Client-Driven Pointer Analysis ⋆

Samuel Z. Guyer and Calvin Lin

Department of Computer Sciences,
The University of Texas at Austin, Austin, TX 78712, USA,
{sammy,lin}@cs.utexas.edu

Abstract. This paper presents a new *client-driven* pointer analysis algorithm that automatically adjusts its precision in response to the needs of client analyses. We evaluate our algorithm on 18 real C programs, using five significant error detection problems as clients. 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 the right places in each input program.

1 Introduction

Pointer analysis is critical for effectively analyzing programs written in languages like C, C++, and Java, which make heavy use of pointers and pointer-based data structures. Pointer analysis attempts 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 has disagreed on whether more precise algorithms yield more accurate results, and whether these results are worth the additional cost [23, 22, 16, 10, 20]. Despite these differences, a recent survey claims that the choice of pointer analysis algorithm should be dictated by the needs of the client analyses [15].

```
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.

⋆ This research was funded by NSF CAREER grant ACI-9984660 and DARPA contract #F30602-97-1-0150.

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 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 [17, 14], 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.

For example, consider how context-insensitive analysis treats 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 support they can detect significant and complex program defects. We compare our algorithm against five 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 accuracy simply as the number of errors reported: the analysis is conservative, so fewer error reports always indicates fewer false positives.

This paper 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 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 has 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 various kinds of program analysis, including pointer analysis, has explored ways to reduce the cost of analysis while still producing an accurate result. Our client-driven algorithm addresses this problem specifically for the precision of pointer analysis. It is closely related to demand-driven algorithms and mixed-precision analyses. 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 [18] is the only other algorithm that we are aware of 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. The basic mechanism 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 *typestate* problems [24], 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 [14] 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. Client-driven analysis dynamically varies precision, but still computes an exhaustive solution. Demand-driven pointer analysis is a fixed-precision analysis that computes only the necessary part of the solution. The two ideas are complementary and could be combined to obtain the benefits of both.

Demand interprocedural dataflow analysis [17] also avoids the cost of exhaustive program analysis by focusing on computing 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 [27] 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. Client-driven pointer analysis is more feedback directed: instead of using a heuristic, it determines the need for precision dynamically by monitoring the analysis.

2.2 Pointer analysis for error detection

Automatic error checking of C programs is a particularly compelling application for pointer analysis. One of the major challenges in analyzing C programs is constructing a precise enough model of the store 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 [21, 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 three procedures in the application in order to mimic context-sensitivity. In this paper, our goal is not to propose an alternative technique for detecting errors, but rather to present a pointer analysis algorithm that supports these clients more effectively. Our algorithm detects when imprecision in the store model hampers the client and automatically increases precision in the parts of the program where it's needed.

3 Framework

Our analysis framework is part of the Broadway compiler system, which supports high-level analysis and optimization for C programs [13]. In this section we describe the details of this framework, including the overall architecture, the representation of pointer information, and the implementation of the different precision policies.

We use a lightweight annotation language to specify the client analysis problems [12]. The language is designed to extend compiler support to software libraries; it is not used to describe the application programs. 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. The dataflow analysis framework manages both the pointer analysis and the client analyses, which run concurrently. Analysis is whole-program, interprocedural, and uses an iterative worklist algorithm.

3.1 Pointer representation

Our base pointer analysis can be roughly categorized as an interprocedural "Andersen-style" analysis [1]: it is flow-insensitive, context-insensitive, and inclusion-based. We represent the program store using an enhanced implementation of the storage shape graph [3]. Each memory location—local variable, global variable, or heap-allocated memory–is a node in the graph, with directed *points-to* edges from pointers to their targets. Our algorithm is a "may" analysis: a points-to edge in the graph represents a possible pointer relationship in the actual execution of the program. Conservative approximation of program behavior often leads to multiple outgoing points-to edges. However, when a node has exactly one target it is a "must" pointer, and assignments through it admit strong updates.

3.2 Configurable precision

We can add precision, at a fine grain, to the base pointer analysis: individual memory locations in the store can be either flow-sensitive or flow-insensitive, and individual procedures can be either context-sensitive or context-insensitive. For a flow-sensitive location, we record separate points-to sets and client dataflow facts for each assignment to the location. We maintain this information using interprocedural factored def-use chains, which are similar to SSA form [5] except that we store the phi functions in a separate data structure. For a flow-insensitive location, we accumulate the information from all of its assignments. Our algorithm still visits assignments in order, however, so our flow-insensitive analysis often yields slightly more precise information than a traditional algorithm.

