

# Enabling pointers in automatic program separation

No Institute Given

**Abstract.** This is abstract.

## 1 Introduction

We proposed a type-based approach that can automatically separate one program into two modules that can run separately.

Contribution: previous works do not solve the pointer issues well, we do.

## 2 Related work

## 3 Separation Framework

### 3.1 Program dependence graph

Program Dependence Graph(PDG) plays a key role in our separation framework. We develop a toolchain which can build instruction-grained PDGs for LLVM IRs. Our toolchain wraps up each IR instruction as a graph node, and adds some auxiliary data nodes to help represent the inter-procedural dependences. So, we have two kinds of node in a PDG:

(1) **instruction nodes**: an instruction node is used to represent an LLVM IR instruction;

(2) **data nodes**: used to represent: (1) global values; (2) auxiliary trees that called *parameter trees* for representing the parameter passing.

Accordingly, we have four kinds of edges:

(1) **control dependence edges**: used to represent a control dependence;

(2) **data dependence edges**: used to represent a data dependence;

(3) **parameter tree edges**: used to connect the corresponding data nodes to form a parameter tree that represents a function parameter or a global variable;

(4) **call edges**: used to connect the call site with the entry site of callee function. For indirect calls, we use a type-based approximation to connect the call site with all possible target callees with the same type.

Topologically, a PDG for a program with multiple functions is an inter-procedural extension of a program with only one function. So, accordingly, to build a PDG for a large program with multiple functions, we build a PDG for each function first, and then "glue" these small PDGs together. In other words, we first model all the intra-procedural dependences. Then, based on the caller-callee relationships, we build the corresponding *parameter trees*[1] to model the inter-procedural dependences. We also use the parameter trees to represent the global data structures, and construct the data dependences between the global variables and their caller functions. The benefits of using the parameter trees are: (1)we can clearly represent the memory layout of the function parameters(or global variables) with pointers;(2)we can easily represent the data flows in a parameter passing process between a caller function and a callee function;(3)we can construct the PDG modularly and accurately.

**Modeling intra-procedural dependences** The intra-procedural dependences consist of both control dependences and data dependences. The control dependences within a function can be captured easily with the classical post-dominator-tree-based algorithm[4]. Computing the data dependences is a little harder since we need to process two different kinds of data dependences separately. They are:

**definition-use dependence**: Given two instructions  $I_d$  and  $I_u$ , if a variable is defined in  $I_d$ , and then used in  $I_u$ , then we say there is a definition-use data dependence from  $I_d$  to  $I_u$ , noted as  $I_d \rightarrow I_u$ .

**flow dependence**: Given two instructions  $I_r$  and  $I_w$ , if  $I_r$  reads memory that was written by  $I_w$  (RAW: read after write), then we say there is a flow dependence from  $I_w$  to  $I_r$ , noted as  $I_w \rightarrow I_r$ .

An example of two kinds of data dependences is listed in Fig.2.

The definition-use dependences can be detected easily because LLVM uses the SSA form. So, in order to detect a definition-use data dependence, it suffices to retrieve each operand in an instruction  $I_u$ , and add a dependence from the instruction  $I_d$  that defines this operand to  $I_u$ .

In LLVM, only store instructions can write data into memory and only load instructions can read data from memory. By our definition, clearly the flow dependencies in LLVM only exist between the store instructions and load instructions. So, our algorithm for computing all of intra-procedural flow de-

```

A: %x = alloca i32, align 4
...
B: store i32 2, i32* %x, align 4

```

(a) a def-use dependence  $A \rightarrow B$

```

A: store i32 2, i32* %x, align 4
...
B: %0 = load i32* %x, align 4

```

(b) a flow dependence  $A \rightarrow B$

Fig. 2: An example of two kinds of data dependences

pendencies can be described as follows:

In procedure  $P$ , for each load instruction  $l_i$ , check each store instruction  $s_i$  in  $P$ , if  $l_i$  and  $s_i$  access the same memory location (e.g. %x in Fig.2 (b)),  $s_i \rightarrow l_i$  is a flow dependence. Clearly we make an approximation here since we ignore considering the sequence of the store/load instructions, which may lead to some redundant dependences being added.

