Modular Heap Abstraction-Based Memory Leak Detection for Heap-Manipulating Programs

Longming Dong, Ji Wang, Liqian Chen
National Laboratory for Parallel and Distributed Processing
School of Computer, National University of Defense Technology
Changsha, 410073, P.R.China
Email: donglongming22@163.com, jiwang@ios.ac.cn, lqchen@nudt.edu.cn

***Abstract*—Heap-manipulating programs allow flexible manipulations over dynamically allocated, shared, and mutable heap cells via pointers that point to not only linked data structures but also their pointer fields. Therefore, memory leak detection for these programs requires precise field-sensitive pointer alias information, which make the problem more challenging. In this paper, we present a field and context sensitive algorithm for detecting memory leaks in heap-manipulating programs. First, we propose a modular heap abstraction based on member-access distances and alias bit-vector domain as the escape model of each procedure; Then, based on procedural summaries characterized by this modular heap abstraction, an efficient context-sensitive memory leak detection is proposed in an on-demand way. Experimental evaluation about a set of large C benchmark programs shows that the proposed approach is scalable with satisfied precision as expected.**

***Keywords*—heap-manipulating programs, memory leak detection, modular heap abstraction, field and context sensitive analysis**

I. Introduction

Heap-manipulating programs are those storing and processing data with dynamic and shared data-structures, such as lists and trees. These programs are common in various application domains. For example:

- Operation systems, such as Linux and FreeRTOS, usually manage tasks through priority queues or hash tables.
- Most device drivers manage various devices through complex data structures composed of shared singly- and doubly-linked lists.
- Web servers, such as Apache, receive and store requests with various collections.
- Management information systems also denote views or store data queried from the database with kinds of collections.

Heap cells are allocated, aggregated, separated or deallocated dynamically during a running of a heap-manipulating program. Moreover, complex alias relations between these cells may appear during the running. A cell may be referenced by many pointer variables or other cells, due to the pointer variables or the pointer fields used to operate these memory cells directly or indirectly. Therefore, it is more challenging to decide whether an allocated heap cell is correctly deallocated in heap-manipulating programs than other programs. Memory leak is one of the most common errors in software systems. It can cause memory-intensive or long-time running programs to crash.

A recent representative static memory leak detection tool (called Fastcheck) is proposed by Cherem et al. [1] based on Guarded Value-Flow. Fastcheck tracks the flow of memory cells from all allocation points to all deallocation points and can detect memory leaks by checking whether there is exactly one *source-sink* pair along all paths. The analyzer uses a unification-based points-to analysis, which partitions the whole memory into disjoint regions with equivalence classes. However, once an allocated cell is linked to another cell via a pointer field assignment, theses two cells fall into an equivalent region. In this case, Fastcheck cannot tell whether all cells in the same region are eventually deallocated based on this kind of pointer alias analysis. Hence, in order to detect this kind of memory leak, we have to analyze heap-manipulating programs in a field-sensitive way.

On the other hand, field-sensitive memory detection needs to describe the memory layout [2], [3], [4], which will increase the storage overhead of the analysis to store intermediate memory states. Therefore, field-sensitive analysis may not scale to large programs. Hence, we need to design heap abstractions over field-sensitive memory models. Furthermore, in large software systems, due to modular design, a large number of procedures may be called many times and there may be recursive procedures.

Furthermore, the modular, summary-based analysis [5], [6], [7], [8] offers three key advantages over the whole program analysis: such as reuse of analysis results, scalability and parallelizability. Procedures in programs are analyzed in a particular context, and thus, an additional caller to the same procedure may require re-analysis of the whole procedure. However, summary-based analysis allows reuse of analysis results. Procedural summaries also describe precondition and postcondition about the procedure based on big-step semantics, abstracting away its internal details. So, we can spend less memory to store only procedural summaries after analyzing a procedure, which can scale to large programs easily. In modular analysis, any two procedures that do not have a caller/callee relationship can be analyzed independently. Thus, such characteristic naturally enables decomposition of large programs into many small pieces, and each of which can be analyzed in parallel.

In this paper, we propose a modular heap abstraction based on member-access distances and alias bit-vector domain that abstracts memory model field-sensitively. Then, we present procedural summaries based on this modular heap abstraction and a filed and context sensitive interprocedural memory leak detection algorithm. In the end, we implement a prototype tool (*Heapcheck*) for detecting memory leaks of C programs. As corroborated by our experiments, modular heap abstraction is crucial for field and context sensitive memory leak detection in heap-manipulating programs. In summary, this paper makes the following contributions:

- **A Modular Heap Abstraction based on Member-access Distances and Alias Bit-vector Domain.** We present an extended pointer structure based on member-access distances that distinguish different fields to abstract the memory layout reachable from the pointer variable in a field-sensitive way. Then, pointer alias for each member field with certain distance is described by bit vectors in a modular way, which are useful for memory leak detection.
- **A Field and Context Sensitive Memory Leak Detection Algorithm.** We present a field and context sensitive memory detection based on fixpoint iteration algorithm in an on-demand style. Procedural summaries are built for each procedure based on this modular heap abstraction, and a pair of mapping $\langle \bar{\alpha}, \bar{\beta} \rangle$ is proposed to instantiate procedural summary at each call and return site.
- **Experimental Evaluations.** Experimental evaluation for a set of large C programs shows that our approach can be scalable with satisfied precision in detecting memory leaks for large heap-manipulating programs as expected.

The remainder of the paper is organized as follows. Section II introduces the basic syntax of heap-manipulating programs. Section III presents the modular heap abstraction based on member-access distances and bit-vector domain. Section IV details our interprocedural memory leak detection algorithm. Section V shows experimental results. Section VI presents related work. Section VII shows conclusions and future work.

II. Heap-manipulating Programs

The composite data structures (*struct* in C) are used to represent aggregate cells in heap-manipulating programs, and each pointer field is used to point to another cell. In this paper, we omit data fields and their operations to get a minimal set of syntax shown in Fig. 1, as they do not change the shape of the heap.

In Fig. 1, $PVar$ denotes the set of pointer variables, and e denotes a program expression. Basic pointer assignments include: a pointer is set $null$, the value of one pointer (or the pointer field) is copied to another pointer (or another pointer field), a heap cell is allocated or deallocated. The basic control statements (such as: *If-Then-Else* and *While*) can be transformed into *switch* statement [9], herein: e denotes test condition, c_i is the constant condition of each branch, n_i denotes the corresponding successor. Heap-manipulating programs also support the call and return of the procedure.

$$
\begin{aligned}
& p,\ q \in PVar \\
& f_1,\ f_2,\ \dots\ f_i\ \dots,\ f_m \in Fields \\
& AsgnStmt := \ p = null |\ p \to f_i = null |\ p = q | \\
& \qquad\qquad p = q \to f_i |\ p \to f_i = q |\ p = malloc() |\ p = free() \\
& SwitchStmt := \ switch\, e\, \{c_1 : n_1,\ \dots,\ c_j : c_j,\ \dots,\ c_k : n_k,\ \dots\} \\
& CallStmt := \ e = g(e_1,\ \dots,\ e_k) \\
& ReturnStmt := \ return\ \ e \\
& Stmt := \ AsgnStmt |\ SwitchStmt |\ CallStmt |\ ReturnStmt \\
& SequenceStmt := \ Stmt; Stmt
\end{aligned}
$$

Figure 1. Basic statements of heap-manipulating programs

For the sake of simplicity, we require that the assignment statements satisfy a SSA (Static Single Assignment)-like form shown in Fig. 1. And complex assignment statements can be transformed into standard ones by instrumenting auxiliary pointer variables shown in Table I.

