Graph-based automatic program separation

Shen Liu

(Version 1.5, last modified 08/07/2016)

Abstract

Type conversion between C and protocol buffer is an important issue in our project. When the separation is done and the RPC tool begins working, we must automatically restore all the arguments for each RPC function in the receiver process, which means the function parameter types need to be transmitted between two processes fully and exactly. Unfortunately, the type system that protocol buffer supports is quite weak. To make our project automatically run in the end, we have to design a type conversion protocol to let protocol buffer automatically convert some advanced C types(e.g. pointer) into protocol buffer types.

1 Background

We use gRPC, which is fully based on google protocol buffer, to deal with RPC issues in our project. In gRPC, a C type must be packed to protocol buffer "`message`" type in a .proto file(IDL file) for further transmission. For example, if you have a C function `int foo(int x)` which needs to be called remotely, then in your .proto file, the argument type `int` can be packed in protocol buffer as follows:

```
message M{                    message M{
  int64 x=1;       or           int32 x=1
}                                        }
```

Next, protocol buffer will automatically generate a group of read/write APIs for each message. Here is an API for x's value assignment:

```
void set_x(::google::protobuf::int32 value);
```

and an API for getting the value of x:

```
inline ::google::protobuf::int32 M::x() const {
  return x_;
}
```

Here is a more complex C-protobuf type conversion sample:

```
typedef struct{                              message Circle{
  int x;                                       message Point
      {
  int y;                                         int64 x=1;
}Point;                                          int64 y=2;
                           --->                }
typedef struct{                                double radius
   =1;
  Point center;                              }
  double radius;
}Circle;
```

Our project can automatically finish this conversion for all scalar types and simple composite types as "Circle". However, when parameter types become more and more complex, especially for those structures with multi-level pointers, generating a correct .proto file automatically as before will be a real challenge. To achieve this goal, a possible way is designing a type-conversion protocol to make our project work more intelligently. Simply speaking, for any C type input, first we use such a protocol to convert it into an integer array(encoding), and then construct the "message" type in .proto. On the receiver side, we do array parsing to restore the original C types(decoding) instead of parsing the complex .proto file.

2 Type system and encoding/decoding rules

In this draft we only use a small subset of C type system to show how the encoding/decoding idea works. Here is how our toy type system looks like:

```
Type t := int | t* | struct {t1; t2; ... ;tn}
              | tname S
```

Any pair of form (`type,value`) based on this type system will be encoded as an array of bytes(`bytes[] lst`), and the first byte(see table 1) in this array denotes what type this array corresponds to.

lst[0]	type
0	int
1	pointer
2	struct $\{t_1;t_2;...;t_n\}$
3	tname S

Table 1: type mapping rules

As we can see from Table 1, any encoding/decoding operation related to type `tname S` requires knowing the associative type `struct{...}` of `tname S`. In our framework, we use a name-type mapping table(e.g. table 2) to map each name string which represents a struct to its corresponding `struct` type.

name	struct
S1	struct {int;int;}
S2	struct {int;int*;}
S3	struct {int;int*; S1*;}
...	...

Table 2: A name–type mapping table example

Besides, in each round for encoding, we also use an auxiliary table called pointer table to record each pointer value that ever appeared. By doing this we can identify some complex function arguments(e.g. circular linked list).