Besides, an alias analysis step is required in finding the flow dependences. In our work, we use the DSA algorithm[5] to help compute the aliases because it is a kind of open-source, context-sensitive and field-sensitive algorithm that can be applied to large server applications[6].

Once we finished constructing the PDG for each procedure, we can start to model the inter-procedural dependences to construct the PDG for the whole program.

**Modeling inter-procedural dependences** The inter-procedural data dependency fully depends on the function parameter passing. So, the core part of modeling inter-procedural data dependency is how to represent the data dependency between the **actual parameters (arguments)** and the **formal parameters**. To achieve the field-sensitivity and represent the data layout of parameters in memory as precisely as possible, we use trees to model function parameters. In this paper, these trees are called *parameter trees*.

Once we construct the parameter trees for a parameter  $p$ , we can clearly represent the field-sensitive data flow in  $p$ 's parameter passing by connecting its corresponding tree nodes. If  $p$  is a pointer, we need to further model the dependency associated with the value that  $p$  points to.

We model 4 parameter trees for each parameter in total. They are:

**actual-in tree:** used to represent a new actual parameter before entering the callee function;

**actual-out tree:** used to represent the modified actual parameter when a call is finished;

**formal-in tree:** the callee analog of actual-in tree, used to represent the formal parameter before the callee execution;

**formal-out tree:** the callee analog of actual-out tree, used to represent the modified formal parameter after the callee execution.

For each function<sup>1</sup>  $f$ , we only need to build its formal-in and formal-out trees once because  $f$  only has one function body. On the other hand, the number of actual-in and actual-out trees depends on how many calls we have in a program. If  $f$  is directly called by another function  $q$ , we build the actual-in and actual-out trees at the call site, which is within  $q$ . So, each direct call for  $f$  matches exactly one pair of formal-in and formal-out trees.

As we mentioned before, using parameter trees can show the memory layout of variables very clearly. Each node in a parameter tree actually represents a piece of abstract memory whose size corresponds to the type associated with the node. Our algorithm for modeling a function parameter  $p$ 's parameter tree is shown in Algorithm 1. The algorithm is fully based on  $p$ 's type, noted as  $type(p)$ . First, we take  $type(p)$  as the tree root, if it is a basic type<sup>2</sup>, we leave  $type(p)$  as a single leaf node and stop, and in this case all  $p$ 's parameter trees shrink to single nodes accordingly; if  $p$  is a pointer, we insert  $type(*p)$  into the tree as the only child of  $type(p)$  and process  $type(*p)$  to build subtrees recursively; if  $type(p)$  is a structure type, we retrieve each field  $f_i$  in  $p$  and insert  $type(f_i)$  as a child of  $type(p)$ , and then use  $type(f_i)$  as the input to build a subtree. In Fig.4, we use a typical structure type to show how a type-based parameter tree looks like.

One problem about representing parameters with tree structures is some parameters may have recursive data structures(e.g. linked list) that can lead to parameter trees of infinite depth. In our framework, we use the  $k$ -limiting approach[3] to forcibly expand the tree to level  $k$  only to prevent infinite expansions. For simplicity, we let  $k=1$ . An example of building a 1-limiting parameter tree for a recursive data structure is listed in Fig.6.

---

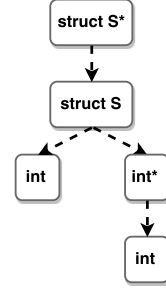
<sup>1</sup> Library functions(e.g. *scanf*, *printf*, *exit*...) will not be unfolded as subgraphs but represented as common nodes instead.

