Computing Must and May Alias to Detect Null Pointer Dereference*

Xiaodong Ma, Ji Wang, and Wei Dong

National Laboratory for Parallel and Distributed Processing, P.R. China
{xd.ma,wj,wdong}@nudt.edu.cn

Abstract. This paper presents a novel algorithm to detect null pointer dereference errors. The algorithm utilizes both of the must and may alias information in a compact way to improve the precision of the detection. Using may alias information obtained by a fast flow- and context- insensitive analysis algorithm, we compute the must alias generated by the assignment statements and the must alias information is also used to improve the precision of the may alias. We can strong update more expressions using the must alias information, which will reduce the false positives of the detection for null pointer dereference. We have implemented our algorithm in the SUIF2 compiler infrastructure and the experiments results are as expected.

1 Introduction

Null pointer dereference is a kind of common errors in programs written in C. If a pointer expression (including a pointer variable) which points to NULL is dereferenced, the program will fail. If a pointer expression is uninitialized or freed, we call it an invalid pointer expression. Dereferencing an invalid pointer expression may not crash the program, but will get a wrong datum. Therefore, dereferencing a NULL pointer or an invalid pointer are regarded as null pointer dereference errors in this paper.

Alias information is needed to detect the null pointer dereference error. For example, dereferencing $*e$ after statement $e = NULL$ or $free(e)$ will cause an error. It should be noticed that dereferencing any expression which may be alias of $*e$ also possibly causes a null pointer dereference error. Thus a conservative algorithm needs the *may alias* information. Of course, we know that although a conservative algorithm does not miss any real error, it may produce many false alarms. This paper makes attempt to find much information to improve the precision of static analysis for null pointer dereference. Statement $e = malloc()$ assigns e with a non-NULL value, thus e can be dereferenced and $*e$ can be written after this statement under the condition that the l-value of e is not changed. If e' is the alias of e before every possible execution of this statement

* This work is supported by National Natural Science Foundation of China(60725206, 60673118 and 90612009), National 863 project of China(2006AA01Z429), Program for New Century Excellent Talents in University under grant No. NCET-04-0996.

T. Margaria and B. Steffen (Eds.): ISoLA 2008, CCIS 17, pp. 252–261, 2008.

and its l-value cannot be changed, then we can say that dereferencing e' or write $*e'$ is also a valid operation. If we do not have the *must alias* information, we may report errors, which could be false alarms actually.

Figure 1 illustrates the usage of must and may alias information. Statement 3 will not cause error because $*z$ is the must alias of $*y$. Statement 7 may cause error because y is the may alias of $*x$ and $*x$ has been nulled at statement 6. It is clear that must alias information can make the detection algorithm more precise. However, the must alias information has not been well used in the existing error finding techniques.

```
int **x, *y, *z;
...
1. y = malloc();
2. z = y;
3. *z = 5;
4. if(...)
5.   x = &y;
6. *x = NULL;
7. *y = 5;
```

Fig. 1. Usage of must and may alias

There are some null pointer dereference detection tools, such as [13], [2] and [9]. But they have not exploited the must alias information. There is a little work about computing and exploiting the must alias information in C programs. To the best of our knowledge, the only work is [1]. It defines an extended form of must alias and uses the result to improve the precision of def-use computation. In this paper, our algorithm computes the must and may alias information of a set of k-limiting expressions [7] from the result of a fast flow- and context- insensitive alias analysis algorithm and the must alias information is used to detect the null pointer dereference errors. A tool prototype has been implemented and the initial experimental results are as expected.

This paper is organized as follows. We first introduce the points-to graph and the method for computing l-values of expressions in Section 2, then the details of our must alias computation algorithm in Section 3 and null pointer dereference detection algorithm in Section 4. Section 5 gives the experimental results. The related work and conclusions are in Section 6.

2 Points-to Graph

We suppose the result of the flow- and context- insensitive alias analysis algorithm is a points-to graph. In this section, we introduce the definition of points-to graph and describe how to compute the l-value of an expression in a points-to graph. Then we introduce the concept of may alias and define the must alias used in this paper.

A *location set* is an abstraction of the memories. It is a pair of the form $(name, offset)$ where $name$ describes a memory block and $offset$ is the offset

within that block. Notice that for a variable we just use its name as the *name* and for the dynamically allocated heaps we use the allocation-site abstraction. We consider an array as a variable with the type of its elements.

