PDG Edge Classification

New Requests

- Use TYPE_SUBTYPE for node and edges.
- SUBTYPEs are disjointed; union of SUBTYPES is TYPE.
- create nodes for annotation instruction (must be a sink node).
  - INST_ANNO_LOCAL
  - INST_ANNO_GLOBAL
- Create edges for ANNO_FUNC and ANNO_VAR.
  - ANNO_FUNC: from function entry node to annotation node.
  - ANNO_VAR: from variable node to annotation node.
- CONTROLDEP Edges
  - CONTROLDEP_CALLINV
  - CONTROLDEP_CALLRET
  - CONTROLDEP_ENTRY
  - CONTROLDEP_BR
  - CONTROLDEP_IND_BR ?
  - Every instruction node should be reachable from the FUNC_ENTRY node through CONTROLDEP_* edges.
  - **Assertion**: every instruction in the function body that are not in control blocks is directly reachable from the FUNC_ENTRY node through CONTROLDEP_* edges. Control dependence edge preserves the control structure inside a function when there is no goto statement in the program.
- Maintain version for the PDG specification and separated PDG implementation.
- PDG capture C at the moment. Need more changes on handling C++ in future.
- For unhandled nodes and edges, label them with TYPE_OTHERNODE, TYPE_OTHEREDGE.
  - if an attributes can apply to more than one subtypes, make it an attribute of the type (a field).
- GLOBALVAR_GLOBAL / GLOBALVAR_LOCAL
- Remove "sensitive" info from the annotation node.
- In C program, add function pointer (invoke) and indirect branch examples.
- In C program, add two level pointer example.

Edges Types

The followings are the edges types defined in our current PDG implementation.

```
enum class EdgeType
{
  CALL,
  DATA_RET,
  CONTROL,
  DATA_DEFUSE,
  DATA_RAW,
  DATA_READ,
  DATA_ALIAS
  PARAMETER_IN,
  PARAMETER_OUT,
  PARAMETER_FIELD
};
```

> add annotation edges

Node Types

```
INST, (INST)
FORMAL_IN,
FORMAL_OUT,
ACTUAL_IN,
ACTUAL_OUT,
RETURN, (INST_RET)
FUNC_ENTRY,
GLOBAL_VAR,
CALL (INST_CALL)
```

> add Annotation node type

Example Program

We use the following example to demonstrate all edges in PDG.

```c
#include <string.h>

 char __attribute__((annotate("sensitive"))) *key ;
 char *ciphertext;
 unsigned int i;

void greeter (char *str, int* s) {
     char* p = str;
     printf("%s\n", p);
     printf(", welcome!\n");
     *s = 15;
 }

 void initkey (int sz) {
     key = (char *) (malloc (sz));
     // init the key randomly; code omitted
     for (i=0; i<sz; i++) key[i]= 1;
 }

int encrypt (char *plaintext, int sz) {
     ciphertext = (char *) (malloc (sz));
     for (i=0; i<sz; i++)
         ciphertext[i]=plaintext[i] ^ key[i];
     return sz;
 }

 int main (){
     int age = 10;
     char __attribute__((annotate("sensitive"))) username[20], text[1024];
     printf("Enter username: ");
     scanf("%19s",username);
     greeter(username, &age);
     printf("Enter plaintext: ");
     scanf("%1023s",text);

     initkey(strlen(text));
     int sz = encrypt(text, strlen(text));
     printf("Cipher text: ");
     for (i=0; i<strlen(text); i++)
         printf("%x ",ciphertext[i]);
     printf("encryption length: %d", sz);
     return 0;
 }
```

Call Edge

Description:

CALL edge connects a call site with the function entry point of callee. It indicates the control flow transition from the caller to callee.

Line: Red dash line with label {CALL}.

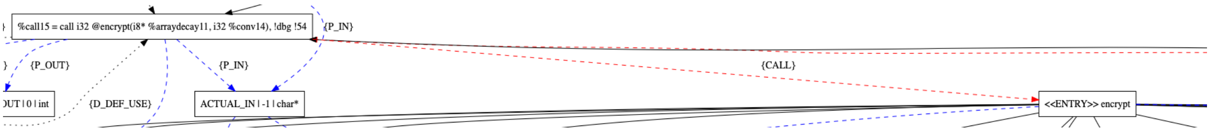
Nodes:

Src: CALL

Dst: FUNC_ENTRY

Example

In the example program, the main function calls the encrypt function.



