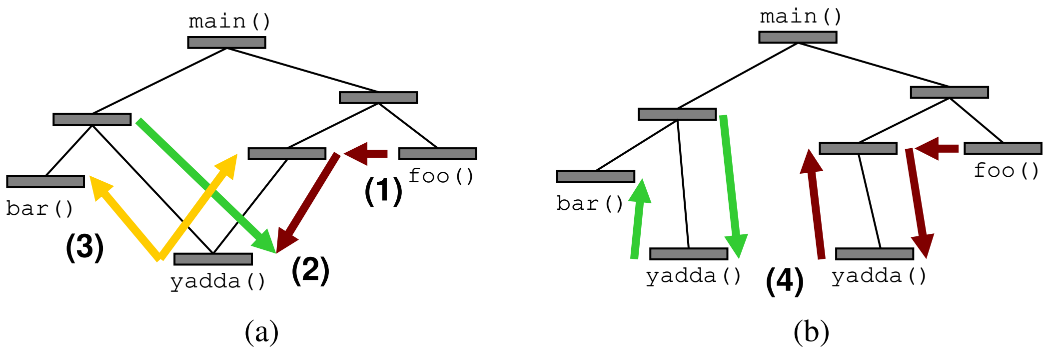


Fig. 2. Our framework implements context sensitivity by cloning. (a) Context insensitivity allows unrealizable paths: information generated at `foo()` *(1)* is merged at the shared procedure `yadda()` *(2)* and flows to the call site in `bar()` *(3)*. (b) With cloning *(4)*, the two paths are kept separate.

representation to make each calling context explicit. To apply context sensitivity to a procedure our compiler creates a logical clone of the procedure for each of its call sites. Figure 2 shows an example of this cloning process on a program's call graph. In the figure the `yadda()` procedure is used at two different call sites. Cloning the procedure provides a separate copy for each call site. This approach provides a uniform view of the program structure, independent of context sensitivity. Specifically, since our analysis algorithm associates dataflow facts with program points, cloning a procedure provides a separate set of program points, thereby keeping dataflow facts from the different calling contexts separate. Recent research has produced more efficient methods of implementing context-sensitivity [23,34,35], but we show in Section 6 that the amount of context-sensitivity needed is typically small.

To analyze a context-insensitive procedure we create a single instantiation and merge the information from all of its call sites. Since our analysis is interprocedural, we still visit all of the calling contexts. However, when no changes occur to the input flow values, the analyzer can often skip over a context-insensitive procedure call, which helps the analysis converge quickly. The main drawback of context-insensitive analysis is that it suffers from the unrealizable paths problem [35]: analysis information from one call site flows back to all the other call sites. Figure 2 shows how this problem can affect dataflow analysis. Without context sensitivity, dataflow information merges at the shared procedure `yadda()` and flows back to all call sites. As a result dataflow information from `foo()` flows to `bar()`, an unrealizable path. Cloning the shared procedure keeps separate the information from the two callers.

3.2 Memory representation

Our representation of the objects in a program is based on the storage shape graph [4], adapted for C programs, and it includes a number of improvements developed in more recent work [9,22,35]. The memory representation plays two important roles in the analysis: (1) it manages the memory abstraction, including the granularity of the memory model and the mapping from abstract memory locations to concrete memory locations, and (2) it provides per-object flow sensitivity by managing how flow values are associated with objects. For flow-sensitive objects the system builds factored def-use chains and associates a separate flow value with each def; for flow-insensitive objects the system only maintains a single flow value and does not need to compute reaching definitions.

The nodes of our storage shape graph represent all addressable objects in memory, including variables, structures, arrays, and heap allocated memory. We decompose complex objects into finer structures in order to more accurately model their behavior. For example, each field of a structure is represented by a separate node, and each instantiation of a structure includes a full set of these field nodes. We represent all the elements of an array with a single node.

The nodes that represent program variables (local and global) are indexed by their declarations. This scheme produces the expected behavior for context-sensitive procedures: each clone of the procedure has its own set of local variables, and thus its own set of nodes, thereby keeping the analysis information separate in each calling context.

We index heap-allocated memory according the program location of the allocation—typically, a call to `malloc()` or `calloc()`. By using the program location as the index, we obtain the same naming behavior for heap allocated memory as for local variables: in the context-insensitive case, our system generates one node for each static call to `malloc()`, while in the context-sensitive case, it generates one node for each call to `malloc()` in each procedure clone.

```
void main()                 void foo(int * p)
{                           {
  int x;                      if (some_condition)
  int y;                        (*p) = 6;
                              else
  foo(&x);                      (*p) = 7;
  foo(&y);                    // -- Phi function: merge x or y ?
}                           }
```

Fig. 3. Our variation of SSA form separates ϕ functions that occur in different contexts.

For flow-sensitive objects our framework records their uses and defs and organizes them into factored use-def chains [31]. This data structure is similar in spirit to SSA form [6], and it is well-suited for efficient dataflow analysis, especially for sparse problems, such as pointer analysis, constant propagation, and many kinds of error checking. The analyzer associates dataflow facts, such as points-to sets and client lattice flow values, with each def of an object in memory (node in the storage shape graph). At a use of the object, the analysis can quickly retrieve the current values by following the use-def chain.

Our representation of factored use-def chains differs from traditional SSA form because it is designed to overcome some of the limitations of SSA form. These limitations become apparent when performing interprocedural analysis in the presence of pointers. The example code in Figure 3 shows the two of these problems. The procedure `foo` modifies a variable indirectly through a pointer, but since the modification occurs in a conditional branch, SSA form requires a ϕ function at the confluence point to merge the flow values. The first problem is that the procedure is called in two places, with two different input values for `p`, address of `x` and address of `y`. However, the mergepoint in `foo` only merges one of the variables, depending on the calling context. The second problem is that the analyzer only discovers the need for these mergepoints during the pointer analysis. Therefore, we cannot separate the computation of use-def chains from the pointer analysis.

Previous work addresses this problem by creating a synthetic name for the target of the pointer `p` and adding a mergepoint for that synthetic variable [35]. Using this approach, the analyzer must compute a binding between actual arguments and synthetics arguments at each call site. Our approach is simpler and more flexible: we store the use-def chains in a separate data structure and avoid modifying the program at all. The analyzer represents the use-def chains for each variable as a directed graph consisting of use nodes and def nodes, each of which is associated with a program location. For the example in Figure 3, our analyzer creates a separate set of use-def chains for `x` and `y`. In addition, it creates these use-def chains on the fly, as it discovers each calling context and set of input arguments.

For flow-insensitive objects our framework only maintains a single flow value and simply accumulates updates into that value. Flow insensitivity significantly reduces the cost of analysis because there is no need to compute reaching definitions for these objects. This savings is particularly significant for global variables and heap-allocated memory, whose reaching definitions can span many procedures. For example, a frequently modified global variable might have a complex web of use-def chains that wind throughout the whole program.

```
          // Traditional                Our implementation
p = &x;   //  p -> {x}                      p -> {x}
q = p;    //  q -> {x}                      q -> {x}
p = &y;   //  p -> {x,y}, q -> {x,y}        p -> {x,y}, q -> {x}
```

Fig. 4. Our implementation of flow-insensitive analysis is more precise than the traditional definition because we respect the ordering of the statements.

Our implementation of flow insensitivity deviates from the traditional definition of flow insensitivity because we still visit statements in program order. As a result, our flow-insensitive analysis can be more precise than an analysis that computes the same result independent of the order in which it visits statements. Figure 4 shows an example that highlights the difference in our algorithm. In a traditional flow-insensitive analysis, the presence of an assignment, such as `q = p`, forces the two variables to always be equal. In our implementation, we take advantage of the fact that the second assignment to `p` occurs after the assignment `q = p`, and therefore it cannot affect the value of `q`. Note that we continue to use iterative analysis even for flow-insensitive variables, which ensures correctness in loops.

For each flow-sensitive object we store its defs in a list that is ordered so that we can quickly find reaching definitions at any program location [35]: a def is never preceded in the list by another def that dominates it. We can find the nearest reaching def by searching the list linearly: the first def that dominates the current program location is the nearest reaching definition. We use the same algorithm to insert new defs in the list. This approach is not as fast, asymptotically, as the dominator skeleton tree proposed by Chase et al. [4], but it works well in practice.

3.3 Analysis algorithm

The analysis algorithm performs two main tasks. First, it analyzes statements in the program and builds our modified SSA form, which represents data dependences for the various nodes in the program, including pointers. Second, it manages client dataflow analysis problems through a series of hooks. Overall convergence of the analysis occurs when all of the individual analyses converge.

