An Implementation Overview of an IDL Generation Framework Based on DSA

Aftab Hussain[1], Vikram Narayanan[2], Anton Burtsev[3]
University of California, Irvine
[1]*aftabh@uci.edu*, [2]*mail2@domain.com*, [3]*aburtsev@uci.edu*

Abstract

In this work, we describe the implementation of an IDL Generator Framework [1] that is designed to generate IDL (Interface Description Language) code for large codebases in C like the linux kernel. The framework is built upon the Data Structure Analysis framework proposed by Chris Lattner [2]. We also highlight the issue of callee to callee data transfer in DSA, and describe how it impacts the information we extract for IDL generation.

1 Introduction

Our IDL Generator Framework generates a specific type of Interface Description Language Code (based on the specifications in [3]) for any codebase in C. The IDL helps to define boundaries between different components of the codebase.

The motivation behind defining such boundaries lies in the fact that it helps developers apply appropriate measures to prevent vulnerabilities in one part of the system to affect other parts of the system [4]. Isolating components in this manner is particularly crucial for large codebases, such as the Linux kernel, where defining such boundaries becomes a non-trivial task. This is because such large systems are monolithic in design whose components show a high degree of functional interconnectivity with each other [5–7]. As a result there has been a push towards redesigning large systems in the systems community. One such project is the Deker Project [4, 8] which targets the decomposition of the Linux kernel into subsystems.

In order to define boundaries in the Linux kernel, Spall [3] built an interface description language that can also be used for other large systems. From here onwards we shall refer to this description language as IDL. In a nutshell, IDL helps in determining what data (e.g. structure fields) a component (e.g. a module) exposes to other components in a system. An exposure of a data in a component (or domain) to the outer domain, entails an access to the data from the outer domain. Such an access may be a *read* or a *write* of the data. A simple example is shown in Figures 1 and 2. In Fig.1a, we see a function dummy_setup, defined in an isolated domain (dummy.c), writing on the field priv_flags of the net_device structure variable dev. Assuming that dummy_setup can be called from outside the domain, dummy.c, the IDL reports a projection of the net_device structure field dev for the function dummy_setup, as shown in Fig. 1b. For a real scenario, the IDL should represent the same projection for all functions in the system that may invoke dummy_setup. An elaborate description of all the features of IDL are given in [3].

Once the IDL is specified, the next major challenge is how to efficiently generate the IDL for a large codebase. A static analysis of the entire codebase would be required which takes into consideration the most common syntactic idioms or features of the codebase being analysed. In this work, we are primarily focussed on decomposing the Linux kernel. Its features include the use of function pointers, structure variables and fields, and aliasing.

We found DSA (Data Structure Analysis) [2, 9] to be a promising static analysis technique that particularly helps us with handling function pointers, recursion, and tracking the use of structure field accesses throughout a large program. It was also found to be relatively efficient in analyzing big code (1-3 seconds for analyzing 100K-200K lines of C code [9]). Although there are some drawbacks of DSA, (which we elaborate upon in Sec. 5, due to its advantages mentioned above, we built the IDL

```
//Isolated Domain (dummy.c)
#include <linux/netdevice.h>
...
static void dummy_setup (struct
net_device ∗ dev){
    dev −> priv_flags |=
    IFF_LIVE_ADDR_CHANGE;
}
...
```

(a) Code

```
projection dummy_setup.dev {
  unsigned int priv_flags;
}
rpc void dummy_setup (projection
dummy_setup.dev ∗ dev)
...
```

(b) IDL

Figure 1: (a) Structure field access in a function, defined in an isolated domain. (b) IDL for dummy_setup.

Generator Framework using DSA. We used DSA libraries from the SMACK [10] tool, a modular software verification toolchain and self-contained software verifier. This library is originally taken from LLVM's poolalloc library [11].

The rest of this work is organized as follows: In Section 2, we discuss how our framework is supposed to work with the help of DSA. In Section 3, we give implementation details of our framework. In Section 4, we show how to setup your system in order to execute our framework. In Section 5, we show how DSA may leak context sensitive information leading to a loss in precision while generating projection information (Sec. 5).

