Software Validation via Scalable Path-Sensitive Value Flow Analysis

Nurit Dor
Tel Aviv University
nurr@tau.ac.il

Stephen Adams Manuvir Das Zhe Yang
Microsoft Research
<sra,manuvir,zhey>@microsoft.com

ABSTRACT

In this paper, we present a new algorithm for tracking the flow of values through a program. Our algorithm represents a substantial improvement over the state of the art. Previously described value flow analyses that are control-flow sensitive do not scale well, nor do they eliminate value flow information from infeasible execution paths (*i.e.*, they are path-insensitive). Our algorithm scales to large programs, and it is path-sensitive.

The efficiency of our algorithm arises from three insights: The value flow problem can be "bit-vectorized" by tracking the flow of one value at a time; dataflow facts from different execution paths with the same value flow information can be merged; and information about complex aliasing that affects value flow can be plugged in from a different analysis.

We have incorporated our analysis in ESP, a software validation tool. We have used ESP to validate the Windows operating system kernel (a million lines of code) against an important security property. This experience suggests that our algorithm scales to large programs, and is accurate enough to trace the flow of values in real code.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Reliability, Verification

Keywords

Value flow, alias analysis, path-sensitive analysis

1. INTRODUCTION

Value flow analysis has been studied under many different names in many different areas. For example, the dataflow analysis community has studied value flow analysis as reaching definitions [24, 5], the functional programming community has studied it as flow analysis and closure analysis [22, 29], and the symbolic evaluation community has studied it as strongest post-conditions [16].

Regardless of name or area, value flow analysis answers the following question: Which memory locations hold a given value of interest [at a given program point [along a given execution path]]? Values of interest are generally symbolic; for instance, we may be interested in tracing the flow of an untrusted value passed in as an argument to a function in a trusted codebase. We refer to the set of memory locations that hold a value as its "value alias set".

In recent years, software validation tools based on interprocedural value flow analysis have come of age [4, 10, 15]. These tools use exhaustive analysis to find all potential errors of a given kind in a program. They use value flow analysis to determine whether a given statement operates on a given value of interest. For instance, in a given program, does the function call `Lock(p→mLock)` operate on the lock object that was created by a preceding function call `l = NewLock()`? If not, the program may contain a locking error. The answer depends on whether `p→mLock` is in the value alias set of `l`.

The difficult problem that limits the adoption of software validation tools is that there is no known algorithm for value flow analysis that is both accurate enough for use in these tools, and scalable enough to run on large programs. In this paper, we present such an algorithm. Our algorithm is scalable, and it is path-sensitive, in that it eliminates value flow information from infeasible execution paths.

Our algorithm is based on three different insights, each of which allows us to reuse results from previous work in a new context. The technical merit of this paper lies in the adaptation and combination of these ideas to produce a useful result.

The first insight is that value flow analysis is a separable dataflow problem [24] *even in the presence of memory aliasing*, if we factor out the computation of memory aliases from the value flow analysis. It is well known that separable dataflow problems can be bit-vectorized, resulting in analyses with a much smaller memory footprint for each subproblem [23]. We can therefore compute value alias sets for one value of interest at a time.

The second insight is that for a given value of interest, most branches in a program do not impact its value alias set, even though they affect the concrete execution state of the program. Therefore, we can use the selective merging tech-

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

nique described in [10] to design a scalable path-sensitive analysis that tracks only relevant branches.

Our "value flow simulation" algorithm identifies relevant branches as follows: We symbolically evaluate the program, generating symbolic states that include both the execution state and the value alias set. At merge points in the control flow graph, if two symbolic states have the same value alias set, we produce a single symbolic state by merging their execution states. Otherwise, we process the symbolic states separately. This approach avoids the cost of full path-sensitive analysis, yet captures relevant correlations.

The third insight is that when computing value alias sets in programs with memory aliasing, we can use compact placeholders to represent memory aliases instead of representing all memory aliases explicitly in the value alias sets. This implicit representation of value alias sets allows us to combine precise value flow analysis with well-known scalable memory alias analyses [9, 18, 11], even when the imprecision of scalable analyses may lead to very large alias sets.

The motivation for our work is to enable software validation tools. More generally, many important analyses are instances of value flow analysis. For instance, call graph construction involves tracking the flow of function names to indirect call sites, and memory alias analysis itself involves tracking the flow of memory addresses to dereference operations. Value flow simulation provides a scalable, path-sensitive method for solving these and other problems.

In this paper, we make the following contributions:

- We present value flow simulation, a new framework for inter-procedural, context-sensitive, path-sensitive value flow analysis. The framework can be instantiated by fixing the choice of transfer functions for value alias sets and symbolic execution states.
- We introduce the notion of a value alias set, and provide a transfer function on these sets. Value alias sets allow a client analysis to perform strong updates even in the presence of memory aliasing. We show how information from an off-the-shelf memory alias analysis can be used to conservatively and efficiently update value alias sets.
- We use value flow simulation in ESP, a software validation tool for C code. Our experience using ESP to validate the Windows operating system kernel against an important security property suggests that:
  - Value flow simulation is scalable. ESP is able to trace the flow of user input values through a million lines of kernel code.
  - Value flow simulation is accurate. Of the roughly 500 input values tracked by ESP, it fails to accurately track the flow (leading to false error reports) of 30 values, a failure rate of 6%.

The rest of this paper is organized as follows: In Section 2, we provide motivating examples for our approach. In Section 3, we present inter-procedural value flow simulation. In Section 4, we present a series of transfer functions on value alias sets. In Section 5, we describe our experiment on the Windows kernel. We summarize related work in Section 6, and conclude in Section 7.

2. EXAMPLES

In this section, we use examples to explain value alias sets and value flow simulation, and to demonstrate how they improve upon previous work.

2.1 Value alias sets

Consider the snippet of (simplified) code from an operating system kernel shown in Figure 1 (a). Suppose we are interested in enforcing the property that kernel functions should only release locks that have previously been acquired.