> line number in IR (IR example code)

DATA_RET

Description:

DATA_RET edge connects the return value to the call instruction in the caller function. It indicates the data flow from the return instruction to the call instruction.

Line: Red dash line with label {D_RET}.

Node Definition:

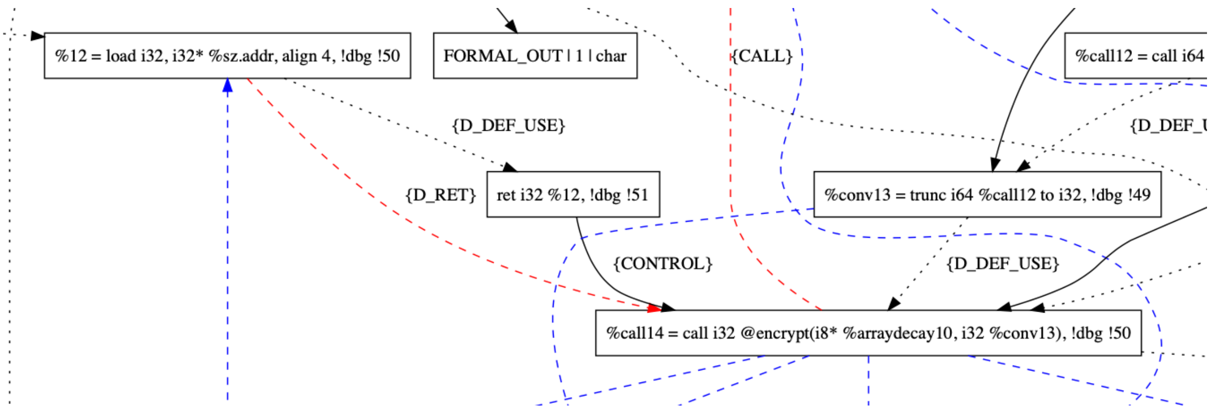
Src: INST

Dst: CALL

Example

The call to function **encrypt** returns an int value. So, there is a DATA_RET edge from the returned value to the call site of **encrypt** in the **main** function.

%12 = load i32, i32* %sz.addr, align 4 → %call14 = call i32 @encrypt(i8* %arraydecay10, i32 %conv13)



CONTROL

Description:

CONTROL edge connects two nodes that have control dependence.

1. Function entry point is connected with all the instructions in the entry basic block.
2. For two basic blocks that have control dependence, we connect the terminator in the dependence basic block to all the dependent instructions.
3. The return instructions in the callee are connected with the call instruction in the caller.

Line: Black solid line with label {CONTROL}.

Node Definition:

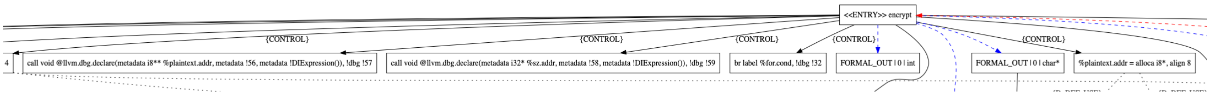
Src: INST / ENTRY

Dst: INST / CALL

Example

Case 1:

> (CONTROLDEP_ENTRY)



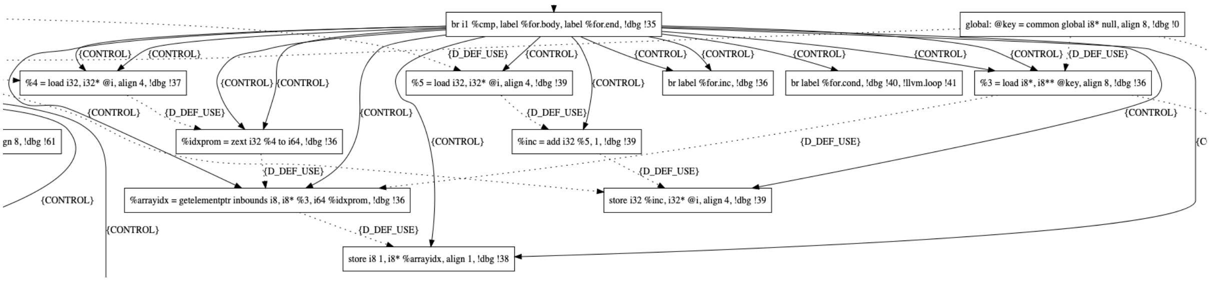
Case 2:

> (CONTROLDEP_BR)

separate case: *(CONTROLDEP_IND_BR)*

In the **initkey** function, the

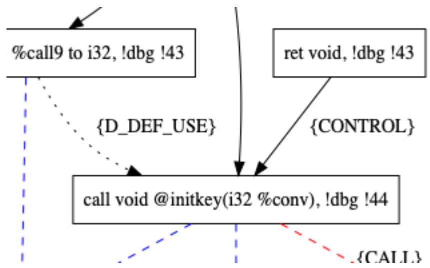
e is a for loop. The following graph corresponds to the for loop.