2 Framework Description

In this section, we first present the basic approach of our framework, giving a description of the DSA process - the backbone of our technique (Sec. 2.1). Then we illustrate via examples how the phases of DSA provide the necessary information required for our framework to generate IDL (Sec. 2.2).

2.1 Approach

Our main goal with IDL-Generator is to output projections for functions. A projection of a function shows how it and *its callees* access (read/write) a data structure that is passed on to it. To do so we take the help of DSA [9], which yields a graph from which we apply our pass to extract relevant data read write information to generate the projections for the IDL.

2.1.1 DSA

In a nutshell, DSA comprises of the following phases or analyses on the code: (1) *Local analysis (LA)* which constructs the data structure usage information for each function, creating local nodes for each function. (2) *Bottom-up analysis (BU)* which inlines or copies data structure usage information of each function into the nodes of each of its callers. (3) *Top-down analysis (TD)* which performs the reverse process of BU, i.e., copies information of each function into the nodes of each of its callees. The additional TD phase helps in resolving calls made via function pointers. From here onwards, we shall use the term "copying (or inlining or cloning) from or to a function" to mean copying from or to a node representing the function.

2.2 Illustration

Let us now examine how DSA actually generates the necessary information for the IDL Generator by means of an artificial example. Consider the example in Fig. 2, where data var is exposed by function a because it transitively calls function c, where var is accessed (written on). Our IDL should therefore report a projection of data var for a. Similarly, it should report this projection for all functions that can lead to an access on var. Let us now see how DSA can help us gather projection info.

For the example in Fig. 2, LA and BU phases help transmit information of the write access in function c to all functions preceding it in the call chain. It may appear that LA and BU are sufficient for obtaining the information we need. However, with the next example, we shall demonstrate why the third phase, TD, is needed.

```
void a(int var) {
    //does nothing on var
     b(var)
}
void b(int var){
    //does nothing on var
    c(var)
}
void c(int var){
    //WRITES on var
    var = var∗3;
}
```

Figure 2: Function a(int var) transitively exposing data via a chain of function calls.

```
int Global = 10;
typedef struct list {
int Data;
}list;
void fun(int∗ X){
    // write
    (∗X)+=Global;
}
 void accessF(struct list∗ L, void (∗FP)(int∗)){
         // should report r/w's reported by function pointed by FP
         FP(&L−>Data);
 }
 void passF(struct list∗ L){
     // should report r/w's reported by function accessF
     accessF (L,fun);
 }
```

Figure 3: Example showing call made via a function pointer.

```
Function: accessF
Projects structure: list
Read:
Write:

Function: passF
Projects structure: list
Read:
  offset:  int Data
Write:
  offset:  int Data
```

(a)

```
Function: accessF
Projects structure: list
Read:
 offset:  int Data
Write:
 offset:  int Data

Function: passF
Projects structure: list
Read:
 offset:  int Data
Write:
 offset:  int Data
```

(b)

Figure 4: (a) Projection information for IDL when applying only LA and BU phases of DSA on code in Fig. 3. (b) Projection information for IDL when applying all the phases of DSA on code in Fig. 3.

Function pointer example. Now let us see another example with a function pointer that demonstrates the need for the third phase of DSA (the TD phase). Consider the code in Fig. 3. The call chain for this code is,

passF −−−−[call1]−−−−> accessF −−−−[call2]−−−> fun,

where passF is at the top of the call chain and fun is at the bottom. Also note that, [call1] is a direct call and [call2] is an indirect call (call through function pointer, FP). Assuming we have LA done on all the functions, let us simulate how BU would work on this example.

In the BU process, while processing fun, at this stage, we cannot resolve [call2], because we do not know accessF calls fun, therefore it is completely ignored and no read/write information is cloned from it into accessF. Moving on to accessF, we can only resolve call1, and copy read/write information of accessF to passF. Thus, if we were to only apply LA and BU analyses on this code, we would get the IDL projections shown in Fig. 4a. The projections fail to capture that structure field Data is accessed via accessF. However, note that passF can still capture the accesses in fun because when BU processes passF, the indirect callee's (fun's) BU graph is cloned and merged into the graph of the function where the call site became resolved (passF) [2]. Note, in Fig. 3, if there were no indirect call in accessF, no indirect callee would have been registered for accessF as it would not contain a callsite. Consequently, fun would not be inlined to passF.