For the example program, the analysis would need to check that on every path where the lock created at line `k` is released, it was previously acquired. The analysis must be able to update the state of the lock to "acquired" after line `l`. The problem is that the call on line `l` operates on a pointer expression that refers to an unknown memory location. In previous approaches, either the analysis involves an expensive shape/heap analysis [22, 27], or the analysis is forced to perform a weak update, meaning that the lock may or may not have been acquired. This would lead to a false error report at the call to `UnLock`. We would like an analysis that can perform a strong update even if the call operates on an arbitrary pointer expression, without relying on expensive shape analysis.[1]

Our solution is to introduce the notion of a value alias set. Intuitively, the value alias set for a given value of interest includes those expressions in the program that evaluate to the given value at a particular point. For instance, at line `l` in the example, the value alias set for the lock created at line `k` is {`p`→`mLock`}. This means that regardless of which memory location `p`→`mLock` refers to, the value held by that memory location is the value created at line `k`. As a result, we can perform a strong update on the state of the lock at line `l`.

In Section 4, we formally define value alias sets and provide a transfer function for value alias sets.

2.2 Value flow simulation

Consider the code shown in Figure 1 (b). Suppose we are interested in enforcing the property that the kernel should not dereference `NULL` pointers passed in from user code.

For the example program, the analysis would need to track the flow of the values passed in through pointer parameters `o` and `p` to dereference operations in the code of `KernelEntryPoint`. The analysis must track the correlation that `q` holds the value of user pointer `o` only if `o` ≠ `NULL`. In other words, the analysis must be path-sensitive. Previous approaches for path-sensitive value flow analysis [5, 4] do not scale to large programs.

As mentioned in Section 1, we can bit-vectorize the problem by tracking the value flow of parameters `o` and `p` independently. This bit-vectorization leads to two orthogonal improvements in scalability: First, the size of the dataflow facts is reduced. We need to compute and store value alias sets for only one parameter at a time. Second, fewer branches in the code are relevant. In the example, the second branch is irrelevant to the value flow of parameter `o`, while the first branch is irrelevant to the value flow of parameter `p`.

Example 1. Consider an analysis that tracks the value

[1]Recent work by Aiken et al [2] allows for strong updates without shape analysis, when the aliasing relationships in the code satisfy certain constraints (see Section 6).

```
(a)   void Process() {
      k: p->mLock = NewLock();

      l: Lock(p->mLock);
         ...
         UnLock(p->mLock);
      }

(b)   void KernelEntryPoint(int *o, int *p) {
        if (o != NULL)
          q = o;
        else
          q = malloc();

        if (p != NULL)
          r = p;
        else
          r = malloc();

      m: *q = data1;
         *r = data2;
       }
```

Figure 1: Code snippets from an OS kernel.

flow of parameter `o` by computing symbolic states, as described in Section 1. The possible symbolic states at line `m` inferred by three possible analyses are described below. Each symbolic state includes the value alias set, and a simulation state. Value alias sets have two components: the "Must" set (shown as {}) includes expressions that *definitely* evaluate to the tracked value, while the "May" set (shown as $\{\}^?$) includes expressions that *possibly* evaluate to the tracked value.[2]

Full path-sensitive analysis. There are four feasible paths to line `m`, resulting in the states:

$[\{o\}\{o\}^?, o = \texttt{NULL}, q = \texttt{new}, p = \texttt{NULL}, r = \texttt{new}]$
$[\{o\}\{o\}^?, o = \texttt{NULL}, q = \texttt{new}, p \neq \texttt{NULL}, r = p]$
$[\{o, q\}\{o, q\}^?, o \neq \texttt{NULL}, q = o, p = \texttt{NULL}, r = \texttt{new}]$
$[\{o, q\}\{o, q\}^?, o \neq \texttt{NULL}, q = o, p \neq \texttt{NULL}, r = p]$

These states capture the correlation that variable `q` is in the value alias set, meaning that the statement at line `m` dereferences parameter `o`, iff $o \neq \texttt{NULL}$, so there is no error.

Standard dataflow analysis. Symbolic states are merged at join points in the CFG. Therefore, the single symbolic state at line `m` is $[\{o\}\{o, q\}^?, \top]$[3]. This state allows the analysis to consider the possibility that $o = \texttt{NULL}$ and `q` is in the value alias set, leading to a false error report at line `m`.

Value flow simulation. The symbolic states from the two legs of the first branch have different value alias sets, and are therefore not merged. The second branch does not affect the value alias sets of parameter `o`. Therefore, the four symbolic states arising from the second branch are merged into two states at `m`:

$[\{o\}\{o\}^?, o = \texttt{NULL}, q = \texttt{new}]$
$[\{o, q\}\{o, q\}^?, o \neq \texttt{NULL}, q = o]$

These states capture the desired correlation, but drop the irrelevant correlation from the second branch. □

[2]By definition, $Must(va) \subseteq May(va)$.
[3]$\top$ represents all possible execution states.

3. VALUE FLOW SIMULATION

In this section, we present value flow simulation, an algorithm for inter-procedural, context-sensitive, and path-sensitive value flow analysis. Value flow simulation is based on the "property simulation" algorithm previously described in depth in [10]. The key difference is that property simulation propagates abstract finite state machine states as its dataflow facts, whereas value flow simulation propagates value alias sets. We refer the reader to either [10] or [13] for further details on the two algorithms, including termination and complexity arguments.

Here, we provide pseudocode for the full inter-procedural version of value flow simulation, and we provide an overview of the two key ingredients of the algorithm: selective merging of symbolic states, and efficient handling of procedure calls.

3.1 A simple imperative language

Value flow simulation operates on the supergraph [26] of a program. The supergraph includes a standard CFG for every procedure, and edges representing inter-procedural control flow. The supergraph contains the following kinds of nodes and edges:

Compute:	Assignment statements, single successor
Branch:	Conditional branch, True/False successors
Call:	"Parameterless" procedure call
	Successor: *Entry* node of callee
Entry:	Procedure entry, single successor
Exit:	Procedure exit
	Successors: *CallReturn* nodes of callers
CallReturn:	Return from procedure call

We assume that all indirect calls have been replaced with direct calls using some form of call graph analysis [25], and that procedure calls have been simplified to parameterless calls (*c.f.* [21]). For example, a call `foo(x+y,z*2)` to function `foo(int a, int b)` is replaced with:

```
t1 := x+y; t2 := z*2; a := t1; b := t2; foo()
```

3.2 Domains

We use appropriately named accessor functions to extract edge information from CFG nodes and vice versa.

