Descriptions for files and folders found in this repo

Data

- binutils_compiled folder: compiled binaries for binutils
  - SW: binutils v2.26.1
  - ARCH: ARM, MIPS, x86
  - OP: O0-O3
- final_data folder: all json files containing info on ACFGs. The files in this folder are used for input into Gemini. Info contains namely the function name ("f_name"), successor(s) for every node with the index in the overarching list corresponding to the index of a node in the ACFG ("succs"), number of nodes in ACFG ("n_num"), and the attributes for each node in the ACFG ("features")
  - SW: OpenSSL v1.0.1u, v1.0.1f
  - ARCH: ARM, MIPS, x86
  - OP: O0-O3
  - GCC v5.4
- openssl_compiled folder: all compiled binaries corresponding to the specifications in final_data. This is the beginning point where the binaries in this folder are input into the ACFG extractor and the output are supposed to be the files in the final_data folder
- openssl_decomp folder: decompiled from binaries in openssl_compiled folder. For input for LLM approaches
- partial_test_fn_100.txt: names of the functions randomly sampled to form the reduced test dataset
- decomp_test.json: to be used together with partial_test_fn_100.txt to get all files needed to be used to obtain code explanations using gpt-3.5-turbo (see get_summaries.py)
- summaries.csv: output from get_summaries.py
- ssl_data.csv: all files to be used to get retrieval results for both Gemini and LLM
- for_llm.csv: all files to be used to get AUC for both Gemini and LLM (random pairiwse, 50-50 positive and negative samples)

Code (LLM approaches)

- get_summaries.py: obtain code explanations using gpt-3.5-turbo with pseudo C functions as input. All code explanations stored in summaries.csv
- get_similarities_sbert.py: get sentence embeddings from code explanations using SBERT
- get_sim_codebertscore.py: obtain F1 score from pseudo C functions using CodeBERTScore

Code (Gemini)

- Refer to https://github.com/xiaojunxu/dnn-binary-code-similarity for official code to implement Gemini
- Refer to https://github.com/qian-feng/Gencoding for unofficial code to implement ACFG extractor

- See section below for a detailed explanation on how the ACFG extraction works and how to modify the code to run it correctly.

ACFG Extraction Code (Gencoding/qian-feng, as per Genius)

- Note that the procedure for ACFG extraction is the same for Genius and Gemini
- Requirements
  - IDA Pro 7.1
  - Python 2.7.18
  - Networkx 2.2
- Only scripts in the raw-feature-extractor folder are relevant for this task
- How to run the extraction code
  - C:\path\to\ida64.exe -SC:\path\to\Gencoding-master\raw-feature-extractor\preprocessing_ida.py -c C:\path\to\elf
    - This is the command to generate the ida file. When this is run, IDA Pro will open up and starts to analyse the input file
    - -S switch is to execute a script file when the database is open
    - -c switch is to disassemble a new file and to delete the old database, if any
    - https://hex-rays.com/products/ida/support/idadoc/417.shtml -> documentation for IDA Pro command line switches
  - raw_graphs.py
    - Line 3 sys.path.insert(0, ‘C:/Python27/Lib/site-packages’)
      - This is for IDA Pro to detect the networkx package installed for Python 2.7.x
      - Change to your Python 2.7.x path
  - preprocessing_ida.py
    - Line 21 Set ‘path’ to the path where you want to output the ida files to
- What was the ACFG extraction process like?
  - Reference to this Chinese blog: https://www.cnblogs.com/lqerio/p/15572511.html#1.3%20%E8%BE%93%E5%87%BAACFG
  - preprocessing_ida.py is the main programme to extract the ACFG
  - From preprocessing_ida.py, line 22 cfgs = get_func_cfgs_c(FirstSeg()) is the main function which does (1) retrieval of CFGs, and (2) extraction of attributes which will make up the ACFG

```python
if __name__ == '__main__':

        args = parse_command()
        path = args.path
        analysis_flags = idc.GetShortPrm(idc.INF_START_AF)
        analysis_flags &= ~idc.AF_IMMOFF
        # turn off "automatically make offset" heuristic
        idc.SetShortPrm(idc.INF_START_AF, analysis_flags)
        idaapi.autoWait()
        cfgs = get_func_cfgs_c(FirstSeg())
        binary_name = idc.GetInputFile() + '.ida'
        fullpath = os.path.join(path, binary_name)
        pickle.dump(cfgs, open(fullpath,'w'))
        print binary_name
        idc.Exit(0)
```