To analyze a context-sensitive procedure, we treat each call site as a completely separate instantiation, which keeps information from different call sites separate. More efficient methods of implementing context-sensitivity exist [26], but we show in Section 6 that we can often avoid it altogether. 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, the analyzer can often skip over a context-insensitive procedure call when no changes occur to the input values, which helps the analysis converge quickly. The main drawback of this mode is that it suffers from the unrealizable paths problem [26], in which information from one call site is returned to a different call site.

3.3 Heap objects

For many C programs, manipulation of heap allocated objects accounts for much of the pointer behavior. Our pointer analyzer contains two features that help improve analysis of these objects. First, the analyzer generates one heap object for each allocation site in each calling context. This feature can dramatically improve accuracy for programs that allocate memory through a wrapper function around `malloc()`. It also helps distinguish data structure elements that are allocated by a single constructor-like function. By making these functions context-sensitive, we produce a separate object for each invocation of the function.

Second, heap objects in the store model can represent multiple memory blocks at runtime. For example, a program may contain a loop that repeatedly calls `malloc()` — our analyzer generates one heap object to represent all of the instances. We adopt the multiple instance analysis from Chase et al. [3] to determine when an allocation truly generates only one object. Previous work has also referred to this flow-sensitive property as *linearity* [11].

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 tighter coupling between the pointer analyzer 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 interface between the pointer analyzer and the client is simple, but it is the core mechanism that allows the framework to tailor its precision for the particular client and target program.

The implementation of this algorithm adds two components to our analysis framework: a *monitor* that detects and tracks loss of information during program analysis,

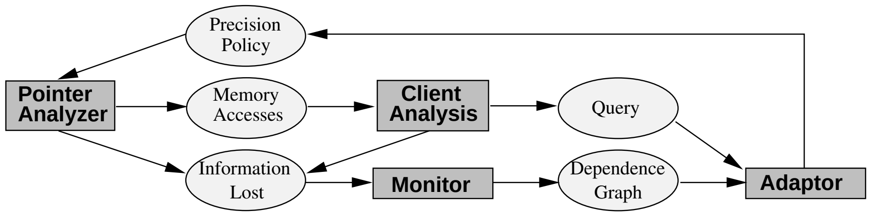

Fig. 2. Our analysis framework allows client analyses to provide feedback, which drives corrective adjustments to the precision.

and an *adaptor* that uses the output of the monitor to determine corrective adjustments to 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 takes over and performs its tasks—afterword 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 need more precision. The framework reruns the analysis with the customized precision policy. Figure 2 shows a diagram of the system.

Even though the algorithm detects information loss during analysis, it waits until the analysis is complete to change precision. One reason for this is pragmatic: our framework cannot change precision during analysis and recompute the results incrementally. There is a more fundamental reason, however: 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. 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 [3]. Therefore, a pointer assignment is polluting if it combines one 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 to 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 merged conflicting information from different call sites.
- Flow-insensitive assignment: multiple assignments to a single memory location merge conflicting information.
- Control-flow merge: the SSA phi 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,* and they always refer to the offending pointer:

- 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 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 of all the directly polluting assignments to that memory location—for our experiments we only record the parameter passing or flow-insensitive assignment cases. 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.

The 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*, 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, if necessary, 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 it 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.

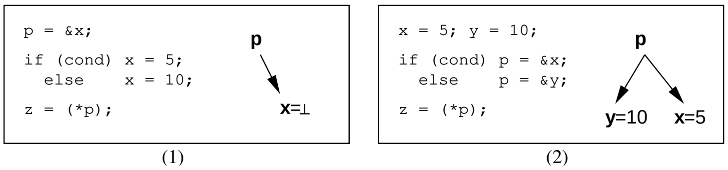

Fig. 3. Both code fragments assign bottom to z: in (1) x is responsible, in (2) p is responsible.

Figure 3 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 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 an memory location affect other location. 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

When analysis is complete, the client has an opportunity to use the results for its purposes, such as checking for error states or applying an optimization. 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 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 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. It inspects the list of directly polluting assignments labeling each node (if there are any) and determines the appropriate 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.

4.4 Chaining precision

In addition to addressing each polluting assignment, the adaptor increases precision along the whole 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. The motivation for this chaining is to ensure that intermediate locations preserve the additional accuracy provided by fixing 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. Our view is 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 our annotation language. This language 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, but due to space limitations we cannot present the full problem specification.

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.

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 in between the proper open and close calls. The purpose of this analysis client is to detect possible violations of this usage rule. The first line in Figure 4 defines the flow value for this analysis, which consists of the two possible states, "Open" and "Closed".