Once we have such auxiliary tables, we can easily construct the encoding/decoding rules for our type system as follows:

```
Basic value conversion functions:

intToBytes(int): convert an integer to a byte string.

symbolToBytes(S): convert a symbol S to a byte string.

ptrToBytes(int): convert a pointer address(int) to a
   byte string.

bytesToInt(bytes[]): convert a byte string to an
   integer.

bytesToSymbol(bytes[]): convert a byte string to a
   symbol.

bytesToPtr(bytes[]) convert a byte string to a
   hexadecimal integer.

dereference(int): return the value that a pointer
   points to.

getTypeFromTable(S): look up S in the mapping table
   and return its
                    associative type.

Definition encode: (type,value) (t,v) -> bytes[]
  match t with
  | int     => 0::intToBytes(v)
  | t*      => 1::ptrToBytes(v)::encode(t,dereference(v
     ))
  | struct {(t1,v1);...;(tn,vn)}
            => 2::intToBytes(n)::encode(t1,v1)::...::
               encode(tn,vn)
  | tname S
            => 3::symbolToBytes(S)::encode(
               getTypeFromTable(S),v)
  end.


Definition decode bytes[] lst =
  match lst[0] with
    | 0 => ((int, bytesToInt(lst[1...4])), lst+5)

    | 1 => let ((t1,dereference(v)), l1) = decode (lst
       +5) in ((t1*, v), l1)
            //v = bytesToPtr(lst[1...4])

    | 2 => let n = bytesToInt(lst[1]) in
            let ((t1,v1),l1) = decode (lst+5) in
        let ((t2,v2),l2) = decode l1 in
            ...
           let ((tn,vn),ln) = decode l_{n-1} in
           (struct {(t1,v1);(t2,v2);...;(tn,vn)}, ln)

    | 3 => let S = bytesToSymbol(lst[5...5+length(S)
       -1]) in
                                /*length(S) =
                                   bytesToInt(lst
                                   [1...4])*/
           decode(lst+offset)   /* offset = 1+4+length(S
              )*/
  end
```

Now consider a circular linked list example:

```
typedef struct Node{
 int val;
 Node* next;
}Node_t;

Node_t *head = (Node_t*) malloc(sizeof(Node_t));
    //head: 0x0004
Node_t *tail = (Node_t*) malloc(sizeof(Node_t));
    //tail: 0x0008

head->val = 10;
head->next = tail;

tail->val = 20;
tail->next = head; // circular linked list
```

Assume that we want to send this circular linked list from sender to receiver, then the encode/decode process is as follow:

```
encode(Node_t *head, 0x0004)

= 1::ptrToBytes(0x0004)
   ::encode(Node_t, dereference(0x0004))
   /* dereference(0x0004) = {10,0x0008} */

   (pointer table: {0x0004})

= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
      ::encode(getTypeFromTable(Node_t),{10,0x0008})

   (pointer table: {0x0004})

= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
   ::encode(struct Node{int, struct Node*},{10,0x0008
      })

   (pointer table: {0x0004})

= 1::ptrToBytes(0x0004)
   ::3::symbolToBytes(Node_t)
      ::2::intToBytes(2) /*two fields*/
         ::encode(int,10)
         ::encode(Node_t*,0x0008)
```

```
    (pointer table: {0x0004})


= 1::ptrToBytes(0x0004)
    ::3::symbolToBytes(Node_t)
        ::2::intToBytes(2) /*two fields*/
            ::0::intToBytes(10)
            ::1::ptrToBytes(0x0008)
                ::3::symbolToBytes(Node_t)
                    ::encode(getTypeFromTable(Node_t),
                       dereference(0x0008))

    (pointer table: {0x0004, 0x0008})


= 1::ptrToBytes(0x0004)
    ::3::symbolToBytes(Node_t)
        ::2::intToBytes(2) /*two fields*/
            ::0::intToBytes(10)
            ::1::ptrToBytes(0x0008)
                ::3::symbolToBytes(Node_t)
                    ::encode(struct Node{int, struct Node
                       *}, {20,0x0004})

    (pointer table: {0x0004, 0x0008})



= 1::ptrToBytes(0x0004)
    ::3::symbolToBytes(Node_t)
        ::2::intToBytes(2) /*two fields*/
            ::0::intToBytes(10)
            ::1::ptrToBytes(0x0008)
                ::3::symbolToBytes(Node_t)
                    ::2::intToBytes(2) /*two fields*/
                        ::0::intToBytes(20)
                        ::1::ptrToBytes(0x0004)
                    ...
                    (0x0004 is in pointer table already,
                        stop here)

    (pointer table: {0x0004, 0x0008})
```

0x0004 appears again, which means there must be a circle, to remember all pointer values we need an extra data structure for pointer storage and comparison.

