Field-sensitive Function Pointer Analysis Using Field Propagation for State Graph Extraction

Bo Huang, Xiang Ling, Guoqing Wu
Computer School of Wuhan University, Wuhan 430072, China
Email: huangbowhu@gmail.com, lingxianglx@gmail.com, wgq@whu.edu.cn

Abstract— **Accurate state graph is important to static program analysis. In order to extract reliable state graphs from programs, which contains function pointers, for model analysis in software engineering, we proposed a field-sensitive may-alias analysis using field propagation, which tries to discover deeply implied aliasing relations by propagating the aliases to variables from their nested structures. For the high complexity of alias analysis, several optimizations are proposed for performance with the price of potentially losing few aliases. Unlike Most previous works which adopted various kinds of approximations that would compromise the reliability of the results, our field-propagation analyzer makes sure that the solutions of queried pointers are sound in a flow-insensitive manner, because none of our adopted optimizations would bring in approximation to the results.**

Index Terms— **Function Pointer Analysis, May-alias, Demand Driven, Field Propagation**

I. INTRODUCTION

In model checking or software verification, state graph extraction, which aims to build state graphs that reflect the control flows of the programs, is needed for those works based on automaton models. There are mainly two methods of call graph extraction: dynamic extraction and static analysis. In dynamic extraction, programs are learnt by run-time study [1], [2], in which the behaviors are sampled and recorded. In static analysis, which is the concentration of our work, the source codes of programs are formally studied to extract the invocation relations in every function call.

In C programs, the key problem of static call graph extraction is the existence of function pointers, which hinders the analyses to determine which functions are actually called. For dynamic binding of function pointers, static analysis needs to scan over the whole program and tries to find all those potential referenced functions. Function pointer analysis is a special case of pointer analysis, which could be classified into two categories: point-to analysis and alias analysis. Point-to analysis concerns about the point-to sets of pointers, which would be propagated and accumulated in every pointer assignment. Alias analysis concerns about the transitivity of values or relation of memory locations between the pointers and/or other variables. Unlike point-to analysis which requires every pointer must points to a memory block explicitly in the analyzing scope, alias analysis can make decision on only pointer aliases, so that it is able to find the relations between pointers and their solutions with incomplete program source code. This kind of characteristics of alias analysis is useful when analyzing those programs which contains third party libraries.

In point-to analyses, Andersens algorithm [3], and Unification-based algorithm [4], [5] are the representative early works in pointer analysis. Andersen proposed a complete solution of program analysis in C language, including a point-to analysis, which contains an inference system based on language semantics. Steensgaard [4] improved the Andersens algorithm to nearly linear time by unifying the locations that pointed by aliasing pointers, so as to reduce distinguishable locations considerably, but losing the precision in the result. Das et al. [5] changed Steensgaards unification by one level unification, and consequently their analyzer could finish analysis with the accuracy of Andersens and the performance of Steensgaards experimentally. Unlike these three analyses that treat different fields of aggregate types in a field-based manner, Pearce et al. [6] proposed a complete solution of field-sensitive inter-procedural analysis for the first time with the discussion of the complexity of their algorithms.

As for alias analysis, Horwitz and Susan [7] point out that in flow-insensitive may alias analysis, it is NP-hard to find out whether two arbitrary expressions may alias. To make alias analysis applicable, we usually need some approximations for a compromise with the performance and precision. Jones et al. [8] proposed a k-limiting method which limits the length of expressions under k, limiting the upper bound size of the expression set under a polynomial scale to the size of the variable set. Deutsch and Alain [9] proposed Symbolic Access Path (SAP), a structure which uses one of the repetition parts and its repeating count to represent a whole expression, so that a short SAP could represent a long expression with repeating sub-structures. But it has no help to those expressions without repeating sub-structure. Milanova et al. [10] proposed a flow-insensitive alias analysis, in which the directional relation alias is regarded as bidirectional relation equivalence. As the result, all the variables could be separated into different disjoint equivalence sets, where pointers could be solved within them. However, irrelative pointers would be put into one

This research is supported by Special Funds for Shenzhen Strategic New Industry Development (JCYJ20120616135936123), National Natural Science Foundation of China (91118003, 61003071), and the Fundamental Research Funds for the Central Universities (3101046, 201121102020006).

doi:10.4304/jsw.8.7.1592-1603

set by the bidirectional transitivity of equivalence relation, leading to imprecisions.

Besides the analyses above, demand-driven analysis is grouped by the demand-driven characteristics. In this kind of analyses, pointer analysis procedures are driven by queries, and during the procedure, the analyzer could skip irrelative parts of program to avoid useless processes and to obtain better performance empirically, without any change to the theoretical complexity of problem. Heintze and Tardieu [11] proposed a demand-driven point-to analysis, which repeatedly applies a sort of rules until the point-to sets of queried variables are built. In these processes, new variables will be created for different fields of structure, and the location relation of the fields and their structures are ignored. Zheng and Rugina [12] proposed a demand-driven alias analysis, in which the alias relation is attached with semantics, converting the May-alias problem into CFL reachability problem. However, it cannot reduce the complexity theoretically. So the authors use some techniques, such as gradual exploration, concurrent exploration, tuning for queries, etc., for a better performance.

Most of the previous works adopted approximations which would compromise the accuracy of the result. This article presents a field-sensitive flow-insensitive function pointer alias analysis, called Field Propagation Analysis. Our analysis is applied in another project to help users construct accurate and reliable state graphs of programs for further studying. In II, Unified Access Path and Acyclic Constraint Graph, are introduced. The former one forms our language of alias analysis, and the latter one is the basic structure of our analysis algorithm. In III, the algorithms of field propagation analysis are entirely illustrated. And in following sections, experiment results are studied and discussed.

II. Analysis Language and Data Structure

A. Unified Access Path

Access Path [9], [13], [14] is a notation of variables, which is usually used in field-sensitive analysis. Every access path consists of a list of variables, reference expressions or components separated by dots. For example, $p \rightarrow field$ in C language is denoted by access path *p.field. In access paths, there exist at least two indirect relations: component relation denoted by dots, and reference relation denoted by asterisks. To make it simpler, a unified access path or UAP notation is introduced in our analysis algorithm to denote different language elements, which include variables, pointers and functions, respectively. In the analysis algorithm, each element is denoted by one or more strings concatenated with one or more dots .. There are 4 types of compositions considered as follows.