```
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. 4. Annotations for tracking file state: to properly model files and files descriptors, we associate the state with an abstract "handle".

To track this state, we annotate the various library functions that open and close files. Figure 4 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 4 shows an annotation for the `fgets()` function, which emits an error if the file could be closed.

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 are 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* [25, 21] 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 5, 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()` and `read()`,

```
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. 5. Annotations defining the Taint analysis: taintedness is associated with strings and buffers, and can be transferred between them.

and less obvious ones such as `readdir()` and `getenv()`. Figure 5 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 5 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 5 shows the annotation for the `syslog()` function, which is often the culprit.

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 the level that they are trusted. All data read from these sources is labeled likewise. The first line of Figure 6 defines the three levels of trust in our analysis—internal (trusted), locally trusted (for example, local files), and remote (untrusted).

We start by annotating functions that return fundamentally untrusted data sources, such as Internet sockets. Figure 6 shows the annotations for the `socket()` function. The level of trust depends on the kind of socket being created. When the 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

```
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. 6. 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.

`open()` is untrusted, then we treat all data read from that file descriptor as untrusted. The annotations in Figure 6 implement this policy.

As with the Taint analysis above, 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()`.

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 a 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 bug, but the bug 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. We 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.

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 it is possible for the program to send the contents of a file to a remote client, where the name of the file read 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 7 defines the flow value to track different kinds of sources and sinks of data. We can distinguish between different kinds 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. 7. Annotations to track kinds 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 kind. In addition, like the other analyses, we associate this kind with any data read from it. We check for FTP behavior in the `write()` family of routines, shown in Figure 7, by testing both the buffer and the file descriptor.

5.3 Programs

Table 1 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 software. Second, many of them are system tools or daemons that have significant security implications because they interact with remote clients and provide privileged services. Finally, several of them are specific versions of programs that are identified by security advisories as containing format string vulnerabilities. In addition, 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. Each bar on the graph shows the 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

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 propy	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 1. 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 the compiler internal representation—the table is sorted on this column.

fastest algorithm on that program, 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. In addition, Figure 13 shows the actual analysis times, averaged over all five clients.

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

- In 87 out of 90 cases the client-driven algorithm equals or beats the **accuracy** of the best fixed-precision policy. The other three cases appear to be anomalies, and we believe we can address them.
- In 64 of those 87 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.

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.