A *points-to graph* is a directed graph with the location sets as nodes. There is a directed edge from one node to another if one of the memory locations represented by the first node may point to one of the memory locations represented by the second one.

Figure 2 is a segment of C code and its corresponding flow-insensitive points-to graph. *heap*_1 denotes all the memory locations allocated at line 1. *heap*_2.32 is the "next" field of the structure allocated at line 2. Different fields of a structure are represented by different location sets.

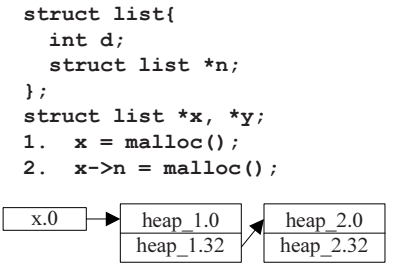

Fig. 2. An example segment of C code and the points-to graphs

Given a points-to graph, we can compute the l-value of an expression, which is a set of the possible location sets used to store its value. Besides all the location sets occurring in the points-to graph, we introduce another kind of location set called "virtual location set". It is in the form of "&l" where l is a location set. We define the l-value of an expression e on the points-to graph G by a function $ll(e, G)$ as following.

$ll(const, G) = \emptyset \quad ll(null, G) = \emptyset \quad ll(x, G) = \{x.0\}$
$ll(\&e, G) = \{\&l \mid l \in ll(e, G)\} \quad ll(e.f, G) = \{n.f \mid n.0 \in ll(e, G)\}$
$ll(*e, G) = \quad \{l' \mid \&l' \in ll(e, G) \text{ or } ((l, l') \in G, l \in ll(e, G))\}$
$ll(e \rightarrow f, G) = \{n.f \mid (l, n.0) \in G, l \in ll(e, G)\}$

May alias is discussed widely, such as [6], [8], [10] and [12]. Alias information is computed at the control flow graph vertex, that is, before the execution of each statement–we call it program point. It is obvious that there may be more than one path through each program point. At a program point, if the l-value of expression e_1 and e_2 may be the same, then e_1 and e_2 have may alias relationship. It is possible that the l-value of e_1 in some execution path is the same as that of e_2 in another execution path. In this case, e_1 and e_2 may not be alias actually.

In order to make the definition of must alias clear, we use the postfix form of an expression e, which is defined by a function $pf(e)$ in the following, where x is a variable.

$pf(const) = const \quad pf(x) = x \quad pf(\&e) = e\& \quad pf(*e) = e* \quad pf(e.f) = e.f$
$pf(e{\rightarrow}n) = e*.n$

We say that e_1 is the must alias of e_2 at a program point p if e_1 and e_2 have the same l-value at every possible execution of p. But in some cases, the l-value of e_1 or e_2 may not be defined. For example, $x{\rightarrow}n{\rightarrow}n$ does not have l-value before the execution of statement 2 in Figure 2.

Let $e_1 = e_1'\omega$, $e_2 = e_2'\omega$, where ω can be empty. If e_1' and e_2' have the same l-value, then we think e_1 and e_2 have must alias relationship.

3 Computing Must Alias

3.1 Must Alias Data Flow Fact

Based on a flow- and context- insensitive may alias analysis, we compute the fixpoint of data flow fact of must alias at the program point before each statement. The must alias relation is an equivalence relation, that is, it is reflexive, symmetric and transitive. For example, if e_1 is the must alias of e_2 and e_2 is the must alias of e_3, then e_1 is the must alias of e_3. May alias relation is also reflexive, symmetric, but not transitive, because two may alias pairs may be generated in different execution paths. Supposing r is a must alias relation on the expression set E, we can get $E\backslash r = \{C_r^1, C_r^2, ..., C_r^n\}$.

The data flow fact of must alias used in our algorithm is a tuple (r, M) where r is a must alias relation on E and M is a map from the equivalence class with respect to r to the location sets in the points-to graph. In other words, $M(C_r^i)$ denotes all the possible l-values of the expressions in C_r^i.

Let r_1, r_2 be must alias relation on E, we define $r_1 \preceq r_2$ if and only if $\forall e_1 \in E, \forall e_2 \in E, < e_1, e_2 > \in r_1 \Rightarrow < e_1, e_2 > \in r_2$. Thus the partial order of the data flow fact is defined as $(r_1, M_1) \sqsubseteq (r_2, M_2)$ if and only if $r_1 \preceq r_2$ and $\forall e \in E$, $M_1([e]_{r_1}) \supseteq M_2([e]_{r_2})$. Two special elements are also defined: the top element $\top$ and the bottom element $\bot$. For any data flow fact d, we have $d \sqsubseteq \top$ and $\bot \sqsubseteq d$.