- Pointer reference: if variable p is a pointer, then UAP $p.*$ denotes variable $*p$, which stands for the variable that pointed by the pointer p.
- Fields: if b is a field of a structure, UAP $a.b$ denotes $a.b$ in source code, and UAP $p.*.b$ denotes $p \rightarrow b$.
- Function parameters: if a function f has N parameters, then $f.\rho_i$ denotes the i-th parameter of the function f, while $i \in [0, N)$.
- Return Value: if a function f has return value, then $f.\gamma$ denotes the return value of the function f.

Finally, every UAP has a syntax shown in Fig.1.

```
<UAP> ::= ID | ID.<part-list>
<part-list> ::= <component>
              | <part-list>.<component>
<component> ::= ID | * | ρi |γ
```

Figure 1. Syntax of UAPs: ID refers to the name of a variable, field or function. Since components are concatenated with dots, each UAP has a separation set P, where $P(a) = (p, s)|p.s = a$

Especially, $P(a) = \emptyset$ when the UAP a is simply an ID. In each separation $(p, s) \in P(a)$ of UAP a, p is one of prefix UAP of a, and denotes a pointer or a variable of aggregate type; s is one of the suffix fields of a, and represents the field that resides in p. By introducing UAP, all forms of variables can be uniformly denoted, avoiding distinguishing different levels of pointers and normal variables, so that we can analyze pointers with a very simple language as follow.

$$a \supseteq b$$

We use = to represent mutual constraint, such that $a = b \Leftrightarrow a \supseteq b \wedge b \supseteq a$. So the constraint relation is formally reflexive. Considering its transitivity, the UAPs can be stored in a structure of Acyclic Constraint Graph, whose topological ordering could help to sort those UAPs in right order for analysis

B. Acyclic Constraint Graph

Acyclic constraint graph (ACG) is an improved constraint graph by adding a layer facility to sort all the variables into a specific topology.

Common constraint graph is defined as directed graph, 2-tuple (V, E), where V is the set of vertices or variables, and E is the set of directed edges or constraints.

ACG is a 4-tuple (V, X, E, F), where V is the set of variables, the same as its definition in constraint graph; L is a layer mapping that maps each variable to a layer; X is the set of vertices; E is the set of edges, and F is a mapping that helps to find the vertex in which any given variable lies. Formally

$$X \subseteq 2^V \wedge (\forall s, t \in X)(s \cap t = \emptyset) \wedge \bigcup X = V$$
$$F = \{v \mapsto s | s \in X \wedge v \in s\}$$
$$E = \{(s, t) | s, t \in X \wedge (L(s) < L(t))\}$$

The vertices of ACG are sets of variables, in which variables are in the same layer. Edges of ACG must direct from vertices in lower layers to those in higher layers. Fig.2 shows an example of ACG that used in graph based analysis.

An ACG can be transformed into common constraint graph by replacing each vertex by the complete graph of variables. Contrarily, a common constraint graph can be

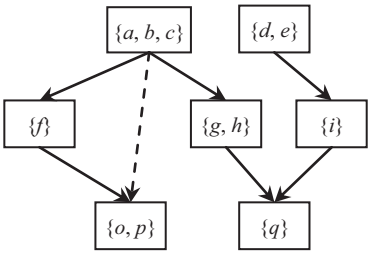

Figure 2. An Example of ACG, the dashed edge means it is implied by transitivity of other edges

transformed into ACG by merging every directed cycle into one vertex, which is also the essential idea of ACG construction.

In fact, ACG is a kind of Directed Acyclic Graph (DAG), whose topological ordering represents a corresponding partially ordered set (or poset). Since each ACG contains two sets, variable set V and vertex set X, and their correspondent relation, however, there must be two posets in every ACG: posets $(X, \supset)$ and $(X, \supseteq)$. ACG sorts variables and vertices by constraints, a kind of partial ordering, leading two effects that help analysis. Firstly, ACG implies the constraint sets of variables within its poset $(V, \supseteq)$. Secondly, we can build a fix point of ACG in a constraint order, which is one of the key points of our analysis algorithm.

ACG can be constructed by constraint insertions. After each common graph edge insertion, cycles can be collapsed by merging all the variables on the cycle into one vertex. From a point-to perspective, all pointers on the same cycle shares the same point-to set [15], [16]; from a graph perspective, constraint cycle could deduct complete component by the transitivity of constraint, leading all variables aliasing to each other. The canonical algorithm of online cycle elimination travels the edges from a specific node to find collapse the cycles, which needs O(V+E) visits, where V and E are the count of vertices and edges of the graph respectively. Considering to the complexity of analysis is measured with variables and constraints, let v stands for the number of variables, and V stands for the number of vertices in ACG; let t stands for the number of constraints, and T stands for the number of edges in ACG. We can easily get that $V \leq v$ and $T \leq t$. So the complexity could be expressed with v and t. The insertion of a constraint to ACG has two steps: cycle detection and cycle decomposition. The former step is essentially a depth first searching with O(t) times visits. And in the later step, all vertices are merged together with O(v) times visits. So eliminating a cycle could finish in O(v+t) times visits.

III. Analysis Framework of Field Propagation

Our analysis framework consists of two major steps: Constraint Extraction and Field Propagation. At first, all assignments are filtered by our analysis module of GCC, from which the original constraints are extracted and stored into a specified text file. And then an individual program, which contains the field propagation algorithm, analyzes the original constraints in the text file, tries to solve all the function pointers in order to generate a complete call graph.

A. Constraint Extraction

Constraint extraction is a mechanism that extracts constraints from various types of the assignments. In C programs, there are at least 4 types of assignment should be taken into consideration:

- Explicit equation assignment: The characteristic of this form is the equation operator =, with two operands: the left hand side and the right hand side. Especially, the right hand side can be an address expression like &x, or reference expression like *x.
- Argument to parameter assignment: In function calls, arguments are passed into functions as parameters, while both arguments and parameters are semantically variables. So there are implicit assignment in every function calls with arguments.
- Return value assignment: This form of assignment can be treated as a special case of equation assignment, in which the right hand side is a function call expression. In GCC, it is another kind of statement distinguished from equation assignment statement.
- Specialized assignment: C programs usually contain memory operations, which semantically act like assignment. For example, the *memcpy(arg1, arg2, arg3)* function call implies an assignment of $*$*arg2* to $*$*arg1*. And the *pthread_create(arg1, arg2, arg3, arg4)*, for another example, will pass *arg3* as the argument to the thread routine pointed by *arg2*. If the source codes of such functions are included in the project, our pointer analysis should correctly deduce the implied constraints, otherwise, our analysis need manual helps to point out their implications.