On applying TD analysis on the graph output of BU, we can resolve the missing information for function pointer FP in the function containing the actual call (accessF) in Fig. 3. The projections after applying TD are shown in Fig. 4b.

3 Implementation Details

In this section, we give the implementation details of our framework. In Sec. 3.1, we highlight the main components of the framework and where its execution begins. In Sec. 3.2 and Sec. 3.3 we elaborate upon the parsing process and how the passes are executed, respectively. In Secs. 3.4 and 3.5, we outline the alias analysis and function pointer support capabilities in DSA, respectively.

3.1 Source Outline

The complete source code of IDL-Generator is available in [1]. Here is a brief description of its main components: (1) lib/DSA implements the core part of the DSA algorithm. It consists of a set of passes (or analyses) like Local.cpp, BottomUpClosure.cpp, and TopDownClosure.cpp, and a set of supporting data structures like DSGraph.cpp, and DSCallGraph.cpp. (2) lib/ dsaGenerator consists of the pass that builds upon the previous analyses or passes to generate the idl file. Each pass's work is done by their respective runOnModule() functions. During execution, runOnModule is invoked by the LLVM system, LegacyPassManager.cpp [12].

The starting point of the framework is tools/dsaGenerator. It contains the main function [13] which begins by parsing the input bitcode using parseIR(), then initializes and adds all the required passes to be performed to the PassRegistry (as shown in [14]), and then executes the passes. The scheduling of the passes are handled by PassManager. In the following subsections, we present the details of the parsing process and the executions of the passes.

3.2 Parsing input bitcode

The parsing is performed by the parseIR() function, which is an LLVM function that parses the bitcode input and returns a unique_ptr to a Module [15]. Module is the top level container of all LLVM IR objects. All passes are invoked on this container. parseIR() triggers the following call chain to perform the parsing process: getFileOrSTDIN –> getFile –> getFileAux (opens the input file for reading) –> getOpenFileImpl. getOpenFileImpl [16] is important for it sets up the buffer and the map with the data. This function calls getMemoryBufferForStream which reads the file into a SmallString data structure, which acts as the buffer. SmallString uses the SmallVector [17] data structure which is essentially a variable sized array. getMemoryBufferForStream then finally calls getMemBufferCopy [18], which initializes the buffer by getNewUninitMemBuffer [19], and then invokes memcpy to copy the data to the buffer. Details of how the map is built, and how both the map and buffer are used to set up the contents of the Module have been omitted.

3.3 Executing the Passes

As mentioned in Sec. 3.1, tools/dsagenerator.cpp registers all passes that are to be performed. The passes are added via getAnalysisUsage [20] using addRequired [21] (See [22] for more information on getAnalysisUsage() method). The passes that are added are LocalPass, BottomUpClosure, TopDownClosure, DSAGenerator. The passes are eventually executed in the same order. Note that adding TopDownClosure using getAnalysisUsage() automatically adds the preceding passes, LocalPass and BottomUpClosure, since both are its prerequisite passes.

As already mentioned, all passes use LLVM runOnModule to execute on Module container, which contains the LLVM IR Objects of the input code to be analyzed. LocalPass generates the first intermediary graph (a DSGraph) from Module. Then BottomUpClosure accesses Module to obtain the intermediary graph, and generates a new graph, which is again processed upon in the same manner by TopDownClosure.

3.3.1 Local Pass [23]

This pass builds the local graphs of the functions in the input program using GraphBuilder [24] constructor. Its implementation is given in lib/DSA/Local.cpp. It visits the instructions using visitor methods of the LLVM friend class InstVisitor. For example, here is a visit to a call site, It also builds the call graph using DSGraph's buildCallGraph [25] function. DSGraph provides the core data structure functionality (like cloning/inling, merging, dead node deletion) for all kinds of graphs. An important part of our IDL is distinguishing between reads and writes. The Local pass helps in this by taking appropriate action depending on the type of instruction it visits. It records a read when a load instruction is visited in the IR obtained from the bitcode [26]. Similarly it records a write when a store instruction is visited in the IR [27]. This read/write information is stored in each node of the graph (the DSNode).