May alias information can be deduced from the data flow fact. If $M([e_1]_r) \cap M([e_2]_r) \neq \emptyset$, then we say that $\forall e_x \in [e_1]_r$, $\forall e_y \in [e_2]_r$, e_x and e_y have the may alias relation. If two data flow facts d_1 and d_2 satisfy $d_1 \sqsubseteq d_2$, then the must alias pairs in d_2 is a superset of that in d_1. Because the possible l-values of each expression in d_1 is also a superset of that in d_2, it is easy to prove that the may alias pairs deduced from d_2 is a subset of that from d_1. d_2 has more must alias pairs and less may alias pairs than that of d_1, thus d_2 is more precise than d_1.

We use data flow analysis to compute the data flow fact at each program point. Initially, the data flow fact at each program point is $\bot$. "join" operation "$\sqcup$" is defined to compute the fixpoint of the data flow fact at each program point. In order to define the join operation of the data flow fact, we define that of the equivalence relation. Join operation $\vee$ of two equivalence relations r_1 and r_2 is defined as the transive closure of the union of r_1 and r_2, that is, $r_1 \vee r_2 = closure(r_1 \cup r_2)$. Thus $r_1 \vee r_2$ is also an equivalence relation.

The join operation of data flow fact is defined as:

$(r_1, M_1) \sqcup (r_2, M_2) = (r_1 \vee r_2, M')$, M' satisfies $M'(C^i_{r_1 \vee r_2}) = (M_1([e_1]_{r_1}) \cap M_2([e_1]_{r_2})) \cap ... \cap (M_1([e_n]_{r_1}) \cap M_2([e_n]_{r_2}))$ where $C^i_{r_1 \vee r_2} = \{e_1, ..., e_n\}$.

Of course, for any data flow fact d, we have $\top \sqcup d = \top$, $\bot \sqcup d = d$.

3.2 Must Alias Analysis

In this subsection, we will show the effect of statements on data flow facts, that is, how a statement produces a new data flow fact from an input data flow fact.

Some auxiliary functions need to be defined first.

$deref(e) = stars(e) - addr(e)$. Where $stars(e)$ is the number of character '*' in e and $addr(e)$ is the number of character '&'. The result of $deref(e)$ is the dereference depth of e. It is easily to know that $\forall e \in E, deref(e) \geq -1$.

$may(e, (r, M)) = \{e' | M([e']_r) \cap M([e]_r) \neq \emptyset, e' \in E\}$ is the set of expressions which are the may aliases of e.

$lchg(e, (r, M)) = \{e'\omega | e' \in may(e, (r, M)), deref(ew) > deref(e), e'w \in E\}$. The result of this function is all the expressions whose l-value may be changed to different location sets by a statement which assigns a value to e.

We use a transfer function to define the effect of a statement. Three kinds of statements are considered: the allocation statement $e = malloc()$; the free statement $free(e)$ and the assignment statement $e_0 = e_1$. Note that we think $free(e)$ has the same effect as that of $e = NULL$, so e_1 in the assignment statement is supposed not to be NULL.

The transfer function for $e = malloc()$.

$[e = malloc()](r, M) = (r', M')$ where r' satisfies the following condition.

(1) $\forall e_0 \notin lchg(e, (r, M))$, $\forall e_1 \notin lchg(e, (r, M))$: $< e_0, e_1 > \in r \Rightarrow < e_0, e_1 > \in r'$;

(2) $\forall e_0 \in may(e, (r, M)), e_0 \notin lchg(e, (r, M))$ and $\forall e_1 \in may(e, (r, M)), e_1 \notin lchg(e, (r, M))$, $< e_0, e_1 > \in r \Rightarrow < e_0\omega, e_1\omega > \in r'$ whenever $deref(e_0\omega) > deref(e_0)$ and $e_0\omega \in E$, $e_1\omega \in E$.

(3) there are no more relation pairs in r' other than that generated by rules (1) and (2).

It can be proved that r' is also an equivalence relation.

M' is defined on the equivalence class with respect to r'.