When the compiler scans through the program, variables and functions are represented by unique UAPs in Constraint Extraction module. And each assignment related statement is parsed into constraint, a form of asymmetric binary relation, written as ordered pair , which means that the constraint set of variable a includes that of b. In summary, the constraint extraction rules are listed in Table.I.

TABLE I.
RULES OF CONSTRAINT EXTRACTION

	Program	Constraint
1	a=b	$a \supseteq b$
2	a=&b	$a.* = b$
3	return a	$f.\gamma = a$
4	f(a_1, a_2, ...)	$f.\rho_1 \supseteq a_1$, $f.\rho_2 \supseteq a_2$, ...

B. Field Propagation

After scanning through a translation unit, all the original constraints form an initial ACG. The originality

means that these constraints are discovered directly by the source code of the program. Next, we will try to find out all the implicit constraints and add them to the ACG to form a complete solution for pointer analysis. An inference system is introduced to help to discover the implied relations between function pointers and their pointed functions. The inference system consists of three rules stated below:

$$[\text{trans}]\frac{a \supseteq b \ \wedge b \supseteq c}{a \supseteq c}[\text{alias}]\frac{a \supseteq b}{a.* = b.*}[\text{field}]\frac{a \supseteq b}{a.\mu \supseteq b.\mu}$$

The transitive rule shows the transitivity of constraint relation on UAP set; the alias rule demonstrates that pointer constraint implies mutual constraints between the values that the pointers point to; and the field rule illustrates the field constraints implied by other constraints. The alias rule can be applied along with constraint extraction; transitive implication lies in the ACG with its reachability, and the field rule is applied in a form of a series of graph operations, called field propagation.

As our analyzer runs in a demand-driven way, before the analysis starts, a set of queried UAPs should be provided. Unlike some demand-driven analyses [11], [12], which could skip some parts of program in demand, however, our analysis still need to scan through the program and analyze every part in constraint extraction. That is why we say our analysis is semi-demand driven. The complexity of constraint extraction is much less than that of field propagation which constitutes the main body of the analysis. So a complete scan makes limited effect to the performance, and does not change the complexity of the whole analysis procedure.

The field rule in the inference system above expands and propagates the field constraints from one UAP to others in order to discover implied constraints. Fig.3 depicts a simple example, in which, the constraints of queried UAP $q.*.pv.*$ is discovered by field propagation.

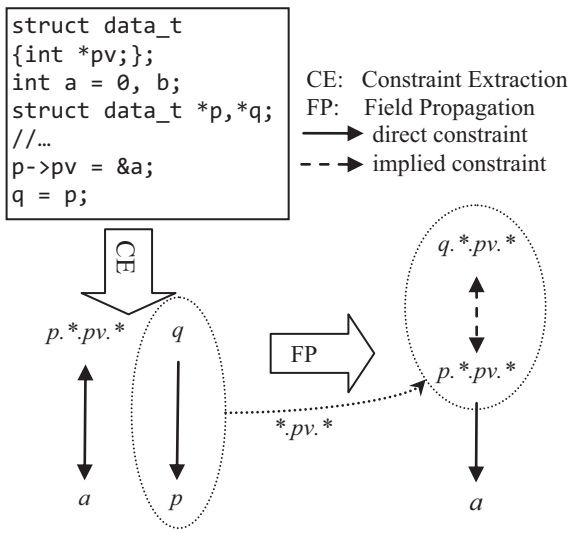

Figure 3. An example of implied constraint

A key issue is then followed, that is how the fields are propagated to other concerning UAPs, making the queried UAPs could reach their constraint variables with the edges of ACG. The value of a UAP, or a variable, depends on not only the UAP itself, but also the prefix UAPs (see introduction of separation of UAP in section 2.1) in which it resides. Fig.4 shows an indirect prefix constraint of this kind. Recursively, the prefix UAPs also have their indirect constraints depending on their prefix UAPs. Formally, L represents the UAP set of a program; A: L2L maps UAPs to their constraint sets, which is defined as

$$\begin{aligned} A(a) &= D(a) \cup I(a) \\ D(a) &= \{b \in L | a \supseteq b\} \\ I(a) &= \{p.s \in L | p \in A(p') \wedge \\ &\quad (p', s) \in P(a') \wedge a' \in D(a)\} \end{aligned}$$

$D(a)$ represents the direct part of the constraint set of UAP a, which is generated in the constraint extraction, and is equivalent to the initial ACG. And $I(a)$ represents the indirect part, which is implied by the rules of transitivity and field implication.

Given an initial ACG G and a UAP a, $D(a)$ represents the sub-graph rooted from a. To demonstrate $I(a)$ with graph, an example is provided in Fig.4, which depicts how the sub-graph rooted by queried UAP $p.*.f$ is generated. The UAPs without square and solid edges are elements contained in initial graph. Those dashed UAPs edges are new elements created in the field propagation for resolving the queried UAP. The doted ellipses and arrows illustrate the effected sub-graphs and direction of field propagation. After three steps of propagation, UAP f and g are included in $A(p.*.f)$.

Since A is recursively defined, the constraint sets need to be generated in a reversed order of the topology of constraint graph, because constraint set of each UAP depends on the ones of successive UAPs. The basic field propagation algorithm is provided in Fig.5.

In the field propagation algorithm, reach(p) represents the set of UAPs and edges in the reachability topology from UAP p, and the set is generated before further calculation. And K is a limit of K-limiting [13] to prevent recursive UAPs from exploding the UAP space. So the Propagate process will definitely stop after propagations of all original UAPs finish, or the UAP set contains all UAPs that isnt longer than K.

To discuss the complexity of the recursive field propagation algorithm above, we should clarify the propagation sequence first. For each queried UAP, all its prefix parts are visited and propagated (in line 7), as well as their successors in ACG. As the result, all UAPs in ACG would be visited in the worst cases. The actual propagation for each separation is an edge traversal from the prefix part (in line 8 through 10), in which the whole sub-graph is copied, with the suffix field appending to all UAPs in the sub-graph. In worst cases, the whole ACG is copied in each propagation process.