get_func_cfgs_c found is the first function called in preprocessing_ida.py

```python
def get_func_cfgs_c(ea):
        binary_name = idc.GetInputFile()
        raw_cfgs = raw_graphs(binary_name)
        externs_eas, ea_externs = processpltSegs()
        i = 0
        for funcea in Functions(SegStart(ea)):
                funcname = get_unified_funcname(funcea)
                func = get_func(funcea)
                print i
                i += 1
                icfg = cfg.getCfg(func, externs_eas, ea_externs)
                func_f = get_discoverRe_feature(func, icfg[0])
                raw_g = raw_graph(funcname, icfg, func_f)
                raw_cfgs.append(raw_g)

        return raw_cfgs
```

get_func_cfgs_c as seen in func.py

- As seen in the screenshot above, the function first gets the input binary as specified in the command line, then instantiates an raw_graphs object with the binary_name.
- The function processpltSegs() first gets the list of all segments (line 224) and for each segment (line 225), it gets the linear address of the start of the segment (226), then uses this address to get the name of the segment (line 227). If the segment name contains '.plt', 'extern' or '.MIPS.stubs', the start and end addresses of that segment is used (lines 228-230) to get the function name and the current address of that function. The 2 dictionaries, funcdata and datafunc, are used to store the function name and address information, where for funcdata the key is the function name while the value is the function address whereas it is the opposite for datafunc (as seen in the reversed dictionary names), where the key is the function address and the value is the function name. Hence, processpltSegs() returns a dictionary of function names and their corresponding addresses, and a dictionary of function addresses and their corresponding names (the dictionaries are the reverse of each other).

```python
def processpltSegs():
        funcdata = {}
        datafunc = {}
        for n in xrange(idaapi.get_segm_qty()):
                seg = idaapi.getnseg(n)
                ea = seg.startEA
                segname = SegName(ea)
                if segname in ['.plt', 'extern', '.MIPS.stubs']:
                        start = seg.startEA
                        end = seg.endEA
                        cur = start
                        while cur < end:
                                name = get_unified_funcname(cur)
                                funcdata[name] = hex(cur)
                                datafunc[cur]= name
                                cur = NextHead(cur)
        return funcdata, datafunc
```