Table I
TRANSFORMING FOR HEAP-MANIPULATING PROGRAMS

Pointer Statement	Instrumental Statement
$p \to f_i = q \to f_j$	$pt_0 = q \to f_j;\ p \to f_i = pt_0$
$p = p \to f_i$	$pt_1 = p \to f_i;\ p = pt_1$
$p \to f_i = malloc$	$pt_2 = malloc;\ p \to f_i = pt_2$
$p = q \to f_i \to f_j$	$pt_3 = q \to f_i;\ p = pt_3 \to f_j$
$p \to f_i = free()$	$pt_4 = p \to f_i;\ pt_4 = free()$

A configuration for heap-manipulating programs is a valuation of pointer variables with allocated heap addresses. The constant *null* is represented by a distinguished symbol $\perp$. We can denote such a configuration through a heap state.

Definition 1 (Heap State). A heap state is the pair $\langle H, S \rangle$. Herein, $H = \langle V, E \rangle$, where the node V denotes the set of all the heap cells, and the edge $E : V \xrightarrow{F} V$ denotes the points-to relations between cells via their fields F. $S : PVar \to V$ describes that the values of pointer variables are the addresses of heap cells.

Example 1. Fig. 2 shows a simple heap state with the formal tuples and the shape graph respectively.

$$
\begin{aligned}
& PVar = \{x, y, z\} \\
& V = \{heap\_0, heap\_1, heap\_2, heap\_3, heap\_4\} \\
& S = \{\langle x, heap\_0 \rangle, \langle y, heap\_1 \rangle, \langle z, heap\_4 \rangle\} \\
& E = \{heap\_0 \xrightarrow{f_1} heap\_3, heap\_0 \xrightarrow{f_2} heap\_1, \\
& \qquad heap\_1 \xrightarrow{f_2} heap\_2, heap\_3 \xrightarrow{f_1} heap\_4\}
\end{aligned}
$$

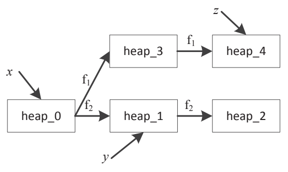

Figure 2. A simple heap state

III. Modular Heap Abstraction

In this section, we extend the basic pointer structure based on member-access information to support field-sensitive analysis, and then propose a modular heap abstraction by combining member-access information and alias information. Then, we show abstract semantics of basic statements for heap-manipulating based on the modular heap abstraction.

A. Extended pointer structures

The basic structure of pointer variable can be denoted as $p : \{f_1,\ f_2,\ ...,\ f_n\}$, which describes that the pointer p points to a cell with *n* pointer fields. In a field-sensitive analysis, the distance away from the heap cell pointed to by current pointer in the heap can be also said to be the times of dereferencing member pointer field of the pointer. For example, the times of dereferencing member pointer field f_m of the pointer variable p in the pointer expression $p \rightarrow f_m$ is 1, which is also said to be that the heap arrived at through dereferencing member pointer field f_m one times is 1 away from the heap pointed to by p.

We know from the basic pointer statements in Fig. 1 that the maximum distance away from the cell pointed to directly by the pointer assignment is 1 (e.g., the distance of the heap cell referenced by the field f_m away from p is 1 in $p \rightarrow f_m = q$). Therefore, if we can obtain the precise alias information of heap cells in the distance less than 2, then we can decide whether the pointer assignment causes any memory leak. So, in this sense, detecting memory leaks for the heap-manipulating programs has local property.

Definition 2 (Member-access Distances $\mathbb{D}$). The member-access distances $\mathbb{D}$ is a pair $\langle\{0,1,2\},\{+,-\}\rangle$, where 0 and 1 denote precise values of distances respectively — zero and one, and 2 is an abstract value of distances denoting the values equal to or greater than two. The operations $+,-$ are shown in Table II(a), II(b), where $\bot$ denotes this operator cannot be applied to these operands, and $\top$ denotes the whole set $\{0,1,2\}$.

Table II
OPERATIONS OVER $\mathbb{D}$

(a) Subtraction over $\mathbb{D}$

–	0	1	2
0	$\bot$	$\bot$	$\bot$
1	{1}	{0}	$\bot$
2	{2}	{1,2}	$\top$

(b) Addition over $\mathbb{D}$

+	0	1	2
0	{0}	{1}	{2}
1	{1}	{2}	{2}
2	{2}	{2}	{2}

The expression $d_1 - d_2$ (or $d_1 + d_2$) denotes the distance obtained by d_2 steps *backward* (or *forward*) along a pointer field from the heap cell with the distance d_1 away from the current pointer. In Table II(a), as 2 is an abstract value denoting that cells (called *summary node*) can be arrived at through exactly two or more than two times of pointer member dereferencing, the expression $2-1$ describes 1 step backward from the current distance greater than 1 away from the current pointer. Hence, if the starting distance is exactly 2, then the result is 1; otherwise (greater than 2), the result is the abstract value i.e. 2. Similarly, $2-2=\top$.

Definition 3 (Extended Pointer Structure). An extended pointer structure of a pointer p is denoted as $\tau_p^\sharp : \{f_1 : \langle dist, 2^{PVar}\rangle;\ f_2 : \langle dist, 2^{PVar}\rangle;\ ...;\ f_n : \langle dist, 2^{PVar}\rangle\}$, where $dist \in \mathbb{D}$ and the set 2^{PVar} is the set of other pointer variables that also point to the cells that can be accessed in the $dist$ times of dereferencing the pointer member field f_m $(1 \le m \le n)$.

There are two special elements in the alias pointer power set 2^{PVar}: $\emptyset$ denotes there are no other pointer variables pointing to the cell but which do exist in the heap; however, $\bot$ denotes that the value of pointer (or pointer field) is $null$ and this cell has not yet been allocated in the heap.

The function $\mathbb{FD}_{\tau_x^\sharp} : Fields \times \mathbb{D} \rightarrow 2^{PVar}$ is defined to obtain the set of alias pointers, when given a pointer field and the distance away from the pointer variable x. For example, $\mathbb{FD}_{\tau_x^\sharp}(f_2,1)=\{y\}$ in Fig. 2. Here, we can show the extended structures of the pointer variables x, y and z in Fig. 2.

$$\tau_x^\sharp : \{f_1 : \langle 0,\emptyset\rangle, \langle 1,\emptyset\rangle, \langle 2,\{z\}\rangle; f_2 : \langle 0,\emptyset\rangle, \langle 1,\{y\}\rangle, \langle 2,\emptyset\rangle\}$$

$$\tau_y^\sharp : \{f_1 : \langle 0,\emptyset\rangle, \langle 1,\bot\rangle; f_2 : \langle 0,\emptyset\rangle, \langle 1,\emptyset\rangle, \langle 2,\bot\rangle\}$$

$$\tau_z^\sharp : \{f_1 : \langle 0,\emptyset\rangle, \langle 1,\bot\rangle; f_2 : \langle 0,\emptyset\rangle, \langle 1,\bot\rangle\}$$

Definition 4 (Abstract Heap State). The abstract heap state $\mathbb{S}^\sharp$ at any point of the heap program *HP* can be composed of extended structures of all pointer variables, as: $\mathbb{S}^\sharp = \{\tau_{p_i}^\sharp | p_i \in PVar(HP)\}$.