<sup>2</sup> In C, this means types formed by four basic type specifiers *char*, *int*, *float*, *double* and four modifiers *signed*, *unsigned*, *short* and *long*.

```

struct S{
    int i;
    int *p;
};
...
int a = 3;
struct S s = {2, &a};
struct S *sp = &s;
func(sp);

```



(a) sp: a pointer which points to a structure      (b) the type-based parameter tree of sp

Fig. 4: Example: a type-based parameter tree

---

**Algorithm 1:** building a parameter tree

---

Parameter type  $t := \text{base type} \mid t' * \mid \text{struct } \{t_1; t_2; \dots; t_n\}$ .  
 $\text{appendSubtrees}(\text{root}, t_1, t_2, \dots, t_n) : \text{append trees } t_1, t_2, \dots, t_n \text{ as the children of } \text{root}$ .  
 $\text{buildTree}(t)$

```

switch  $t$  do
    case  $t'$  *
        | let subtree = buildTree( $t'$ ) in appendSubtree( $t$ , subtree);
    end
    case struct  $\{t_1; t_2; \dots; t_n\}$ 
        | let subtree1 = buildTree( $t_1$ ) in
        | let subtree2 = buildTree( $t_2$ ) in
        | ...
        | let subtreen = buildTree( $t_n$ ) in
        | appendSubtree( $t$ , subtree1, subtree2, ..., subtreen);
    end
    otherwise /*  $t$  is a base type */
        | appendSubtree( $t$ , null); /* Leave  $t$  as a leaf node */
    end
endsw

```

---

We also use the tree data structure to model the global variables for better field-sensitivity. The algorithm for building the global variable trees is completely the same as building the function parameter trees.

**indirect function call** If a function  $f$  is called indirectly, it is impossible for us to exactly locate the position of callee function. Our solution is to approximately retrieve all possible candidate functions with the same type as  $f$  and build actual-in/out trees for each of them. Considering this is still a type-based matching, we have to do some preprocessing to guarantee that in the input programs for separation we have (1) no type cast to or from function pointer types and (2) no assembly code. This preprocessing is necessary because both of the two cases

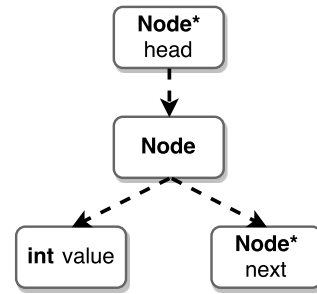
```

struct Node{
    int value;
    Node *next;
};

struct Node *head = NULL;
head = malloc(sizeof(struct Node));
buildlist(head);
...

```

(a) a linked list program



(b) an unfolded parameter tree

Fig. 6: An example of parameter trees with recursive data structure

will lead to the failure of our type-based indirect call matching. While once we finished the preprocessing, this type-based indirect call matching is compatible with the parameter trees construction.

**Example: encrypt a password** Let us see a real example. The toy program in Figure 2 includes a function with simple greeting, a function for password encryption and the main function. In line 2, we mark array `password` as a piece of sensitive data. Figure 3 is the PDG model for this toy program. We omit most of the simple intra-dependences, and only keep the most interesting part about how to use parameter trees to represent the inter-procedural data dependency.

```

1  static char username[20];
2  static char __attribute__((sensitive)) password[20];
3
4  int greeter(char *str){
5      if(str == NULL)
6          return 1;
7      printf("Welcome %s!"\n, str);
8      return 0;
9  }
10
11 int encrypt(char *str, int key){
12     unsigned int i;
13     for(i = 0; i < strlen(str); ++i){
14         str[i] = str[i] - key;
15     }
16     return 0;
17 }
18

```

```

19 int main() {
20     printf("Create your username: \n");
21     scanf("%s", username);
22     if(greeter(username) == 1)
23         printf("Invalid user!\n");
24     printf("Enter your password: \n");
25     scanf("%s", password);
26     printf("password:%s\n", encrypt(password,5));
27     return 0;
28 }