$M'([e_x]_{r'}) = M([e_x]_r)$ if $[e_x]_{r'} \cap lchg(e, (r, M)) = \emptyset$

For the equivalence class which contains expression whose l-value may be changed, we write it in the form of $[e_y\omega]_{r'}$ where $e_y \notin lchg(e, (r, M))$ and $e_y \in may(e, (r, M))$. The possible l-value of this set of expressions is defined as:

$$M'([e_y\omega]_{r'}) = \begin{cases} heap\_i.f_x & : \text{ if } e\omega \in [e_y\omega]_{r'} \text{ and } deref(e_y\omega) = deref(e_y) + 1 \\ heap\_i.f_x \cup M([e_y\omega]_r) & : \text{ if } e\omega \notin [e_y\omega]_{r'} \text{ and } deref(e_y\omega) = deref(e_y) + 1 \\ \emptyset & : \text{ if } e\omega \in [e_y\omega]_{r'} \text{ and } deref(e_y\omega) > deref(e_y) + 1 \\ M([e_y\omega]_r) & : \text{ if } e\omega \notin [e_y\omega]_{r'} \text{ and } deref(e_y\omega) > deref(e_y) + 1 \end{cases}$$

$heap\_i$ is the abstract heap allocated at the current statement. The suffix f_x depends on the types of expression $e_y\omega$. If the l-value of an equivalence class is allocated definitely at the current statement, then we use $heap\_i.f_x$ as its l-value, else we add $heap\_i.f_x$ to the original l-value set to make our computation of l-value conservative. It is clear that the currently allocated heap cannot be dereferenced-which explains why $\emptyset$ occurs in the definition.

The transfer function for $free(e)$.

$[free(e)](r, M) = (r', M')$.

Because we regard the effect of $free(e)$ as assigning NULL to e, the rules for generating r' are the same as that of statement $e = malloc()$.

The definition of M' is divided into two cases.

For the equivalence class $[e_x]_{r'}$ which does not contain any expression in $lchg(e, (r, M))$, we get $M'([e_x]_{r'}) = M([e_x]_r)$.

For the equivalence classes which contains expression in $lchg(e, (r, M))$, we can write it as $[e_y\omega]_{r'}$ where $e_y \in may(e, (r, M))$ and $e_y \notin lchg(e, (r, M))$.

$$M'([e_y\omega]_{r'}) = \begin{cases} \emptyset & : \text{ if } e\omega \in [e_y\omega]_{r'} \text{ and } deref(e_y\omega) > deref(e_y) \\ M([e_y\omega]_r) & : \text{ if } e\omega \notin [e_y\omega]_{r'} \text{ and } deref(e_y\omega) > deref(e_y) \end{cases}$$

The transfer function for $e_0 = e_1$.

$[e_0 = e_1](r, M) = (r', M')$.

The effect of this statement can be divided into two parts: it first destroys the old value of e_0 and then assigns a new value to it. In other words, we replace it with two statements: $e_0 = NULL; e_0 = e_1$. The generation of r' can also be divided into two steps: the generation of equivalence relation r'' after the execution of $e_0 = NULL$ and that of r' after $e_0 = e_1$. We can get r'' by applying the same rules as that for the $e = malloc()$ statement. The generation of r' is defined in the following.

$$r' = \begin{cases} closure(r'' \cup \{< e_0\omega, e_1\omega > | \; deref(e_0\omega) > deref(e_0)\}) & : \text{if (1)} \\ r'' & : \text{else} \end{cases}$$

where $(1) \equiv e_0 \notin lchg(e_0, (r, M)), e_1 \notin lchg(e_0, (r, M))$.

The definition of M' is as following.

$M'([e_x]_{r'}) = M([e_x]_r)$ if $[e_x]_{r'} \cap lchg(e_0, (r, M)) = \emptyset$.

For the equivalence class $[e_y\omega]_{r'}$ which satisfies $[e_y\omega]_{r'} \cap lchg(e_0, (r, M)) \neq \emptyset$, $e_y \in may(e_0, (r, M))$ and $e_y \notin lchg(e_0, (r, M))$, we can have:

$$M'([e_y\omega]_{r'}) = \begin{cases} M([e_1\omega]_r) & : \text{ if (2)} \\ M([e_1\omega]_r) \cup M([e_y\omega]_r) & : \text{ if (3)} \\ bottom & : \text{ else} \end{cases}$$