Our framework solves both the pointer analysis and client analyses using the iterative dataflow analysis algorithm introduced by Kildall [21]. We extend the algorithm in a straightforward way to interprocedural analysis: when the analyzer encounters a procedure call, it immediately begins analyzing the body of the callee procedure. The analysis framework computes dataflow facts by evaluating each statement in program order, looking up flow values for the uses of variables, and updating flow values for defs of variables. It manages this process using a worklist of basic blocks for each procedure.

Flow values come in two varieties: points-to sets (for pointer analysis) and client flow values. Points-to sets are simply sets of nodes in the storage shape graph. For points-to sets the lattice *meet* function, denoted by the $\sqcap$ operator, is the set union operation. Client flow values are named types, organized into an explicit lattice structure. For example, we could model colors using a simple lattice consisting of “Red”, “Green”, and “Blue”, and the pairwise combinations “Yellow”, “Purple”, “Aqua”. The meet function would specify, for example, that “Red” $\sqcap$ “Blue” = “Purple”. This information is specified using our annotation language, which we describe in Section 5.2.

The analysis algorithm includes many engineering details, but for the purposes of describing the client-driven analysis algorithm we only need to describe two parts in detail: assignments and procedure calls. These two components use the lattice meet function to implement flow insensitivity and context insensitivity, respectively. Later in the paper, we show how the client-driven algorithm monitors these components and adjusts precision to avoid using the meet function.

We can divide assignments into two general categories: assignments that dereference a pointer and assignments through a pointer. There are many other assignment forms, but we can handle them as special cases of, or combinations of these two categories. In addition, there are other operators, such as field accesses and arithmetic operators, which our algorithm handles properly but that do not bear on the precision of the analysis. During this process we also record when dataflow information changes and update the worklists accordingly.

We evaluate an assignment of the form `x = *p` using the following steps:

(1) **Dereference `p`.** If `p` is a flow-sensitive variable, then we find the points-to set for `p` by following the use-def chain to its nearest reaching definition and retrieving the points-to set associated with that def. If `p` is a flow-insensitive variables, then we retrieve the points-to set associated with the variable itself – there are no use-def chains. The result of this operation is a set of right-hand-side nodes, R.
(2) **Look up right-hand-side flow values.** For each element r of R, we find the flow values for r using the same process described above for `p`. The result is two sets of flow values, one for the pointer analysis and one for the client analysis: (1) V_p, a set of points-to sets, and (2) V_c, a set of client flow values. Each r contributes one points-to set and one client flow value to each set.
(3) **Merge right-hand-side flow values.** We then compute a single points-to set and a single client flow value to represent all the possible right-hand-side values. We use the lattice meet function to compute these values: $m_p = \sqcap V_p$ and $m_c = \sqcap V_c$.
(4) **Record left-hand-side flow value.** If `x` is a flow-sensitive variable, then we create a def for `x` at the current program location and store the flow values m_p and m_c with the def. If `x` is a flow-insensitive variable, then we look up the current values associated with the variable and merge in the new values using the meet function again.

We evaluate an assignment of the form `*p = y` using the following steps:

- **Dereference `p`.** This step is identical to step (1) above, except that we refer to the resulting points-to set as L, the set of left-hand side nodes. Notice that because our analysis is a "may point to" analysis, `p` can have multiple targets.
- **Look up right-hand-side flow values.** This step is like step (2) above, except that we have only one right-hand side, `y`. The result is a single points-to set and

client flow value for y, v_p and v_c.

- **Assign to left-hand sides.** Since our analysis supports strong updates, this step has two cases: (1) L contains exactly one flow-sensitive variable, so we apply a strong update on the flow values, (2) L contains multiple targets or the targets are flow-insensitive, in which case we apply a weak update. A strong update allows the analyzer to store a new flow value independent of any previous defs of the variable. A weak update forces the analyzer to use the meet function to merge the new flow value with that of the previous reaching definition.

Finally, we evaluate a procedure call `proc(x, y, ...)` using a series of assignments from the actual parameters to the formal parameters. This process is identical to that of the first type of assignment described above. Notice, though, that for context insensitive procedures, the assignment effectively occurs at the entry of the procedure, and therefore produces only a single def for the formal parameter. This forces the analysis to merge the flow values from each callsite using the meet function.

4 Client-driven algorithm

Our client-driven pointer analysis is a two-pass algorithm. The key idea is to use a fast low-precision pointer analysis in the first pass to discover which parts of the program need more precision. The algorithm uses this information to construct a fine-grained customized precision policy for the second pass. This approach requires a tight coupling between the pointer analysis and the client analyses: in addition to providing memory access information to the client, the pointer analyzer receives feedback from the client about the accuracy of the client flow values. For example, the client analysis can report when a confluence point, such as a control-flow merge or context-insensitive procedure call, adversely affects the accuracy of its analysis. The simple interface between the pointer analyzer and the client is the core mechanism that allows the framework to tailor its precision for the particular client and target program.

The implementation of this algorithm (see Figure 5) adds two components to our analysis framework: a *monitor* that detects and tracks loss of information during program analysis, and an *adaptor* that uses the output of the monitor to adjust the precision. During program analysis, the monitor identifies the places where information is lost, and it uses a dependence graph to track the memory locations that are subsequently affected. When analysis is complete the client performs its tasks—after which it reports back to the adaptor with a set of memory locations that are not sufficiently accurate for its purposes. Borrowing terminology from demand-driven analysis, we refer to this set as the *query*. The adaptor starts with the locations in the query and tracks their values back through the dependence graph. The nodes and edges that make up this back-trace indicate which variables and procedures

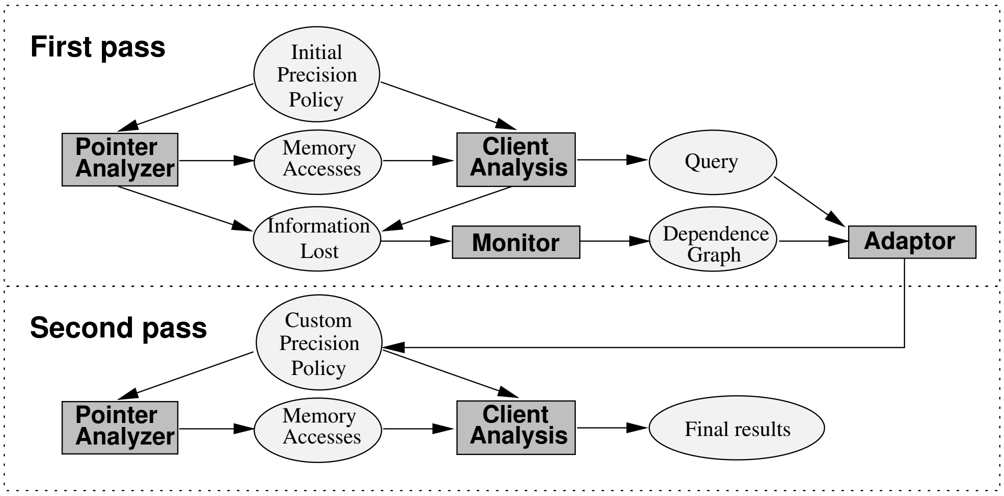


Fig. 5. Our analysis framework allows client analyses to provide feedback, which drives corrective adjustments to the precision.

need more precision. The framework then reruns the analysis with the customized precision policy.

Although the algorithm detects information loss during analysis, it waits until the analysis is complete before changing precision. One reason for this is pragmatic: our framework cannot change precision during analysis and recompute the results incrementally. The other reason is more fundamental: during analysis it is not readily apparent that imprecision detected in a particular pointer value will adversely affect the client later in the program. For example, a program may contain a pointer variable with numerous assignments, causing the points-to set to grow large. However, if the client analysis never needs the value of the pointer then it is not worth expending extra effort to disambiguate it. By waiting to see its impact, we significantly reduce the amount of precision added by the algorithm.

4.1 Polluting Assignments

The monitor runs along side the main pointer analysis and client analysis, detecting information loss and recording its effects. Loss of information occurs when conservative assumptions about program behavior force the analyzer to merge flow values. The analysis algorithm described in Section 3.3 contains several steps that use the lattice meet function to compute these conservative values. In fact, any place where the analyzer uses the lattice meet function can potentially result in loss of information. In particular, we are interested in the cases where accurate, but conflicting, information is merged, resulting in an inaccurate value—we refer to this as a *polluting assignment*.