processpltSegs() function in func.py

- After getting these 2 dictionaries from processpltSegs(), every function in the input binary is iterated through from the start of the first segment (line 115). For each function, the function name ‘funcname’ is retrieved from get_unified_funcname (line 116) and the pointer to the function ‘func’ is retrieved from get_func (line 117).
- Next, the CFG is obtained from the getCfg function which can be found in cfg_constructor.py (line 120). The pointer to the function ‘func’, and the 2 dictionaries obtained from processpltSegs() are passed in as arguments to getCfg. The CFG is a networkx DiGraph (line 18). For every block in control blocks, unvisited nodes are visited and are added to the CFG. At the end of getCfg, the attributingRe function is called which adds 9 attributes to every node in the newly generated CFG, namely numIns, numCalls, numLIs, numAs, numNc, consts, strings, externs, and numTIs. To note that this isn’t the final list of attributes to be used for the ACFG.
- After the CFG is obtained, the get_discoverRe_feature function from discovRe.py is called (line 121), returning a list of 11 attributes including FunctionCalls, LogicInstr, Transfer, Locals, BB, Edges, Incoming, Instrs, between, strings, and consts. Note that this is where the ‘betweeness’ attribute mentioned in the paper comes from. This list of features stored in the variable ‘func_f’ in the main get_func_cfgs_c function is then passed as an argument into the raw_graph constructor along with the function name and the CFG of that function to generate the raw ACFG. (cont. below after the screenshots)

```python
def getCfg(func, externs_eas, ea_externs):
        func_start = func.startEA
        func_end = func.endEA
        cfg = nx.DiGraph()
        control_blocks, main_blocks = obtain_block_sequence(func)
        i = 0
        visited = {}
        start_node = None
        for bl in control_blocks:
                start = control_blocks[bl][0]
                end = control_blocks[bl][1]
                src_node = (start, end)
                if src_node not in visited:
                        src_id = len(cfg)
                        visited[src_node] = src_id
                        cfg.add_node(src_id)
                        cfg.node[src_id]['label'] = src_node
                else:
                        src_id = visited[src_node]

                #if end in seq_blocks and GetMnem(PrevHead(end)) != 'jmp':
                if start == func_start:
                        cfg.node[src_id]['c'] = "start"
                        start_node = src_node
                if end == func_end:
                        cfg.node[src_id]['c'] = "end"
                #print control_ea, 1
                refs = CodeRefsTo(start, 0)
                for ref in refs:
                        if ref in control_blocks:
                                dst_node = control_blocks[ref]
                                if dst_node not in visited:
                                        visited[dst_node] = len(cfg)
                                dst_id = visited[dst_node]
                                cfg.add_edge(dst_id, src_id)
                                cfg.node[dst_id]['label'] = dst_node
                #print control_ea, 1
                refs = CodeRefsTo(start, 1)
                for ref in refs:
                        if ref in control_blocks:
                                dst_node = control_blocks[ref]
                                if dst_node not in visited:
                                        visited[dst_node] = len(cfg)
                                dst_id = visited[dst_node]
                                cfg.add_edge(dst_id, src_id)
                                cfg.node[dst_id]['label'] = dst_node
        #print "attributing"
        attributingRe(cfg, externs_eas, ea_externs)
        # removing deadnodes
        #old_cfg = copy.deepcopy(cfg)
        #transform(cfg)
        return cfg, 0
```

getCfg(func, externs_eas, ea_externs) from cfg_constructor.py

```python
def attributingRe(cfg, externs_eas, ea_externs):
        for node_id in cfg:
                bl = cfg.node[node_id]['label']
                numIns = calInsts(bl)
                cfg.node[node_id]['numIns'] = numIns
                numCalls = calCalls(bl)
                cfg.node[node_id]['numCalls'] = numCalls
                numLIs = calLogicInstructions(bl)
                cfg.node[node_id]['numLIs'] = numLIs
                numAs = calArithmeticIns(bl)
                cfg.node[node_id]['numAs'] = numAs
                strings, consts = getBBconsts(bl)
                cfg.node[node_id]['numNc'] = len(strings) + len(consts)
                cfg.node[node_id]['consts'] = consts
                cfg.node[node_id]['strings'] = strings
                externs = retrieveExterns(bl, ea_externs)
                cfg.node[node_id]['externs'] = externs
                numTIs = calTransferIns(bl)
                cfg.node[node_id]['numTIs'] = numTIs
```

attributingRe function from cfg_constructor.py

```python
def get_discoverRe_feature(func, icfg):
    start = func.startEA
    end = func.endEA
    features = []
    FunctionCalls = getFuncCalls(func)
    #1
    features.append(FunctionCalls)
    LogicInstr = getLogicInsts(func)
    #2
    features.append(LogicInstr)
    Transfer = getTransferInsts(func)
    #3
    features.append(Transfer)
    Locals = getLocalVariables(func)
    #4
    features.append(Locals)
    BB = getBasicBlocks(func)
    #5
    features.append(BB)
    Edges = len(icfg.edges())
    #6
    features.append(Edges)
    Incoming = getIncommingCalls(func)
    #7
    features.append(Incoming)
    #8
    Instrs = getIntrs(func)
    features.append(Instrs)
    between = retrieveGP(icfg)
    #9
    features.append(between)

    strings, consts = getfunc_consts(func)
    features.append(strings)
    features.append(consts)
    return features
```