Lets denote the UAPs in ACG with v_1, v_2, v_3 vn respectively, and the procedure starts from v_1, but the actual propagation starts from v_n reversely. Suppose there are totally V UAPs in the ACG initially, and let V_i and

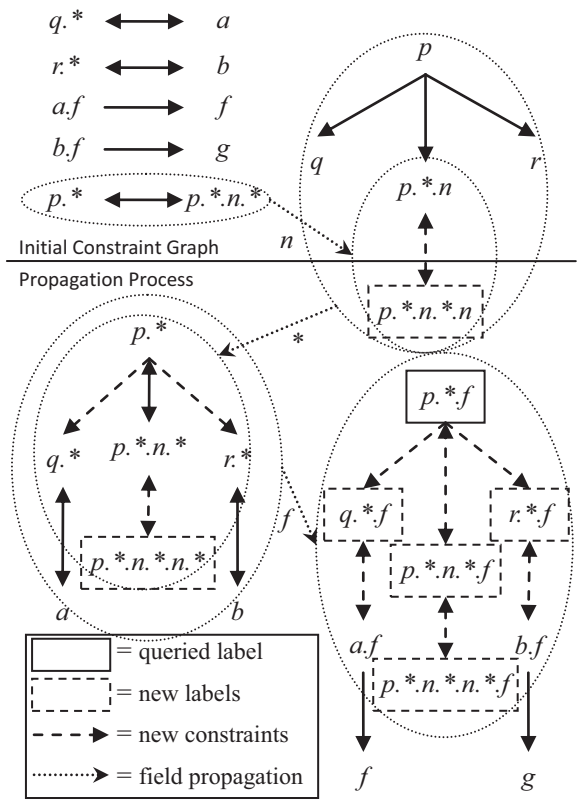

Figure 4. An example process of field propagation

```
1Procedure Propagate (v)
2   If v is marked as propagated yet Then
3       Exit Procedure
4   Mark v as PROPAGATED
5   For each separation (p, s) ∈ P(v)
6       For each node v_p in reach(p)
7           Propagate (v_p)
8       For each edge (v_l, v_r) in reach(p)
9           If |v_l.s| ≤K ∧ |v_r.s| ≤K Then
10              ACGInsert (v_l.s, v_r.s)
```

Figure 5. Basic algorithm of field propagation

T_i represent the number of nodes (or UAPs) and edges (or constraints) in acyclic sub-graph that start from v_i. These sub-graphs may overlap with each other, but every UAP has a *PROPAGATED* flag to mark whether the UAP has been processed, and each UAP would be processed exactly only once (assured by line 2 through 4). In the chain of propagation, every sub-graph are replicated and accumulated into previous sub-graphs, like a rolling snow ball. The times of insertions in each time of propagation are list in Table.II.

In the worst cases, the edge sets of sub-graphs are overlapped and for any i, $T_i \approx T$. Then we can infer that the total complexity of propagation procedures measured by count of constraint insertions.

TABLE II.
TIMES OF INSERTIONS IN EACH TIME OF PROPAGATION

	Increment	**Total**
v_n	$V \cdot T_n$	$V \cdot (T_n + 1)$
v_{n-1}	$V \cdot T_{n-1} \cdot (T_n + 1)$	$V \cdot (T_{n-1} + 1) \cdot (T_n + 1)$
...	...	...
v_1	$V \cdot T_1 \cdot \prod_{i=2}^{n} (T_i + 1)$	$V \cdot \prod_{i=1}^{n} (T_i + 1)$

$$O(V \cdot \prod_{i=1}^{n} (T_i + 1)) = \begin{cases} O(V \cdot T^n), & \text{when } n \ll V \\ O(V \cdot T^V), & \text{when } n \approx V \end{cases}$$

And in the worst case, the whole constraint set of the preceded UAP is copied in every time of propagation, leading the whole graph might grow exponentially. This worst case would only happen when the data structure of program is a deep nested aggregate type. But in most programs, there are plenty of irrelevant data structures, dispersing the variables into different sub-graphs and making n far smaller than V. So we believe that the performance of our algorithm would be polynomial to the count of constraints in most cases.

C. Optimizations

The basic algorithm is time exhausting in practice, several performance optimizations are introduced to make our analysis applicable for our application.

1) Folding Right Recursive UAPs: It has been discussed above that UAP set is diverging when it involves recursive sub-structures. For example, the constraint $p \supseteq p.*.next$ is usually met in list traversals. With the field rule, it could repeatedly derive out an infinite sequence of constraints, like $p.*.next \supseteq p.*.next.*.next$, $p.*.next.*.next \supseteq p.*.next.*.next.*.next$, etc. These constraints are really exists but most of them are useless because no other constraint refers them. So these UAPs with recursive sub-structures could be folded into short forms.

As it is referred above, $P(a)$ is the separation set of UAP a. As every separation (p, s) of UAP is a pair of components, let $dom(P)$ and $ran(P)$ represent the sets of former and latter components respectively in the separations. That is

$$\begin{aligned} dom(P) &= \{p | (\exists s)(p, s) \in P\} \\ ran(P) &= \{s | (\exists p)(p, s) \in P\} \end{aligned}$$

A UAP a can be said right recursive, when . And f folds a in a way of getting rid of the latter part of the separation, where

$$f(a) = \begin{cases} p, & (\exists (p, s) \in P(a)) s \in ran(P(p)) \\ a, & \text{otherwise} \end{cases}$$

A UAP needs to be folded for several times until it is non-recursive. So in analysis, all UAPs like $p.*.next.*$

$.next.*.next$ are folded into non-recursive forms like $p.*.next$.

Folding right recursive UAPs can bring approximation to the common pointer analysis, causing that, for example, different elements of an iterative container become undistinguishable. It introduces approximation to the result of static constant containers, in which pointers in different positions should be distinguishable. In practice, however, those iterative containers are dynamically allocated and managed. So that when anyone of the elements is referred in a program point, all the functions stored in respective fields in the container are potentially involved. From this point, the approximation would have no effect to the accuracy of analysis to dynamic data structures. Similar problem is encountered by previous works [6], and is overcome by a similar mechanism.

After getting rid of those recursive UAPs, the total number of UAPs in a program has an upper bound. Let l stand for the maximum distance between the level of a function pointer type field and the level of the top level aggregate type that the pointer resides, and let k stand for the maximum level of pointer used in program. Then the maximum length of UAP is $l+lk$, because every field may have the maximum level of pointer type, and program may use variables of maximum level pointer type. So UAP set should have a polynomial upper bound of V^{l+lk}, where V is the number of identifiers in program.

In our observation, some data structures have a very large recursive span, which encumbers the whole analysis greatly. In this case, k-limiting is the last effective method to control the size of UAP set, as well as the ACG size. As stated before, the theoretical maximum length of UAP is $l + lk$, which is hard to reach for actual programs, because few pointers have the maximum pointer level of l, and few aggregates have maximum depth of k, in our observation, most programs can be solved appropriately in a UAP length of 10 to 12. In other words, practically most pointers could be solved in a set of UAPs whose length is not exceeding 12. By limiting the upper bound of UAPs, the size of ACG could be efficiently reduced.