For "may" pointer analysis smaller points-to sets indicate more accurate information—

a points-to set of size one is the most accurate. In this case the pointer relationship is unambiguous, and assignments through the pointer allow strong updates [4]. Therefore, a pointer assignment is polluting if it combines two or more unambiguous pointers and produces an ambiguous pointer.

For the client analysis, information loss is problem-specific, but we can define it generally in terms of dataflow lattice values. We take the compiler community's view of lattices: higher lattice values represent better analysis information. Lower lattice values are more conservative, with lattice bottom denoting the worst case. Therefore, a client update is polluting if it combines a set of lattice values and produces a lattice value that is lower than any of the individual members.

We classify polluting assignments according to their cause. In our framework there are three ways that conservative analysis can directly cause the loss of information [8]. We will refer to them as *directly polluting assignments*, and they can occur in both the pointer analysis and the client analysis:

- Context-insensitive procedure call: the parameter assignment merges conflicting information from different call sites.
- Flow-insensitive assignment: multiple assignments to a single memory location merge conflicting information.
- Control-flow merge: the SSA ϕ function merges conflicting information from different control-flow paths.

The current implementation of the algorithm is only concerned with the first two classes. It can detect loss of information at control-flow merges, but it currently has no corrective mechanism, such as node splitting or path sensitivity, to remedy it.

In addition to these classes, there are two kinds of polluting assignments that are caused specifically by ambiguous pointers. These assignments are critical to the client-driven algorithm because they capture the relationship between accuracy in the pointer analysis and accuracy in the client. We refer to them as *indirectly polluting assignments*:

- Weak access: the right-hand side of the assignment dereferences an ambiguous pointer, which merges conflicting information from the pointer targets.
- Weak update: the left-hand side assigns through an ambiguous pointer, forcing a weak update that loses information.

4.2 Monitoring Analysis

During analysis, the monitor detects the five kinds of polluting assignments described above, both for the client analysis and the pointer analysis, and it records this information in a directed dependence graph. The goal of the dependence graph

Code	Imprecision	Effect	Monitor action
`foo(5);` `foo(6);`	Context insensitive	Param to foo = $\perp$	Add a node for **`foo`** Label *needs context-sensitivity*
`bar(&a);` `bar(&b);`	Context insensitive	Param to bar points-to **`a`** or **`b`**	Add a node for **`bar`** Label *needs context-sensitivity*
`x = 5;` `x = 6;`	Flow insensitive	**`x`** = $\perp$	Add a node for **`x`** Label *needs flow-sensitivity*
`p = &a;` `p = &b;`	Flow insensitive	**`p`** points-to **`a`** or **`b`**	Add a node for **`p`** Label *needs flow-sensitivity*

`if (c)`
`  x = 5;`
`else`
`  x = 6;`
	Path insensitive	**`x`** = $\perp$	Currently no action

Table 2
For each type of polluting assignment the monitor adds a node into the graph and labels it according to the action needed to improve the precision.

Code	Initially	Effect	Monitor action
`x = y;`	**`y`** = $\perp$	**`x`** = $\perp$	Add a node for **`x`** Add edge **`x`** $\rightarrow$ **`y`**
`p = q;`	**`q`** points-to **`a`** or **`b`**	**`p`** points-to **`a`** or **`b`**	Add a node for **`p`** Add edge **`p`** $\rightarrow$ **`q`**

Table 3
Complicit assignments track imprecision backwards across assignments, allowing the system to find the polluting assignments that are the sources of the imprecision.

is to capture the effects of polluting assignments on subsequent parts of the program.

Each node in the graph represents a memory location whose analysis information, either points-to set or client flow value, is polluted. The graph contains a node for each location that is modified by a directly polluting assignment, and each node has a label that lists all of the directly polluting assignments to that memory location. The monitor builds this graph online by adding nodes to the graph and adding assignments to the labels as they are discovered during analysis. These nodes represent the sources of polluted information, and the labels indicate how to fix the imprecision. Table 2 shows examples of polluting assignments and the actions taken for both pointers and constant propagation as an example client.

The dependence graph contains two types of directed edges. The first type of edge represents an assignment that passes polluted information from one location to another. We refer to this as a *complicit assignment* (see Table 3), and it occurs whenever the memory locations on the right-hand side are already represented in the dependence graph. The monitor creates nodes for the affected left-hand side locations and adds edges from those nodes back to the right-hand side nodes. Note that the direction of the edge is opposite the direction of assignment so that we can trace dependences backward in the program. The second type of edge represents indirectly polluting assignments. The monitor adds nodes for the left-hand side locations and adds a directed edge from each of these nodes back to the offending pointer variable. This kind of edge is unique to our analysis because it allows our algorithm to distinguish between the following two situations: (1) an unambiguous pointer whose target is polluted, and (2) an ambiguous pointer whose targets have precise information.

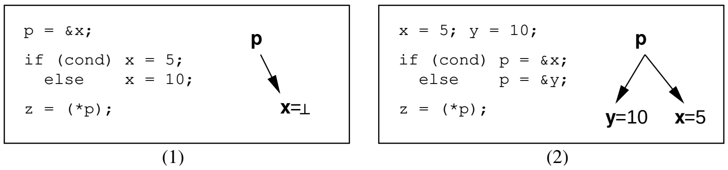


Fig. 6. Both code fragments assign bottom to `z`: in (1) `x` is responsible; in (2) `p` is responsible.

Figure 6 illustrates this distinction using constant propagation as an example client. Both code fragments assign lattice bottom to `z`, but for different reasons. Case (1) is caused by the polluted value of `x`, so the monitor adds an edge in the dependence graph from `z` back to `x`. Case (2), however, is caused by the polluted value of the pointer `p`, so the monitor adds an edge from `z` to `p`.

We store the program locations of all assignments, but for performance reasons the monitor dependence graph is fundamentally a flow-insensitive data structure. As a result, the algorithm cannot tell which specific assignments to a memory location affect other locations. For example, a location might have multiple polluting assignments, some of which occur later in the program than complicit assignments that read its value. In most cases, this simplification does not noticeably hurt the algorithm, but occasionally it leads to overly aggressive precision, particularly when it involves global variables that are used in many different places and for different purposes.

4.3 Diagnosing Information Loss

After the first pass of the analysis, the client provides feedback to the adaptor, in the form of a query, indicating where it needs more accuracy. The adaptor uses the dependence graph to construct a precision policy specifically tailored to obtain the desired accuracy. The output of the adaptor is thus a set of memory locations that need flow-sensitivity and a set of procedures that need context-sensitivity. The new precision policy applies to both the pointer analysis and the client analysis.

The client query consists of a set of memory locations that have “unsatisfactory” flow values. For example, if the client tests a variable for a particular flow value but finds lattice bottom, it could add that variable to the query. The goal of the adaptor is to improve the accuracy of the memory locations in the query. The corresponding nodes in the dependence graph serve as a starting point, and the set of nodes reachable from those nodes represents all the memory locations whose inaccuracy directly or indirectly affects the flow values of the query. The key to the efficiency of our algorithm is that this subgraph is typically much smaller than the whole graph—we rarely to need to fix *all* of the polluting assignments.

The adaptor starts at the query nodes in the graph and visits all of the reachable nodes in the graph. This traversal effectively computes a backwards slice of the program that includes all the assignments that may generate or propagate inaccuracy to the nodes in the query. The adaptor collects the labels on the nodes and applies the specified corrective measures: for polluting parameter assignments it adds the corresponding procedure to the set of procedures that need context-sensitivity; for flow-insensitive assignments it adds the corresponding memory location to the set of locations that need flow-sensitivity.

Before making any changes to the policy, the adaptor checks each proposed precision enhancement. It verifies that flow-sensitivity will help by making sure that there are actually multiple assignments to the object. For example, a polluting assignment could occur in a loop: a variable has a precise flow value until the second time we visit the loop body. It might help to peel the loop, but the current implementation just prunes out these cases.

To verify that context-sensitivity will help, the adaptor re-evaluates the parameters at each callsite. For the client analysis, we make sure that the object in question actually has different values at different call sites. For pointers, we perform two checks: we make sure that the points-to sets are different at different call sites, and we check that any client objects reachable from those pointers have different states. If the final incoming flow values are not different, then we prune the procedure from the context-sensitive set.

4.4 Chaining precision

In addition to addressing each polluting assignment, the adaptor increases precision along the path from each polluting assignment back to the original query nodes. When it finds a node that needs flow-sensitivity, it also applies this additional precision to all the nodes back along the path. When it makes a procedure context-sensitive, it also determines the set of procedures that contain all the complicit assignments back along the path, and it adds that set to the context-sensitive set. This chaining ensures that intermediate locations preserve the additional accuracy that comes from fixing the polluting assignments.