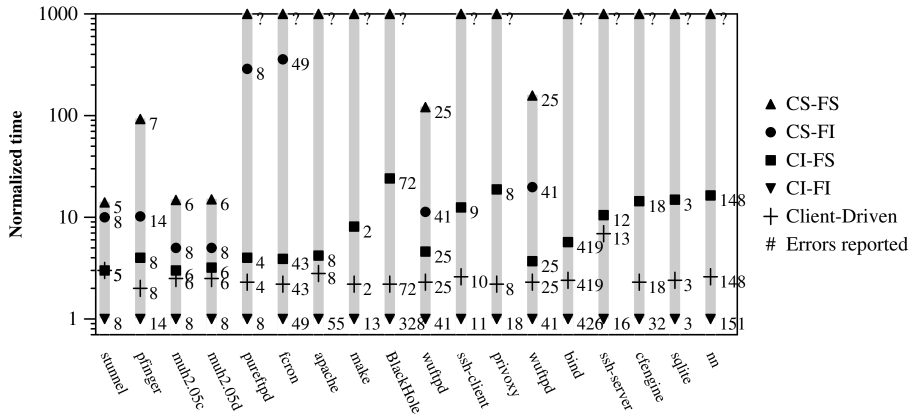

Fig. 8. Checking file access requires flow-sensitivity, but not context-sensitivity. The client-driven algorithm beats the other algorithms because it makes only the file-related objects flow-sensitive.

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.

For the problems that do require more precision, the client-driven algorithm consistently outperforms the fixed-precision algorithms. Table 2 provides 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. (In this table, hyphens represent the number 0.) These statistics show that client analyses often need some extra precision, but only a very small amount. In particular, the clients that benefit from context-sensitivity only need a tiny fraction of their procedures analyzed in this way.

6.1 Client-specific results

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

Figure 8 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.

Figure 9 shows the results for detecting format string vulnerabilities. The taintedness analysis that we use to detect format string vulnerabilities generally requires no extra precision. 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

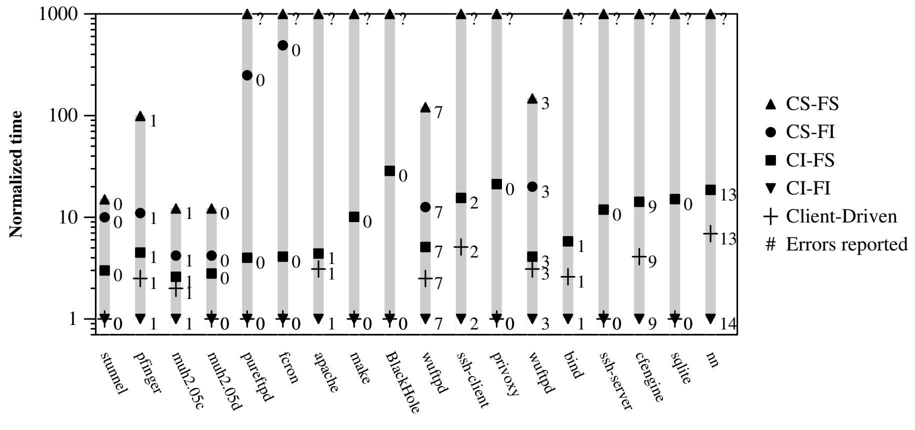

Fig. 9. Detecting format string vulnerabilities rarely benefits from either flow-sensitivity or context-sensitivity—the client-driven algorithm is only slower because it is a two-pass algorithm.

programs. The high false positive rates observed in previous work [21] are probably due to the use of equality-based analysis.

Figure 11 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 obtained 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 confirm their absence in the newer versions. Our analysis also found 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 10 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 12 shows the results for detecting FTP-like behavior, which is the most challenging problem because it depends on the states of multiple memory locations and multiple client analyses. However, our analysis does properly detect exactly those program points in the two FTP daemons that perform the "get" or "put" file transfer functions.