The algorithm references the following domains:

N	CFG nodes
F	Functions
S	Simulation states
VA	Value alias sets
$C = N \times VA \times VA$	Analysis coordinates

Transfer functions that lookup the expression associated with a CFG node and compute new information are prefixed by *M*. Functions subscripted by *va* and *s* apply to value alias sets and simulation states, respectively. *Into* functions map information from callers to callees, *Out* functions map information from callees to callers, and *Over* functions preserve information at call sites that cannot be modified by callees.

3.3 Overview

The pseudo code for value flow simulation (VFS) is given in Figure 2. Given a variable `t` and a program point `n` at which `t` is assigned the tracked value (we refer to `n` as the "creation point"), the algorithm computes value alias sets for the value produced at `n`, at all program points.

```
   global
1    Worklist : 2^C
2    Info : C → S
3    Summary : (F × VA) → 2^C

   procedure Solve(Global G = [N, E, F] Value = [t, n])
   begin
4    start := (entry(fn(n)), ⊥, ⊥)
5    c := (succ(n), ⊥, {t}{t}^?)
6    Info(start) := ⊤
7    AddCoord(start, c, ⊤)

8    while Worklist ≠ ∅
9      Remove a coordinate c = [n, id, a] from Worklist
10     switch(n)

11       case n ∈ Call:
12         aInto := MInto_va(n, a)
13         sInto := MInto_s(n, Info(c))
14         AddCoord(c, (entry(callee(n)), aInto, aInto), sInto)
15         aOver := MOver_va(n, a)
16         sOver := MOver_s(n, Info(c))
17         foreach c' = [n', id', a'] ∈ Summary(callee(n), aInto)
18           aRet := Combine(aOver, MOut_va(a'))
19           sRet := Combine(sOver, MOut_s(Info(c')))
20           AddCoord(c, (retNode(n), id, aRet), sRet)
21       case n ∈ CallReturn:
22         AddCoord(c, (succ(n), id, a), Info(c))
23       case n ∈ Entry:
24         AddCoord(c, (succ(n), id, a), Info(c))
25       case n ∈ Exit:
26         AddSummary(fn(n), id, c)

27       case n ∈ Compute:
28         a' := M_va(n, a)
29         s' := M_s(n, Info(c))
30         AddCoord(c, (succ(n), id, a'), s')
31       case n ∈ Branch:
32         s' := MTrue(n, Info(c))
33         s'' := MFalse(n, Info(c))
34         AddCoord(c, (succ_T(n), id, a), s')
35         AddCoord(c, (succ_F(n), id, a), s'')
   end

   procedure AddCoord(c, c', s')
   begin
36   if s' ≠ ⊥
37     preds(c') ∪= {c}
38     if s' ⋢ Info(c')
39       Info(c') ⊔= s'
40       Worklist ∪= {c'}
   end

   procedure AddSummary(f, id, c)
   begin
41   if id = ⊥
42     foreach m ∈ callSites(f)
43       Info((m, ⊥, ⊥)) := ⊤
44       Info((entry(fn(m)), ⊥, ⊥)) := ⊤
45       AddCoord((entry(fn(m)), ⊥, ⊥), (m, ⊥, ⊥), ⊤)
46       AddCoord(((m, ⊥, ⊥), (entry(f), ⊥, ⊥), ⊤)
47   Summary(f, id) ∪= {c}
48   Worklist ∪= preds((entry(f), id, id))
   end
```

Figure 2: VFS: An algorithm for path-sensitive, context-sensitive inter-procedural value flow analysis.

If there are multiple creation points, a separate instance of the algorithm is used for each creation point, resulting in a bit-vectorized analysis. The advantages of this approach are that (a) each instance of the algorithm has a smaller memory footprint because it computes alias sets for a single created value, and (b) the individual instances are independent, meaning that we can parallelize the computation of value flow queries for a client analysis across as many machines as are available.

VFS is a worklist algorithm that is similar to the interprocedural analysis of Reps, Horwitz and Sagiv (RHS, [26]). The algorithm dynamically creates and explores a space of "analysis coordinates", by applying transfer functions at CFG nodes and pushing value alias sets to successors, until no more coordinates can be produced.

Context-sensitivity. Coordinates can be understood conceptually as the result of "exploding" the supergraph in two ways: First, every CFG is exploded into multiple clones, one for every value alias set with which the function is entered. Second, every cloned CFG node is exploded into multiple coordinates, one for every value alias set produced at the node. A call from a call site with a given value alias set resolves to the clone of the callee with the matching value alias set. In this way, the domain of dataflow facts is used to achieve a measure of context-sensitivity. This is the key insight behind both RHS and VFS.

Procedure calls are handled efficiently by maintaining procedure summaries that grow dynamically, and plugging in summaries in place of transfer functions at call nodes.

Path-sensitivity. VFS is a path-sensitive analysis. The goal of VFS is to traverse only feasible execution paths.

VFS incorporates a symbolic evaluator that uses transfer functions to compute strongest post-conditions. Every coordinate is associated with a simulation state produced by the symbolic evaluator. The simulation state represents (over-approximates) the set of execution states under which the tracked value has the given set of value aliases at that program point. When a coordinate is propagated through a *Branch* node, the simulation state is used to rule out infeasible branch directions.

It is important to understand that VFS provides only partial path-sensitivity. If two different execution paths to the same point result in the same coordinate, then their simulation states are merged. If they result in different coordinates, their simulation states are kept apart. In this way, the domain of dataflow facts is used to keep execution paths apart to some degree. This is the key insight behind both [10] and VFS.

Example 2. The coordinates produced by VFS at line m of the example program in Figure 1, along with the associated simulation states, are given below:

$$\langle \mathtt{m} :, \bot, \{\mathtt{o}\}\{\mathtt{o}\}^? \rangle \rightarrow [\mathtt{o} = \mathtt{NULL}, \mathtt{q} = \mathtt{new}]$$
$$\langle \mathtt{m} :, \bot, \{\mathtt{o}, \mathtt{q}\}\{\mathtt{o}, \mathtt{q}\}^? \rangle \rightarrow [\mathtt{o} \neq \mathtt{NULL}, \mathtt{q} = \mathtt{o}] \qquad \square$$

VFS performs two fixed point computations: One fixed point computation explores the space of coordinates, while the other weakens the simulation state (*i.e.*, expands the set of execution states) associated with each coordinate.