The decoding process for the generated `bytes[] lst` can be illustrated as follows:

```
decode(bytes[] lst)
= decode(1::ptrToBytes(0x0004)
            ::3::symbolToBytes(Node_t)
                ::2::intToBytes(2)
                    ::0::intToBytes(10)
                    ::1::ptrToBytes(0x0008)
                        ::3::symbolToBytes(Node_t)
                            ::2::intToBytes(2)
                                ::0::intToBytes(20)
                                ::1::ptrToBytes(0x0004))

= decode(3::symbolToBytes(Node_t)
            ::2::intToBytes(2)
                ::0::intToBytes(10)
                ::1::ptrToBytes(0x0008)
                    ::3::symbolToBytes(Node_t)
                        ::2::intToBytes(2)
                            ::0::intToBytes(20)
                            ::1::ptrToBytes(0x0004)) in ((t1
                                *, 0x0004), null)

= decode(2::intToBytes(2)
            ::0::intToBytes(10)
            ::1::ptrToBytes(0x0008)
                ::3::symbolToBytes(Node_t)
                    ::2::intToBytes(2) /*two fields*/
                        ::0::intToBytes(20)
                    ::1::ptrToBytes(0x0004)) in ((
                         Node_t*, 0x0004), null)
```

Once we have the name `Node_t`, we can directly look up and retrieve its associative struct type in the mapping table, and then restore a new list on the receiver side like:

```
Node_t* head = (Node_t*) malloc(sizeof(Node_t));
```

During the left decoding process, we set up a table(see table 3) which records the values on both sender and receiver sides for each pointer, to help us conveniently restore the sender side point-to relationships in the receiver side.

pointer	value in sender	value in receiver
head	0x0004	0x0012
tail	0x0008	0x0016
head→next	0x0008	0x0016
tail→next	0x0004	0x0012
...	...	...

Table 3: Pointer values in both sender and receiver

For example, assume in the receiver side pointer `head` equals to 0x0012, and its sender counterpart equals to 0x0004, then we have an entry like head–0x0004–0x0012 in our table. After we finished the restoration, we can also use this table to check whether the old pointer-to relationships are maintained correctly.

Let's continue decoding the bytes above, we have:

```
= decode(2::intToBytes(2)
          ::0::intToBytes(10)
          ::1::ptrToBytes(0x0008)
             ::3::symbolToBytes(Node_t)
                ::2::intToBytes(2)
                   ::0::intToBytes(20)
                   ::1::ptrToBytes(0x0004))

= decode(3::symbolToBytes(Node_t)
            ::2::intToBytes(2)
               ::0::intToBytes(20)
               ::1::ptrToBytes(0x0004))
  head->val = 10;
  head->next = (Node_t*)malloc(sizeof(Node_t)); //
     assume head->next = 0x0016
  ("Node_t" can be directly retrieved from the name-
     type mapping table)

  Pointer table:
  Pointer        sender    receiver
  head           0x0004     0x0012
  head->next     0x0008     0x0016

= decode(empty)

  head->next->val = 20;
  head->next->next = (Node_t*)malloc(sizeof(Node_t))
     ;//assume 0x0020

  Pointer table:
  Pointer              sender    receiver
  head                 0x0004     0x0012
  head->next           0x0008     0x0016
  head->next->next     0x0004     0x0020(wrong value!)
```

By checking the pointer table we know pointer "`head→next→next`" should be an alias of pointer "`head`". So the new allocated value for "`head→next→next`" should be updated immediately from 0x0020 to 0x0012 as follows:

```
= decode(empty)

  head->next->val = 20;
  head->next->next = 0x0012;
  or
  head->next->next = head;

  Pointer table:
```

```
Pointer               sender    receiver
head                  0x0004    0x0012
head->next            0x0008    0x0016
head->next->next      0x0004    0x0012
```

3 Separation Framework

3.1 Program dependence graph

Program Dependence Graph(PDG) plays a key role in our separation framework. To build a PDG for a large program with multiple functions, we first build a PDG for each function, which means we first model all the intra-procedural dependences. Then, based on the caller-callee relationships, we build corresponding parameter trees[1] to model the inter-procedural dependences. We also use tree structures to represent the global data structures, and construct the data dependences between the global variables and their caller functions.

