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.

```
#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;
 }
```

CONTROLDEP

Description:

CONTROLDEP edges are consists of five subtypes:

1. CONTROLDEP_ENTRY
2. CONTROLDEP_BR
3. CONTROLDEP_IND_BR

4. CONTROLDEP_CALLINV
5. CONTROLDEP_CALLRET

Line: Black solid line with label {CONTROLDEP_[TYPE]}.

Case 1: **CONTROLDEP_ENTRY**

Description:

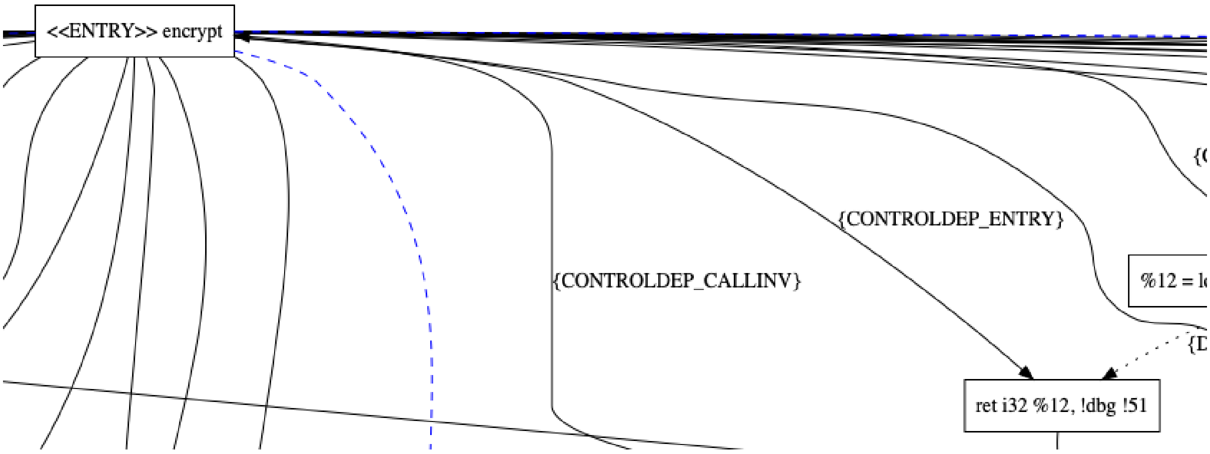
Connects the function entry node with all the instructions in the function body.

Edge type: CONTROLDEP_ENTRY

From node: ENTRY, **IR Instruction:** N/A, **IR line number**: N/A, **Source line number**: N/A

To node: INST, **IR Instruction**: ret i32 %12, !dbg !92, **IR line number**: 140, **Source line number**: 26

Example



Case 2: **CONTROLDEP_BR**

Description:

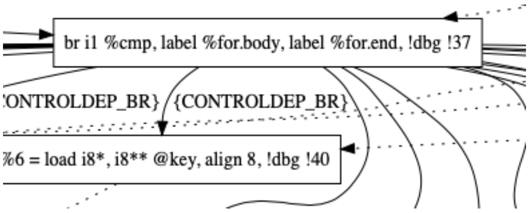
Connects the control dependence block's terminator to the instructions in the control dependent block.

Edge type: CONTROLDEP_BR

From node: INST, **IR Instruction:** br i1 %cmp, label %for.body, label %for.end, !dbg !78, **IR line number**: 108, **Source line number**: 24

To node: INST, **IR Instruction**: %6 = load i8*, i8** @key, align 8, !dbg !81, **IR line number**: 117, **Source line number**: 25

Example


br i1 %cmp, label %for.body, label %for.end, !dbg !37
ONTROLDEP_BR}
{CONTROLDEP_BR}
%6 = load i8*, i8** @key, align 8, !dbg !40

Case 3: CONTROLDEP_IND_BR

Need an example from Rob.

