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 from I_r, noted as $I_w \rightarrow I_r$.

An example of two kinds of data dependences is listed in Fig.1.

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. 1: 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.1 (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[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 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[2], 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.2, we use a typical structure type to show how a type-based parameter tree looks like.

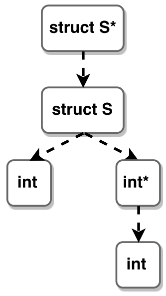
One problem about representing parameters with tree structures is some parameters may have recursive data structures(e.g. linked list) that can lead to parameter trees of infinite depth. In our framework, we use the $k-$limiting approach[3] to forcibly expand the tree to level k only to prevent infinite 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.3.

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

[2] 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. 2: Example: a type-based parameter tree

Algorithm 1: building a parameter tree

Parameter type t := *base type* | $t^{'}*$ | *struct* $\{t_1;\ t_2;\ ...\ ;\ t_n\}$.
appendSubtrees$(root,\ t_1,\ t_2,\ ...\ ,\ t_n)$: append trees $t_1, t_2, ... , t_n$ as the children of *root* .
buildTree(t)
switch t **do**
  case $t^{'}*$
    let *subtree* = *buildTree*$(t^{'})$ in *appendSubtrees*$(t,\ subtree)$;
  end
  case *struct* $\{t_1;\ t_2;\ ...\ ;\ t_n\}$
    let $subtree_1$ = *buildTree*(t_1) in
    let $subtree_2$ = *buildTree*(t_2) in
    ...
    let $subtree_n$ = *buildTree*(t_n) in
    appendSubtrees$(t,\ subtree_1,\ subtree_2,\ ...\ ,\ subtree_n)$;
  end
  otherwise `/* t is a base type */`
    appendSubtrees$(t,\ null)$; `/* Leave t as a leaf node */`
  end
endsw

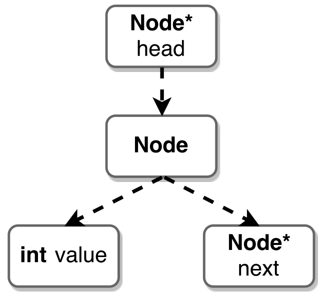
We also use the tree data structure to model the global variables for better field-sensitivity. The algorithm for building the global variable trees is completely the same as building the function parameter trees.

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. 3: 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 4 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.

In Figure 5, 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 ($9 \rightarrow 11$).

```c
static char username[20];
static char __attribute__(sensitive) password[20];

int greeter(char *str){
  if(str == NULL)
   return 1;
  printf("Welcome %s!"\n, str);
  return 0;
}

int encrypt(char *str,int key){
  unsigned int i;
  for(i = 0; i < strlen(str); ++i){
    str[i] = str[i] - key;
  }
  return 0;
}

int main(){
  printf("Create your username: \n");
  scanf("%s", username);
  if(greeter(username) == 1)
    printf("Invalid user!\n");
  printf("Enter your password: \n");
  scanf("%s", password);
  printf("password:%s\n", encrypt(password,5));
  return 0;
}
```

Fig. 4: A toy C program to encrypt a password.

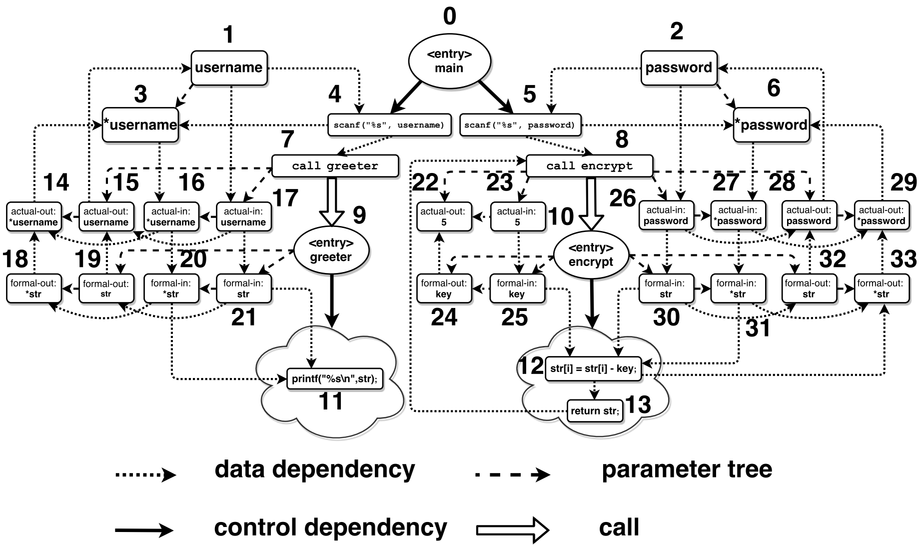


Fig. 5: The program dependence graph of our toy program.

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 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 4) 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 ← source_nodes;
current_node ← 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 ← Pop(Queue);
    foreach successor node n_s of current_node do
        if s_n is uncolored and (n → 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 after separation

4.1 Basic workflow

The slices we have by the graph-based separation algorithm above are just two sets of functions and global variables. We call these slices "*raw slices*" because these code can not be executed directly for missing the necessary context. For example, in the above encryption program, if we want to run the sensitive slice with function *encrypt* and *main*, we will be stuck at line 21 and 22 since the variable *username* and function *greeter* are defined in another slice.