In the above definition, $(2) \equiv e_1 \notin lchg(e_0, (r, M)) \wedge e_0\omega \in [e_y\omega]_{r'} \wedge \forall\omega' : deref(e_1) < deref(e_1\omega') < deref(e_1\omega) \Rightarrow e_1\omega' \notin lchg(e_0, (r, M))$. If (2) is

satisfied, we are sure that the l-value of $e_y\omega$ is changed and the corresponding l-value of the source equivalence class is not changed. Thus we can replace the l-value of $[e_y\omega]_{r'}$ with that of $[e_1\omega]_r$. (3) $\equiv e_1 \notin lchg(e_0,(r,M)) \wedge e_0\omega \notin [e_y\omega]_{r'} \wedge \forall\omega' : deref(e_1) < deref(e_1\omega') < deref(e_1\omega) \Rightarrow e_1\omega' \notin lchg(e_0,(r,M))$. In this case, the l-value of the source equivalence class is not changed, too. But we are not sure whether $[e_y\omega]_r$ is the target equivalence class. So we use the union operation to make our algorithm conservative. In the third case, *bottom* denotes all the possible location sets of $e_y\omega$. This value is used because the l-value of the source class may be changed and the target class may not be assigned with the l-value of the source class.

In the inter-procedural analysis phase, we add a sequence of statements which assign the formal parameters with the corresponding real arguments before stepping into the called procedure. After exiting from the called procedure p, we clean the must information of the local expressions of p. That is, there is no expression which is the must alias of a local expression except itself.

4 Null Pointer Dereference Detection

The null pointer dereference detection algorithm which will be presented is based on the results of the must alias analysis. The strong updates derived from the must alias information can make the detection algorithm more precise. In this section, we suppose the data flow fact of must alias information has already been computed at the program point before each statement.

In order to detect null dereference error, we use a data flow fact to describe the allocation information. It is a function $A : E \rightarrow \{true, false\}$. $A(e) = true$ denotes that the pointer expression e points to a valid memory location. $A(e) = false$ means that e points to $NULL$ or other invalid memory location, such as an uninitialized or freed one.

As in the computing of must alias, there is a function describing the allocation information at each program point. We compute its fixpoint. The top value of the allocation information $A_\top$ is defined as: $\forall e \in E(A_\top(e) = true)$ and the bottom $\forall e \in E(A_\bot(e) = false)$. $A_1 \sqsubseteq A_2$ iff $\forall e \in E(A_1(e) = false \vee A_2(e) = true)$. $A_1 \sqcup A_2 = A'$ where $\forall e \in E(A'(e) = A_1(e) \vee A_2(e))$. The initial allocation information at each program point is $A_\bot$.

Each type of statements can be regarded as a transfer function which takes a must alias data flow fact (r, M) and an allocation data flow fact A as inputs and outputs allocation data flow fact A'.

The transfer function for $e = malloc()$.
$[e = malloc()]_A((r,M),A) = A'$ where:

$$A'(e_y) = \begin{cases} false : & \text{if } e_y \in lchg(e,(r,M)) \\ true \ : & \text{if } e_y \notin lchg(e,(r,M)) \text{ and } < e_y, e > \in r \\ A(e_y) : & \text{else} \end{cases}$$

The transfer function for $free(e)$.
$[free(e)]_A((r, M), A) = A'$ where:

$$A'(e_y) = \begin{cases} false: & \text{if } e_y \in may(e,(r,M)) \text{ or } e_y \in lchg(e,(r,M)) \\ A(e_y): & \text{else} \end{cases}$$

The transfer function for $e_0 = e_1$.
$[e_0 = e_1]_A((r, M), A) = A'$.

The definition of A' is divided into two parts. The first is for the expressions which are not in $lchg(e_0, (r, M))$ and the second is for the other expressions. We also rewrite expressions in the second part in the form of $e_y\omega$ where $e_y \in may(e_0, (r, M))$ and $e_y \notin lchg(e_0, (r, M))$.

$A'(e_x) = A(e_x)$ if $e_x \notin lchg(e_0, (r, M))$.

$$A'(e_y\omega) = \begin{cases} A(e_1\omega) & : \text{ if (2)} \\ A(e_1\omega) \wedge A(e_y\omega) & : \text{ if (3)} \\ false & : \text{ else} \end{cases}$$

The inter-procedural analysis is in the similar way as that in the must alias analysis. A sequence of statements which assign real arguments to the corresponding formal parameters are inserted at the entry of the called procedure. $A(e)$ is assigned with $false$ after exiting from procedure p if e is the local expression of p.