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 two kinds of auxiliary nodes to help represent the inter-procedural dependences. So, we have three kinds of node in a PDG in total:

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

(2) **entry node**: used to represent the entry point of a function;

(3) **parameter tree node**: used to build the parameter trees.

Accordingly, we have four kinds of edges:

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

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

(3) **parameter tree edge**: used to connect the parameter tree nodes;

(4) **call edge**: used to connect the call site with the entry node 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.

3.1.1 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[2]. Computing the data dependences is a little harder since we need to process two different kinds of data dependences separately, the **def-use dependences** and the **flow dependences**(**true data dependences**). The def-use dependence represents the data dependency between two statements with the definition-use association(DUA,see Fig 1.a), and the flow dependence represents the dependency between memory read and write operations(read-after-write,see Fig 1.b). In this paper, both the **anti-**dependence (write-after-read) and the **output-**dependence(write-after-write) are not in our consideration, and we only compute the flow dependence.

```
int x = 0;     /*define x*/
...
int y = x + 1; /*use x*/
...
```

(a) def-use dependence

```
int a = 3;
int *p = &a;
*p = 5;     /*memory write*/
int b = a;  /*memory read*/
```

(b) flow dependence

Figure 1: An example of two kinds of data dependences

The definition-use dependences can be detected easily by finding all variables in a statement and then adding a dependence between it and the statement that defines the variable. For flow dependencies, since they only exist between the memory read and write statements, our algorithm for computing all of intra-procedural flow dependencies can be described as follows:

Step 1: Check each memory operation in the program, if it is a memory read, do Step 2; otherwise, continue to check the next memory operation.

Step 2: For each memory read r_i, check each memory write operation w_i in this function, if r_i and w_i access the same memory location, $r_i \rightarrow \mathrm{w}_i$ is a flow dependence. The alias analysis algorithm is required here, in our work, we use the DSA algorithm[3] 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[4].

3.1.2 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**. If the function parameter is a pointer, we need to further model the dependency associated with the value that the pointer points to. To represent the data layout of parameters in memory as precisely as possible, we model 4 parameter trees for each parameter:

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

actual-out tree: used to represent the 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 formal parameter after the callee execution.

For each function[1] 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 actual-in and actual-out trees. However, if 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, which means some redundant actual trees will be built inevitably.

The process of building a parameter tree is shown in Algorithm 1. Each incoming parameter p corresponds to a root. If p is a pointer, we insert the value that p points to into the tree as the child of p and process this value to build subtrees recursively; otherwise if p is a struct, we retrieve each field f_i in p and insert f_i as a child of p, and then use f_i as input to build a subtree; if p is neither a pointer nor a struct, which means it must be a variable of atomic type[2](e.g. int, float, char...), p will be inserted as a leaf. Actually the parameter tree for an atomic variable always shrinks to a single node.

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. To solve this problem Graf[1] proposes an algorithm to unfold the recursive data structures. In our framework, we directly borrow that algorithm to help build parameter trees. A example of building parameter trees with recursive data structures is listed in Figure 2.

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.

[1]Library functions(e.g.scanf, printf, exit...) will not be unfolded as subgraphs but represented as common nodes.

[2]For simplicity, we take `string` as an atomic type also.