It is easy to see that the number of abstract heap states based on extended pointer structures is finite. Assume the number of pointer variables is pn and the maximum number of pointer fields is fn. Then the maximum number of abstract states is $[fn \times 3 \times (2^{pn}+1)]^{pn}$, where $2^{pn}+1$ is the total number of alias pointer power set 2^{PVar}.

B. Modular heap abstraction

From the above section, we know that the number of abstract heap states based on extended pointer structures is mainly dominated by the factor— the number of pointer variables in heap-manipulating programs. On the other hand, large programs are composed of kinds of procedures with the caller/callee relationships, and the number of pointers read or written in a procedure is usually small. So, we can encode alias pointer sets of heap states modularly to reduce the complexity.

The base pointer set in a procedure (called *BPPS* for short) includes global pointers, formal parameter pointers, local pointers and return pointers. The pointer variables read or written in a procedure cannot exceed the scope of the base procedural pointer set. We maintain a fixed variable ordering for pointer variables in the *BPPS* of a procedure. Then, the alias pointer sets of the extended pointer structure of a pointer in a procedure can be denoted as a bit vector based on the variable ordering.

Definition 5 (Alias Bit-Vector, $\mathbb{ABV}$). If the base pointer set in a procedure ranges over: $BPPS = \langle p_0, p_1, ..., p_{n-1}\rangle$, we can denote an alias pointer set in the extended pointer structure $\tau_x^\sharp$ with a bit-vector $\overrightarrow{r}_x^\sharp \in \{0,1\}^n$. Given a field f_m and a distance d, $\overrightarrow{r}_x^\sharp(f_m,d)[i]=1$ if and only if: $p_i \in \mathbb{FD}_{\tau_x^\sharp}(f_m,d)$.

Accordingly, we maintain two particular elements in $\overrightarrow{r}_x^\sharp$: $\bot$ and 0. $\overrightarrow{r}_x^\sharp(f_m,d)=\bot$ means $\mathbb{FD}_{\tau_x^\sharp}(f_m,d)=\bot$, $\overrightarrow{r}_x^\sharp(f_m,d)=0$ means $\mathbb{FD}_{\tau_x^\sharp}(f_m,d)=\emptyset$. As an example, in Fig. 2, extended pointer structures of variables *x*, *y* and *z* can be described by alias bit-vectors as follows:

$$BPPS = \langle x, y, z\rangle \text{ with variable ordering: } x \prec y \prec z$$

$$\overrightarrow{r}^{\sharp}_{x} : \{f_1 : \langle 0,0\rangle, \langle 1,0\rangle, \langle 2,001\rangle; f_2 : \langle 0,0\rangle, \langle 1,010\rangle, \langle 2,0\rangle\}$$

$$\overrightarrow{r}^{\sharp}_{y} : \{f_1 : \langle 0,0\rangle, \langle 1,\bot\rangle; f_2 : \langle 0,0\rangle, \langle 1,0\rangle, \langle 2,\bot\rangle\}$$

$$\overrightarrow{r}^{\sharp}_{z} : \{f_1 : \langle 0,0\rangle, \langle 1,\bot\rangle; f_2 : \langle 0,0\rangle, \langle 1,\bot\rangle\}$$

In order to maintain the mapping between variables and indices, we define a pair of bijective function $\langle \iota, \iota^{-1}\rangle$, such that $\iota(p_i) = i$ and $\iota^{-1}(i) = p_i$. For example, in the above basic procedural pointer set $BPPS$, $\iota(x) = 0$ and $\iota^{-1}(0) = x$.

Definition 6 (Alias Bit-Vector Domain). We define a so-called *alias bit-vector* domain $\mathbb{ABVD} = \langle \mathbb{ABV}, \{\overrightarrow{+}, \overrightarrow{-}\rangle\}$, whose operations are defined in Table III(a), III(b).

Table III
OPERATIONS OVER $\mathbb{ABVD}$

(a) Bitwise addition

$\overrightarrow{+}$	0	1
0	0	1
1	1	1

(b) Bitwise subtraction

$\overrightarrow{-}$	0	1
0	0	0
1	1	0

Definition 7 (Bitwise Less between Alias Bit-Vectors $\overrightarrow{\leq}$). Given two alias bit-vectors $\overrightarrow{r}^{\sharp}_{1x}$ and $\overrightarrow{r}^{\sharp}_{2x}$ of dimension n in the same procedure, if it holds that $\forall 0 \leq i < n.\ \overrightarrow{r}^{\sharp}_{1x}[i] \leq \overrightarrow{r}^{\sharp}_{2x}[i]$, then $\overrightarrow{r}^{\sharp}_{1x} \overrightarrow{\leq} \overrightarrow{r}^{\sharp}_{2x}$.

So, we can denote the alias bit-vector as a lattice $\mathbb{ABVL} = \langle \mathbb{ABV}, \{\overrightarrow{+}, \overrightarrow{-}\}, \overrightarrow{\leq}\rangle$.

Definition 8 (Containment between two extended pointer structures). If $\overrightarrow{r}^{\sharp}_{1x}$ and $\overrightarrow{r}^{\sharp}_{2x}$ for two extended structure $\tau^{\sharp}_{1x}$ and $\tau^{\sharp}_{2x}$ in the same procedure hold $\forall f_m \in Fields\ \forall d \in \mathbb{D}. \overrightarrow{r}^{\sharp}_{1x}(f_m, d) \overrightarrow{\leq} \overrightarrow{r}^{\sharp}_{2x}(f_m, d)$, then $\tau^{\sharp}_{1x} \subseteq \tau^{\sharp}_{2x}$.

Obviously, $\tau^{\sharp}_{1x} = \tau^{\sharp}_{2x} \equiv \tau^{\sharp}_{1x} \subseteq \tau^{\sharp}_{2x} \wedge \tau^{\sharp}_{1x} \supseteq \tau^{\sharp}_{2x}$.

Definition 9 (Compatibility between two extended pointer structures). For two extended structures $\tau^{\sharp}_{1x}$ and $\tau^{\sharp}_{2x}$ in the same procedure, if it holds that $\tau^{\sharp}_{1x} \subseteq \tau^{\sharp}_{2x} \vee \tau^{\sharp}_{1x} \supseteq \tau^{\sharp}_{2x}$, then we say they are *compatible*.

Definition 10 (Containment between two abstract heap states). If for two abstract states $\mathbb{S}^{\sharp}_1$ and $\mathbb{S}^{\sharp}_2$ in the same procedure holds $\forall \tau^{\sharp}_x \in \mathbb{S}^{\sharp}_1\ \exists \tau'^{\sharp}_x \in \mathbb{S}^{\sharp}_2.\ \tau^{\sharp}_x \subseteq \tau'^{\sharp}_x$, then $\mathbb{S}^{\sharp}_1 \subseteq \mathbb{S}^{\sharp}_2$.

An abstract heap state may be various forms, due to the alias relations between pointers. In order to compare abstract states, we should denote an abstract heap state with a canonical form (called a saturated abstract state).

Definition 11 (Saturated abstract state). An abstract state $\mathbb{S}^{\sharp}$ is *saturated* in a procedure with $BPPS = \langle p_0, p_1, ..., p_{n-1}\rangle$, if and only if it satisfies the following three properties:

1) **Anti-reflexivity**. $\forall p_i \in BPPS. \overrightarrow{r}^{\sharp}_{p_i}(f_m, 0)[i] = 0$.
2) **Symmetry**. $\forall p_i, p_j \in BPPS.\ \overrightarrow{r}^{\sharp}_{p_i}(f_m, 0)[j] = 1 \rightarrow \overrightarrow{r}^{\sharp}_{p_j}(f_m, 0)[i] = 1$.
3) **Transitivity**. $\forall p_i, p_j, p_t \in BPPS.\ \overrightarrow{r}^{\sharp}_{p_i}(f_m, d_1)[j] = 1 \wedge \overrightarrow{r}^{\sharp}_{p_j}(f_m, d_2)[t] = 1 \rightarrow \overrightarrow{r}^{\sharp}_{p_i}(f_m, d_1 + d_2)[t] = 1$.

