Clamav funcation call flow

(bytecode scan using llvm for JIT)

bytecode scan JIT mode with llvm explained

by eqmcc

Description

this document explains llvm's roll in clamav's bytecode scan with JIT mode
how it works in general:

1. write 'c-like' code according to virus signature or behavior
2. compile 'c-like' code to bytecode via clambc-compiler
3. load to db and trigger the bytecode on certain match
4. checking bytecode result and report match case

about llvm

http://llvm.org/

The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.

lib for llvm at clamav/libclamav/c++/llvm/lib

about bytecode in clamav

http://lwn.net/Articles/387426/

0.96's bytecode engine is the new release's most fundamental change, and has sparked its share of controversy. In previous releases, the creators of the virus signatures stored in ClamAV's database were limited to pattern-matching techniques to recognize malware. With the bytecode engine, signature creators can now

develop "logical" signatures that involve heuristics, complex routines, and even unpacking file contents for examination. It also theoretically allows signature creators to examine new file formats without waiting for the main ClamAV program to support them explicitly.

ClamAV can run bytecode-engine signatures through a built-in interpreter or through a Just-In-Time (JIT) compiler built with LLVM. The syntax of the signature definition language is described as "C-like," and although it has not been formally described in the project documentation, it is partially described in the ClamAV code itself inside the bytecode_api.h header file.

The developers responded with an explanation of the security measures taken to protect hosts from malicious or problematic routines in bytecode signatures. First, all bytecode distributed by the project will come with embedded source code that can be examined by the user with the clambc utility. Second, all bytecodes in the virus database will be cryptographically signed by the project to verify their integrity. Third, bytecodes themselves have access only to the limited ClamAV API, cannot access system calls or memory, and can only read from the currently-scanned file. Finally, bounds-checking and other security measures are inserted by the compiler and by LibClamAV itself. In addition, the entire feature can be deactivated with a simple line in the freshclam.conf configuration file.

Data structures

Test case

1. source-code and bytecode

1.1 source code

test_bytecode.c

```
VIRUSNAME_PREFIX("test_bytecode")
VIRUSNAMES("A","B")
TARGET(7)
SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(magic)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
DEFINE_SIGNATURE(magic,"61616262")        // the pattern as "aabb" in hex
SIGNATURES_END
bool logical_trigger (void)
{
        // @ clamav-bytecode-compiler/obj/Release/lib/clang/1.1/include/bytecode_local.h
        return count_match(Signatures.magic) != 1; // if "aabb" match count is '1', it's not a virus
}

int entrypoint (void)
{
        int count = count_match(Signatures.magic);
        if ( count == 3)    foundVirus("B"); // 3 matches of "aabb", find virus B
        else foundVirus ("A"); // other case, find virus A
        return 0;
}
```

1.2 source code

1.2.1 compile

compile the source code to bytecode file *test_bytecode.cbc* via following command:

```
clambc-compiler test_bytecode.c -o test_bytecode.cbc -O2
```