3.3.2 Bottom-Up Closure Pass [28]

This pass performs interprocedural bottom up analysis of graphs. It implements the BUDataStructures [29] class. Its implementation is given in lib/DSA/BottomUpClosure.cpp. The calculateGraph function [30] is where the inlining of the functions is carried out.

3.3.3 Top-Down Closure Pass

This pass performs top-down analysis of graphs. It implements the TDDataStructures [31] class. Its implementation is given in lib/DSA/TopDownClosure.cpp.

3.3.4 IDL Generator Pass [32]

This pass extracts information from the resulting graph of the DSA algorithm and generates the IDL. Its implementation is given in lib/dsaGenerator/DSAGenerator.cpp. This pass scans the graph generated by the DSA algorithm (reads the DSNodes), extracting relevant information. Each DSNode provides accessor methods, which can be used to obtain certain information. For example the accessor methods read_offset_begin()([33]) and write_offset_begin() provide the indexes of where information of read and written variables are stored in a DSNode.

The call graph of the IDL Generator pass (DSAGenerator.cpp) is given in Fig. 5.

3.4 Alias Analysis with DSA

DSA does not currently handle global aliases [34]. According to the developers of DSA, DSA is broken on global aliasing, as it does not handle the aliases of parameters correctly [35]. The address taken analysis pass helps find which functions are address taken in a module, where functions are considered to be address taken if they are either stored, or passed as arguments to functions [36]. This pass is executed before the local pass of DSA.

Traditional alias analysis algorithms like Andersen's analysis and Steensgard analysis can be built on top of DSA. They have been implemented by Lattner et al. as additional passes (local, steens−fi, steens−fs, and an−ders, ds−aa). For details the reader is referred to Chapter 4 of Lattner's thesis [2].

3.5 Function Pointer Handling in DSA

Any call that involves a call instruction is known as a *callsite*. The invoked function in the callsite is known as the *callee*. A callsite is represented by the DSCallSite data structure [37], which is a wrapper

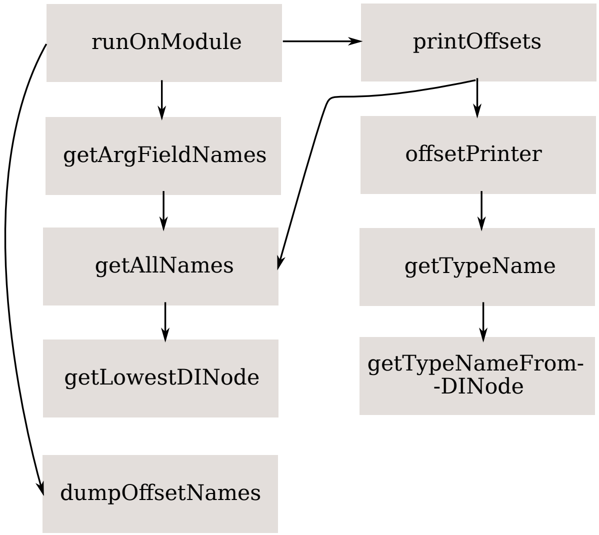

Figure 5: Call graph of IDL Generator Pass (DSAGenerator.cpp)

around the actual data structure for representing a callsite: CallSite [38]. A callee is represented by a DSNode and the LLVM data structure Function [39]. DSCallSite [37] provides the handles for the callee function and the function arguments. The logic to distinguish an indirect call from a direct call is provided in the method FunctionTypeOfCallSite. The IR for an indirect call, e.g. fp(devops); [40], is shown below:

```
[local] visiting call:tail
% call void %fp(%struct.device_operations∗ %devops)
% #5, !dbg !134.
```

This instruction is visited in the Local pass as shown here [41] in the debug output. The code that visits this instruction is shown here [42].

4 Setup

In this section, we first show how to setup LLVM in your system, which is required for the IDL-Generator framwork (Sec. 4.1). Then we show how to setup IDL-Generator in your system (Sec. 4.2).

4.1 LLVM Setup Instructions

Use the source for LLVM 3.8.1 [43].