Any abstract state can become *saturated* by the $Saturate$ operation, shown in Alg. 1.

Algorithm 1 Saturate($\mathbb{S}^{\sharp}$,$BPPS = \langle p_1, p_2, ..., p_{n-1}\rangle$)

```
modified ← true;
while modified do
  modified ← false;
  for each $p_i$ in $BPPS$ do
    for each $f_m$ in $Field$ do
      //Anti-reflexivity
      $\overrightarrow{r}^{\sharp}_{p_i}(f_m, 0)[i] \leftarrow 0$;
      for each $j$ in $\{j | \overrightarrow{r}^{\sharp}_{p_i}(f_m, 0)[j] = 1\}$ do
        //Symmetry
        if $\overrightarrow{r}^{\sharp}_{p_j}(f_m, 0)[i] = 0$ then
          modified ← true;
          $\overrightarrow{r}^{\sharp}_{p_j}(f_m, 0)[i] \leftarrow 1$;
        end if
      end for
      for each $j$ in $\{j | \overrightarrow{r}^{\sharp}_{p_i}(f_m, d_1)[j] = 1\}$ do
        //Transitivity
        if $\overrightarrow{r}^{\sharp}_{p_j}(f_m, d_2)[t] = 1 \wedge \overrightarrow{r}^{\sharp}_{p_i}(f_m, d_1 + d_2)[t] = 0$ then
          modified ← true;
          $\overrightarrow{r}_{p_i}(f_m, d_1 + d_2)[t] \leftarrow 1$;
        end if
      end for
    end for
  end for
end while
```

Join Operation. We explore an operation $Join$ that merges two abstract states into one shown in Fig. 3. The $Join$ operation can reduce the number of states at program point to speed up the termination of iteration.

$$Join(\mathbb{S}^{\sharp}_1, \mathbb{S}^{\sharp}_2) = \begin{cases} \mathbb{S}^{\sharp}_1 & \text{if } \mathbb{S}^{\sharp}_1 \subseteq \mathbb{S}^{\sharp}_2 \\ \mathbb{S}^{\sharp}_2 & \text{if } \mathbb{S}^{\sharp}_1 \supseteq \mathbb{S}^{\sharp}_2 \\ \mathbb{S}^{\sharp}_1 \cup \mathbb{S}^{\sharp}_2 & \text{otherwise} \end{cases}$$

Figure 3. Join operator

Note that the *Join* operation may cause the loss of precision. However, any memory leaks are guaranteed not to be missed after $Join$ operation. For $\mathbb{S}^{\sharp}_1 \subseteq \mathbb{S}^{\sharp}_2$, according to Definition 10, $\mathbb{S}^{\sharp}_2$ has more alias pointers information about every heap cell pointed to or referenced via some pointer field by each pointer than $\mathbb{S}^{\sharp}_1$. Therefore, if some statement causes a memory leak in the abstract state $\mathbb{S}^{\sharp}_2$, it will also cause the same memory leak in the abstract state $\mathbb{S}^{\sharp}_1$. On the contrary, if some statement causes a memory leak in the abstract state $\mathbb{S}^{\sharp}_1$, the same memory leak is not necessary to occur in the abstract state $\mathbb{S}^{\sharp}_2$, because the leaked cell in $\mathbb{S}^{\sharp}_1$ may also be referenced by some pointer in $\mathbb{S}^{\sharp}_2$. Hence, if we get two abstract states $\mathbb{S}^{\sharp}_1$ and $\mathbb{S}^{\sharp}_2$ at the same program point, to be sound, we should preserve the smaller abstract state $\mathbb{S}_1^\sharp$ when joining these two states, as shown in Fig. 3.

C. Abstract semantics of basic statements

Only the pointer assignments may directly cause memory leaks, owing to modifying points-to relations among the heaps in heap-manipulating programs. So, we give operational semantics about pointer assignments based on the above modular heap abstraction, shown in Fig. 4.

We write $\mathbb{S}^\sharp$ to denote an environment mapping each live pointer p_t to an extended structure $\tau_{p_t}^\sharp$. The subscripts u, v, w and t range over $\{0, 1, ..., n-1\}$, the field variable f_m ranges over $Fields$, and the distance variable d ranges over $\mathbb{D}$. We define $\overrightarrow{\tau}_{p_t}^\sharp(f_m, d) \triangleq \{\overrightarrow{\tau}_{p_t}^\sharp(f_m, d) | \forall f_m \in Fields \wedge \forall d \in \mathbb{D}\}$.

Detecting memory leaks is fairly straightforward in Fig. 4:

(a) We should check whether there are other pointers dereferencing this cell pointed to by the current pointer through the member access of pointer fields, such as **Rule** 1, 3, 4, 6;
(b) We should check whether there are other pointers pointing to this cell referenced by the pointer field of the current pointer, such as **Rule** 2, 5;
(c) We should check whether all pointer fields of the current pointer are $null$ or pointed to by other pointers, like **Rule** 7.

The semantics of the *Return* statement can be described by a sequence of basic pointer assignments. Firstly, all pointers in the set of local pointers $LVar_f$ should be assigned to *null*, and then the return variable ret_f of the function be assigned with the expression e of the *Return* statement if the function has a return pointer variable.

$$[[return\ \ e]](\mathbb{S}^\sharp) = [[l_k = null; ret_f = e]](\mathbb{S}^\sharp) \quad l_k \in LVar_f$$

Fig. 5 gives the semantics of compositional statements. We analyze each statement sequentially according to the basic operational semantics. In *switch* statement, the choice of branch is made according to the truth of test condition e under the initial abstract state $\mathbb{S}_0^\sharp$. Note that if the truth of test condition is non-deterministic, every branch should be chosen to analyze. So, our memory detection is partially path-sensitive.

$$\frac{S_1^\sharp = [[Stmnt_1]](S_0^\sharp) \quad S_2^\sharp = [[Stmnt_2]](S_1^\sharp)}{S_2^\sharp = [[Stmnt_1; Stmnt_2]](S_0^\sharp)}$$

$$\frac{[[e]](\mathbb{S}_0^\sharp) = [[c_k]](\mathbb{S}_0^\sharp)}{[[switch\ e\ \{c_1 : n_1, ..., c_j : n_j, ..., c_k : n_k, ...\}]](\mathbb{S}_0^\sharp) \rightsquigarrow [[n_k]](\mathbb{S}_0^\sharp)}$$

Figure 5. Abstract semantics for compositional statements

IV. Interprocedural Memory Leak Detection

The transformation of the procedure based on modular heap abstraction can be described with the big-step abstract semantics. If the procedure is: $f(p_0, p_1, ..., p_{k-1})$ with the return pointer variable ret_f, we can give its abstract semantics as: $\mathbb{S}_{Of}^\sharp = [[f(p_0, p_1, ..., p_{k-1})]](\mathbb{S}_{If}^\sharp)$, wherein: $\mathbb{S}_{Of}^\sharp$ is the postcondition after the running of the callee f under the precondition $\mathbb{S}_{If}^\sharp$.