$Field = \texttt{NULL} + FieldNames$
$OneLevelExpr ::= Var \mid Var \rightarrow Field$
$PointerExpr ::= Var \mid PointerExpr \rightarrow Field$

$LhsExpr = OneLevelExpr$
$RhsExpr = OneLevelExpr \mid \texttt{CREATE}$
$Statement = LhsExpr \times RhsExpr$

$MustSet = 2^{PointerExpr}$
$VA = MustSet$

$\rho : Env = PointerExpr \rightarrow Location$
$\sigma : Store = Location \rightarrow Value$
$\sigma^{-1} : StoreInv = Value \rightarrow 2^{Location}$

Correctness Criteria: $\mathbf{M} : MustSet \times Env \rightarrow 2^{Location}$
$\mathbf{M}[\![must]\!]\,\rho = \sigma^{-1}(\texttt{CREATE})$

Figure 3: A simple pointer assignment language.

4. VALUE ALIAS SETS

In the previous section, we treated value alias sets as abstract dataflow facts. In this section, we provide a domain and transfer function for value alias sets. The goal of the transfer function is to (a) put as few expressions as possible in the May set, so that spurious value flow is minimized, and (b) put as many expressions as possible in the Must set, so that a client analysis can perform strong updates.

The technical contributions of this section are a novel method for combining memory alias analysis with value flow analysis, and an implicit representation for memory aliases in value alias sets.

We first describe a simple language of pointer assignments, and use this language to formalize the notion of value alias sets and transfer functions. We then define progressively less precise but more concise abstract domains and transfer functions for value alias sets. We explain the intuition behind each step, and informally justify its correctness.

4.1 A simple pointer assignment language

The domains given in Figure 3 define a simple language of pointer assignments. A program in this language consists of a series of statements that manipulate pointers and scalars. The assumption is that a number of memory locations have already been allocated. These locations can be accessed through dereference expressions formed from *PointerExpr*. There is a distinguished scalar value, `CREATE`, which is the single value of interest. After the execution of every statement, some set of memory locations hold this value. The value alias set contains exactly the set of pointer expressions that refer to these memory locations.

We use two macros, $\rightarrow^*$ and $\rightarrow^+$, for pointer expressions: $\texttt{e} \rightarrow^* \texttt{e}'$ appends access path $\texttt{e}'$ consisting of zero or more dereferences to access path $\texttt{e}$, while $\texttt{e} \rightarrow^+ \texttt{e}'$ appends access path $\texttt{e}'$ consisting of one or more dereferences to $\texttt{e}$. For example, $(\texttt{x} \rightarrow \texttt{f}) \rightarrow^* (\texttt{g} \rightarrow \texttt{h}) \equiv \texttt{x} \rightarrow \texttt{f} \rightarrow \texttt{g} \rightarrow \texttt{h}$.

We use a semantics in which the domain of the environment includes all pointer expressions, not just variables. Such an environment can be obtained from the standard semantics by composing the usual variable environment and the store. Our environment captures the shape of the store (as a prefix-closed partition of pointer expressions [6]), including the memory aliasing structure, at every point.

4.2 A concrete transfer function

A concrete transfer function on value alias sets is given in Figure 4. We use the standard Gen/Kill formulation.

It is important to understand the distinction between value aliases and memory aliases. Expressions $\texttt{e}_1$ and $\texttt{e}_2$ are value aliases iff $\sigma\ \rho\ \texttt{e}_1 = \sigma\ \rho\ \texttt{e}_2$. Expressions $\texttt{e}_1$ and $\texttt{e}_2$ are memory aliases iff $\rho\ \texttt{e}_1 = \rho\ \texttt{e}_2$. These definitions show that memory aliasing implies value aliasing. Therefore, there are two ways in which an assignment can cause $\texttt{e}_1$ to become value aliased with $\texttt{e}_2$: Either the assignment copies the value stored in $\rho\ \texttt{e}_2$ to $\rho\ \texttt{e}_1$, or the assignment sets $\rho\ \texttt{e}_1$ equal to $\rho\ \texttt{e}_2$.

MustGen adds expressions to the value alias set to match the two possibilities above. If an assignment causes $\texttt{lhs} \rightarrow^* \texttt{fs}$ to evaluate to the tracked value, we add $\texttt{lhs} \rightarrow^* \texttt{fs}$ to the value alias set. For all expressions $\texttt{e}$ that are memory aliases of `lhs`, $\texttt{e} \rightarrow^* \texttt{fs}$ now also evaluates to the tracked value. Hence, $\texttt{e} \rightarrow^* \texttt{fs}$ is added to the value alias set. *MustGen* uses *LocAlias*, an oracle that returns true iff two pointer expressions refer to the same memory location in the current environment, to identify memory aliases.

MustKill removes any expression whose prefix is updated by the assignment, as it may no longer evaluate to the tracked value. Note that if the expression continues to evaluate to the value after the update, it is re-introduced by *MustGen*.

4.3 An abstract transfer function

The concrete transfer function in Figure 4 is not computable, because it relies on a concrete memory aliasing oracle. We approximate the semantics of value aliasing using the abstract transfer function shown in Figure 5. The abstract transfer function uses *MustLocAlias* and *MayLocAlias* as its memory aliasing oracle. $\mathit{MustLocAlias}(\texttt{e}_1, \texttt{e}_2)$ under-approximates must aliases; it returns true only if $\texttt{e}_1$ and $\texttt{e}_2$ must refer to the same memory location. $\mathit{MayLocAlias}(\texttt{e}_1, \texttt{e}_2)$ over-approximates may aliases; it returns true if $\texttt{e}_1$ and $\texttt{e}_2$ may refer to the same memory location. *MustLocAlias* and *MayLocAlias* can be implemented using any off-the-shelf memory alias analysis [31, 8, 19].