Using the allocation information, we can decide whether an expression causes null pointer dereference. If an expression $e\omega$ is read or written and the following conditions are satisfied: (1) $A(e) = false$; (2) $deref(e\omega) > deref(e)$, then we say that a null pointer dereference may occur.

5 Experiment

We have implemented the prototype of our algorithm in the SUIF2 compiler infrastructure and evaluated it with the test cases from samate [11]. The description of the test cases and the results of our experiment are listed in Table 1.

NPD stands for "Null Pointer Dereference" in Table 1. From the results of the experiment, we can see that our method has a good precision. It should be

Table 1. Experiment with the test cases from samate

Case IDs	Reports	Bugs	Description
1760	1	1	Ordinary NPD.
1875	1	1	NPD through array element.
1876	1	1	NPD through array element in branch condition.
1877	1	1	NPD within branch of switch statement.
1879	1	1	NPD caused inter-procedurally.
1880	0	0	Dereferencing inter-procedurally without NPD.
1934	0	0	NPD within unreachable branch of if statement.

noticed that in the program No. 1394, there is a dereference of a null pointer in the branch of an if statement, but the condition cannot be satisfied. Our tool uses the allocation information to decide whether some simple condition expressions can be satisfied. For example, if $A(e) = true$, then we can know the value of $e \neq NULL$ is true.

6 Related Work and Conclusions

Must alias information is very useful for many analysis like constant propagation, register allocation and dependence analysis [3]. However, not much work has been done for must alias analysis [5]. In most cases, it is the side effect of a may alias analysis and is used during the process of may alias analysis in order to improve the precision. [6] defines must alias in an optimistic manner: if during the analysis a pointer only points to one object, then it is treated as a must alias. This definition may miss the must alias information between some expressions which have heap locations in their access path. [1] introduces an extended must alias analysis to handle dynamically allocated locations and this result is used to improved def-use information. CALYSTO [2] can detect null pointer dereference errors. It embraces the ESC/Java [4] philosophy of combining the ease of use of static checking with the powerful analysis of formal verification. It is fully automatic, performing inter-procedural analysis. PSE [9] is also a null pointer dereference detection tool. It tracks the flow of a single value of interest from the point in the program where the failure occurred back to the point in the program where the value may have originated. In other words, it can work in a demand-driven fashion.

In this paper, we propose a novel must alias analysis algorithm. Using the result of a fast, imprecise may alias analysis, it can compute the must alias relation between complex expressions and improve the precision of the may alias at the same time. Exploiting the must alias information, a precise null pointer dereference detection algorithm is also proposed. In the futrue, we will improve the scalability of our tool and use it to check some real world applications.

References

1. Altucher, R.Z., Landi, W.: An extended form of must alias analysis for dynamic allocation. In: POPL 1995: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 74–84. ACM, New York (1995)
2. Babić, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: Proceedings of 30th International Conference on Software Engineering (ICSE 2008), May 10–18 (2008)
3. Emami, M.: A practical interprocedural alias analysis for an optimizing/parallelizing c compiler. Master's thesis, McGill University (1993)
4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 234–245. ACM, New York (2002)

5. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: PASTE 2001: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 54–61. ACM, New York (2001)
6. Hind, M., Burke, M., Carini, P., Choi, J.-D.: Interprocedural pointer alias analysis. ACM Transactions on Programming Languages and Systems 21(4), 848–894 (1999)
7. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of lisp-like structures. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 244–256. ACM, New York (1979)
8. Liang, D., Harrold, M.J.: Efficient points-to analysis for whole-program analysis. In: ESEC / SIGSOFT FSE, pp. 199–215 (1999)
9. Manevich, R., Sridharan, M., Adams, S., Das, M., Yang, Z.: Pse: Explaining program failures via postmortem static analysis. In: Richard, N. (ed.) Proceedings of the 12th International Symposium on the Foundations of Software Engineering (FSE 2004)November 2004. ACM, New York (2004)
10. Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. In: PLDI 1999: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, pp. 77–90. ACM Press, New York (1999)
11. Samate test cases, `http://samate.nist.gov`
12. Steensgaard, B.: Points-to analysis in almost linear time. In: Symposium on Principles of Programming Languages, pp. 32–41 (1996)
13. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. ACM Trans. Program. Lang. Syst. 29(3), 16 (2007)