By aggressively chaining the precision, we also avoid the need for additional analysis passes. The initial pass computes the least precise analysis information and therefore covers all the regions of the program for which more precision might be beneficial. Any polluting assignments detected in later passes would necessarily occur within these regions and thus would already be addressed in the customized precision policy. We validated this design decision empirically: subsequent passes typically discover only spurious instances of imprecision and do not improve the quality of the client analysis.

5 Experiments

In this section we describe our experiments, including our methodology, the five error detection clients, and the input programs. The query that these clients provide to the adaptor consists of the set of memory locations that trigger errors. We compare both the cost and the accuracy of our algorithm against four fixed-precision algorithms. In Section 6 we present the empirical results.

We run all experiments on a Dell OptiPlex GX-400, with a Pentium 4 processor running at 1.7 GHz and 2 GB of main memory. The machine runs Linux with the 2.4.18 kernel. Our system is implemented entirely in C++ and compiled using the GNU g++ compiler version 3.0.3.

5.1 Methodology

Our suite of experiments consists of 18 C programs, five error detection problems, and five pointer analysis algorithms—four fixed-precision pointer algorithms and our client-driven algorithm. The fixed-precision algorithms consist of the four possible combinations of flow-sensitivity and context-sensitivity—we refer to them in the results as *CIFI*, *CIFS*, *CSFI*, and *CSFS*. For each combination of program, error problem, and pointer analysis algorithm, we run the analyzer and collect a variety of measurements, including analysis time, memory consumption, and number of errors reported.

The number of errors reported is the most important of these metrics. The more false positives that an algorithm produces, the more time a programmer must spend sorting through them to find the real errors. Our experience is that this is an extremely tedious and time consuming task. Using a fast, inaccurate error detection algorithm is false economy: it trades computer time, which is cheap and plentiful, for programmer time, which is valuable and limited. We believe that it is preferable to use a more expensive algorithm that can reduce the number of false positives, even if it has to run overnight or over the weekend. When two algorithms report the same number of errors, we compare them in terms of analysis time and memory consumption.

In some cases, we know the actual number of errors present in the programs. This information comes from security advisories published by organizations such as CERT and SecurityFocus. We have also manually inspected some of the programs to validate the errors. For the client-driven algorithm we also record the number of procedures that it makes context-sensitive and the number of memory locations that it makes flow-sensitive. Unlike previous research on pointer analysis, we do not present data on the points-to set sizes because this metric is not relevant to our algorithm.

5.2 Error detection clients

We define the five error detection client analyses using an annotation language [14], which allows us to define simple dataflow analysis problems that are associated with a library interface: for each library routine, we specify how it affects the flow values of the problem. The language also provides a way to test the results of the analysis and generate reports. For each analysis problem we show some representative examples of the annotations.

These error detection problems represent realistic errors that actually occur in practice and can cause serious damage. Like many error detection problems, they involve data structures, such as buffers and file handles, that are allocated on the heap and manipulated through pointers. The lifetimes of these data structures often cross many procedures, requiring interprocedural analysis to properly model. Thus, they present a considerable challenge for the pointer analyzer.

TBD: Show lattices for the error checking clients.

5.2.1 File access errors

Library interfaces often contain implicit constraints on the order in which their routines may be called. File access rules are one example of this kind of usage constraint. A program can only access a file between the proper open and close calls. The purpose of this client analysis is to detect possible violations of this usage rule. The first line in Figure 7 defines the flow value for this analysis, which consists of the two possible states, “Open” and “Closed”. Figure 8 (a) depicts the lattice structure for this dataflow analysis.

To track this state, we annotate the various library functions that open and close files. Figure 7 shows the annotations for the `fopen()` function. The `on_entry` and `on_exit` annotations describe the pointer behavior of the routine: it returns a pointer to a new file stream, which points to a new file handle. The `analyze` annotation sets the state of the newly created file handle to open. At each use of a file stream or file descriptor, we check to make sure the state is open. Figure 7 also shows an annotation for the `fgets()` function, which emits an error if the

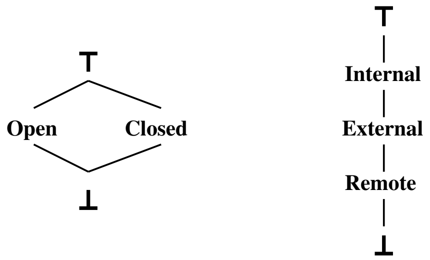
```
property FileState : { Open, Closed }  initially Closed

procedure fopen(path, mode)
{
  on_exit { return --> new file_stream --> new file_handle }
  analyze FileState {  file_handle <- Open  }
}

procedure fgets(s, size, f)
{
  on_entry { f --> file_stream --> handle }
  error if (FileState : handle could-be Closed)  "Error:␣file␣might␣be␣closed";
}
```

Fig. 7. Annotations for tracking file state: to properly model files and files descriptors, we associate the state with an abstract “handle”.

file could be closed.



(a) (b)

Fig. 8. Each `property` annotation implies a dataflow lattice. The nesting structure of the property values implies the meet function.

5.2.2 Format string vulnerability (FSV)

A number of output functions in the Standard C Library, such as `printf()` and `syslog()`, take a format string argument that controls output formatting. A format string vulnerability (FSV) occurs when untrusted data ends up as part of the format string. A hacker can exploit this vulnerability by sending the program a carefully crafted input string that causes part of the code to be overwritten with new instructions. These vulnerabilities represent a serious security problem that have been the subject of many CERT advisories.

To detect format string vulnerabilities we define an analysis that determines when data from an untrusted source can become part of a format string. We consider data to be *tainted* [33,27] when it comes from an untrusted source. We track this data through the program to make sure that all format string arguments are *untainted*.

Our formulation of the Taint analysis starts with a definition of the Taint property, shown at the top of Figure 9, which consists of two possible values, `Tainted` and `Untainted`. We then annotate the Standard C Library functions that produce tainted data. These include such obvious sources of untrusted data as `scanf()`

```
property Taint : { Tainted, Untainted }  initially Untainted

procedure read(fd, buffer_ptr, size)
{
  on_entry { buffer_ptr --> buffer }
  analyze Taint { buffer <- Tainted }
}

procedure strdup(s)
{
  on_entry { s --> string }
  on_exit { return --> string_copy }
  analyze Taint { string_copy <- string }
}

procedure syslog(prio, fmt, args)
{
  on_entry { fmt --> fmt_string }
  error if (Taint : fmt_string could-be Tainted)  "Error:␣tainted␣format␣string.";
}
```

Fig. 9. Annotations defining the Taint analysis: taintedness is associated with strings and buffers and can be transferred between them.

and `read()`, and less obvious ones such as `readdir()` and `getenv()`. Figure 9 shows the annotations for the `read()` routine. Notice that the annotations assign the `Tainted` property to the contents of the buffer rather than to the buffer pointer. We then annotate string manipulation functions to reflect how taintedness can propagate from one string to another. The example in Figure 9 annotates the `strdup()` function: the string copy has the same Taint value as the input string.

Finally, we annotate all the library functions that accept format strings (including `sprintf()`) to report when the format string is tainted. Figure 9 shows the annotation for the `syslog()` function, which is often the culprit in FSV attacks.

5.2.3 Remote access vulnerability

Hostile clients can only manipulate programs through the various program inputs. We can approximate the extent of this control by tracking the input data and observing how it is used. We label input sources, such as file handles and sockets, according to their level of trust. All data read from these sources is labeled likewise. The first line of Figure 10 defines the three levels of trust in our analysis—internal (trusted), locally trusted (for example, local files), and remote (untrusted). Figure 8 (b) depicts the lattice structure for this dataflow analysis. Notice that the nesting of the three property values creates a vertical lattice, which captures the fact that `Remote` is more conservative than `External`, which is more conservative than `Internal`.

We start by annotating functions that return fundamentally untrusted data sources, such as Internet sockets. Figure 10 shows the annotations for the `socket()` function. The level of trust depends on the type of socket being created. When the

```
property Trust : { Remote { External { Internal }}}

procedure socket(domain, type, protocol}
{
  on_exit { return --> new file_handle }
  analyze Trust {
    if (domain == AF_UNIX)  file_handle <- External
    if (domain == AF_INET)  file_handle <- Remote
  }
}

procedure open(path, flags)
{
  on_entry { path --> path_string }
  on_exit { return --> new file_handle }
  analyze Trust { file_handle <- path_string }
}
```

Fig. 10. Annotations defining the Trust analysis. Note the cascading effect: we only trust data from a file handle if we trust the file name used to open it.

program reads data from these sources, the buffers are marked with the Trust level of the source.