Build Create a build directory (build) outside the LLVM source directory (llvm). Copy the clang project (version 3.8.1, available here [44]) inside llvm/tools. Then do the following inside build,

```
$cmake −G "Unix Makefiles" −DLLVM_TARGETS_TO_BUILD
 ="X86" −DCMAKE_BUILD_TYPE="Release"
 ..llvm && make −j32
```

Install Inside the build do,

```
$sudo cmake −−build . −−target install
```

4.2 DSA-IDL-Generator Setup Instructions

Clone from [1] to get DSA-IDL-Generator source code.

Build Create build directory in source directory. In build do,

```
$cmake .. && make −j32
```

Run In build directory containing DSA-IDL-Generator executable and test.bc:

```
$./dsagenerator <test.bc>
```

Preparing the input The input is a compiled unit of the program we want to analyze in the form of a .bc file. The output is an .idl file, which consists of the projections. The compiled unit of the programs needs to be obtained as follows if using clang:

```
$clang −O1 −g −emit−llvm sampleProgram.c
−c −o sampleProgram.bc
```

5 Imprecise Top Down Information Transfer in DSA

In this Section, we show how DSA, in particular, its top-down phase, may leak context sensitive information leading to a loss in precision while generating projection information.

While the DSA algorithm is intended to be "fully-context sensitive", we found there to be some loss of information, which arises due to the unification approach of its algorithm. Based on the DSA algorithm presented in [2], we see that the top-down analysis phase copies a caller's information to all its callee functions. Prior to this, information from all the callees of a function are copied into it (by virture of the bottom-up analysis phase). This results in improper information flows, which can lead to a loss of precision in the information we gather for the projections.

In Sec. 5.1, we show a callee of a function can get data structure information from another callee of the same function.

5.1 Callee-to-Callee Information Transfer

Information from one callee of a function can be copied into another callee of the same function. We illustrate this with the example in Fig. 8a. The call graph for this program is given in Fig. 6.

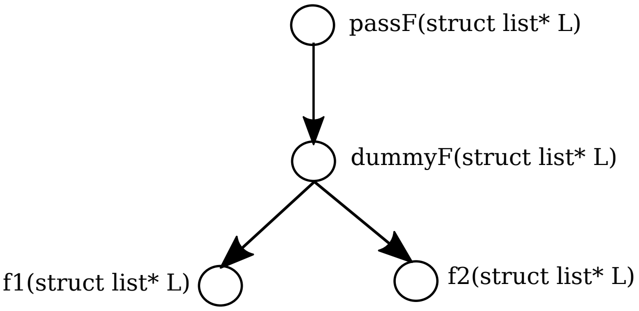

Figure 6: Call graph of code in Fig. 8a

In this code, we see dummyF calling two functions, f1 and f2, both of which carry out different kinds of accesses on the structure field Cell. The BU phase copies these access information to dummyF and to passF. The TD phase, then copies all the information from the top of the call graph to the bottom. As a result, f1 and f2 receive one another's access information. In other words, f1 and f2 incorrectly record they both read and write on Cell. This is also revealed by the IDL projection information in Fig. 8b, which is generated when analyzing the resulting graph DSA produces for this example.

Another example with function pointers is shown in Fig. 9 (program in Fig. 9a, and projections in Fig. 9b).

6 Conclusion

In this work, we have examined the effectiveness of implementing DSA to track caller-callee relationships induced by direct and indirect calls. A framework for implementing DSA is presented, and the flow of data structure information is observed using the framework. The data structure flow information is used to support IDL generation. In this work we also show how the top down transfer of information, a phase of the DSA, can be inconsistent.