```

Figure 2. A toy C program to encrypt a password

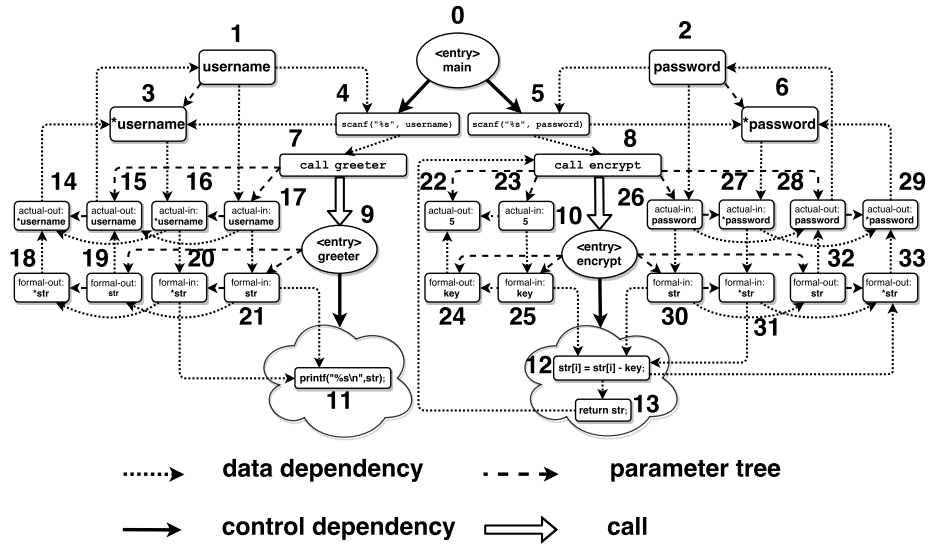


Fig. 7: The program dependence graph of our toy program.

In Figure 3, we construct two trees for global arrays `username` and `password` separately. Considering all arrays passed to functions are converted into pointers in C, we directly represent them with the form of `char*` instead of `char[]`, and by doing this we can show the data dependency better. As you can see on the upper-left part, node 1, which is the root of the global variable tree for `username`, represents the pointer that points to the `username` string; while node 3 represents the content that pointer points to, i.e., the string itself. By algorithm 1 clearly node 3 is the child of node 1, and also a leaf since we take string as an atomic type just as we mentioned before. In the main function, we need to set up a def-use data dependence edge ( $1 \rightarrow 4$ ), and a flow dependence data

dependence edge ( $4 \rightarrow 3$ ) since the operation `scanf` uses the pointer "username" first and then modifies the value that pointer "username" points to.

Next, we have a data dependence edge ( $4 \rightarrow 7$ ) because we take the call operation (node 7) as a kind of use of "username". We use a call edge ( $7 \rightarrow 9$ ) to connect the call site (node 7) in the caller function (main) and the entry node of the callee function (greeter). The function body of greeter is abstracted as a cloud-like region, connected with its entry node through an abstract control dependence edge ( $9 \rightarrow 11$ ).

Now we can use parameter trees to represent the parameter passing between main and greeter. We let the call node (node 7) dominate the actual-in (node 16, node 17) and actual-out (node 15, node 14) trees, and the entry node of greeter (node 9) dominate the formal-in (node 21 and 20) and formal-out (node 19 and 18) trees. There are two kinds of data flow when the parameter is a pointer: one is the data flow of the pointer itself, and the other is the data flow of the value that pointer points to (pointee). On this graph, the data flow of the pointer itself (username) can be represented by a node sequence  $1 \rightarrow 17 \rightarrow 21 \rightarrow 19 \rightarrow 15 \rightarrow 1$ , which is a loop actually. Accordingly, the data flow of the pointee of "username" can be represented with the sequence  $3 \rightarrow 16 \rightarrow 20 \rightarrow 18 \rightarrow 14 \rightarrow 3$ . However, there is only memory read (node 11) but no write operation in function greeter, so we only have two data dependence edges ( $21 \rightarrow 11$ ) and ( $20 \rightarrow 11$ ), which both come from the caller to callee.