Case 3: (CONTROLDEP_CALLINV)

Case 4: (CONTROLDEP_CALLRET)

The return instruction in the **initkey** function is connected with the call site of **initkey** in the **main** function.



DATA_DEF_USE

Description:

DATA_DEF_USE edge connects two nodes with def and use. It is directly computed from the LLVM def-use chain.

Line: Black dotted line with label {D_DEF_USE}.

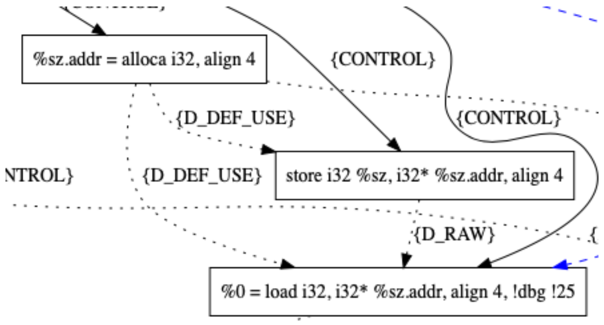
Node Definition:

Src: INST / GLOBAL_VAR

Dst: INST / GLOBAL_VAR

Example

The load instruction **%0 = load i32, i32* %sz.addr, align 4**, loads from a stack address allocated by an alloca instruction. Thus, there is a def-use data dependence between the register holding the stack address (%sz.addr) and the load instruction.



DATA_RAW

Description:

DATA_RAW connects two nodes with read after write dependence. This is flow sensitive. We use memory dependency LLVM pass to compute this information.

Line: Black dotted line with label {D_RAW}.

Node Definition:

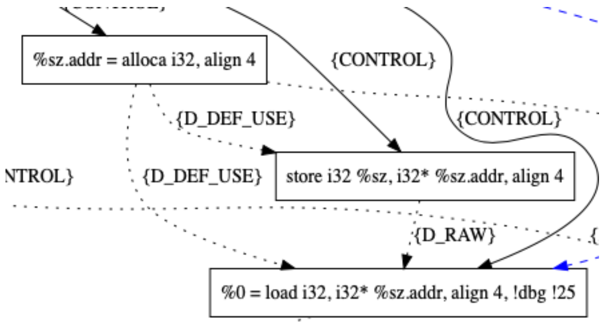
Src: INST

Dst: INST

Example

The load instruction reads from the address which is written by the store instruction.

store i32 %sz, i32 * %sz.addr, align 4 → %0 = load i32, i32* %sz.addr, align 4



DATA_ALIAS

Description:

DATA_ALIAS edge connects two nodes that have may_alias relations. Note that if two nodes n1 and n2 have may_alias relation, then there are two DATA_ALIAS edges exist between them, one from n1 to n2 and one from n2 to n1. This is because the alias relation is undirectional. We currently use SVF to reason about may_alias information.

Line: Black dotted line with label {D_ALIAS}.

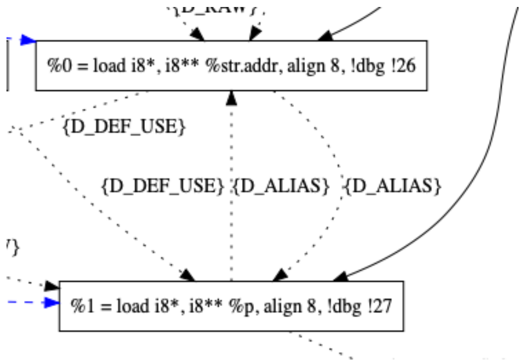
Node Definition:

Src: INST

Dst: INST

Example

In the greeter function, we a local variable **char* p** is defined and it's an alias to the **str** argument. The value of p is stored in %1 register and the argument str is stored in the register %0. Thus, there is an alias edge from %0 and %1, and an alias edge from %1 to %0.



Parameter Trees

In the **main** function, there is a call to the **encrypt** function, which takes takes a **char*** as the first argument. The parameter trees for the char* argument are consists of two nodes: the root node represents the char* parameter and the char child node represent the pointed char.