The Trust analysis has two distinguishing features. First, data is only as trustworthy as its least trustworthy source. For example, if the program reads both trusted and untrusted data into a single buffer, then we consider the whole buffer to be untrusted. The nested structure of the lattice definition captures this fact. Second, untrusted data has a domino effect on other data sources and sinks. For example, if the file name argument to `open()` is untrusted, then we treat all data read from that file descriptor as untrusted. The annotations in Figure 10 implement this policy.

As with the earlier Taint analysis, we annotate string manipulation functions to propagate the Trust values from one buffer to another. We generate an error message when untrusted data reaches certain sensitive routines, including any file system manipulation or program execution routines, such as `exec()`.

5.2.4 Remote FSV

The Taint analysis defined above tends to find many format string vulnerabilities that are not exploitable security holes. For example, consider a program that uses data from a file as part of a format string. If a hacker can dictate the name of the file or can control the contents of the file, then the program contains a remotely exploitable vulnerability. If a hacker cannot control the file, however, then the program still contains a vulnerability, but the vulnerability does not have security implications.

To identify exploitable format string vulnerabilities more precisely, we can combine the Taint analysis with the Trust analysis, which specifically tracks data from remote sources. No new dataflow analyses are needed. We simply revise the error test so that it only emits an error message when the format string is tainted and it comes from a remote source.

5.2.5 FTP behavior

The most complex of our client analyses checks to see if a program can behave like an FTP (file transfer protocol) server. Specifically, we want to determine if the program could possibly send the contents of a file to a remote client, where the name of the file is determined, at least in part, by the remote client itself. This behavior is not necessarily incorrect: it is the normal operation of the two FTP daemons that we present in our results. We can use this error checker to make sure the behavior is not unintended (for example, in a finger daemon) or to validate the expected behavior of the FTP programs.

We use the Trust analysis defined above to determine when untrusted data is read from one stream to another. However, we need to know that one stream is associated with a file and the other with a remote socket. Figure 11 defines the flow value to track different type of sources and sinks of data. We can distinguish between different type of sockets, such as “Server” sockets, which have bound addresses for listening, and “Client” sockets, which are the result of accepting a connection.

```
property FDKind : { File, Client, Server, Pipe, Command, StdIO }

procedure write(fd, buffer_ptr, size)
{
  on_entry { buffer_ptr --> buffer
             fd --> file_handle }
  error if ((FDKind : buffer could-be File) &&
            (Trust  : buffer could-be Remote) &&
            (FDKind : file_handle could-be Client) &&
            (Trust  : file_handle could-be Remote))
      "Error:␣possible␣FTP␣behavior";
}
```

Fig. 11. Annotations to track type of data sources and sinks. In combination with Trust analysis, we can check whether a call to `write()` behaves like FTP.

Whenever a new file descriptor is opened, we mark it according to the type. In addition, like the other analyses, we associate this type with any data read from it. We check for FTP behavior in the `write()` family of routines, shown in Figure 11, by testing both the buffer and the file descriptor.

5.3 Programs

Table 4 describes our input programs. We use these particular programs for our experiments for a number of reasons. First, they are all real programs, taken from open-source projects, with all of the nuances and complexities of production soft-

Program	Description	Priv	LOC	CFG nodes	Procedures
stunnel 3.8	Secure TCP wrapper	yes	2K / 13K	2264	42
pfingerd 0.7.8	Finger daemon	yes	5K / 30K	3638	47
muh 2.05c	IRC proxy	yes	5K / 25K	5191	84
muh 2.05d	IRC proxy	yes	5K / 25K	5390	84
pure-ftpd 1.0.15	FTP server	yes	13K / 45K	11,239	116
crond (fcron-2.9.3)	cron daemon	yes	9K / 40K	11,310	100
apache 1.3.12 (core only)	Web server	yes	30K / 67K	16,755	313
make 3.75	make		21K / 50K	18,581	167
BlackHole 1.0.9	E-mail filter		12K / 244K	21,370	71
wu-ftpd 2.6.0	FTP server	yes	21K / 64K	22,196	183
openssh client 3.5p1	Secure shell client		38K / 210K	22,411	441
privoxy 3.0.0	Web server proxy	yes	27K / 48K	22,608	223
wu-ftpd 2.6.2	FTP server	yes	22K / 66K	23,107	205
named (BIND 4.9.4)	DNS server	yes	26K / 84K	25,452	210
openssh daemon 3.5p1	Secure shell server	yes	50K / 299K	29,799	601
cfengine 1.5.4	System admin tool	yes	34K / 350K	36,573	421
sqlite 2.7.6	SQL database		36K / 67K	43,333	387
nn 6.5.6	News reader		36K / 116K	46,336	494

Table 4
Properties of the input programs. Many of the programs run in privileged mode, making their security critical. Lines of code (LOC) is given both before and after preprocessing. CFG nodes measures the size of the program in our compiler's internal representation—the table is sorted on this column.

ware. Second, many of them are system tools or daemons that have significant security implications because they provide privileged services and interact with remote clients. Finally, several of them are specific versions of programs that are identified by security advisories as containing format string vulnerabilities. In these cases, we also obtain subsequent versions in which the bugs are fixed, so that we can confirm their absence.

We present several measures of program size, including number of lines of source code, number of lines of preprocessed code, and number of procedures. The table is sorted by the number of CFG nodes, and we use this ordering in all subsequent tables.

6 Results

We measure the results for all combinations of pointer analysis algorithms, error detection clients, and input programs—a total of over 400 experiments. We present the results in five graphs, one for each error detection client (see Figures 12-16). Each bar on the graph shows the accuracy and performance of the different analysis algorithms on the given program. To more easily compare different programs we normalize all execution times to the time of the fastest algorithm on that pro-

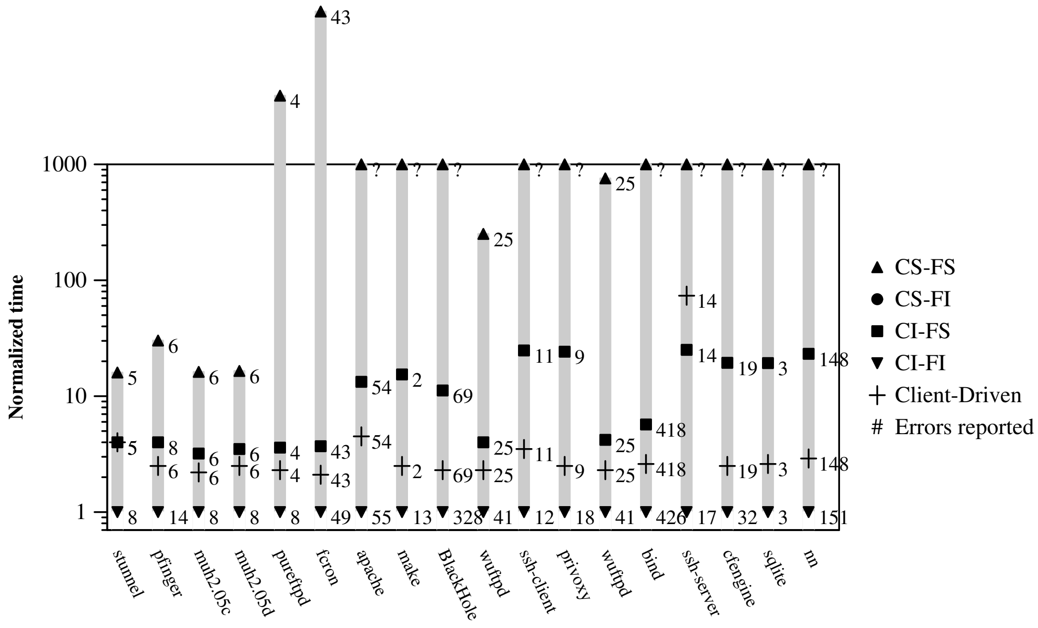


Fig. 12. Checking file access requires flow-sensitivity, but not context-sensitivity. The client-driven algorithm beats the other algorithms because it makes only the file-related objects flow-sensitive.