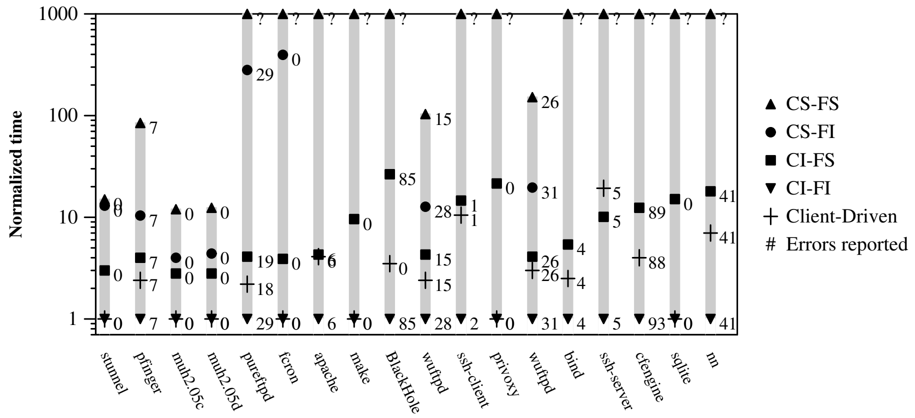

Fig. 10. Detecting remote access vulnerabilities can require both flow-sensitivity and context-sensitivity. In these cases the client-driven algorithm is both the most accurate and the most efficient.

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.

7 Conclusions

This paper presents a new client-driven approach to managing the tradeoff between cost and precision in pointer analysis. We show that such an approach is needed: 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.

Acknowledgments. We would like to thank Kathryn S. McKinley, Michael Hind, and the anonymous reviewers 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.

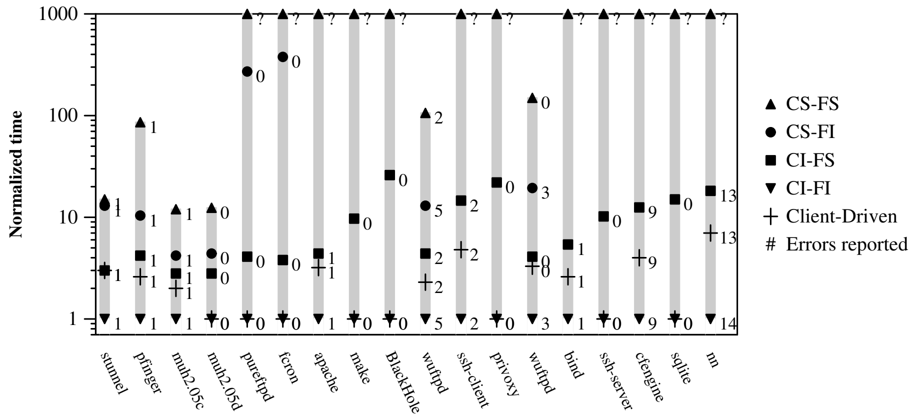

Fig. 11. Determining when a format string vulnerability is remotely exploitable is a more difficult, and often fruitless, analysis. The client-driven algorithm is still competitive with the fastest fixed-precision algorithm, and it even beats the other algorithms in three of the cases.

3. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.
4. F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Computational Complexity*, pages 253–267, 1996.
5. 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.
6. M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ACM Sigplan Notices, pages 35–46, 2000.
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. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, October 2000.
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. 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.
13. 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.
14. N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

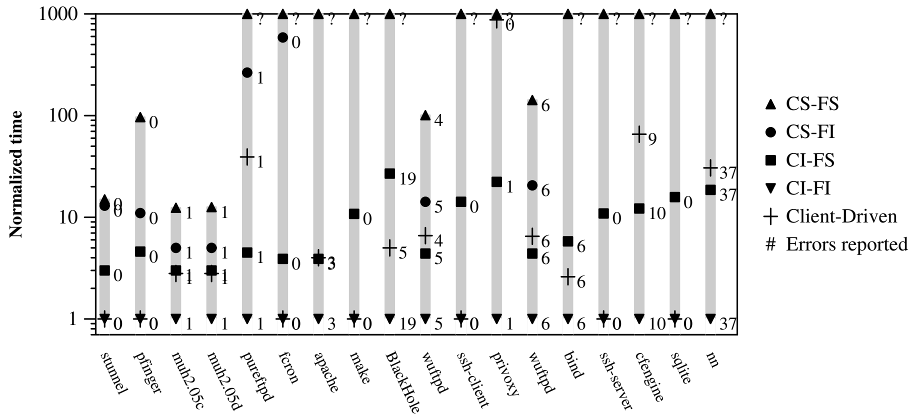

Fig. 12. Detecting FTP-like behavior is the most challenging analysis. In three cases (WU-FTP, privoxy, and CFEngine) the client-driven algorithm achieves accuracy that we believe only the full-precision algorithm can match—if it were able to run to completion.