Case 4: **CONTROLDEP_CALLINV**

Description:

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

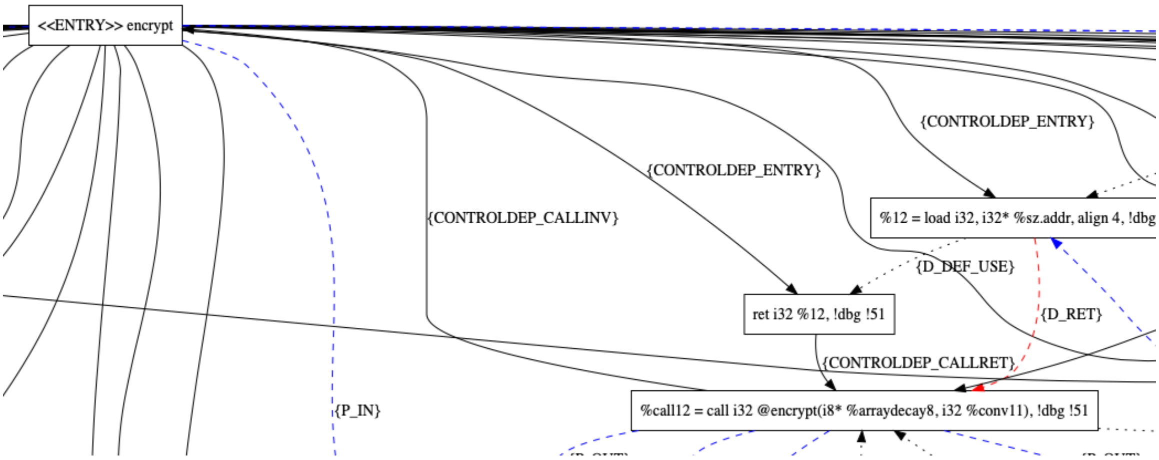
Edge type: D_RET

From node: CALL, **IR Instruction:** %call12 = call i32 @encrypt(i8* %arraydecay8, i32 %conv11), !dbg !124, **IR line number**: 173, **Source line number**: 39

To node: FUNC_ENTRY, **IR Instruction**: N/A, **IR line number**: N/A, **Source line number**: N/A

Example

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



Case 5: **CONTROLDEP_CALLRET**

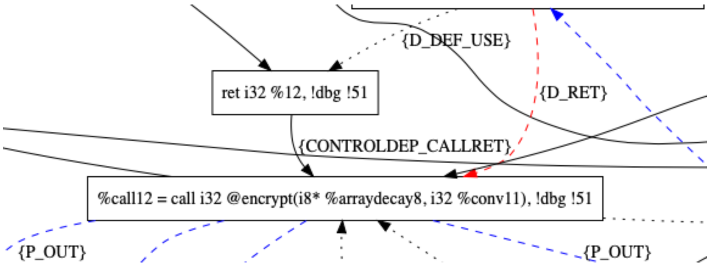
Description:

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

Edge type: CONTROLDEP_CALLRET

From node: INST, **IR Instruction:** ret i32 %12, !dbg !92, **IR line number**: 140, **Source line number**: 26

To node: CALL, **IR Instruction**: %call12 = call i32 @encrypt(i8* %arraydecay8, i32 %conv11), !dbg !124, **IR line number**: 173, **Source line number**: 39



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.