PARAMETER_IN

Description:

PARAMETER_IN edge represents interprocedural data flow from caller to callee. It connects

1. actual_parameter_in_tree node and formal_in tree nodes

2. formal_in_tree node and the IR variables that corresponds to this formal_in_tree nodes in the callee.

Line: Blue dash line with label {P_IN}.

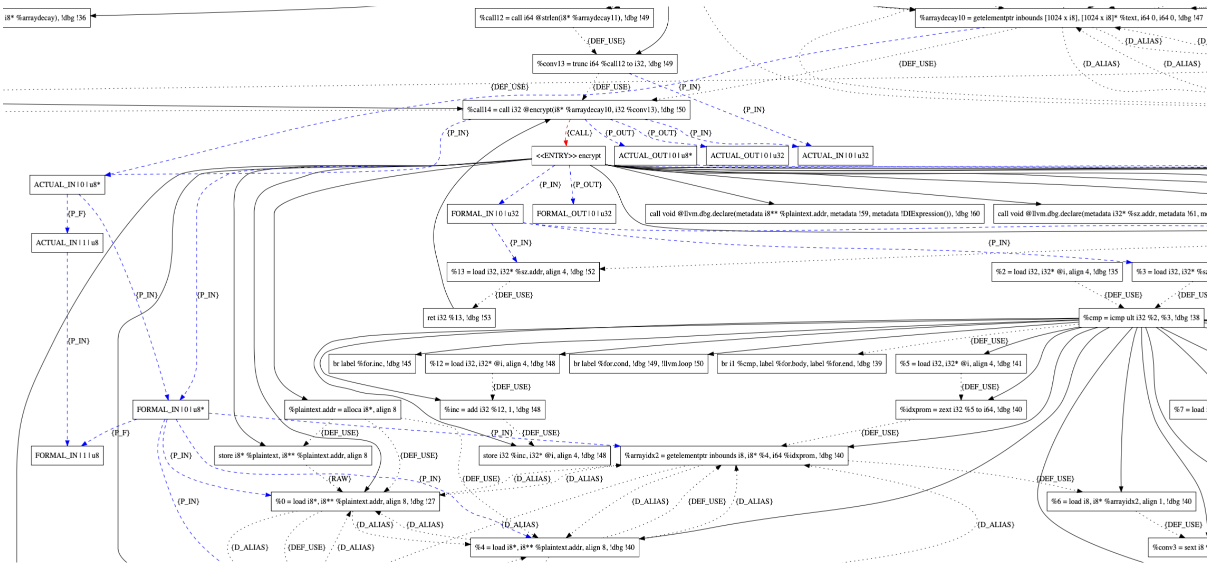
Node Definition:

Src: Actual in tree node / Formal in tree node

Dst: Formal in tree node / INST

Example

Start with the call to **encrypt** function, the %arraydecay10 is passed as an argument to the call. Thus, there is a parameter in edge, from the %arraydecay10 to the actual_in node **ACTUAL_IN | 0 | u8***. Then, the actual in node is linked with the formal in tree node which is defined for the first formal parameter **text**. Next, the value of **text** parameter is stored to a the stack address %plaintext.addr. So, any load from the stack address will return the value of text parameter (in this case, it's the char pointer). In the example, a load instruction **%4 = load i8*, i8** %plaintext.addr, align 8** loads from the stack address. Thus, there is a parameter_in edge from the formal tree node of text to the load instruction.



PARAMETER_OUT

Description:

PARAMETER_OUT edge represents data flow from callee to caller. It connects

1. arguments modified in callee to formal_out_tree node.

2. formal_out_tree node to actual_out_tree node.

3. actual_out_tree node to the modified variable in caller.

Line: Blue dash line with label {P_OUT}.