In this section, procedural summary based on modular heap abstraction is firstly described. Then the instantiation of procedural summary is given at the call and return site. In the end, we present a fixpoint iteration algorithm to calculate procedural summary in an on-demand manner.

A. Procedural summary

Procedural summary related to memory leaks describes the transformation of the heap layout about cells before and after procedure, which are reachable from global pointer variables, formal pointer arguments and the return value of the procedure.

Table IV
Six categories of alias in our procedural summary

	Argument	Global	Return
Argument	Arg&Arg	Arg&Glob	
Global	Glob&Arg	Glob&Glob	Glob&Ret
Return		Ret&Glob	

The kinds of alias information in procedural summary can be divided into 6 categories according to modular heap abstraction shown in Table IV. To be more clear, we need to know extended pointer structures over formal arguments, global pointers, and the return pointer at call and return sites of the procedure.

- **Argument**. In heap-manipulating programs, heap cells which the caller passes to the callee are reachable via only global pointers or pointer arguments. Heap cells reached by a pointer argument may be pointed to by another pointer argument or a global pointer variable. Thus, we record alias relationships between pointer arguments (called Arg&Arg), as well as between pointer arguments and global pointers (called Arg&Glob) in the precondition of the procedure.
- **Global**. Global pointers destroy the modularity in heap-manipulating programs: they can be passed into the callee at the call site, and can also be escaped from the callee at the return site. So, we record alias relationships between global pointers and pointer arguments (called Glob&Arg), as well as among global pointers (called Glob&Glob) in the precondition of the procedure. And, we record alias relationships between global pointers and the return variable (called Glob&Ret), as well as between global pointers (called Glob&Glob) in the postcondition of the procedure.
- **Return**. In heap-manipulating programs, heap cells reached by the return variable may be also pointed to by global pointers. So, when the procedure returns, we should record alias relationships between the return variable and global pointers (called Ret&Glob) in the postcondition of the procedure.

1) $[[p_u = null]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp[\vec{r}^\sharp_{p_v}(f_m,d)[u] \leftarrow 0, \vec{r}^\sharp_{p_u}(f_m,d) \leftarrow \bot] | v \neq u\}$ if $\exists w \neq u \wedge l \in \mathbb{D}.\ \vec{r}^\sharp_{p_w}(f_m,l)[u] \neq 0 \vee \vec{r}^\sharp_{p_u}(f_m,0) = \bot$
$\{$***memory_leak***$\}$ otherwise

2) $[[p_u \rightarrow f_i = null]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp[\vec{r}^\sharp_{p_v}(f_m,2) \leftarrow \vec{r}^\sharp_{p_v}(f_m,2) \overrightarrow{-} \vec{r}^\sharp_{p_u}(f_i,l) \overrightarrow{+} \vec{r}^\sharp_{p_u}(f_j,l) | v \in \{v | \vec{r}^\sharp_{p_v}(f_m,1)[u] = 1 \vee \vec{r}^\sharp_{p_v}(f_m,2)[u] = 1\} \wedge l \in \{1,2\} \wedge f_j \in Fields - \{f_i\}, \vec{r}^\sharp_{p_u}(f_i,l) \leftarrow \bot | l \in \{1,2\}, \vec{r}^\sharp_{p_w}(f_i,l) \leftarrow \bot | l \in \{1,2\} \wedge w \in \{w | \vec{r}^\sharp_{p_u}(f_m,0)[w] = 1]\}\}$ if $\vec{r}^\sharp_{p_u}(f_i,1) \neq 0$
$\{$***memory_leak***$\}$ otherwise

3) $[[p_u = p_v]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp_1[\vec{r}^\sharp_{p_u}(f_m,d) \leftarrow \vec{r}^\sharp_{p_v}(f_m,d)] | \mathbb{S}^\sharp_1 \in [[p_u = null]]\mathbb{S}^\sharp\}$ if $\exists w \neq u \wedge l \in \mathbb{D}. \vec{r}^\sharp_{p_w}(f_m,l)[u] \neq 0 \vee \vec{r}^\sharp_{p_u}(f_m,0) = \bot$
$\{$***memory_leak***$\}$ otherwise

4) $[[p_u = p_v \rightarrow f_i]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp_1[\vec{r}^\sharp_{p_u}(f_m,0) \leftarrow \vec{r}^\sharp_{p_v}(f_m,1), \vec{r}^\sharp_{p_v}(f_m,1)[u] \leftarrow 1] | \mathbb{S}^\sharp_1 \in [[p_u = null]]\mathbb{S}^\sharp\}$
if $\exists w \neq u \wedge l \in \mathbb{D}. \vec{r}^\sharp_{p_w}(f_m,l)[u] \neq 0 \vee \vec{r}^\sharp_{p_u}(f_m,0) = \bot$
$\{$***memory_leak***$\}$ otherwise

5) $[[p_u \rightarrow f_i = p_v]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp_1[(\vec{r}^\sharp_{p_w}(f_i,1) \leftarrow \vec{r}^\sharp_{p_v}(f_m,0))[v] \leftarrow 1, \vec{r}^\sharp_{p_w}(f_m,2) \leftarrow \vec{r}^\sharp_{p_v}(f_m,1) \overrightarrow{+} \vec{r}^\sharp_{p_v}(f_m,2),$
$(\vec{r}^\sharp_{p_t}(f_m,2) \leftarrow \vec{r}^\sharp_{p_t}(f_m,2) \overrightarrow{+} \vec{r}^\sharp_{p_v}(f_m,0) \overrightarrow{+} \vec{r}^\sharp_{p_v}(f_m,1) \overrightarrow{+} \vec{r}^\sharp_{p_v}(f_m,2))[v] \leftarrow 1] | t \in \{t | \vec{r}^\sharp_{p_t}(f_m,1)[u] = 1 \vee \vec{r}^\sharp_{p_t}(f_m,2)[u] = 1\} \wedge w \in \{w | \vec{r}^\sharp_{p_u}(f_m,0)[w] = 1\} \cup \{u\} \wedge \mathbb{S}^\sharp_1 \in [[p_u \rightarrow f_i = null]](\mathbb{S}^\sharp)\}$ if $\vec{r}^\sharp_{p_u}(f_i,1) \neq 0$
$\{$***memory_leak***$\}$ otherwise

6) $[[p_u = malloc]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp_1[\vec{r}^\sharp_{p_u}(f_m,d) \leftarrow 0] | \mathbb{S}^\sharp_1 \in [[p_u = null]]\mathbb{S}^\sharp\}$ if $\exists w \neq u \wedge l \in \mathbb{D}.\ \vec{r}^\sharp_{p_w}(f_m,l)[u] \neq 0 \vee \vec{r}^\sharp_{p_u}(f_m,0) = \bot$
$\{$***memory_leak***$\}$ otherwise

7) $[[p_u = free()]](\mathbb{S}^\sharp) = \{\mathbb{S}^\sharp[\vec{r}^\sharp_{p_w}(f_m,d)[t] \leftarrow 0, \vec{r}^\sharp_{p_w}(f_m,d)[u] \leftarrow 0, \vec{r}^\sharp_{p_t}(f_m,d) \leftarrow \bot, \vec{r}^\sharp_{p_u}(f_m,d) \leftarrow \bot] | t \in \{t | \vec{r}^\sharp_{p_u}(f_m,0)[t] = 1\} \wedge w \in \{w | \vec{r}^\sharp_{p_u}(f_m,0)[w] = 0 \wedge w \neq u\}\}$ if $\vec{r}^\sharp_{p_u}(f_m,1) \neq 0 \vee \vec{r}^\sharp_{p_u}(f_m,1) = \bot$
$\{$***memory_leak***$\}$ otherwise

Figure 4. Abstract semantics of basic pointer assignments

B. Interprocedural summary instantiation

We should instantiate the callee's summary with the caller's heap abstraction at a call site for detecting memory leaks. The instantiation consists of mapping extended structures of the heap abstraction at the call site into parameterized ones in the summary of the callee. Alias information captured at the call site should be also reflected in instantiation. The output of the instantiation consists of extended structures over global pointers and the return variable after the call.