Algorithm 1: building a parameter tree

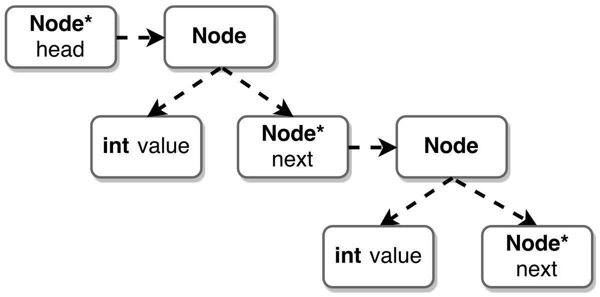
```
$BuildTree(p,T)$
Input  : $p$– parameter; $T$– a pointer which points to the tree root
Output: a constructed tree represented by $T$

$T \rightarrow root = p;$
if $p$ is a pointer then
    $T \rightarrow root \rightarrow firstchild = dereference(p)$ ;            /* insert *p */
    $BuildTree(dereference(p), T \rightarrow root \rightarrow firstchild);$
else
    if $p$ is a struct then
        for each field $f_i$ in $p$ do
            $T \rightarrow root \rightarrow appendchild(f_i);$
            $BuildTree(f_i, T \rightarrow root \rightarrow getchild(i));$
        end
    else
        ;                                            /* atomic parameter */
    end
end
```

```
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

Figure 2: An example of parameter trees with recursive data structure

3.1.3 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

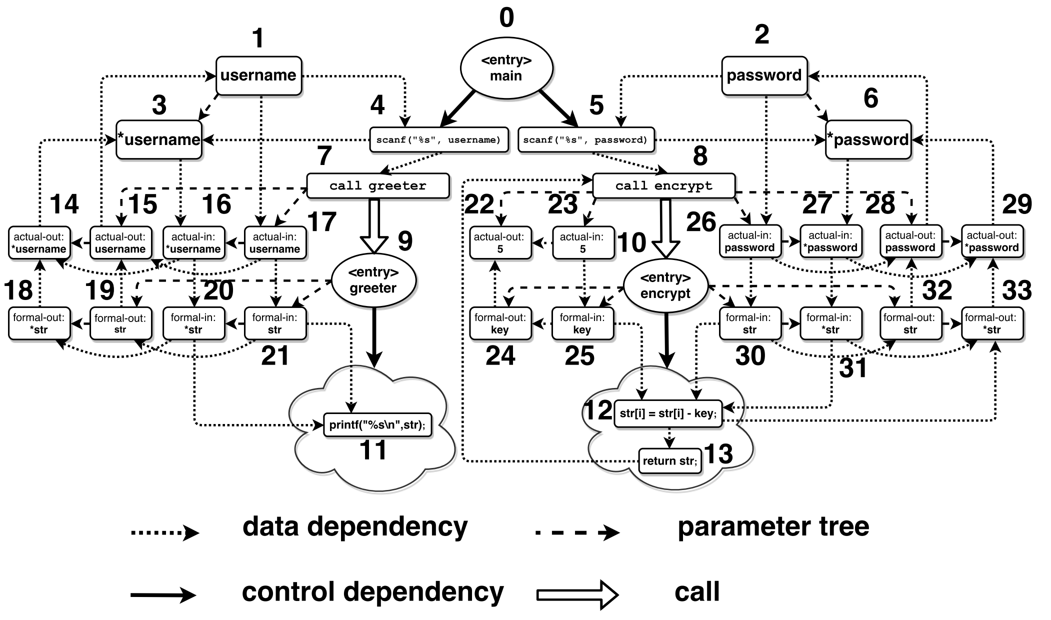


Figure 3: The program dependence graph of our toy program.

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 → 4), and a flow dependence data dependence edge (4 → 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 → 7) because we take the call operation(node 7) as a kind of use of "username". We use a call edge(7 → 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 → 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 → 17 → 21 → 19 → 15 → 1, which is a loop actually. Accordingly, the data flow of the pointee of "username" can be represented with the sequence 3 → 16 → 20 → 18 → 14 → 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 → 11) and (20 → 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 → 12) since the formal parameter "key" is read by "`str[i] = str[i] - key;`". One important data dependence edge in `encrypt` is (12 → 33), which represents that we update the buffer that the pointer "str" points to inside the callee function. Finally, we have a a data dependence edge (13 → 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 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 colored nodes and the other consists of all uncolored nodes. In the set of 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 sensitive global variables will be separated from the original program as an independent slice, and we call it "sensitive slice". On the other hand, the left global variables and functions that are non-sensitive form another slice, named as "non-sensitive slice" for simplicity.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.