15. 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.
16. M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, January 2001.
17. 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.
18. J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, October 1994.
19. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.
20. E. Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, 1995.
21. 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.
22. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. *Lecture Notes in Computer Science*, 1302, 1997.
23. 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.
24. R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliabiity. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.
25. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl, 3^{rd} Edition*. O'Reilly, July 2000.
26. 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.
27. 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.

Added Precision:	# of Procedures set C-S					% of Memory Locations set F-S				
Client Program:	File Access	FSV	Remote Access	Remote FSV	FTP Behavior	File Access	FSV	Remote Access	Remote FSV	FTP Behavior
stunnel-3.8	-	-	-	-	-	0.20	–	–	–	0.19
pfinger-0.7.8	-	-	1	-	-	–	0.53	0.20	0.53	0.61
muh2.05c	-	-	-	-	6	0.10	–	–	0.07	0.31
muh2.05d	-	-	-	-	6	0.10	–	–	–	0.33
pure-ftpd-1.0.15	-	-	2	-	9	0.13	–	0.12	–	0.10
fcron-2.9.3	-	-	-	-	-	–	–	0.03	–	0.26
apache-1.3.12	-	2	8	2	10	0.18	0.91	0.89	1.07	0.83
make-3.75	-	-	-	-	-	0.02	–	–	–	2.19
BlackHole-1.0.9	-	-	2	-	5	0.04	–	0.24	–	0.32
wu-ftpd-2.6.0	-	-	-	-	17	0.09	0.22	0.34	0.24	0.08
openssh-3.5p1-client	1	-	10	-	-	0.06	0.55	0.35	0.56	0.96
privoxy-3.0.0-stable	-	-	-	-	5	0.01	–	–	–	0.10
wu-ftpd-2.6.2	-	4	-	4	17	0.09	0.51	0.63	0.53	0.23
bind-4.9.4-REL	-	2	1	1	4	0.01	0.23	0.14	0.20	0.42
openssh-3.5p1-server	1	-	13	-	-	0.59	–	0.49	–	1.19
cfengine-1.5.4	-	1	4	3	31	0.04	0.46	0.43	0.48	0.03
sqlite-2.7.6	-	-	-	-	-	0.01	–	1.47	–	1.43
nn-6.5.6	-	1	2	1	30	0.17	1.99	1.82	2.03	0.97

Table 2. The precision policies created by the client-driven algorithm. Different clients have different precision requirements, but the amount of extra precision needed is typically very small.

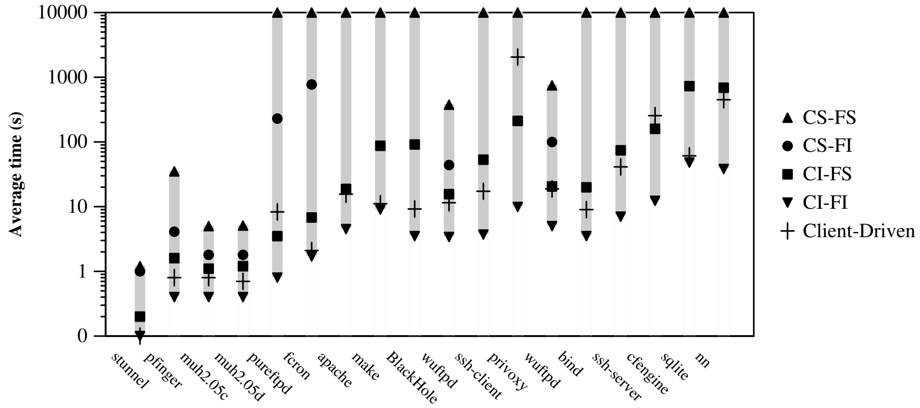

Fig. 13. The client-driven algorithm performs competitively with the fastest fixed-precision algorithm.