Given the caller with the alias bit-vector domain $\mathbb{ABVD_R}$ based on $BPPS_R = \langle r_0, r_1, ..., r_{m-1}\rangle$ and variable ordering $r_0 \prec r_1 \prec ... \prec r_{m-1}$, and the callee $f(p_0,\ p_1,\ ...,\ p_{k-1})$ with the alias bit-vector domain $\mathbb{ABVD_E}$ based on $BPPS_E = \langle e_0,\ e_1,\ ...,\ e_{n-1}\rangle$ and variable ordering $e_0 \prec e_1 \prec ... \prec e_{n-1}$, assume the call statement is: $ret = f(t_0,\ t_1,\ ..., t_{k-1})$; $p_0,\ p_1,\ ...,\ p_{k-1}$ are formals, $t_0,\ t_1,\ ...,\ t_{k-1}$ are actuals; ret_f is the formal return pointer variable of the procedure f; $g_0,\ g_1,\ ...,\ g_{h-1}$ are the global pointers. We can denote the mapping between two environments by two pairs $\langle\alpha,\beta\rangle$: $BPPS_R \underset{\alpha}{\overset{\beta}{\leftrightarrows}} BPPS_E$ and $\langle\bar{\alpha},\bar{\beta}\rangle$: $\mathbb{ABVD_R} \underset{\bar{\alpha}}{\overset{\bar{\beta}}{\leftrightarrows}} \mathbb{ABVD_E}$.

- $BPPS_R \xrightarrow{\alpha} BPPS_E$

$$\begin{cases} \alpha(t_i) = p_i & : \ t_i \text{ is an actual parameter} \\ \alpha(g_i) = g_i & : \ g_i \text{ is a global variable} \\ \alpha(r_i) = \bot & : \ \text{other} \end{cases}$$

- $\mathbb{ABVD_R} \xrightarrow{\bar{\alpha}} \mathbb{ABVD_E}$

$$\bar{\alpha}(\vec{r}^\sharp_{r_i}[j]) = \vec{r}^\sharp_{\alpha(r_i)}[\iota_E(\alpha(\iota_R^{-1}(j)))], \quad \forall j.\ 0 \leq j < m$$

$$\bar{\alpha}(\vec{r}^\sharp_{r_1} \overrightarrow{+} \vec{r}^\sharp_{r_2}) = \vec{r}^\sharp_{\alpha(r_1)} \overrightarrow{+} \vec{r}^\sharp_{\alpha(r_2)}$$

$$\bar{\alpha}(\vec{r}^\sharp_{r_1} \overrightarrow{-} \vec{r}^\sharp_{r_2}) = \vec{r}^\sharp_{\alpha(r_1)} \overrightarrow{-} \vec{r}^\sharp_{\alpha(r_2)}$$

- $BPPS_E \xrightarrow{\beta} BPPS_R$

$$\begin{cases} \beta(ret_f) = ret & : \ ret_f \text{ is a return variable} \\ \beta(g_j) = g_j & : \ g_j \text{ is a global variable} \\ \beta(e_j) = \bot & : \ \text{other} \end{cases}$$

- $\mathbb{ABVD_E} \xrightarrow{\bar{\beta}} \mathbb{ABVD_R}$

$$\bar{\beta}(\vec{r}^\sharp_{e_i}[j]) = \vec{r}^\sharp_{\beta(e_i)}[\iota_R(\beta(\iota_E^{-1}(j)))], \quad \forall j.\ 0 \leq j < n$$

$$\bar{\beta}(\vec{r}^\sharp_{r_1} \overrightarrow{+} \vec{r}^\sharp_{r_2}) = \vec{r}^\sharp_{\beta(r_1)} \overrightarrow{+} \vec{r}^\sharp_{\beta(r_2)}$$

$$\bar{\beta}(\vec{r}^\sharp_{e_1} \overrightarrow{-} \vec{r}^\sharp_{e_2}) = \vec{r}^\sharp_{\beta(e_1)} \overrightarrow{-} \vec{r}^\sharp_{\beta(e_2)}$$

The relations between $\langle\alpha,\beta\rangle$ and $\langle\bar{\alpha},\bar{\beta}\rangle$ could also be denoted with the following diagram, saying that:

$$\begin{cases} \bar{\alpha} \circ \vec{r}^\sharp = \vec{r}^\sharp \circ \alpha \\ \bar{\beta} \circ \vec{r}^\sharp = \vec{r}^\sharp \circ \beta \end{cases}$$

$$\begin{tikzcd} BPPS_R \arrow[r, "\vec{r}^\sharp"] \arrow[d, shift right, "\alpha"'] & \mathbb{ABVD}_R \arrow[d, shift right, "\bar{\alpha}"'] \\ BPPS_E \arrow[r, "\vec{r}^\sharp"] \arrow[u, shift right, "\beta"'] & \mathbb{ABVD}_E \arrow[u, shift right, "\bar{\beta}"'] \end{tikzcd}$$

Figure 6. Relation between $\langle\alpha,\beta\rangle$ and $\langle\bar{\alpha},\bar{\beta}\rangle$

So, we can represent this heap relations between the caller and the callee like Fig. 7.

C. Summarizing procedures via fixpoint iterations

To summarize a procedure, we must know what postconditions the procedure outputs under various preconditions. After the few calls of the same function, we summarize a procedure from top to down in the call flow graph in an on-demand way.

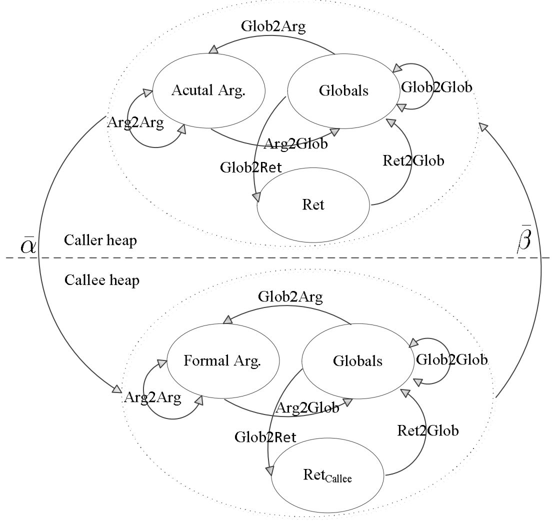

Figure 7. Relation between the caller and callee about abstract heaps

We use a fixpoint iteration algorithm to find heap abstract states at the exits of the procedure, shown in Alg. 2. The input of the fixpoint iteration algorithm includes the procedure and optional heap abstraction. The initial state of the procedure is that:

- If it is on the top of the call graph, all the pointers, such as the local pointers, the global pointers and formals, are set $\perp$.
- Otherwise, it could be obtained according to the heap state of its caller.

A data structure—stack with two operations is defined as:

- $pop()$: pop a element from the top of the stack.
- $push()$: push a element into the stack.

We maintain two stacks during fixpoint iteration:

- M_f with the basic element $\langle Pre, Post\rangle$, where each precondition Pre has one postcondition $Post$, is used to store the summary for the procedure f.
- W with the basic element $\langle s, \mathbb{S}^\sharp\rangle$ is used to store the statement s which needs to be analyzed and the heap abstract states at the program point before this statement.