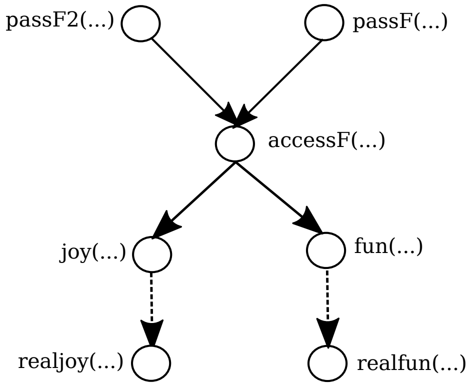

Figure 7: Call graph of code in Fig. 9a. (Dashed directed edges represent an indirect call.)

```
#include "stdio.h"
#include "stdlib.h"

typedef struct list {
int Data;
int Block;
int Cell;
}list;

void f1(struct list* L){
L->Cell=10;
}

void f2(struct list* L){
   if (L->Cell>0){printf("test");}
}

void dummyF(struct list* L){
   f1(L);
   f2(L);
}

void passF(struct list* L){
      dummyF(L);
}
```

(a)

```
Function: f1
Projects structure: list
Read:
   offset:   int Cell
Write:
   offset:   int Cell

Function: f2
Projects structure: list
Read:
   offset:   int Cell
Write:
   offset:   int Cell

Function: dummyF
Projects structure: list
Read:
   offset:   int Cell
Write:
   offset:   int Cell

Function: passF
Projects structure: list
Read:
   offset:   int Cell
Write:
   offset:   int Cell
```

(b)

Figure 8: (a) Example code with branching.
(b) Projection information generated for the code.

```
#include "stdio.h"
#include "stdlib.h"

int Global = 10;

typedef struct list {
int Data;
int Block;
int Cell;
}list;

void realfun(struct list* L){
     L->Block=24;
}

void realjoy(struct list* L){
     L->Cell=24;
}

void fun(struct list* L){
    realfun(L);
}

void joy(struct list* L){
    realjoy(L);
}

 void accessF(struct list* L,
 void (*FP)(struct list* L)){
    FP(L);
 }

 void passF(struct list* L){
    accessF (L,fun);
 }

  void passF2(struct list* L){
    accessF (L,joy);
 }
```

(a)

```
Function: realfun
Projects structure: list
Read:
Write:
 offset:   int Block
 offset:   int Cell

Function: realjoy
Projects structure: list
Read:
Write:
 offset:   int Block
 offset:   int Cell

Function: fun
Projects structure: list
Read:
Write:
 offset:   int Block
 offset:   int Cell

Function: joy
Projects structure: list
Read:
Write:
 offset:   int Block
 offset:   int Cell

Function: accessF
Projects structure: list
Read:
Write:
 offset:   int Block
 offset:   int Cell

Function: passF
Projects structure: list
Read:
Write:
 offset:   int Block

Function: passF2
Projects structure: list
Read:
Write:
 offset:   int Cell
```

(b)

Figure 9: (a) Example code with branching and function pointers. (b) Projection information generated for the code.

References

[1] Jiten Thakkar, Vikram Narayanan, Aftab Hussain. DSA-IDL-Generator Framework. `https://github.com/AftabHussain/DataStructureAnalysis/tree/dsa_llvm3.8`.

[2] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[3] Sarah Spall. kIDL: interface definition language for the kernel. Technical report, 2016.

[4] Charles Jacobsen. Lightweight capability domains: Toward decomposing the linux kernel. Master's thesis, University of Utah, 2016.

[5] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[6] Jim Mauro and Richard McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[7] Jonathan Levin. *Mac OS X and iOS Internals: To the Apple's Core*. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2012.

[8] Deker Project. `https://www.flux.utah.edu/project/deker`.

[9] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[10] Michael Emmi, Zvonimir Rakamaric. SMACK. `https://github.com/smackers/smack/tree/d609e00b9a40be15eb587d05cf4206d26b70fabd`.

[11] poolalloc library, LLVM. `https://github.com/llvm-mirror/poolalloc`.

[12] LegacyPassManager (LLVM). `http://llvm.org/doxygen/LegacyPassManager_8cpp_source.html#l01688`.

[13] Main function of DSA-IDL-Gen Framework. `https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/tools/dsaGenerator/dsaGenerator.cpp#L54`.

[14] Adding passes in DSA-IDL-Generator. `https://github.com/AftabHussain/DataStructureAnalysis/blob/dsa_llvm3.8/tools/dsaGenerator/dsaGenerator.cpp`.

[15] Module class (LLVM). `http://llvm.org/doxygen/classllvm_1_1Module.html#details`.