The consequence of using *MayLocAlias* is that we may include expressions that are not value aliases in the value alias set. Therefore, we extend the value alias set to include a Must set and a May set. The Must set contains only expressions that must evaluate to the tracked value. The May set contains all expressions that may evaluate to the tracked value. It is a mechanism for distinguishing between expressions that may be in the concrete value alias set, and expressions that are definitely not in the concrete value alias set (*i.e.*, it is a concise representation of the "must not" set).

Correctness. In order for the transfer function to be conservative, the Must set should under-represent the concrete value alias set, while the May set should over-represent the concrete value alias set.

In *MustGen*, we replace *LocAlias* with *MustLocAlias*, so fewer expressions are added to the Must set. In *MustKill*, we replace *LocAlias* with *MayLocAlias*, so more expressions are removed from the Must set. Therefore, the Must set is a subset of the concrete value alias set. In *MayGen*, we

$LocAlias : MemoryAliasEnv \times PointerExpr \times PointerExpr \to Boolean$

$M_{va} : Statement \times VA \times MemoryAliasEnv \to VA$

$M_{va}(v, must, aenv) \;=\; (must - MustKill(v, must, aenv)) \cup MustGen(v, must, aenv)$

$$MustGen(\langle \texttt{lhs}, \texttt{rhs} \rangle, must, aenv) = \{\texttt{e} \mid LocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} = \texttt{CREATE})\} \cup \{\texttt{e} \to^* \texttt{fs} \mid LocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} \to^* \texttt{fs} \in must)\}$$

$$MustKill(\langle \texttt{lhs}, \texttt{rhs} \rangle, must, aenv) = \{\texttt{e} \to^* \texttt{fs} \mid LocAlias(aenv, \texttt{e}, \texttt{lhs})\}$$

Figure 4: Concrete transfer function for value alias sets.

$MustLocAlias : MemoryAliasEnv \times PointerExpr \times PointerExpr \to Boolean$
$MayLocAlias : MemoryAliasEnv \times PointerExpr \times PointerExpr \to Boolean$

$MaySet = 2^{PointerExpr}$
$VA = MustSet \times MaySet$

Correctness Criteria: $\mathbf{M}^? : MaySet \times Env \to 2^{Location}$
$\mathbf{M}[\![must]\!]\,\rho \subseteq \sigma^{-1}(\texttt{CREATE})$
$\mathbf{M}^?[\![may]\!]\,\rho \supseteq \sigma^{-1}(\texttt{CREATE})$

$M_{va}(v, \langle must, may \rangle, aenv) \;=\; \langle M_{must}(v, must, aenv), M_{may}(v, may, aenv) \rangle$
$M_{must}(v, must, aenv) \;=\; (must - MustKill(v, must, aenv)) \cup MustGen(v, must, aenv)$
$M_{may}(v, may, aenv) \;=\; (may - MayKill(v, may, aenv)) \cup MayGen(v, may, aenv)$

$$MustGen(\langle \texttt{lhs}, \texttt{rhs} \rangle, must, aenv) = \{\texttt{e} \mid MustLocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} = \texttt{CREATE})\} \cup \{\texttt{e} \to^* \texttt{fs} \mid MustLocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} \to^* \texttt{fs} \in must)\}$$

$$MustKill(\langle \texttt{lhs}, \texttt{rhs} \rangle, must, aenv) = \{\texttt{e} \to^* \texttt{fs} \mid MayLocAlias(aenv, \texttt{e}, \texttt{lhs})\}$$

$$MayGen(\langle \texttt{lhs}, \texttt{rhs} \rangle, may, aenv) = \{\texttt{e} \mid MayLocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} = \texttt{CREATE})\} \cup \{\texttt{e} \to^* \texttt{fs} \mid MayLocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} \to^* \texttt{fs} \in may)\}$$

$$MayKill(\langle \texttt{lhs}, \texttt{rhs} \rangle, may, aenv) = \{\texttt{e} \to^* \texttt{fs} \mid MustLocAlias(aenv, \texttt{e}, \texttt{lhs})\}$$

Figure 5: Abstract transfer function for value alias sets.

$GlobalMayAliasesOf : MaySet \to 2^{PointerExpr}$

Correctness Criteria: $\mathbf{M}[\![must]\!]\,\rho \subseteq \sigma^{-1}(\texttt{CREATE})$
$\mathbf{M}^?[\![GlobalMayAliasesOf\,(may)]\!]\,\rho \supseteq \sigma^{-1}(\texttt{CREATE})$

$$MayGen(\langle \texttt{lhs}, \texttt{rhs} \rangle, may, aenv) = \{\texttt{lhs} \mid \texttt{rhs} = \texttt{CREATE}\} \cup \{\texttt{lhs} \to^* \texttt{fs} \mid \texttt{rhs} \to^* \texttt{fs} \in GlobalMayAliasesOf\,(may)\}$$

$$MayKill(\langle \texttt{lhs}, \texttt{rhs} \rangle, may, aenv) = \{\texttt{lhs} \mid \texttt{lhs} \in Var\}$$

Figure 6: Implicit May alias sets.

$$MayGen(\langle \texttt{lhs}, \texttt{rhs} \rangle, may, aenv) = \{\texttt{lhs} \mid \texttt{rhs} = \texttt{CREATE}\} \cup \{\texttt{lhs} \mid \texttt{rhs} \in GlobalMayAliasesOf\,(may)\}$$

Figure 7: Avoid search for expressions prefixed by `rhs`.

$$MustSet = 2^{OneLevelExpr}$$

$$MustGen(\langle \texttt{lhs}, \texttt{rhs} \rangle, must, aenv) = \{\texttt{e} \in MultiLevelExpr(k) \mid MustLocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} = \texttt{CREATE})\} \cup \{\texttt{e} \rightarrow^* \texttt{fs} \in MultiLevelExpr(k) \mid MustLocAlias(aenv, \texttt{e}, \texttt{lhs}) \wedge (\texttt{rhs} \rightarrow^* \texttt{fs} \in must)\}$$

Figure 8: Finite representation for Must sets. *MultiLevelExpr*(k) **includes all pointer expression with at most** k **levels of derefefence (***i.e.* **occurrences of** $\rightarrow$**).**

replace *LocAlias* with *MayLocAlias*, so more expressions are added to the May set. In *MayKill*, we replace *LocAlias* with *MustLocAlias*, so fewer expressions are removed from the May set. Therefore, the May set is a superset of the concrete value alias set.