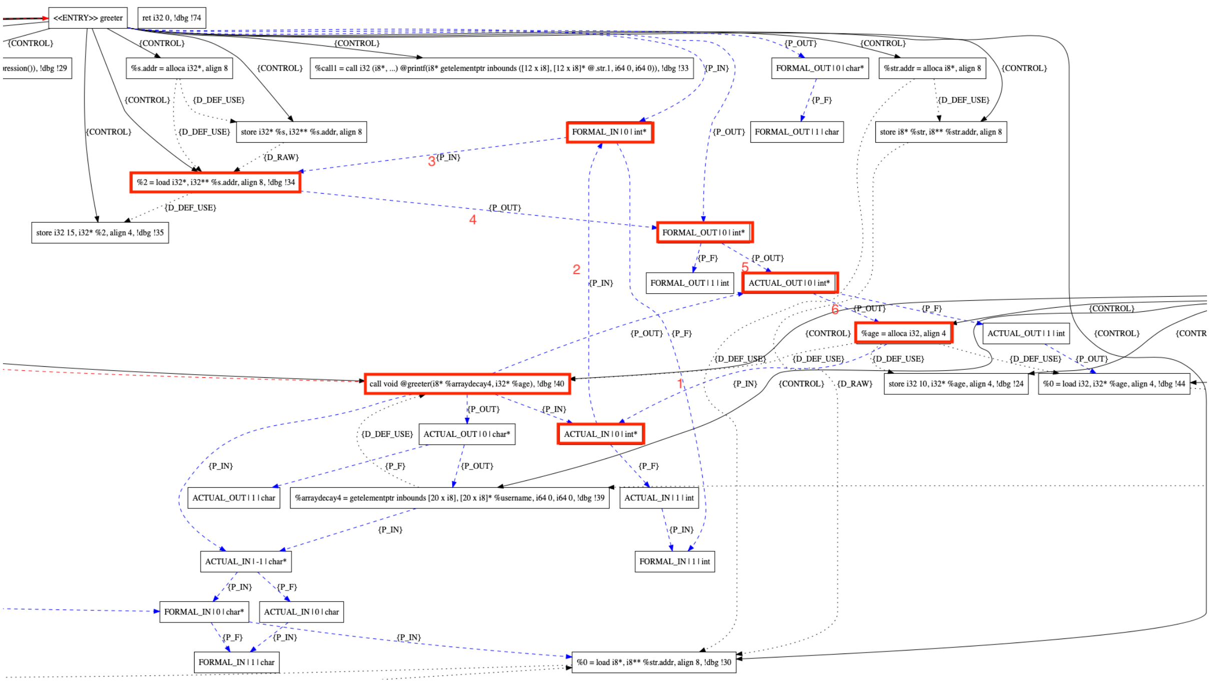
Node Definition:

Src: INST / FORMAL_OUT / ACTUAL_OUT

Dst: INST / FORMAL_OUT / ACTUAL_OUT

Example

The relevant nodes are highlighted by red rectangle box. And the flow order is shown on the edges. The %age variable in main is passed to function **greeter** and get modified in the function. Thus, there is an actual out edge connects the second actual parameter with the %age variable.



PARAMETER_FIELD

Description:

PARAMETER_FIELD edge connects a parent parameter tree node to a child parameter tree node.

Line: Blue dash line with label {P_F}.

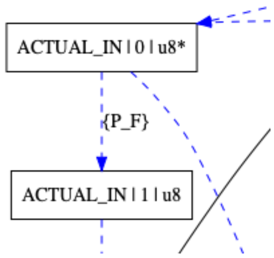
Node Definition:

Src: ACTUAL_IN / ACTUAL_OUT / FORMAL_IN / FORMAL_OUT

Dst: ACTUAL_IN / ACTUAL_OUT / FORMAL_IN / FORMAL_OUT

Example

The actual out tree for the first parameter in the encrypt function. The type of the parameter is char*, thus, there are two nodes represent in the parameter trees: the root node represents the pointer and the child node represent the pointed char.



Annotations

We use attributes to annotate a sensitive data source.

Local annotation

If we annotate a local variable, then the annotation information is stored in the function.

Example:

```
Source Code:
int main() {
  ...
  char __attribute__((annotate("sensitive"))) username[20];
  ...
}

IR:
@.str.2 = private unnamed_addr constant [10 x i8] c"sensitive\00", section "llvm.metadata"

define i32 @main() {
  %username = alloca [20 x i8], align 16
  %username1 = bitcast [20 x i8]* %username to i8*
  call void @llvm.var.annotation(i8* %username1, i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str.2, i32 0, i32 0), i8* getelem
}
```

Global annotation

If we annotate a global variable, then the annotation information is stored in the global scope.

```
Sourcce code:
char __attribute__((annotate("sensitive"))) *key;

IR:

@key = common global i8* null, align 8, !dbg !0
@llvm.global.annotations = appending global [1 x { i8*, i8*, i8*, i32 }] [{ i8*, i8*, i8*, i32 } { i8* bitcast (i8** @key to i8*), i8*
```

Some Issues

Some edges actually represents data flow and control flow instead of data dependency and control dependence.

For example, the return control edge represents control flow.