[16] getOpenFileImpl function (LLVM). `https://github.com/llvm-mirror/llvm/blob/051e787f26dbfdc26cf61a57bc82ca00dcb812e8/lib/Support/MemoryBuffer.cpp#L330`.

[17] SmallVector class (LLVM). `http://llvm.org/doxygen/classllvm_1_1SmallVector.html#details`.

[18] getMemoryBufferCopy function (LLVM). `https://github.com/llvm-mirror/llvm/blob/051e787f26dbfdc26cf61a57bc82ca00dcb812e8/lib/Support/MemoryBuffer.cpp#L118`.

[19] getNewUninitMemBuffer function (LLVM). `https://github.com/llvm-mirror/llvm/blob/051e787f26dbfdc26cf61a57bc82ca00dcb812e8/lib/Support/MemoryBuffer.cpp#L130`.

[20] getAnalysisUsage function (DSA-IDL-Gen). `https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/include/dsaGenerator/DSAGenerator.h#L31`.

[21] addRequired function (LLVM). `http://llvm.org/doxygen/PassAnalysisSupport_8h_source.html#l00066`.

[22] getAnalysisUsage function (LLVM). `http://llvm.org/docs/WritingAnLLVMPass.html#specifying-interactions-between-passes`.

[23] Local pass (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/lib/DSA/Local.cpp.

[24] GraphBuilder class (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/lib/DSA/Local.cpp#L151.

[25] buildCallgraph function (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/lib/DSA/DSGraph.cpp#L1577.

[26] visitLoadInst function. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/DSA/Local.cpp#L395.

[27] vistStoreInst. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/DSA/Local.cpp#L429.

[28] BottomUpClosure pass (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/lib/DSA/BottomUpClosure.cpp.

[29] BUDataStructures class (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/9043612b5977ac91afe9667d57c4b87ef8af0d66/include/dsa/DataStructure.h#L230.

[30] calculateGraph Function. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/DSA/BottomUpClosure.cpp#L640.

[31] TDDataStructures class (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/9043612b5977ac91afe9667d57c4b87ef8af0d66/include/dsa/DataStructure.h#L335.

[32] DSAGenerator pass (DSA-IDL-Generator Framework). https://github.com/AftabHussain/DataStructureAnalysis/blob/dsa_llvm3.8/lib/dsaGenerator/DSAGenerator.cpp.

[33] read_offset_begin() usage. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/dsaGenerator/DSAGenerator.cpp#L375.

[34] DSA does not do global alias analysis. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/DSA/TypeSafety.cpp#L69.

[35] Global alias analysis not working in DSA. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/DSA/Local.cpp#L306.

[36] Address taken analysis. https://github.com/AftabHussain/DataStructureAnalysis/blob/c92597f23fe64a3851a82a92c3effcb2f34ab5a3/lib/DSA/AddressTakenAnalysis.cpp#L1.

[37] DSCallSite data structure (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/92fcca27d70335d3b1493121bbb4478c703a2114/include/dsa/DSSupport.h#L154.

[38] CallSite data structure (LLVM). https://github.com/llvm-mirror/llvm/blob/release_38/include/llvm/IR/CallSite.h.

[39] Function data structure (LLVM). https://github.com/llvm-mirror/llvm/blob/release_38/include/llvm/IR/Function.h.

[40] Sample indirect call in an input program to DSA-IDL-Generator. https://github.com/AftabHussain/DataStructureAnalysis/blob/92fcca27d70335d3b1493121bbb4478c703a2114/example/device_manager.c#L79.

[41] Visit of a function pointer instruction in debug output. https://github.com/AftabHussain/DataStructureAnalysis/blob/92fcca27d70335d3b1493121bbb4478c703a2114/example/debug.out#L88.

[42] visitCallInst function (DSA). https://github.com/AftabHussain/DataStructureAnalysis/blob/08f042cba4155912c98c2837873bdefd6ad8640d/lib/DSA/Local.cpp#L976.

[43] Source code of LLVM, release 3.8.1. https://github.com/llvm-mirror/llvm/tree/release_38.

[44] LLVM Releases Download Page. http://releases.llvm.org/download.html.