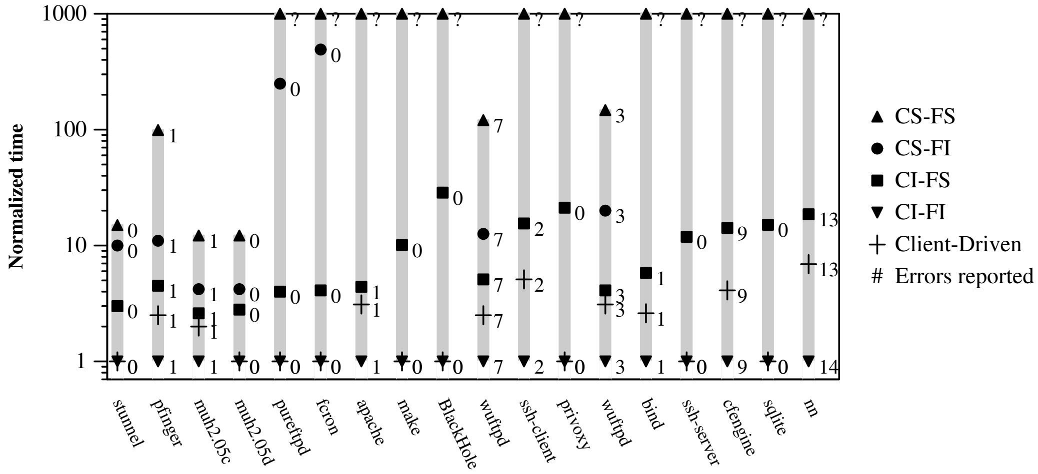


Fig. 13. Detecting format string vulnerabilities rarely benefits from either flow-sensitivity or context-sensitivity—the client-driven algorithm is only slower because it is a two-pass algorithm.

gram, which in all cases is the context-insensitive, flow-insensitive algorithm. Each point on these graphs represents a single combination of error detection client, input program, and analysis algorithm. We label each point with the number of errors reported in that combination. For example, from Figure 12 we see that for the stunnel program, the CI-FI and CS-FI algorithms reported 8 errors, while the other algorithms reported 5 errors. The same bar shows that the client-driven algorithm was as fast as the CI-FS algorithm, but slower than the CI-FI algorithm.

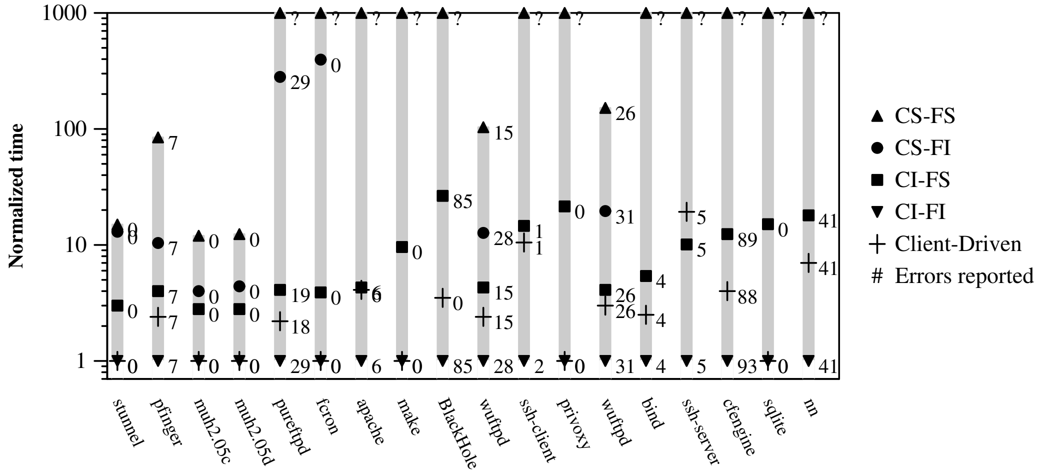


Fig. 14. Detecting remote access vulnerabilities can require both flow-sensitivity and context-sensitivity. In these cases the client-driven algorithm is both the most accurate and the most efficient.

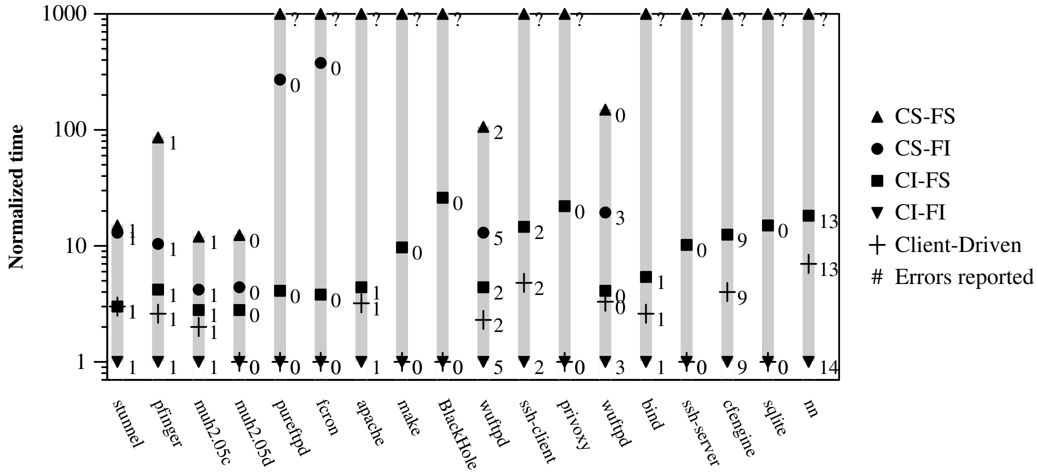


Fig. 15. Determining when a format string vulnerability is remotely exploitable is a more difficult analysis. The execution time of the client-driven algorithm is still competitive with the fastest fixed-precision algorithm.

For the 90 combinations of error detection clients and input programs, we find the following:

- In all cases, the client-driven algorithm equals or beats the **accuracy** of the best fixed-precision policy.
- In 64 of the 90 cases the client-driven algorithm also equals or beats the **performance** of the comparably accurate fixed-precision algorithm. In 29 of these cases the client-driven algorithm is both the fastest ***and*** the most accurate.
- In 19 of the remaining 23 cases the client-driven algorithm performs within a factor of two or three of the best fixed-precision algorithm. In many of these cases the best fixed-precision algorithm is the fastest fixed-precision algorithm, so in absolute terms the execution times are all low.

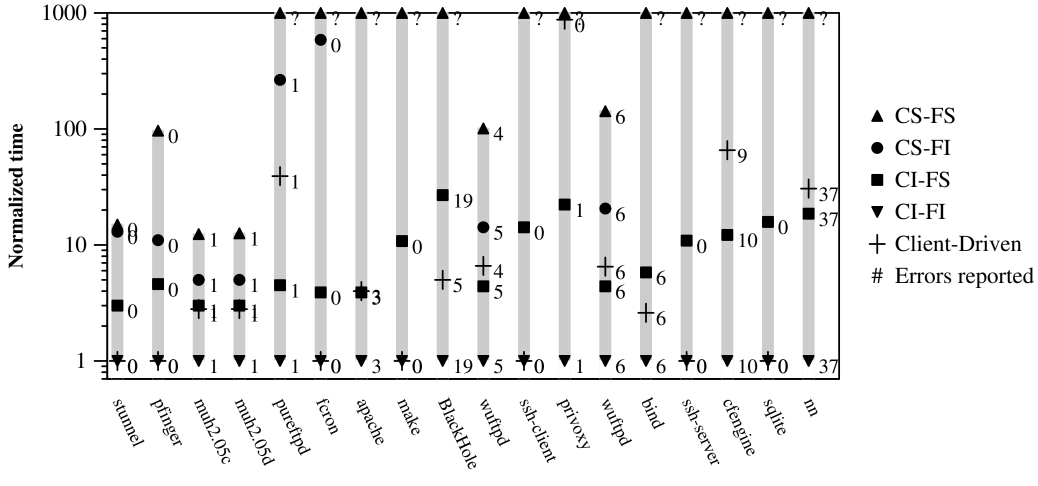


Fig. 16. Detecting FTP-like behavior is the most challenging analysis. In three cases (WU-FTP, privoxy, and CFEngine) the client-driven algorithm achieves accuracy that we believe only the full-precision algorithm can match—if it were able to run to completion.

Note that for many of the larger programs the fully flow-sensitive and context-sensitive algorithm either runs out of memory or requires an intolerable amount of time. In these cases we cannot measure the accuracy of this algorithm for comparison. However, we do find that for the smaller programs the client-driven algorithm matches the accuracy of the full-precision algorithm.