The abstract transfer function makes one of the insights of this paper explicit, namely that value alias sets can be computed in programs with pointer assignments by plugging in information from a conservative memory alias analysis.

4.4 Implicit representation of memory aliases

Efficient value alias set computation requires the use of a relatively imprecise memory alias analysis. The problem is that the more imprecise the alias analysis, the more expressions would be added to the May set. For instance, if the tracked value is assigned to `*p`, every variable potentially pointed to by `p` would be added to the May set.

A key insight of this paper is that if we use a flow-insensitive memory alias analysis, it is possible to avoid adding all memory aliases to the May set. This insight allows us to combine precise value alias analysis with imprecise memory alias analysis in an efficient manner.

A transfer function with "implicit" may sets is shown in Figure 6. The technical change from the previous transfer function is that in *MayGen*, we use *GlobalMayAliasesOf* to compute the flow-insensitive closure of the May set under memory aliasing.[4] $\texttt{e}_1 \in GlobalMayAliasesOf(\texttt{e}_2)$ returns true if $\texttt{e}_1$ and $\texttt{e}_2$ could refer to the same memory location at some point during the execution of the program.

This change to an implicit representation allows us to avoid adding $\texttt{e} \rightarrow^* \texttt{fs}$ to the May set. It requires more careful treatment of *MayKill*. Because an expression in the May set implicitly represents all of its memory aliases, we can no longer remove expressions from the May set due to assignment. The only exception is for variables.

We continue to represent all memory aliases explicitly in the Must set, because these sets are expected to be small.

Correctness. *MayGen*: If `lhs` and `e` are memory aliases, then $\texttt{lhs} \rightarrow^* \texttt{fs}$ and $\texttt{e} \rightarrow^* \texttt{fs}$ are also memory aliases. Therefore, if $\texttt{lhs} \rightarrow^* \texttt{fs}$ is added to the May set, $\texttt{e} \rightarrow^* \texttt{fs}$ will be included by the closure computation wherever the May set is used. *MayKill*: Consider an expression `e` that is aliased with the lhs variable `x`. By removing `x`, we are potentially eliminating `e` from the closure of the May set. If the only reason for including `e` is that `e` refers to the same location as `x`, then the update to `x` must be an update to `e` as well, in which case it is correct to eliminate `e`. Otherwise, `e` will remain in the closure because of some other expression.

[4]The query $\texttt{rhs} \rightarrow^* \texttt{fs} \in GlobalMayAliasesOf(may)$ can be implemented efficiently using well known techniques [9, 18].

4.5 Avoiding prefix-based search in May sets

The transfer function in Figure 6 has the drawback that we must search the May set for all pointer expressions prefixed by `rhs`. We can avoid this search, and further reduce the number of expressions added to the May set, by using the transfer function shown in Figure 7. This transfer function also ensures that only expressions in *OneLevelExpr* are added to May sets. Therefore, the May sets are bounded.

Correctness. The improvement in Figure 7 is that *MayGen* does not add $\texttt{lhs} \rightarrow^+ \texttt{fs}$ to the May set if $\texttt{rhs} \rightarrow^+ \texttt{fs}$ is in the closure of the May set. The key insight is that a correct flow-insensitive memory alias analysis that has processed this assignment must map $\texttt{lhs} \rightarrow^+ \texttt{fs}$ to a superset of the locations mapped by $\texttt{rhs} \rightarrow^+ \texttt{fs}$. Therefore, $\texttt{lhs} \rightarrow^+ \texttt{fs}$ must already be included in the closure of any May set whose closure includes $\texttt{rhs} \rightarrow^+ \texttt{fs}$.

The tricky issue is that memory may-aliasing is not transitive. Therefore, we must still explain why dropping $\texttt{lhs} \rightarrow^+ \texttt{fs}$ from the May set does not eliminate $\texttt{e} \rightarrow^+ \texttt{fs}$ from the closure, for expressions `e` that are memory aliases of `lhs`. The argument is as follows: If `e` and `lhs` are memory aliases, then a correct flow-insensitive memory alias analysis must map $\texttt{e} \rightarrow^+ \texttt{fs}$ to a superset of the locations mapped by $\texttt{lhs} \rightarrow^+ \texttt{fs}$, and vice versa. Therefore, $\texttt{e} \rightarrow^+ \texttt{fs}$ must also be included in the closure of any May set whose closure includes $\texttt{lhs} \rightarrow^+ \texttt{fs}$.

4.6 Finite Must sets

The final complication is that the Must sets may be unbounded. For instance, consider an assignment $\texttt{x} \rightarrow \texttt{f} = \texttt{x}$ within a loop, where `x` holds the tracked value. As shown in Figure 8, we can avoid this problem by arbitrarily limiting the dereference depth of expressions added to the Must set. The effect of this truncation is a potential loss of precision in a client analysis that relies on Must sets to perform strong updates.

4.7 Putting it all together

We have followed the progression of steps above in order to properly explain the intuition behind our transfer function for value alias sets, and in order to justify its correctness. The general theme of the progression is that as long as the May component is conservative, we can arbitrarily limit the amount of information we track in the Must component. Therefore, we can use the Must component to trade-off precision (more Must information) with scalability (less Must information).

In [13], we present our transfer function over a more complete language of assignments. Our language includes a generic *Other* statement that represents an arbitrary computation decorated with Mod and Ref sets. Mod sets include pointer expressions whose location may be updated

by an assignment, whereas Ref sets include pointer expressions whose location may be accessed by an assignment. We use the *Other* case to conservatively handle arithmetic and other complex operations that are difficult to process accurately.

The key insight of the transfer function presented in this section is that we can use certain mathematical properties of conservative flow-insensitive memory alias analyses to create a concise representation for value alias sets.

4.8 Extension: MustNot sets

A weakness of the implicit representation described above is that once a complex pointer expression is added to the May set, it cannot be removed, even if it is assigned something other than the tracked value. This is because the expression implicitly represents all of its memory aliases. We can overcome this weakness by introducing a third component, the MustNot set. Intuitively, any expression from the May component that is killed is added to the MustNot set, but also retained in the May set. The rule for lookup is modified as follows: If the expression is in the Must set, it holds the value. If it is in the MustNot set, it does not hold the value. Otherwise, if it is in the memory alias closure of the May set, it may hold the value.