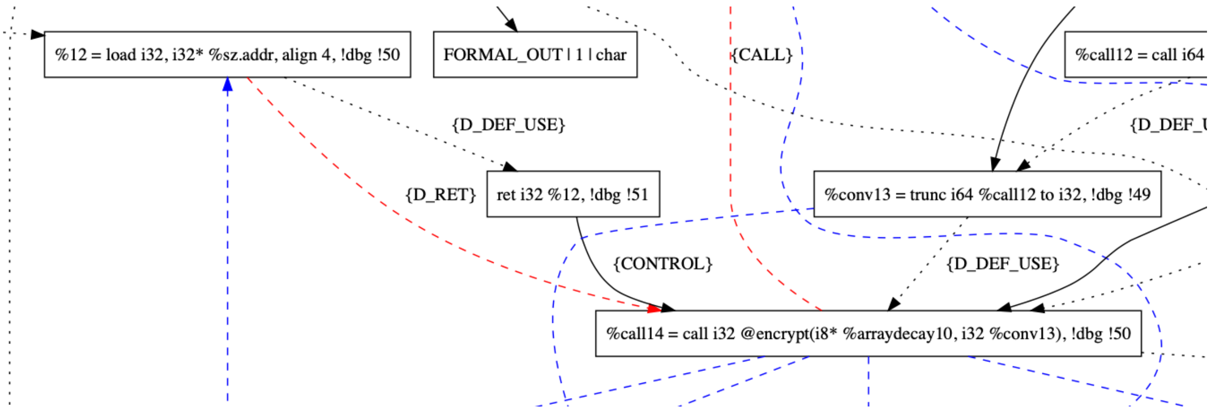
Line: Red dash line with label {D_RET}.

Edge type: D_RET

From node: INST, **IR Instruction:** %12 = load i32, i32* %sz.addr, align 4 , **IR line number**: 140, **Source line number**: 26

To node: CALL, **IR Instruction**: %call14 = call i32 @encrypt(i8* %arraydecay10, i32 %conv13), **IR line number**: 173, **Source line number**: 39

Example


%12 = load i32, i32* %sz.addr, align 4, !dbg !50
FORMAL_OUT | 1 | char
{CALL}
%call12 = call i64
{D_DEF_USE}
{D_DEF_U
{D_RET}
ret i32 %12, !dbg !51
%conv13 = trunc i64 %call12 to i32, !dbg !49
{CONTROL}
{D_DEF_USE}
%call14 = call i32 @encrypt(i8* %arraydecay10, i32 %conv13), !dbg !50

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.

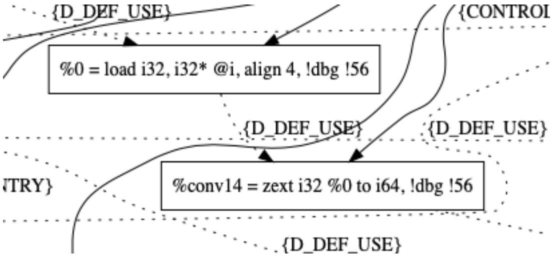
Line: Black dotted line with label {D_DEF_USE}.

Edge type: D_DEF_USE

From node: INST, **IR Instruction:** %0 = load i32, i32* @i, align 4, !dbg !129, **IR line number**: 180, **Source line number**: 41

To node: INST, **IR Instruction**: %conv14 = zext i32 %0 to i64, !dbg !129, **IR line number**: 181, **Source line number**: 41

Example


{D_DEF_USE}
{CONTROL
%0 = load i32, i32* @i, align 4, !dbg !56
{D_DEF_USE}
{D_DEF_USE}
TRY}
%conv14 = zext i32 %0 to i64, !dbg !56
{D_DEF_USE}

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}.