In general, the only cases where a fixed-policy algorithm performs better than the client-driven algorithm are those in which the client requires little or no extra precision. In particular, the format string vulnerability problem rarely seems to benefit from higher levels of precision. In these cases, though, the analysis is usually so fast that the performance difference is practically irrelevant. Figure 18 shows that for these cases, the analysis time for the client-driven algorithm is typically between 1 and 10 seconds.

For the problems that do require more precision, the client-driven algorithm consistently outperforms the fixed-precision algorithms. Tables 5 and 6 provide some insight into this result. For each program and each client, we record the number of procedures that the algorithm makes context-sensitive and the percentage of memory locations that it makes flow-sensitive. From these tables, we draw several conclusions:

- Looking at the columns, we find that different clients have different precision requirements. The file access client, for example, benefits from some flow-sensitivity but not context-sensitivity; the FTP behavior client requires both. These statistics show that client analyses often need some extra precision, but only a very small amount.
- While the client-driven algorithm might needlessly analyze some variables with flow-sensitivity, the amount of such extra precision is minimal. For example, Figure 13 shows that the Format String Vulnerability requires flow-sensitivity

	Percentage of Memory Locations set Flow-Sensitive				
Program	File Access	FSV	Remote Access	Remote FSV	FTP Behavior
stunnel-3.8	0.20	–	–	–	0.19
pfinger-0.7.8	–	0.53	0.20	0.53	0.61
muh2.05c	0.10	–	–	0.07	0.31
muh2.05d	0.10	–	–	–	0.33
pure-ftpd-1.0.15	0.13	–	0.12	–	0.10
fcron-2.9.3	–	–	0.03	–	0.26
apache-1.3.12	0.18	0.91	0.89	1.07	0.83
make-3.75	0.02	–	–	–	2.19
BlackHole-1.0.9	0.04	–	0.24	–	0.32
wu-ftpd-2.6.0	0.09	0.22	0.34	0.24	0.08
openssh-3.5p1-client	0.06	0.55	0.35	0.56	0.96
privoxy-3.0.0-stable	0.01	–	–	–	0.10
wu-ftpd-2.6.2	0.09	0.51	0.63	0.53	0.23
bind-4.9.4-REL	0.01	0.23	0.14	0.20	0.42
openssh-3.5p1-server	0.59	–	0.49	–	1.19
cfengine-1.5.4	0.04	0.46	0.43	0.48	0.03
sqlite-2.7.6	0.01	–	1.47	–	1.43
nn-6.5.6	0.17	1.99	1.82	2.03	0.97

Table 5
The percent of all memory locations in each program that the client-driven algorithm chooses to analyze using flow-sensitivity. We show this value as a percentage because the overall numbers are large. Dashes indicate that no memory locations were analyzed using flow-sensitivity.

for only one benchmark, `nn`, and Table 5 shows that the client-driven algorithm does not use any flow-sensitivity for ten of the benchmarks, and for the others excluding `nn`, it uses very little flow-sensitivity.

- From Figures 12-16, we determine that only seven of the 90 problem instances require any context-sensitivity. From Table 6, we see that only a tiny fraction of procedures are analyzed in this way, suggesting that while faster techniques may exist for implementing context-sensitivity, we can actually avoid it altogether in most cases.

6.1 Client-specific results

The client-driven algorithm reveals some significant differences between the precision requirements of the five error detection problems.

Figure 12 shows the results for the file access client, which benefits significantly from flow-sensitivity but not from context-sensitivity. This result makes sense because the state of a file handle can change over time, but most procedures only accept open file handles as arguments. We suspect that few of these error reports represent true errors, and we believe that many of the remaining false positives could be eliminated using path-sensitive analysis.

		Number of Procedures set Context-Sensitive				
Program	**Total procedures**	File Access	FSV	Remote Access	Remote FSV	FTP Behavior
stunnel-3.8	42	-	-	-	-	-
pfinger-0.7.8	47	-	-	1	-	-
muh2.05c	84	-	-	-	-	6
muh2.05d	84	-	-	-	-	6
pure-ftpd-1.0.15	116	-	-	2	-	9
fcron-2.9.3	100	-	-	-	-	-
apache-1.3.12	313	-	2	8	2	10
make-3.75	167	-	-	-	-	-
BlackHole-1.0.9	71	-	-	2	-	5
wu-ftpd-2.6.0	183	-	-	-	-	17
openssh-3.5p1-client	441	1	-	10	-	-
privoxy-3.0.0-stable	223	-	-	-	-	5
wu-ftpd-2.6.2	205	-	4	-	4	17
bind-4.9.4-REL	210	-	2	1	1	4
openssh-3.5p1-server	601	1	-	13	-	-
cfengine-1.5.4	421	-	1	4	3	31
sqlite-2.7.6	387	-	-	-	-	-
nn-6.5.6	494	-	1	2	1	30

Table 6
The number of procedures in each program that the client-driven algorithm chooses to analyze using context-sensitivity. Dashes indicate that no procedures were analyzed using context-sensitivity.

Figure 13 shows the results for detecting format string vulnerabilities. The taintedness analysis that we use to detect format string vulnerabilities generally requires no extra precision beyond the CI-FI analysis. We might expect utility functions, such as string copying, to have unrealizable paths that cause spurious errors, but this does not happen in any of our example programs. The high false positive rates observed in previous work [27] are probably due to the use of equality-based analysis.

Figure 14 shows the results for remote access vulnerability detection. Accurate detection of remote access vulnerabilities requires both flow-sensitivity and context-sensitivity because the “domino effect” of the underlying Trust analysis causes information loss to propagate to many parts of the program. For example, all of the false positives in BlackHole are due to unrealizable paths through a single function called `my_strlcpy()`, which implements string copying. The client-driven algorithm detects the problem and makes the routine context-sensitive, which eliminates all the false positives.

Figure 15 shows the results for determining the remote exploitability of format string vulnerabilities. We find that this client is particularly difficult for the client-driven analysis, which tends to add too much precision without lowering the false positive count. Interestingly, many spurious FSV errors are caused by typos in the program: for example, `cfengine` calls `sprintf()` in several places without providing the string buffer argument.

For two of the input programs, `muh` and `wu-ftp`, we use two versions of each program: one version known to contain format string vulnerabilities and a subsequent version with the bugs fixed. Our system accurately detects the known vulnerabilities in the old versions and confirms their absence in the newer versions. Our analysis also finds the known vulnerabilities in several other programs, including `stunnel`, `cfengine`, `sshd`, and `named`. In addition, our system reports a format string vulnerability in the Apache web server. Manual inspection, however, shows that it is unexploitable for algorithmic reasons that are beyond the scope of our analysis.

Figure 16 shows the results for detecting FTP-like behavior, which is the most challenging of our problems because it depends on the states of multiple memory locations and multiple client analyses. Even for this more demanding problem, our client-driven analysis properly detects exactly those program points in the two FTP daemons that perform the "get" or "put" file transfer functions. Context-sensitivity helps eliminate a false positive in one interesting case: in `wu-ftp`, a data transfer function appears to contain an error because the source and target could either be files or sockets. However, when the calling contexts are separated, the combinations that actually occur are file-to-file and socket-to-socket.

6.2 Program-specific results

This section describes some of the significant challenges that the input programs present for static analysis.

6.2.1 Function tables

Some of the programs use tables of function pointers that are similar to virtual function tables in C++. Unfortunately, these tables are indexed by strings, making it practically impossible to reduce the number of possible call targets. As a result, the dispatch procedures, which access the table and call through the function pointer, end up significantly polluting the call graph.

6.2.2 Library wrappers

Many of the programs put wrappers around standard library functions or provide their own implementations of these functions. For example, many programs put a wrapper around `strdup()` that handles a null pointer or that exits gracefully if memory is exhausted. The client-driven algorithm works well in these cases because it makes the wrapper functions context-sensitive. Occasionally, however, there are so many calls to these functions that the cost of context-sensitivity explodes.

6.2.3 Custom memory allocators

A few of the programs use custom memory allocators. Apache is particularly problematic because it implements a region-like allocator with semantics unlike the conventional heap or stack allocation. Luckily, there is an option to compile it using the regular malloc interface. In general, though, many analysis tools rely on the semantics of malloc and free to build an accurate model of heap objects: since multiple calls to malloc always return distinct chunks of memory, there is no need to explicitly model the address space.