2) Partitioning Analysis: After folding the recursive UAPs, the UAP set has a polynomial upper bound, however, the total number of UAPs and their constraints are still very huge for large programs. To make the analysis more effective, the UAPs and the constraints need to be reduced as possible as we can. There are mainly two methods to reduce constraints: partitioning analysis and constraint reduction.

Partitioning analysis is based on the natural compositions of program. There are at least 2 levels of compositions: functions and translation units. And someone would refer modules as well. A program is made of several translation units, each of which contains several functions. An identifier is global when it can be used in different functions in the same translation unit. Or it is external when it can be referred in different translation units. By classifying those identifiers, our analysis could be run in different levels of compositions. At first, functions in the same translation unit are processed individually, exporting the constraints in which the UAPs at both sides are global or external. And second, the translation unit is processed, exporting external constraints. And at last, all external constraints are collected together for analyzing. In practice, the layer of translation units could be skipped if program doesnt have too many translation units.

Partitioning analysis divides the constraints into much smaller partitions, in each of which the analysis become efficient because the UAP set of each partition is very small, and most complex graph components are filtered by the boundary of partitions, leaving simplified relations of global or external constraints to the next level of analysis. The whole process is depicted in Fig.6.

In Fig.6, there are 3 categories of UAPs: local, global, and external. As a result, field propagation process of every level becomes a category-based filter. For example, FPs in local level filter out all local UAPs, and FPs in global level filter out all global UAPs, respectively. Before being filtered, queried UAPs should be linked to at least one UAP of outgoing categories. And the linkage information can help the FPs in next levels to generate complete relations between queried pointers and the functions that they pointed. Additionally, the number of levels can be adjusted by different applications, such as adding a modular level for inter-modular analysis which support static and dynamic libraries, or merging the global and external level together in analyzing small programs for performance.

3) Tunable Exploration: The basic algorithm is essentially an aggregation in depth-first order, whose disadvantage is apparent that exploring will not stop until all relevant UAPs are propagated. But in many applications, we wish the analyzer could report some results as quickly as possible. So it is necessary to stop the propagation process after a small wave of propagation, giving the analyzer and users a chance to decide whether or not to continue processing. This technique, called tunable exploration, could help to prune a lot of long UAPs which unlikely contain useful aliases under the belief that aliases could be found with short UAPs.

To make the propagation stop quickly, we modified the Propagate procedure, and marks all UAPs in new constraints (in line10) as PROPAGATED. If any queried UAP doesnt link to any outgoing UAPs, propagation will continue by resetting those new UAPs, until the whole depth-first aggregation is completely performed. Users could also specify a limit count of waves of propagations if analysis runs too long. However, pruning long UAPs can lead to losing some useful constraints as Fig.7 depicts.

The missing of unexplored constraints is caused by the stopping of propagation when queried nodes have different sources of values from different levels of fields. Experimental result shows that the undiscovered solutions are minor, because most nodes have simple origin of values. In fact, the analyzer needs to spend most of time in exploring few solutions precisely, while most solutions can be discovered in short time. Thats the reason of the

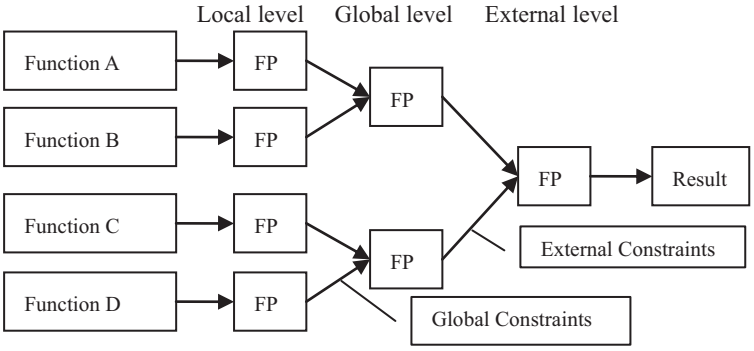

Figure 6. Partitioning analysis, FP means the field propagation process.

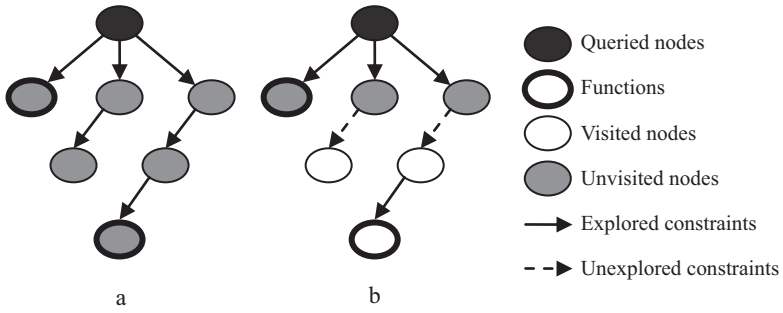

Figure 7. Normal exploration (a) and Tunable exploration with only one round (b)

effectiveness of tunable exploration.

4) Constraint Reduction: Constraint reduction is based on the observation that many functions, especially the string operations and memory controls, may bring in a large number of irrelevant aliases which link different types of aggregates together, and increase the work load of field propagation process tremendously. In worst cases, it makes the ACG a complete graph. So in the step of constraint extraction, the value passing through the parameters of these functions are ignored. Other functions can also be filtered, but the effect is unclear.

D. Field-based Analysis

Field-based analysis, an approximated method for experiment comparison in this article, treats field names as special variables. With this method, all pointers that nested in aggregate types could be solved simply by the help of corresponding field name, instead of data flow.

To implement a field-based analyzer on the basis of field propagation algorithm, field names are omitted in the field propagation process. Instead, we create individual UAPs to stand for different fields, and add aliases of the fields and corresponding variables. Formally, we add 2 rules to our inference system:

$$[\text{field2}]\frac{p.s \supseteq q}{p.s = s \wedge s \supseteq q}[\text{field3}]\frac{q \supseteq p.s}{p.s = s \wedge q \supseteq s}$$

Note that in these rules, s refers to the UAP components that contain field names, and these components have no need to be propagated because they are regarded as variables and the aliases could be transited though them. Other components consist of pointers (*), arguments (ρ_i), and return values (γ) are propagated like field sensitive analyzer, and these components are far less than those components that contain field names, leading the field-based analyzer much quicker than field-sensitive analyzer experimentally.