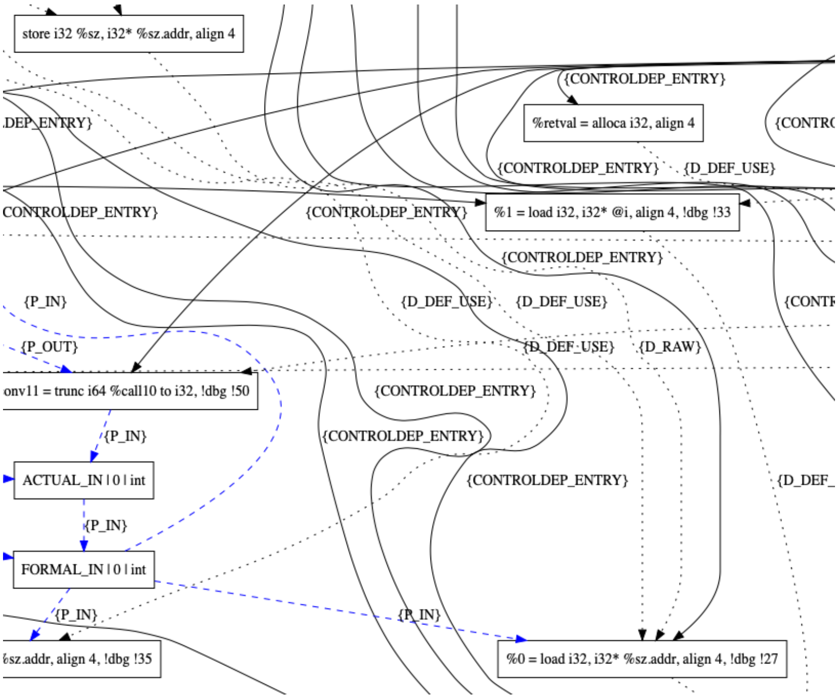
Edge type: D_RAW

From node: INST, **IR Instruction:** store i32 %sz, i32* %sz.addr, align 4, **IR line number**: 95, **Source line number**: 23

To node: INST, **IR Instruction**: %0 = load i32, i32* %sz.addr, align 4, !dbg !68, **IR line number**: 97, **Source line number**: 23

Example

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



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.

Line: Black dotted line with label {D_ALIAS}.

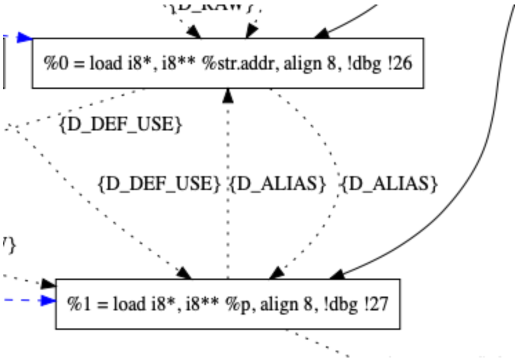
Edge type: D_ALIAS

From node: INST, **IR Instruction:** %0 = load i8*, i8** %str.addr, align 8, !dbg !30, **IR line number**: 33, **Source line number**: 10

To node: INST, **IR Instruction**: %1 = load i8*, i8** %p, align 8, !dbg !31 , **IR line number**: 35, **Source line number**: 10

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.


%0 = load i8*, i8** %str.addr, align 8, !dbg !26
{D_DEF_USE}
{D_DEF_USE}
{D_ALIAS}
{D_ALIAS}
%1 = load i8*, i8** %p, align 8, !dbg !27

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}.

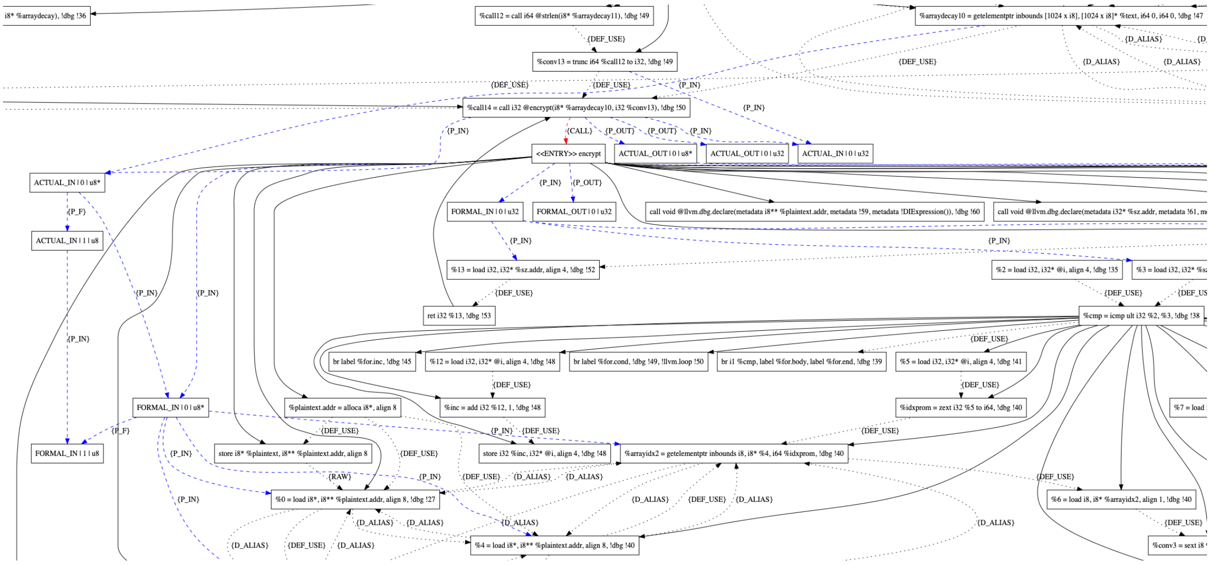
Node Definition:

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.


i8* %arraydecay), !dbg !36
%call12 = call i64 @strlen(i8* %arraydecay11), !dbg !49
%arraydecay10 = getelementptr inbounds [1024 x i8], [1024 x i8]* %text, i64 0, i64 0, !dbg !47
%conv13 = trunc i64 %call12 to i32, !dbg !49
%call14 = call i32 @encrypt(i8* %arraydecay10, i32 %conv13), !dbg !50
<<ENTRY>> encrypt
ACTUAL_OUT | 0 | u8*
ACTUAL_OUT | 0 | u32
ACTUAL_IN | 0 | u32
ACTUAL_IN | 0 | u8*
ACTUAL_IN | 1 | u8
FORMAL_IN | 0 | u32
FORMAL_OUT | 0 | u32
call void @llvm.dbg.declare(metadata i8** %plaintext.addr, metadata !59, metadata !DIExpression()), !dbg !60
call void @llvm.dbg.declare(metadata i32* %sz.addr, metadata !61, m
%13 = load i32, i32* %sz.addr, align 4, !dbg !52
%2 = load i32, i32* @i, align 4, !dbg !35
%3 = load i32, i32* %s
ret i32 %13, !dbg !53
%cmp = icmp ult i32 %2, %3, !dbg !38
br label %for.inc, !dbg !45
%12 = load i32, i32* @i, align 4, !dbg !48
br label %for.cond, !dbg !49, !llvm.loop !50
br i1 %cmp, label %for.body, label %for.end, !dbg !39
%5 = load i32, i32* @i, align 4, !dbg !41
FORMAL_IN | 0 | u8*
%plaintext.addr = alloca i8*, align 8
%inc = add i32 %12, 1, !dbg !48
%idxprom = zext i32 %5 to i64, !dbg !40
%7 = load
FORMAL_IN | 1 | u8
store i8* %plaintext, i8** %plaintext.addr, align 8
store i32 %inc, i32* @i, align 4, !dbg !48
%arrayidx2 = getelementptr inbounds i8, i8* %4, i64 %idxprom, !dbg !40
%0 = load i8*, i8** %plaintext.addr, align 8, !dbg !27
%6 = load i8, i8* %arrayidx2, align 1, !dbg !40
%4 = load i8*, i8** %plaintext.addr, align 8, !dbg !40
%conv3 = sext i8
{DEF_USE}
{D_ALIAS}
{P_IN}
{P_OUT}
{CALL}
{P_F}
{RAW}

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}.