The PDG construction for (encrypt) is similar. The first parameter is just an integer so each parameter tree is simply constructed as a single node. There is a data dependence edge ( $25 \rightarrow 12$ ) since the formal parameter "key" is read by statement "`str[i] = str[i] - key;`". One important data dependence edge in encrypt is ( $12 \rightarrow 33$ ), which represents that we update the buffer that the pointer "str" points to inside the callee function. Finally, we have a data dependence edge ( $13 \rightarrow 8$ ) to represent the data flow of return value.

### 3.2 PDG-based separation algorithm

After we have a PDG for a program, we can separate this program into two slices by partitioning its PDG into two cuts. Simply speaking, we first annotate some sensitive data in the program with the `__attribute__` grammar (see line 2 in Figure 2) and take the corresponding graph nodes as the source, and then start from the source nodes to do a greedy coloring along the data dependence edges on the graph. After the coloring finished, we get two sets of graph nodes, one consists of all the colored nodes and the other consists of all the uncolored nodes. In the set of the colored nodes, we check each node one by one, if it corresponds to a global variable, we directly mark that global variable as sensitive, and if it corresponds to a statement within a function, we mark that function as sensitive. Ultimately, all the sensitive functions and global variables will be



separated from the original input program as an independent slice, and we call this slice a "sensitive slice". On the other hand, the left global variables and functions that are non-sensitive form another slice, and we call it "non-sensitive slice" accordingly. A detailed description of our separation is in Algorithm 2.

---

**Algorithm 2:** PDG-based program separation

---

*Program\_Separate*(PDG, *function\_set*, *global\_set*, *source\_nodes*, *colored\_nodes*, *Queue*)

**Input :**

*PDG*( $N, E$ ) — a directed graph which represents the whole input program

$P$ , where  $N$  is the set of nodes and  $E$  is the set of edges;

*function\_set* — a set which contains all the functions in  $P$ ;

*global\_set* — a set which contains all the global variables in  $P$ ;

*source\_nodes* — a set which contains all the nodes that marked as "sensitive" on the PDG;

*colored\_nodes* — a set which contains all the colored graph nodes during the coloring;

*Queue* — a priority queue used for the greedy graph coloring;

**Output:**

*function\_set* — a set with all sensitive functions colored and non-sensitive functions uncolored;

*global\_set* — a set with all global variables colored and non-sensitive global variables uncolored.

*Initialization :*

*colored\_nodes*  $\leftarrow$  *source\_nodes*;

*current\_node*  $\leftarrow$  NULL ;                      /\* the node we are working on \*/

**foreach** node  $n$  in *source\_nodes* **do**

*Push*(*Queue*,  $n$ );

    Color  $n$ ;

*Insert*(*colored\_nodes*,  $n$ );

**end**

**while** *Queue* is not empty **do**

*current\_node*  $\leftarrow$  *Pop*(*Queue*);

**foreach** successor node  $n_s$  of *current\_node* **do**

**if**  $s_n$  is uncolored **and**  $(n \rightarrow n_s)$  is a data dependence edge **then**

            Color  $n_s$ ;

*Insert*(*colored\_nodes*,  $n_s$ );

*Push*(*Queue*,  $n_s$ );

**end**

**end**

**end**

**foreach** node  $c$  in *colored\_nodes* **do**

    Color  $c$ 's corresponding function  $f_c$  in *function\_set*;

    Color  $c$ 's corresponding global variable  $g_c$  in *global\_set*;

**end**

---

## 4 Inter-module communication

## 5 Evaluation

## 6 Conclusions

## References

1. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM Trans. Programming Languages and Systems*, vol. 12, no. 1, pages 35-46, 1990.
2. J. Graf. Speeding up context-, object- and field-sensitive SDG generation. In *Proc. 9th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, pages 105-114, 2010.
3. D. Liang, M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358-367, 1998.
4. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, 1987.
5. C. Lattner, A. Lanharth, V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of PLDI*, 2007.
6. I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.