IV. Experimental Results

We have selected several programs as analysis targets listed in Table.III. All these programs can be downloaded from GNU.org or their respective official websites. The analysis program runs on a Intel Core Dual T5250 1.5GHz2 CPU with 2GB memory, but uses only one core because the analyzer is single threading. Our analyzer propagates fields in 2 levels of partitions: intra-procedural level and inter-procedural level. And the inter-modular capability is unused in experiment.

From Table.III, most function pointers can be solved. And we have inspected the sources of these programs, and find that most of the unsolvable pointers are exactly null pointers. To demonstrate the accuracy, the analyzer counts all functions that referenced by solved pointers and that have been taken address. Most of the unreferenced functions are passed as callbacks to libraries, such as qsort(), signal(), atexit(), etc.. So actually, the result should be more accurate than it looks. The percentages of solved pointers of field-sensitive and field-based analysis are depicted comparatively as the evidence in Fig.8.

The percentages are not the higher the better, because some of the pointers are used in unreachable parts of program, which can affect the results of field propagation.

TABLE III.
ACCURACY OF THE EXPERIMENT

Program	Version	Solved/Queried	Ref/All[a]	Avg size of solutions	Unreferenced functions
bash	2.05	75/85	338/363	6.72	25
cflow	1.4	22/29	22/24	2.00	2
gawk	3.1.0	71/73	73/81	4.11	8
gettext[b]	0.18	1/1	2/14	2.00	12
grep	2.6.3	16/21	33/36	3.56	3
gzip	1.4	4/4	8/10	4.00	2
make	3.79.1	2/2	24/32	12.00	8
parted	3.0	49/57	272/306	6.20	34
sed	4.2	1/1	2/3	2.00	1
sendmail[c]	8.14.4	32/39	55/88	5.47	33
uucp	1.06.1	75/75	187/210	5.33	23

[a] Count of functions referenced by solved pointers, and count of all functions which have their address taken.
[b] gettext-runtime only
[c] Without static libraries

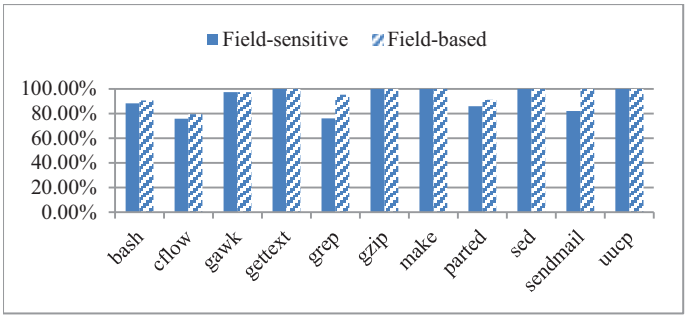

Figure 8. Percentages of solved pointers

While field-based analyzer can still solve them by simply the transitivity of field names, which makes no sense from the view of data flow. Thats the reason why the results of field-based analyzer sometimes have many spurious solutions so as to be unreliable and unsuitable for precise state graph construction, as Fig.9 shows.

The time consumptions of field propagation are listed in Table.IV, in which, we regard constraint insertions as the count of total constraints approximately, because in partitioning analyzing, constraints of each partitions are overlapped with each other, and it needs a lot of time and memory to count them clearly, so we simply count all the insertions as the count of constraints.

The result shows that the propagation time is not exactly polynomial to the size of program (measured as original constraints), because not all the constraints are involved since the propagation process is demand driven.

The time consumption per query varies in different programs, depending on the complexity of the relations among variables. In our observation, it is extraordinarily slow to analyze structures like hash tables and memory control components, in which, plenty uses of void pointer for generality bring in huge number of illogical aliases which contain different types of pointers. It might be avoided by type checking, however, we havent implemented it yet. Currently, we simply isolate the hash tables and memory control components out of the analysis manually for performance, if these components dont involves any function pointers. The most serious disadvantage of field propagation is the time consumption. It needs several orders of magnitude time comparing to field-based analyzer, as Fig.10 shows. Whether it is worth spending so much time for a precise result depends on the requirement of applications.

V. RELATED WORKS AND DISCUSS

The purpose of our analysis is trying to find out those functions are pointed by specific function pointers, which are used in indirect calls, and generating call graphs or state graphs. Unlike some general pointer analyses, the accuracy is of the utmost importance to the results because it can affect the correctness of generated state graphs. Besides the precision requirement, there are other differences to other works in methods and techniques.

A. Comparing to Point-to Analysis

In point-to analysis, analyzer records the point-to set of every pointer, and propagate the set from one pointer to the others by a group of specific rules, which make up an inference system of point-to analysis. Many point-to analyses [6], [17] use an inference system like Table.5

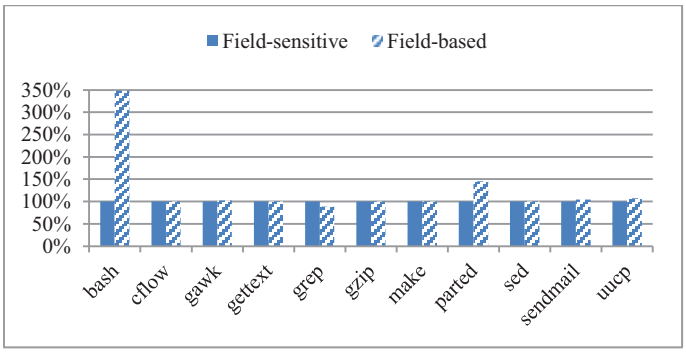

Figure 9. Comparison of the average sizes of solutions of solved pointers

TABLE IV.
ACCURACY OF THE EXPERIMENT

Program	Original Variables	Original Constraints	Total Variables	Constraints Insertions	Propagation Time (ms)	Time (ms) per query
gzip	1,088	20,039	2,855	3,682	266	67
sed	1,437	11,205	2,864	7,345	202	202
gettext	1,163	34,750	2,862	7,900	265	265
bash	16,451	140,426	45,767	654,061	25,226	297
cflow	4,588	22,283	11,995	549,261	18,049	623
sendmail	6,726	75,928	48,689	2,678,439	34,438	884
make	2,646	24,238	16,433	687,158	2,059	1,030
gawk	8,179	47,625	79,463	3,926,304	106,971	1,466
grep	2,887	22,766	85,988	1,086,478	32,404	1,544
uucp	8,350	117,191	651,006	6,934,369	214,811	2,865
parted	10,121	118,401	119,325	6,026,555	183,930	3,227