get_discoverRe_feature function from discovRe.py

- During the creation of the ACFG, the raw_graph object is instantiated with funcname as ‘funcname’ (name of function), the CFG g as ‘old_g’ and the list of features obtained from get_discoverRe_feature as ‘fun_features’. **Hence to note that ‘fun_features’ isn’t the list of attributes we’re looking for in the ACFG**. The attributing() function is called, which then creates the ACFG ‘g’ and the associated features by calling retrieveVec. A list of 8 attributes is returned from retrieveVec, namely consts, strings, offs, numAs, numCalls, numIns, numLIs, and numTIs. **To reiterate, the ACFG is ‘g’ and not ‘old_g’, while the attributes of the ACFG are found in ‘g’ (self.g.node[node_id][‘v’] gives the list of attributes) and were generated from the retrieveVec function, all of which can be found in raw_graphs.py.**

```python
class raw_graph:
    def __init__(self, funcname, g, func_f):
        self.funcname = funcname
        self.old_g = g[0]
        self.g = nx.DiGraph()
        self.entry = g[1]
        self.fun_features = func_f
        self.attributing()
```

```python
    def attributing(self):
        self.obtainOffsprings(self.old_g)
        for node in self.old_g:
            fvector = self.retrieveVec(node, self.old_g)
            self.g.add_node(node)
            self.g.node[node]['v'] = fvector

        for edge in self.old_g.edges():
            node1 = edge[0]
            node2 = edge[1]
            self.g.add_edge(node1, node2)

    def obtainOffsprings(self,g):
        nodes = g.nodes()
        for node in nodes:
            offsprings = {}
            self.getOffsprings(g, node, offsprings)
            g.node[node]['offs'] = len(offsprings)
        return g

    def getOffsprings(self, g, node, offsprings):
        node_offs = 0
        sucs = g.successors(node)
        for suc in sucs:
            if suc not in offsprings:
                offsprings[suc] = 1
                self.getOffsprings(g, suc, offsprings)
```

```python
    def retrieveVec(self, id_, g):
        feature_vec = []
        #numC0
        numc = g.node[id_]['consts']
        feature_vec.append(numc)
        #nums1
        nums = g.node[id_]['strings']
        feature_vec.append(nums)
        #offsprings2
        offs = g.node[id_]['offs']
        feature_vec.append(offs)
        #numAs3
        numAs = g.node[id_]['numAs']
        feature_vec.append(numAs)
        # of calls4
        calls = g.node[id_]['numCalls']
        feature_vec.append(calls)
        # of insts5
        insts = g.node[id_]['numIns']
        feature_vec.append(insts)
        # of LIs6
        insts = g.node[id_]['numLIs']
        feature_vec.append(insts)
        # of TIs7
        insts = g.node[id_]['numTIs']
        feature_vec.append(insts)
        return feature_vec
```

- Eg of the features list: [[8L], ['L'], 15, 0, 0, 7, 0, 0] corresponding to [**consts, strings**, offs, numAs, numCalls, numIns, numLIs, numTIs]
    - #1 **consts (not numeric)**: list of long integers → get the length of the list as the numeric value
    - #2 **strings (not numeric)**: list of strings → get the length of the list as the numeric value
    - #3 offs: # offspring
    - #4 numAs: # arithmetic instructions
    - #5 numCalls: # calls
    - #6 numIns: # instructions
    - #7 numLIs: # logic instructions
    - #8 numTIs: # transfer instructions
    - *** missing betweeness?? → found in 'fun_features' instead of 'v' ***

Type	Attribute name
Block-level attributes	String Constants
	Numeric Constants
	No. of Transfer Instructions
	No. of Calls
	No. of Instructions
	No. of Arithmetic Instructions
Inter-block attributes	No. of offspring
	Betweenness

Table 1: Basic-block attributes