5. EXPERIMENTAL RESULTS

In this section, we describe an end-to-end experiment in which we used value flow simulation in a client software validation tool, ESP. We provide an overview of ESP, explain the property we validated on the Windows operating system kernel, and provide data that sheds some light on the precision and efficiency of value flow simulation.

An alternative methodolgy for evaluating our analysis would be to run it on several benchmarks and gather some statistics that may shed light on its potential impact on a variety of clients. It is not clear that such results would be meaningful. Therefore, we focus on an end-to-end experiment with a client application.

5.1 ESP

ESP is a software validation tool for C/C++ programs [10]. It is run multiple times a week over tens of millions of lines of real, shipped, operating systems code. ESP takes as input a set of C/C++ files and a declarative specification of a correctness property, written as a finite state machine. The property describes how values of interest are created by the program, and how these values are driven to different states during program execution.

ESP uses value flow simulation to track one created value at a time, computing value alias sets at every program point. It uses the transfer function given in Figure 8 (with k set to 1), and a flow-insensitive, context-sensitive memory alias analysis (GOLF, [11]). When a state changing event is encountered, ESP updates the state of the state machine if the argument of the event is in the value alias set. If the argument is in the Must set, ESP performs a strong update on the state. If the argument is not in the Must set, but in the May set, ESP performs a weak update on the state. If the state machine is driven to the special error state, ESP displays a trace from the creation point to the error point.

5.2 The Probe security property

Many operating systems reserve a section of system memory for use by kernel mode components. The kernel provides a set of API functions that can be called from a user program, to perform various tasks. Some of these tasks require the kernel to read data from, or store data into, locations referenced by the pointer values passed in as arguments.

One possible source of security attacks on these systems is that a user program may pass in pointers that point into the reserved section, thereby duping the kernel into exposing or over-writing system data. To prevent this form of attack, kernel code is written with the guideline that all user mode pointers must be "probed", to ensure that they do not point into kernel memory, before they are dereferenced. A version of this finite-state property is shown in Figure 9.

Over the years, the Windows kernel has been heavily reviewed to ensure that this kind of security attack is not possible. In the experiment reported here, we used ESP to ensure that there are no remaining dereferences of unprobed parameters in the kernel.

We chose the Probe property for the following reasons: First, this property is a special case of a more general situation, where "tainted" data from an untrusted source should not be passed to components that expect data to be "untainted" (for instance, see [3]). A well defined set of validation operations move data from the tainted state to the untainted state. Second, this property involves complex value flow as pointers are passed through the code; therefore, it serves as a good test for value flow simulation.

5.2.1 Impact of value flow analysis

The accuracy, efficiency and usability of ESP on the Probe property depend heavily on the value flow analysis engine.

Accuracy. If the argument to the probe operation is not present in the Must set, ESP will not be able to perform a strong update to the `Safe` state. This will lead to false error reports at dereference points. If unrelated pointers are pulled into the May set, ESP will report false errors when those pointers are dereferenced.

Efficiency. If bit-vectorization does not cut down the span of the program on which each value is tracked, ESP will explore too many program points. If value flow simulation creates too many different value alias sets per program point, the performance of ESP will degrade in both space and time.

Usability. The usability of ESP depends on the quality of debugging information provided to the user when an error is reported. An important reason for including value flow simulation in ESP is that the path-sensitivity of the analysis allows ESP to provide error traces that are very much like execution traces.

When an error is detected, ESP walks the predecessor edges in the coordinate space from the error point. The simulation state is displayed at every coordinate along the path. For every summary edge (from a *CallReturn* node to its *Call* node) encountered along the way, a sub-trace is produced by recursively repeating the walk in the coordinate space of the applicable clone of the callee.

Value flow simulation provides only partial path-sensitivity. Therefore, the coordinate space may contain "diamonds"

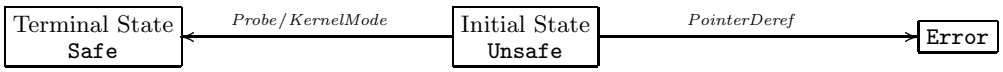

Figure 9: The Probe security property. `Unsafe` is the initial state at every creation point corresponding to a formal parameter of a kernel API function. `Safe` is a terminal state, meaning that the analysis of a parameter can stop, either because the parameter has been probed, or because the API function was invoked from other kernel-mode code.

from branches whose effects were merged away. The backwards walk skips over these diamonds by jumping directly to dominators.

5.3 Measurements

The Windows kernel we analyzed consists of roughly a million lines of code, spread over roughly 9000 functions. The kernel exposes an interface of several hundred callable functions, with roughly 500 pointer parameters. Of these parameters, ESP reported errors on 30 pointers, and validated the rest. All of the reported errors were false positives. The precise error traces made it relatively easy for programmers to examine and rule out false errors. Most of the false errors occur because our implementation is unable to put some expressions with complex selectors, such as `x->f.g.h.j`, in the Must set. We are currently working on improving our implementation.

In order to understand the precision benefit of value alias sets, we also ran ESP with only May sets. The number of false errors increased by a factor of 4, and the error traces quickly became inscrutable, both of which made the the tool unusable.

Performance. For one third of the user pointers, ESP visited just a few functions. For the remaining pointers, ESP visited roughly 4,500 functions (half the program), on average. Because of bit vectorization, ESP is embarrassingly parallel. We used 10 desktop machines to cut down the analysis time by a factor of 10.

On average, ESP visited 50 nodes per function, with 1 value alias set per node. However, there were nodes at which multiple value alias sets were produced, and it was important to keep them apart. Each node was visited roughly 2.6 times, as the simulation state moved up to the fixed point. On a 3.06GHz Intel Xeon™ with 1GB RAM, ESP took 8 minutes to process each pointer that propagated to 4,500 functions. The entire run completed in 11 hours.

The data above suggests that value flow simulation represents a fair trade-off between precision and performance. The results are dependent on the particular property, code base, and client tool we have used. This would be true of any end-to-end experiment.

6. RELATED WORK

The main contribution of this paper is a new value flow analysis that draws on several insights from previous work, and combines these insights in a novel way in order to solve an important problem. Therefore, there are several categories of related work.