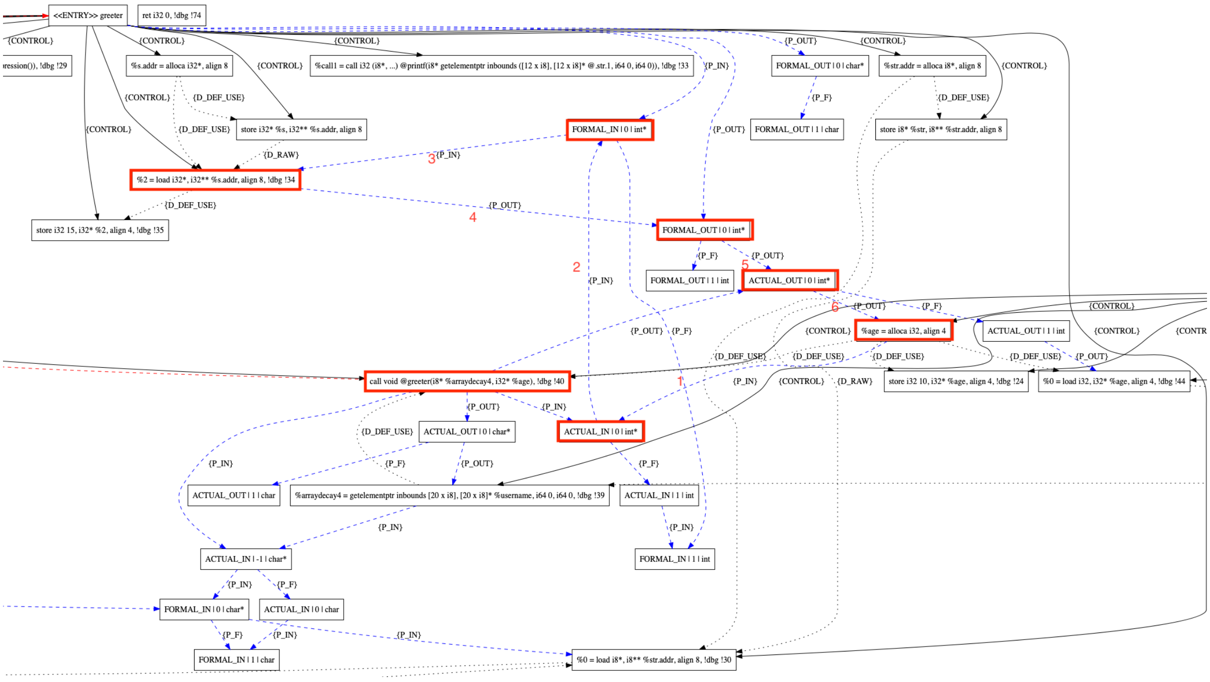
Node Definition:

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.


<<ENTRY>> greeter
ret i32 0, !dbg !74
%s.addr = alloca i32*, align 8
%call1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([12 x i8], [12 x i8]* @.str.1, i64 0, i64 0)), !dbg !33
FORMAL_OUT | 0 | char*
%str.addr = alloca i8*, align 8
store i32* %s, i32** %s.addr, align 8
FORMAL_IN | 0 | int*
FORMAL_OUT | 1 | char
store i8* %str, i8** %str.addr, align 8
%2 = load i32*, i32** %s.addr, align 8, !dbg !34
store i32 15, i32* %2, align 4, !dbg !35
FORMAL_OUT | 0 | int*
FORMAL_OUT | 1 | int
ACTUAL_OUT | 0 | int*
%age = alloca i32, align 4
ACTUAL_OUT | 1 | int
call void @greeter(i8* %arraydecay4, i32* %age), !dbg !40
store i32 10, i32* %age, align 4, !dbg !24
%0 = load i32, i32* %age, align 4, !dbg !44
ACTUAL_OUT | 0 | char*
ACTUAL_IN | 0 | int*
ACTUAL_OUT | 1 | char
%arraydecay4 = getelementptr inbounds [20 x i8], [20 x i8]* %username, i64 0, i64 0, !dbg !39
ACTUAL_IN | 1 | int
ACTUAL_IN | -1 | char*
FORMAL_IN | 1 | int
FORMAL_IN | 0 | char*
ACTUAL_IN | 0 | char
FORMAL_IN | 1 | char
%0 = load i8*, i8** %str.addr, align 8, !dbg !30
{CONTROL}
{D_DEF_USE}
{D_RAW}
{P_IN}
{P_OUT}
{P_F}
1
2
3
4
5
6

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}.

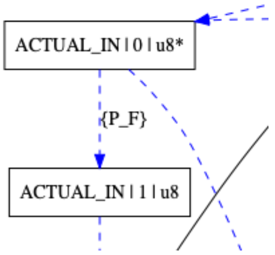
Node Definition:

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.