6.2.4 Internal library state

A number of library routines contain internal state that is not explicitly represented in their interfaces. The string tokenizer function `strtok()`, in particular, can present a challenge for static analysis because it stores non-null input strings in a hidden global variable and returns pointers into the most recently stored string on null inputs. Figure 17 shows how we can easily model this behavior using the annotation language. The global variable `strtok_static_pointer` is synthetic and only exists for the purposes of analysis. The client-driven analysis often decides to make this variable flow-sensitive in order to distinguish between different tokenized strings.

```
global { strtok_static_pointer }
procedure strtok(str, find_str)
{
  on_entry { str --> string
             find_str --> find_string
             strtok_static_pointer --> previous_string  }

  access { string, find_string }

  on_exit {
    if (str == 0) { strtok_static_pointer --> previous_string
                    return --> previous_string  }

    default { strtok_static_pointer --> string
              return --> string }
  }
}
```

Fig. 17. Annotations for `strtok` properly model its internal state.

6.3 Average performance

Figures 18 and 19 show the performance of the different algorithms averaged over all five clients. These two graphs show the actual execution time in seconds and the memory usage in megabytes. We see that the client-driven algorithm is quite efficient. In most cases the client-driven algorithm performs almost as well as the fastest fixed-policy algorithm—the flow-insensitive context-insensitive algorithm. As we saw in Figures 12-16, in the cases where the client-driven algorithm uses more resources, it produces a better result: it takes more time, but it eliminates false positives.

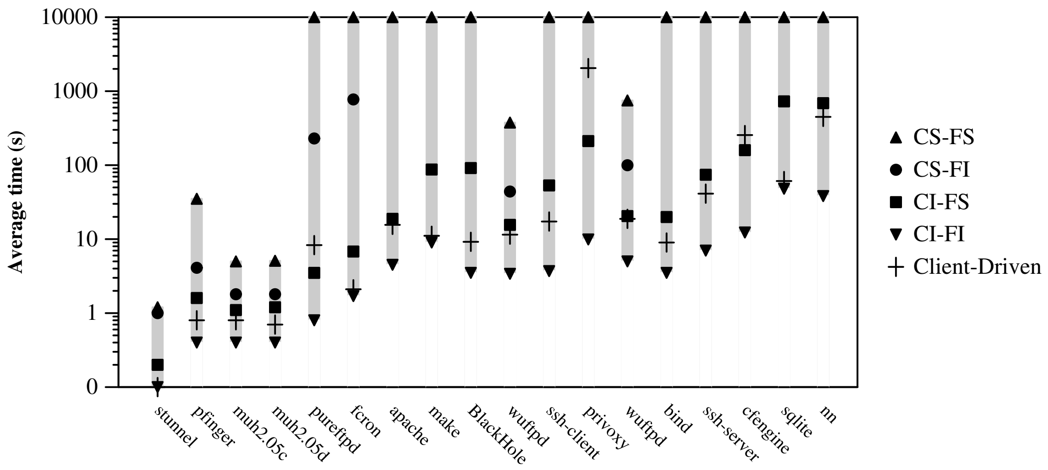


Fig. 18. The client-driven algorithm performs competitively with the fastest fixed-precision algorithm.

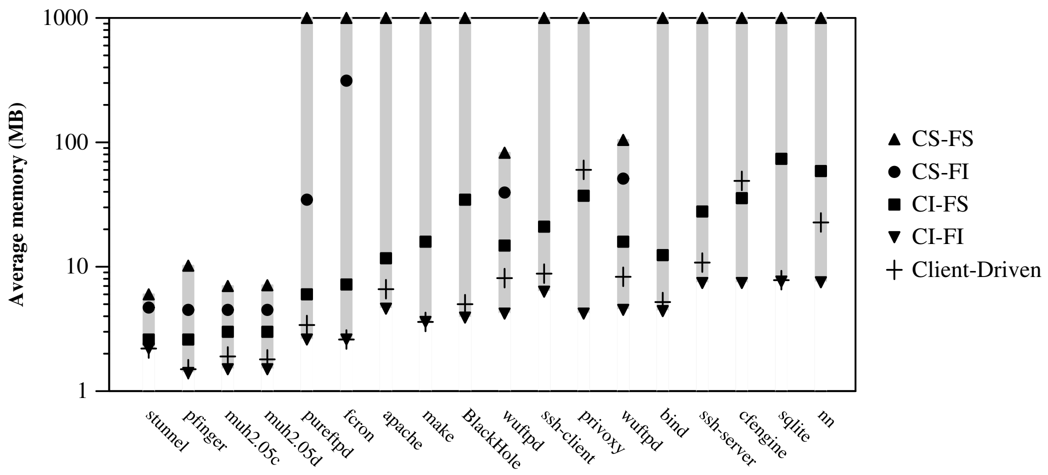


Fig. 19. Memory usage is only a significant problem for the fully context-sensitivity algorithms. More efficient implementations exist, but we find that full context-sensitivity is not needed.

6.4 Discussion and future work

The current implementation of the client-driven algorithm manages just two aspects of precision—flow-sensitivity and context-sensitivity—but precision could be improved by handling other aspects as well. For example, more precise algorithms exist for handling control-flow and for modeling heap objects. While our algorithm can detect information loss in these situations, we currently have no mechanism to address them. We could imagine using path-sensitive techniques when the algorithm detects information loss at a control-flow merge. Similarly, we could employ shape analysis for heap objects that merge information.

The client-driven approach might also be extended to improve its scalability. One promising direction would be to first perform an equality-based pointer analysis, which can scale to programs with a million lines of code [29]. We can then apply our existing client-driven algorithm in places where we detect information loss due to unification. Finally, our algorithm might be combined with other approaches, such as demand-driven analysis [20], to yield further improvements in accuracy and scalability.

7 Conclusions

In this paper we have presented a new client-driven approach to managing the tradeoff between cost and precision in pointer analysis. We have shown that such an approach is needed because no single fixed-precision analysis is appropriate for all client problems and programs. The low-precision algorithms do not provide sufficient accuracy for the more challenging client analysis problems, while the high-precision algorithms waste time over-analyzing the easy problems. Rather than choose any of these fixed-precision policies, we exploit the fact that many client analyses require only a small amount of extra precision applied to specific places in each input program. Our client-driven algorithm can effectively detect these places and automatically apply the necessary additional precision.

Looking to the future, we believe that the client-driven algorithm provides a blueprint for the deployment of more sophisticated but expensive pointer analysis techniques. For example, shape analysis can be extremely expensive, but perhaps in a client-driven framework it can be profitably applied to very small portions of the programs.

Acknowledgments. We thank Teck Tok for his improvements to the implementation of the client-driven analyzer. We also thank Kathryn S. McKinley and Michael Hind for their valuable and insightful comments.

References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, DIKU report 94/19, 1994.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *International SPIN Workshop on Model Checking of Software*, May 2001.

[3] D. Brylow and J. Palsberg. Deadline analysis of interrupt-driven software. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 198–207, September 2003.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.

[5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In ACM, editor, *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[7] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 37, 5, pages 57–68, 2002.

[8] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23, 2001.

[9] 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 '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994.

[10] J. S. Foster, M. Fahndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, pages 175–198, 2000.

[11] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, volume 37 (5) of *ACM SIGPLAN Notices*, pages 1–12, 2002.

[12] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[13] S. Z. Guyer, E. Berger, and C. Lin. Detecting errors with configurable whole-program dataflow analysis. Technical Report TR 02-04, Dept. of Computer Sciences, University of Texas at Austin, February 2002.

[14] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.

[15] S. Z. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*, pages 221–238, August 2000.

[16] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *10th Annual International Static Analysis Symposium*, pages 214–236, June 2003.

[17] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[18] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engeneering (PASTE-01)*, pages 54–61, 2001.

[19] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, January 2001.

[20] S. Horwitz, T. Reps, and M. Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[21] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[22] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN'96 Programming Language Design and Implementation of Programming Languages*, June 1992.

[23] O. Lhotak and L. Hendren. Jedd: A BDD-based relational extension of java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–169, June 2004.

[24] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, September 2003.

[25] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, October 1994.

[26] E. Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[27] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[28] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. *Lecture Notes in Computer Science*, 1302, 1997.

[29] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN'96 Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[30] P. Stocks, B. G. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *International Symposium on Software Testing and Analysis*, pages 21–31, 1998.

[31] E. Stoltz, M. Wolfe, and M. P. Gerlek. Constant propagation: a fresh, demand-driven look. In *Proceedings of the 1994 ACM symposium on Applied computing*, pages 400–404. ACM Press, 1994.

[32] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliabiity. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[33] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl, 3^{rd} Edition*. O'Reilly, July 2000.

[34] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, June 2004.

[35] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[36] S. Zhang, B. G. Ryder, and W. A. Landi. Experiments with combined analysis for pointer aliasing. *ACM SIGPLAN Notices*, 33(7):11–18, July 1998.