6.1 Software validation tools

There has been a resurgence of research in recent years on software validation via static analysis. Most software validation tools rely on some form of value flow analysis. Local analysis tools such as ESC/Java [14] or Fugue [12] do not require scalable analysis but do require annotations on function boundaries. Global analysis tools such as SLAM [4], ESP [10], or CQual [15] would benefit from scalable value flow analysis. PREfix [7] and xgcc [17] are global tools that scale well, because they truncate the search space in an unsound manner.

6.2 Value flow analysis

Value flow analysis has been studied in many areas.

Reaching definitions. In the dataflow analysis community, local value flow analysis has been studied as reaching definitions or def-use chains for years [1]. Systems such as CodeSurfer (slicing, [21]) include methods for global reaching definitions based on flow-sensitive analysis. Bodik and Anik, among others, have proposed methods for path-sensitive reaching definitions [5]. None of these methods appears to have the combination of scalability and precision necessary for use on large, real programs.

Flow analysis. The functional programming community has studied flow-insensitive value flow analysis under the name of flow analysis [22] and its higher-order variant, closure analysis [29]. Flow analyses generally scale well, but lack strong update. Therefore, they are not precise enough for use in imperative programs.

One contribution of this paper is the combination of flow-sensitive reaching definitions with scalable memory alias analyses based on flow analysis.

Strongest post-conditions. Symbolic evaluators such as PREfix [7] perform path-sensitive value flow analysis by computing strongest post-conditions. These methods do not scale well; PREfix scales by truncating the number of execution paths in an unsound manner. ESC/Java performs value flow analysis as part of a weakest precondition computation [14]. It scales by limiting the analysis to local, non-loop code.

6.3 Basis for value flow simulation

Value flow simulation is a framework for efficient interprocedural, context-sensitive, path-sensitive analysis.

Context-sensitive analysis. The context-sensitive aspect of value flow simulation is based heavily on the inter-procedural analysis of Reps et al [26]. We have extended their approach to handle aliasing when moving between different scopes at function call boundaries, and to handle allocators in a context-sensitive manner.

Path-sensitive analysis. The path-sensitive aspect of value flow simulation is based heavily on the merge-based analysis of Das et al [10], which provides an efficient method for com-

puting states in a finite state machine. We have shown that the same approach holds promise for value flow problems as well.

6.4 Memory alias analysis

Our approach relies on the availability of scalable methods for memory alias analysis. In recent years, many such methods have been devised [9, 18]. Our implementation uses GOLF, a context-sensitive, flow-insensitive analysis that scales easily to millions of lines of code [11]. Methods for flow-sensitive alias analysis [31, 8] are much more expensive, and their precision advantage over flow-insensitive methods is open to debate [20].

Using memory alias analysis in other analyses. With the invention of memory alias analyses, well known analyses such as Mod/Ref analysis and program slicing have been extended to handle programs with pointers. Early solutions involved using an alias analysis to replace a pointer expression with all possible targets [28]. This approach suffers from the problem that the less precise the alias analysis, the more inefficient the client analysis.

A popular approach in recent work [15] has been to use unification-based alias analyses such as [30], which map all aliased expressions to a single representative. The result is an implicit representation for free. Our experience is that this form of analysis is too imprecise for use in practical software validation tools.

One contribution of this paper is an implicit representation of memory aliases in value alias sets that supports any flow-insensitive alias analysis. We were able to develop this representation by observing some simple mathematical properties of flow-insensitive alias analysis.

6.5 Strong updates on heap objects

Value alias sets enable strong updates on pointer expressions in the absence of a shape or heap analysis. Recently, Foster et al have provided a flow-insensitive type inference procedure for inferring that some expressions in a procedure are not aliased, allowing for strong updates in certain cases [2]. Our approach is path-sensitive, because of which it is able to perform strong updates in more cases.

7. CONCLUSIONS

In this paper, we have presented value flow simulation, a novel method for context-sensitive, path-sensitive value flow analysis. This algorithm incorporates a transfer function that combines precise value flow computation with scalable memory alias analysis in an efficient manner.

We have used value flow simulation to validate the Windows operating system kernel against an important security property. Our experience suggests that the analysis is both precise enough and scalable enough to serve as an engine for software validation tools.

Acknowledgements

Roman Manevich helped formalize the transfer function for value alias sets. Brian Hackett and Manuel Fahndrich provided valuable feedback on previous drafts. Seth Hallem, Muthu Jagannathan, and Jeff Wallace helped implement parts of ESP. The PREfast team at Microsoft Research provided the compiler front-end for ESP.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 129–140, 2003.

[3] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN '01, 8th Annual SPIN Workshop on Model Checking of Software*, May 2001.

[5] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *Conference Record of the Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 237–251, 1998.

[6] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 55–65, June 2003.

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7):775–802, 2000.

[8] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[9] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[10] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[11] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *8th International Symposium on Static Analysis*, 2001.

[12] R. Deline and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[13] N. Dor, S. Adams, M. Das, and Z. Yang. Path sensitive value flow analysis on large programs. Technical Report MSR-TR-2003-58, Microsoft Research, 2003.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[15] 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, 2002.

[16] D. Gries. *The Science of Programming*. Springer-Verlag, 1987.

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[18] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[19] 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 Engineering (PASTE'01)*, pages 54–61, 2001.

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

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[22] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 244–256, 1979.

[23] U. P. Khedkar and D. M. Dhamdhere. A generalised theory of bit vector data flow analysis. *ACM Trans. Program. Lang. Syst.*, 16(5):1472–1511, 1994.

[24] J. Knoop and B. Steffen. Efficient and optimal bit-vector dataflow analyses: A uniform interprocedural framework. Technical Report Bericht Nr. 9309, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Germany, 1993.

[25] A. Milanova, A. Rountev, and B. G. Ryder. Precise and Efficient Call Graph Construction for C Programs with Function Pointers. *Journal of Automated Software Engineering*, 2004. To appear.

[26] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural data flow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, 1995.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages*, 1999.

[28] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *LNCS 1302, 4th International Symposium on Static Analysis*, Sept. 1997.

[29] O. Shivers. *Control-Flow Analysis Of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[30] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, 1996.

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