The function $callee(s)$ returns the callee of the call statement s, and the function $Abs(s)$ returns the heap abstract state before the program point of the current statement s. $Succ(s)$ contains all the statements which succeed s in the control flow graph. Furthermore, the operations $Join$ and $Saturate$ are applied over the nearly computed abstract states after each transfer function.

Given the number of global pointers is g, the number of formals is f, and the maximum number of pointer fields is fn, then the maximum number of heap abstract states during fixpoint iteration is finite — $[fn \times 3 \times (2^{(g+f)} + 1)]^{(g+f)}$. Therefore, the termination of our memory leak detection is guaranteed.

Algorithm 2 Fixpoint Iteration Algorithm $fix(f, Pre)$

```
Require: A procedure $f$, optional modular heap abstraction $Pre$;
Ensure: $M_f$;
1:  $Abs(n) = \perp,\ \ \forall\ n \in CFG_f$;
2:  $W.push(\langle entryNode, Pre\rangle)$;
3:  while $W \neq \emptyset$ do
4:      $\langle s, \mathbb{S}^\sharp\rangle = W.pop()$;
5:      if $s$ instanceof AsgnStmt, SwitchStmt, ReturnStmt then
6:          $\mathbb{S}_1^\sharp = [[s]]\mathbb{S}^\sharp$;
7:      else if $s$ instanceof CallStmt then
8:          if $\bar{\alpha}(\mathbb{S}^\sharp) \in (M_{callee(s)}).Pre$ then
9:              $\mathbb{S}_1^\sharp = \bar{\beta}(M_{callee(s)}(\bar{\alpha}(\mathbb{S}^\sharp)))$;
10:         else
11:             $M' = fix(callee(s), \bar{\alpha}(\mathbb{S}^\sharp))$;
12:             $\mathbb{S}_1^\sharp = \bar{\beta}(M'(\bar{\alpha}(\mathbb{S}^\sharp)))$ ;
13:         end if
14:     else
15:         //other statements
16:         $\mathbb{S}_1^\sharp = \mathbb{S}^\sharp$;
17:     end if
18:     for all $s' \in Succ(s)$ do
19:         $\mathbb{S}_{new}^\sharp = Saturate(Join(\mathbb{S}_1^\sharp, Abs(s')))$;
20:         if $\mathbb{S}_{new}^\sharp != Abs(s')$ then
21:             $Abs(s') = \mathbb{S}_{new}^\sharp$;
22:             $W.push(\langle s', Abs(s')\rangle)$;
23:         end if
24:     end for
25: end while
26: $M_f.push(Pre, Abs(exitNode))$;
27: return $M_f$;
```

V. Evaluations

We have implemented our field and context sensitive interprocedural memory leak detector (*Heapcheck*) for heap-manipulating programs based on the *Crystal* [9] compiler framework. We have tested our prototype implementation *Heapcheck* on a few large C programs. The experiments are conducted on a 3.0 G java virtual machine on top of a 2.67 GHz Intel Core i5 PC.

A. Precision

We have used both our context-sensitive interprocedural analysis with and without summary to detect memory leaks in some programs from SPEC2000 benchmarks [10] and three open-source applications: Tar, OpenSSH and OpenSSL. The one without summary re-analyzes the procedure at each call site. In Table V, the first column shows the names of programs and the second column total lines of codes. The third column shows the pre-process time, including parsing files, building the control flow graphs and call graph, slicing and transforming. The forth column shows the time of the context-sensitive memory leak detection without summary, and fifth column shows the time of summary-based analysis. The

Table V
RESULTS FOR BENCHMARK PROGRAMS

Programs	Size (Kloc)	Preprocess Time (Sec)	Analysis Time (Sec)		Memory (MB)		Reported bugs
			Without summary	Summary-based	Without summary	Summary-based	
164.gzip	7.7	1.19	0.31	0.33	27	6	0
175.vpr	17	1.84	2.83	1.11	194	86.7	1/1
179.art	1.2	0.32	0.1	0.1	34.4	33	0
186.crafty	21.7	3.13	7.56	6.98	295	258	0
188.ammp	13.2	1.88	1.22	0.21	135	60.2	0
300.twolf	19.9	3.05	7.38	4.31	442	195	0/3
176.gcc	210	8.35	106.62	61.04	4596	920	2/17
tar-1.12	11.7	1.08	18.98	9.09	239	178	0/5
openssh	58.3	20.55	2.61	1.44	186	144.3	2/14
openssl	36	8.47	0.46	0.44	73.5	40.7	6/11

sixth column shows the total used memory without summary and the seventh column shows the memory with summary. The last column shows false positives and alarms in the form of "false positives/total alarms".

The results of these experiments are shown in Table V. We know that the summary-based context-sensitive interprocedural memory leak detection is often better than the analysis without summary both in the used time and memory, because the procedural summary about heap abstract state can be reused in the summary-based analysis. When analyzing the benchmark 176.gcc, our memory detection without summary runs out the memory under the current machine. Hence, we divide the whole source code into two parts and analyze each part separately. However, we can analyze the whole source code using our summary-based memory leak detection under the current machine. We also see that our summary-based interprocedural memory detection is more efficient in both time and memory when analyzing larger programs, because the same procedures are often called more times in larger programs.

The tool produces 51 warnings, 11 of which are false positives. Hence, about one warning out of five is a false positive alarm. Compared with the existing work of [11], which analyzed parts of OpenSSH and OpenSSL with 45 and 22 second respectively under a 1.2GHz Athlon machine with 256MB of memory running Linux RedHat 9 and got 39 total memory leak alarms (25 of which were false positives) with a false positive rate of 64%, we get a lower rate of false positives.

Our bug reporting is based on abstract states, which, as mentioned above, are denoted with extended pointer structures. An alarm reports a pointer assignment which may lead to memory leaks with the current extended pointer structures. *Heapcheck* produces these error abstract states and statements during the forward iterative analysis. We find this technique very useful in identifying true alarms.

Most of true memory leaks in heap-manipulating programs are caused by ignoring judging whether all the sub-level pointer fields are *null* when deallocating the heap cell pointed to by a pointer. In a few cases, the heap cell pointed to by a local pointer variable fails to be released at the return site. The examination of false positives indicates that they are due to three sources of imprecision:

- One source is due to lack of path information, such as numerical test, that could rule out the infeasible pointer assignments.
- Our abstraction may cause loss of precision and thus result in false positives. For instance, we are unable to identify the precise points-to relations more than two distances away from the current pointer variable. This imprecision might impact the precision of memory detection.
- Another source is that the heap cell might be escaped through parameters with complex pointer types, such as a parameter with a type pointer of pointer, during the interprocedural analysis. Currently, our tool do not deal with this case.

Our field and context-sensitive analysis has no false negatives when detecting heap-manipulating programs defined in Fig. 1. However, it also has some false negatives when analyzing C programs, due to ignoring some syntax of C programs. For example, we cannot consider the operator &, the complex pointer types (such as a pointer to a pointer), or the record type in C programs. So, we should analyze these syntax to improve the precision when analyzing C programs in the future.

B. Scalability

In this section, we measure the scalability of our on-demand modular memory detection for three large open-source projects (Apache, Python, and PostgreSQL). We show the time and memory used to detect memory leaks in Table VI. We also collect the total number of procedures and the max pointer number of each procedure, the mean and max number of the analysis per procedure. In the last column, the number of false positives and alarms are shown. We also got about a false positive rate of 20% when analyzing these three large programs.