Clearly, our goal of doing program separation is not only to generate some raw slices but also to make these slices can run as real modules, and then we can use other techniques to protect those sensitive modules specially. So, once we successfully separate a program into two raw slices, we need to wrap up each raw slice with some context wrapper functions to make it become a module that can run independently. Further, we can use RPC (Remote Procedure Call) to accomplish the inter-module communication to make the new modular program work well as if it has never been separated before.

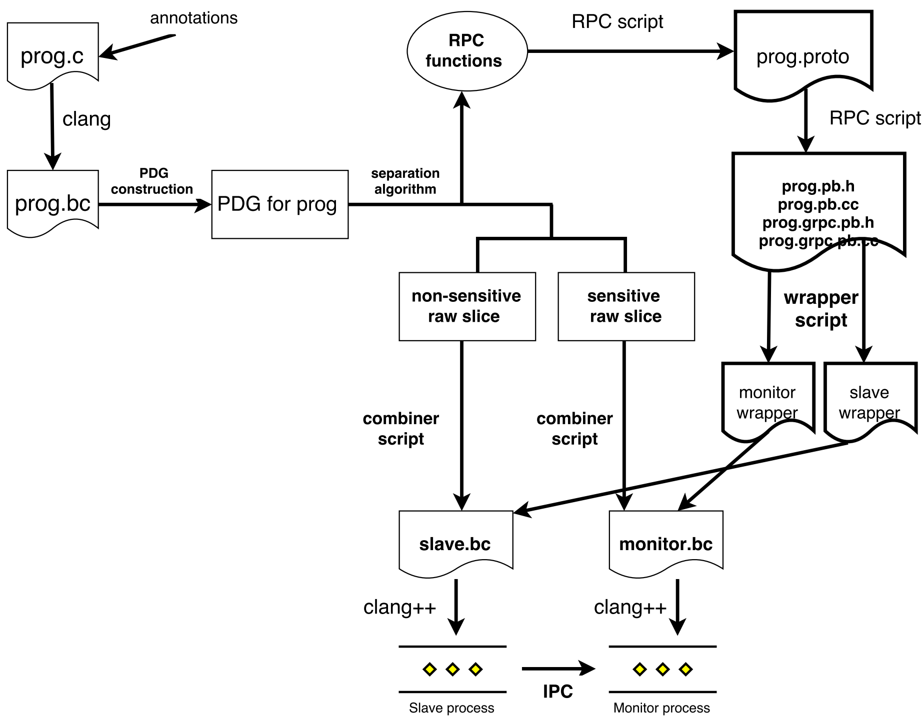


Fig. 6: the automatic program separation framework

We show the whole working process of our separation framework in Fig.6. At the beginning, we add some annotations in the input C program noted as `prog.c`, and then compile the annotated C program into an LLVM IR file noted as `prog.bc`. Then, we construct the program dependence graph for `prog.bc`. By our separation algorithm, we can separate `prog.bc` into two raw slices, one is sensitive while another is non-sensitive. During the separation process, we record all the functions that need to be called remotely by the RPC framework in future, and automatically generate a bunch of interface files with our RPC scripts. Based on these generated interface files, we can use a wrapper script to generate two wrapper files of LLVM IR form, noted as the monitor wrapper and the slave wrapper separately. Next, we use a combiner script to link the monitor wrapper and the sensitive raw slice to get a runnable new IR called `monitor.bc`, and link the slave wrapper and the non-sensitive raw slice to get an IR called `slave.bc`. Finally, we compile `monitor.bc` and `slave.bc` with clang++ to create two processes, one is called slave and the other is called monitor. The inter-process communication between the slave process and the monitor process is also implemented by our RPC framework.

4.2 RPC framework

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

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

or

```
message M{
  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{
  int x;
```

```
message Circle{
  message Point{
```

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

Our RPC script(see Fig.6) 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. Besides, for pointer parameters, we have to do marshalling and demarshalling work for the data that a pointer points to simultaneously. To achieve this goal, a feasible way is to design a type-conversion protocol to make our interface generator work more intelligently. Simply speaking, for any C type input, first we use such a protocol to convert it into an byte array (encoding), and then construct the "message" type in the `.proto` file. On the receiver side, we do array parsing to restore the original C types(decoding) instead of parsing the complex `.proto` file.

Type system and encoding/decoding rules For simplicity, 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 a byte array (`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 as 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:

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 table and return its struct type.

And our encoding/decoding algorithm can be described as follows:
(*Suffer from some latex form errors so just put the listing, later on i will make it more readable with the algorithm style *)

```
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)

    | 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
           decode(lst+offset) /* offset = 1+4+length(S)*/
  end
```

(*Should we add an example here or in the appendix? This is the simplest pointer example i can imagine for illustrating our marshalling/demarshalling algorithm, but it is still too long*)

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
```

Buffer size computation

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.