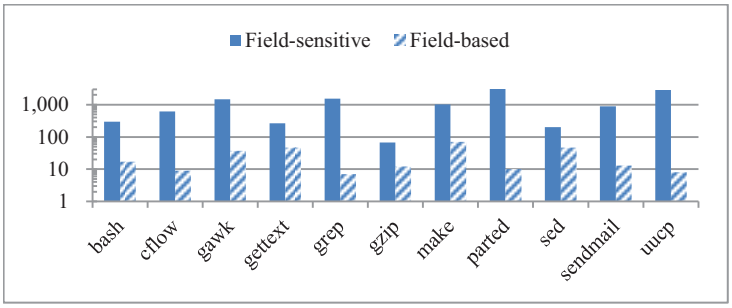

Figure 10. comparison of the time consumption per pointer in milliseconds on a log_10 axis

TABLE V.
COMPARISON OF THE INFERENCE SYSTEM OF POINT-TO ANALYSIS AND ALIAS ANALYSIS

	Point-to inference system		Simulation by alias inference system		
	Preconditions	Conclusions	Preconditions	Conclusions	Deduction
1	$p \supseteq q, r \subseteq p$	$r \supseteq q$	$p.* = q, r \supseteq p$	$r.* = p.* = q$	[alias][trans]
2	$p \supseteq *q, q \supseteq r$	$p \supseteq r$	$p \supseteq q.*, q.* = r$	$p \supseteq q.* = r$	[trans]
3	$*p \supseteq q, p \supseteq r$	$r \supseteq q$	$p.* \supseteq q, p.* = r$	$r = p.* \supseteq q$	[trans]
4	$p \supseteq q \rightarrow s, q \supseteq r$	$p \supseteq r.s$	$p \supseteq q.*.s, q.* = r$	$p \supseteq q.*.s = r.s$	[field][trans]
5	$p \rightarrow s \supseteq q, p \supseteq r$	$r.s \supseteq q$	$p.*.s \supseteq q, p.* = r$	$r.s = p.*.s \supseteq q$	[field][trans]
6	$p \supseteq q \rightarrow s, q \supseteq r$	$p \supseteq r.s$	$p.* = q.*.s, q.* = r$	$p.* = q.*.s = r.s$	[field][trans]

shows. And a simulation by alias inference system is made in passing.

The simulation in Table.V demonstrates that alias inference system could easily simulate all the deductions of point-to analysis, but not vice versa. In Fig.11 is an example which shows that alias analysis can successfully deduce the relation between p and f, but point-to analysis cannot.

What Fig.11 reveals is that the deduction of point-to inference system in Table.5 requires the pointers should stick to memory locations, while alias analysis concerns about the relations among pointers, which are only labels, but taking less attention to what locations the pointers point or where they reside. The location stickiness is an advantage for point-to analysis in performance, because analyzer could take more attention on few locations, instead of the complex relations of the pointers. The disadvantage is that it requires a complete constraint set to make sure that point-to sets could be passed through memory locations. In contrast, alias analyzer could generate chains of pointers that reflect how the values are passed through the variables. If some parts of a program are unavailable temporarily, for example in inter-modular analysis, the chains are broken. After the missing parts are available, the chains are linked. Additionally, after a propagation process on a complete scope of program, internal variables and their constraints could be filtered out to simplify the incomplete constraint graphs. Thats the essential idea of partitioning analysis introduced above. When heap allocation is involved, point-to analyzer needs to analyze the shape of heap in order to assign locations to those pointers that point to heap memory. Function pointer alias analyzer just skips those heap operations because heap location is merely usable in variable aliases; instead, it concentrates on the location of functions and relation of pointers.

Demand-driven pointer analysis [11] suppresses the disadvantage by introducing a special symbol, written as a dot (), which stands for unknown variables that to be solved. That actually acts like partitioning analysis, except that their analysis is performed in a field-based way.

B. Comparing to Other Alias Analysis

Different alias analyses are sharing the same principle of basic model, of which the essential concern is the relations of pointers. Several theoretical works [7], [18] have shown that to precisely solve a may-alias query, it needs exponential times of operations. Most works concentrate their concerns on all kinds of optimizations, which could help to solve alias queries as fast and/or precise as possible. Conclusively, the following optimization techniques are wildly used in previous works:

- Field-insensitive analysis
  Field-insensitive analyzer [12], [19] counts the point-to information of field pointers into aggregate variables where they reside, making the variable set of program much smaller. But the point-to sets of different fields are merged together and hard to distinguish, depending on the test cases.
- Field-based analysis
  Field-based analyzer [3]–[5] [20] treats fields as variables. As our experiment shows, field-based analysis is sometimes effective, but sometimes introduces spurious solutions. Comparing to field-insensitive analysis, field-based analysis is better for function pointer analysis, because functions are usually bound to field names instead of variables.
- K-limiting
  By limiting the length of access paths, analysis could get higher performance. In the other hand, it prevents the analyzer from further exploration, potentially missing some chances to discover deeper implications. However, after all means of optimization, there are always some cases which could cause long access paths and decrease the performance seriously. So K-liming is our last means to control the length of access paths.

There also exist some other techniques, such as regarding constraint relation undirected [10] to lower the complexity of constraint graphs. We absorbed the essences of previous techniques and created several techniques with the help of our observations and experiences. For a graph based analyses, we could benefit in performance from 2 methods primarily: reducing the nodes and reducing the edge. From this aspect, our 4 techniques can be categorized as Table.VI shows.

As stated before, if the actual value of a pointer, at the program point where it is called, is irrelevant to the position in iterative containers, folding the recursive UAPs would constraint the length of recursive UAPs without negative. Partitioning analysis could limit the complexity in small scales, instead of spreading local relations to other parts of programs. The primary inaccuracy of our analysis is brought in with tunable exploration, which would cause some pointers unsolvable. So currently we should avoid complex data structures to get convincing results with a bearable performance.

C. Flow-sensitivity and Context-sensitivity

Flow-sensitivity refers to whether the solutions of the pointers are sensitive to its position in control flow. The reason of using flow-insensitive method in our analysis is that most C programs use static global structures as function tables, and few of the global function pointers would change their values during intra-procedural control flow. Additionally, with the help of SSA in most modern compilers, the flow-sensitivity could be reflected in the value transitivity through the SSA names in intra-procedural analyzing. If the function tables are built dynamically, flow-insensitive analysis is still unreliable, because the analyzer is unaware of the changing of the values.