Table VI
STATISTICS FOR LARGE PROGRAMS

Programs	Size (Kloc)	Preprocess Time (Sec)	Analysis Time (Sec)	Memory (MB)	Procedures	Max Pointer Number per Procedure	Mean Analysis Number per Procedure	Max Analysis Number of Procedures	Reported bugs
Apache	290	31.11	85.09	1148	3312	159	0.28	27	0/5
Python	320	70.22	314.34	2415	5810	260	0.49	17	5/17
PostgreSQL	880	89.05	265.81	1790	9726	199	21.57	225	19/111

The reason why the mean number of the analysis per procedure is very small when analyzing Apache and Python is that we skip analyzing the procedure whose scope does not have any read and write pointer operations, and in fact there are many of this kind of these procedures in these two projects. From Table VI, we know that the time and the memory used during the analysis are usually not related directly to the size of source code and the mean number of analysis per procedure. The reason is that the complexity of the fixpoint iteration algorithm is also dominated by the number of pointer variables and basic pointer assignments in the procedure.

Another phenomenon is that the mean number of the analysis per procedure usually increases with the size of the source code. We discover that the same procedure is possibly called more times in larger programs, and the procedure is encountered more times by our modular memory detection algorithm.

Overall, we believe that these results are encouraging. They suggest that modular heap abstraction for detecting memory leaks is both sufficiently precise to yield a low false positive rate, and sufficiently lightweight to scale to larger programs.

VI. RELATED WORK

In recent years, there has been a large body of research about detecting memory leak errors. We classify the related work mainly into field-insensitive and field-sensitive approaches.

A standard field-insensitive analysis detects whether every allocated cell is deallocated eventually based on a pointer alias analysis. It can be divided into flow-, path-, and context-sensitive and insensitive algorithm. For example, Cherem et al. [1] track the flow of values from allocation points to deallocation points using a guarded value-flow according to source-sink property and presents a practical interprocedural analyzer FastCheck for detecting memory leaks in C programs. Xu and Zhang [12] present a path-sensitive analysis to detect memory leaks based on the constraint solver CVC3. Recently, Xu and Zhang [13] present a more precise path-sensitive interprocedural memory leak detection based on memory state transition graph. Clouseau [14], [4] is a tool to describe those pointer variables responsible for deallocating heap cells based on pointer ownership, and constructs an ownership constraint system to detect memory leaks. But, none of these techniques can detect memory leaks efficiently for heap-manipulating programs, due to lack of field-sensitivity.

Jung and Yi [15] propose parameterized procedural summaries based an escape model in a practical, fully automatic static analyzer (called SPARROW). In addition, Saturn [16] reduces the condition of memory leak to a Boolean satisfiability problem, and then a SAT-solver is used to detect potential errors. Although these two approaches can distinguish different pointer fields when modeling the memory, they cannot represent the reachability relations between pointers.

Field-sensitive analysis involves both the reachability between heap cells and alias between pointer variables and fields. Sagiv et al. [2] propose Three-Valued logic (called TVLA) to divide all memory cells into finite equivalence classes by defining various core and instrumental predicates, and have proven the absence of memory leaks in various list-manipulating programs. However, it has not been used for detecting memory leaks for large programs. Hackett and Rugina [11] propose a region-based shape analysis with tracked locations based on reference counts to reason about individual heap locations and detect memory leaks in a set of three popular open-source C programs efficiently, but have high false positives. Wang et al. [3] have proposed a demand-driven memory leak detection algorithm based on the abstraction of points-to graph, and achieved relatively better balance between scalability and precision. Recently, separation logic [17] is used for reasoning locally about the memory errors, such as SpaceInvader [18] and Xisa [19]. SpaceInvader defines lists recursively in formulae of separation logic, and reasons heap states at each program pointer based on various operational and rewrite rules. It has been used for verifying the memory correctness in various device drivers. Xisa supports user-defined shape invariants about recursively-defined data structures to reason the property of programs under the abstraction interpretation framework [20]. These tools can also verify some other functional properties besides memory leaks. However, these analysis tools need users to give definitions and rules about various data structures in programs, and cannot achieve high scalability.

VII. CONCLUSIONS AND FUTURE WORK

We have presented a field and context sensitive algorithm for detecting memory leaks in heap-manipulating programs and implemented a prototype tool (*Heapcheck*). *Heapcheck* analyzes heap-manipulating programs modularly: it analyzes each procedure from top to bottom among the procedure call graph and produces a partial summary for each procedure under some abstract heap state in an on-demand way. The summary is parameterized by the abstract heap state based on member-access distances and alias bit-vector domain at its entry, so it can be instantiated at different call sites. We have used *Heapcheck* to analyze a set of the C benchmark programs, and got better precision compared with the existing work [11]. We have also used our approach to analyze three large open-sources and got satisfied scalability as expected.

In the future, we would like to adapt our modular heap abstraction to analyzing other memory-related program errors, such as *null* dereferencing, dangle pointer dereferencing and double frees. In order to improve the analysis precision, we would like to extend our modular heap abstraction to k-limited abstraction of distances. We also expect that our procedural summary about heap abstraction is able to be reused by other memory leak detectors or other static analysis tools.

Acknowledgment

This work is funded by National Natural Science Foundation of China (Nos. 61120106006, 61202120, 91118007), and the National High Technology Research and Development 863 Program of China (No. 2011AA010106).

References

[1] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 480–491.

[2] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '99. New York, NY, USA: ACM, 1999, pp. 105–118.

[3] J. Wang, X.-D. Ma, W. Dong, H.-F. Xu, and W.-W. Liu, "Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis," *J. Comput. Sci. Technol.*, vol. 24, pp. 347–356, March 2009.

[4] L. H. David and S. L. Monica, "Static detection of leaks in polymorphic containers," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 252–261.

[5] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv, "A relational approach to interprocedural shape analysis," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 5:1–5:52, February 2010.

[6] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 567–577.

[7] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu, "On inter-procedural analysis of programs with lists and data," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 578–589.

[8] P. Sotin and B. Jeannet, "Precise interprocedural analysis in the presence of pointers to the stack," in *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ser. ESOP'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 459–479.

[9] R. Rugina, M. Orlovich, and X. Zheng, "Crystal: A program analysis system for c," 2006. [Online]. Available: http://www.cs.cornell.edu/projects/crystal

[10] J. Uniejewski, "SPEC Benchmark Suite: Designed for Today's Advanced Systems," SPEC Newsletter 1, Tech. Rep. 1, Fall 1989.

[11] B. Hackett and R. Rugina, "Region-based shape analysis with tracked locations," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 310–323.

[12] Z. Xu and J. Zhang, "Path and context sensitive inter-procedural memory leak detection," in *Proceedings of the 2008 The Eighth International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 412–420.

[13] Z. Xu, J. Zhang, and Z. Xu, "Memory leak detection based on memory state transition graph." in *APSEC*. IEEE, 2011, pp. 33–40.

[14] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive c and c++ memory leak detector," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 168–181.

[15] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proceedings of the 7th international symposium on Memory management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 131–140.

[16] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Trans. Program. Lang. Syst.*, vol. 29, May 2007.

[17] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74.

[18] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, "Scalable shape analysis for systems code," in *Proceedings of the 20th international conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 385–398.

[19] B.-Y. E. Chang and X. Rival, "Relational inductive shape analysis," *SIGPLAN Not.*, vol. 43, pp. 247–260, January 2008.

[20] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252.