Context-sensitivity refers to whether the solutions of the pointers are sensitive to the call site contexts of the functions that they reside, so that every function has

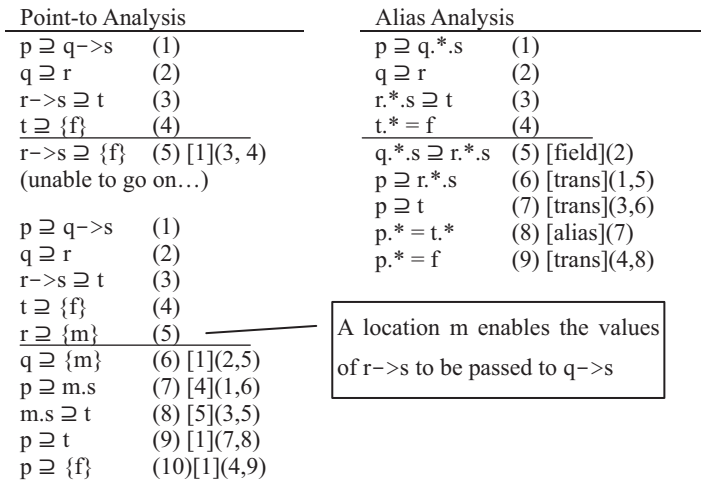

Figure 11. An important difference between point-to and alias analysis

TABLE VI.
COMPARISON OF THE INFERENCE SYSTEM OF POINT-TO ANALYSIS AND ALIAS ANALYSIS

	Node reduction	Edge reduction	Effects to accuracy
Folding of recursive UAP	√		None
Partitioning analysis	√	√	None
Tunable exploration	√	√	With few constraints lost
Constraint reduction		√	Determined by user

different call graphs corresponding to different contexts. So result of context-sensitive analysis is more accurate than result of context-insensitive analysis to reflect the inter-procedural control flow, but it might not be useful for some cases of intra-procedural analysis, such as analysis in unit tests, in which we need a complete control flow of all contexts that the functions are invoked. A context-sensitive analyzer using field propagation is planned in our future works depending on the demand of its applications.

VI. CONCLUSION

We have represented a field-sensitive alias analysis for function pointers, in which the suffix fields of access paths are propagated to prefix parts and their aliases and successors, so as to discover new implied aliases. After the whole process or propagation, analyzer could find the pointed functions in the alias sets of specific pointers. Several optimizations are brought in our analysis for better performance. Although some of the optimizations might cause some alias missing, as an auxiliary tool, users could overcome this disadvantage by either avoiding complex structures, such as raw object, general structures using void pointers and location overlapping, etc., or manually adjusting point-to sets for function pointers, in order to construct a complete reliable state graph for program analysis, which could be applied in program behavior extractions instead of dynamic monitoring [21]–[23].

REFERENCES

[1] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, June 1982.
[2] J. M. Spivey, "Fast, accurate call graph profiling," *Softw. Pract. Exper.*, vol. 34, no. 3, pp. 249–264, Mar. 2004.
[3] A. L. O., "Program analysis and specialization for the c programming language," *PhD Thesis, DIKU, University of Copenhagen*, 1994.
[4] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 32–41.
[5] M. Das, "Unification-based pointer analysis with directional assignments," *SIGPLAN Not.*, vol. 35, no. 5, pp. 35–46, May 2000.
[6] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, Nov. 2007.
[7] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 1–6, Jan. 1997.
[8] N. D. Jones and S. S. Muchnick, "Flow analysis and optimization of lisp-like structures," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '79. New York, NY, USA: ACM, 1979, pp. 244–256.
[9] A. Deutsch, "Interprocedural may-alias analysis for pointers: beyond k-limiting," *SIGPLAN Not.*, vol. 29, no. 6, pp. 230–241, June 1994.
[10] A. Milanova, A. Rountev, and B. G. Ryder, "Precise call graphs for c programs with function pointers," *Automated Software Engineering*, vol. 11, pp. 7–26, 2004, 10.1023/B:AUSE.0000008666.56394.a1.
[11] N. Heintze and O. Tardieu, "Demand-driven pointer analysis," *SIGPLAN Not.*, vol. 36, no. 5, pp. 24–34, May 2001.

[12] X. Zheng and R. Rugina, "Demand-driven alias analysis for c," *SIGPLAN Not.*, vol. 43, no. 1, pp. 197–208, Jan. 2008.
[13] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," *SIGPLAN Not.*, vol. 39, no. 4, pp. 473–489, Apr. 2004.
[14] B.-C. Cheng and W.-M. W. Hwu, "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," *SIGPLAN Not.*, vol. 35, no. 5, pp. 57–69, May 2000.
[15] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 126–135.
[16] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," *SIGPLAN Not.*, vol. 42, no. 6, pp. 290–299, June 2007.
[17] W. Choi and K.-M. Choe, "Cycle elimination for invocation graph-based context-sensitive pointer analysis," *Inf. Softw. Technol.*, vol. 53, no. 8, pp. 818–833, Aug. 2011.
[18] R. Muth and S. Debray, "On the complexity of function pointer may-alias analysis," in *TAPSOFT '97: Theory and Practice of Software Development*, ser. Lecture Notes in Computer Science, M. Bidoit and M. Dauchet, Eds. Springer Berlin / Heidelberg, 1997, vol. 1214, pp. 381–392, 10.1007/BFb0030612.
[19] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards, "Flexible pointer analysis using assign-fetch graphs," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 234–239.
[20] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using cla: a million lines of c code in a second," *SIGPLAN Not.*, vol. 36, no. 5, pp. 254–263, May 2001.
[21] Z. Li and J. Tian, "A software behavior automaton model based on system call and context," *Journal of Computers*, vol. 6, no. 5, pp. 889–896, 2011.
[22] J. Li, Z. Li, and K. Li, "Detection and classification of non-self based on system call related to security," *Journal of Computers*, vol. 4, no. 11, pp. 1117–1124, 2009.
[23] Z. Li and J. Tian, "An approach of trustworthiness evaluation of software behavior based on multidimensional fuzzy attributes," *Journal of Computers*, vol. 7, no. 10, pp. 2572–2577, 2012.

Bo Huang born in 1985, Ph.D. candidate at Computer School, Wuhan University. His research interests include requirements engineering, formal methods, requirements visualization and program synthesis.

Xiang Ling born in 1984, Ph.D. candidate at Computer School, Wuhan University, China. He received his MS degrees in Applied Computer Science in 2008. His research interests include software engineering, compilers and operating systems.

Guoqing Wu born in 1954, professor and Ph.D. supervisor in the Computer School, Wuhan University, China. His main research interests include software requirements engineering and formal methods.