and the *test_bytecode.cbc* looks as bellow:

```
ClamBCafhndcbn`ae|aeegcgefbg```c``a```|bjacflfafmfbfcfmb`cnbicgcnbccafmbecmbgfffecdfdfacdfcc``bcaaap`clamcoincidencejb:4096
test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
Teddaaahdabahdacahdadahdaeahdafahdagahebbeebaeebcdebadaacb`bbadb`bdb`aahdb`db`b
Eaeaaaeb`e|amcgefdgfgifbgegcgnfafmfef``
Gd```hai`@`b`aC``a`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBad@`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBbd@`bcd@D```h`bad@A
b`bad@Ab`bad@Ac`bad@Ac`
A`b`bLadb`b`aa`b`b`b`b`Fagac
Bb`b`gbAd`aaaaeab`b`AcdTaaaaaaab
Bb`bababbaeAh`AodTcab`b@d
Bb`bacabbaeAf`AodTcab`b@dE
Sfeidbeeecendadmdedoe`ebeedfdidhehbbbdgefcgdgoebfigdgefcfofdfefbbibSfeidbeeecendadmdedcehbbbadbblbbbbdbbibSdeadbegdeddehbgcibSceidgdnda
ddeeebeedceoeddedcdldoebdedgdidnd
ddedcdldadbeedoeceidgdndaddeeebeedhbmfafgfifcfibSceidgdndaddeeebeedceoeddedcdldoeednddSSceidgdndaddeeebeedceoeddedfdoebdedgdidndSdde
dfdidndedoeceidgdndaddeeebeedhbmfafgfifcflbbbfcacfcacfcbcfcbcbbib`b`bobob`bdghfef`b`gafdgdgefbgnf`bafcg`bgbgbafafbfbfbb`bifnf`bhfefhg
ceidgdndaddeeebeedceoeedndddSbfofoflf`blfofgfifcfaflfoedgbgifgfgfefbg`bhbfgofifdfibSkgSobob`b`d`bcflfafmfaffgmbbfigdgefcfofdfefmbcfofmf`giflfefbgobofb
fjfobbeeflfefafcgefoblfifbfobcflfafnfgfobacnbacobifnfcflfegdfefobbfigdgefcfofdfefoelfofcfaflfnbhf
bgefdgegbgnf`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfib`babmc`backcSmgSSifnfdg`befnfdgbgig`gofifnfdg`bhbfgofifdfibSkgSifnfdg`bcfofeg
nfdg`bmc`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfibkc
ifff`bhb`bcfofegnfdg`bmcmc`bccib`b`bffofegnfdffeifbgegcghbbbbdbbibkc`bobob`bcc`bmfafdgcfhfefcg`bofff`bblciafafbfbfbldilb`bffifnfdf`bfgifbgegcg`bbdSeflfcg
ef`bffofegnfdffeifbgegcg`bhbbbadbbibkc`bobob`bofdghfefbg`bcfafcgeflb`bffifnfdf`bfgifbgegcg`bad
bgefdgegbgnf`b`ckcSmgSS
```

1.2.2 verify

verify the bytecode info via following command:

```
clambc --info test_bytecode.cbc
```

and output as bellow

```
Bytecode format functionality level: 6
Bytecode metadata:
        compiler version: clambc-0.97.3a-5-gf5dd1d3
        compiled on: (1359881038) Sun Feb  3 03:43:58 2013
        compiled by: user
        target exclude: 0
        bytecode type: logical only
        bytecode functionality level: 0 - 0
        bytecode logical signature: test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
        // ((0<1)|(0>1))means the logic signature will be matched if thesub logic sig's match count is not 1
        virusname prefix: (null)
        virusnames: 0
        bytecode triggered on: files matching logical signature
        number of functions: 1
```

```
number of types: 19
number of global constants: 9
number of debug nodes: 0
bytecode APIs used:
 setvirusname
```

2. run test

2.1 test files

test3.txt

```
aabbxxxxxxxxxxxxxxxxxxxxaabbxxxxxxxxxxxxaabb
```

2.2 test run

```
clamscan --bytecode=yes --bytecode-unsigned=yes    -d test_bytecode.cbc test3.txt
```

comments:
--bytecode=yes : enable bytecode scan
--bytecode-unsigned=yes : load unofficial bytecode

2.2.2 result

```
user@ubuntu:~/clamav$ clamscan --bytecode=yes --bytecode-unsigned=yes test2.txt
test3.txt: test_bytecode.B FOUND
----------- SCAN SUMMARY -----------
Known viruses: 1
Engine version: devel-a62bf02
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
Time: 0.074 sec (0 m 0 s)
```

init

bytecode_init
at libclamav/c++/bytecode2llvm.cpp

```
llvm_install_error_handler(llvm_error_handler);    <===> install_fatal_error_handler
      ErrorHandler = handler;
      ErrorHandlerUserData = user_data;

llvm_start_multithreaded();
      multithreaded_mode = true;
      global_lock = new sys::Mutex(true);
      // We fence here to ensure that all initialization is complete BEFORE we
      // return from llvm_start_multithreaded().
      sys::MemoryFence();

// If we have a native target, initialize it to ensure it is linked in and
// usable by the JIT.
#ifndef AC_APPLE_UNIVERSAL_BUILD // yes for this case
      InitializeNativeTarget();
#else
      InitializeAllTargets();
#endif
```

load bytecode

journey of a bytecode

after bytecode is generated via clambc-compiler, it will go through following steps to fulfill it's destine:

1. load and parser into data structure called cli_bc_ctx
2. pass the structured bytecode into llvm and run it

cli_ loadcbc

```
struct cli_all_bc *bcs = &engine->bcs;
struct cli_bc *bc;
```

bcs->all_bcs = cli_realloc2(bcs->all_bcs, sizeof(*bcs->all_bcs)*(bcs->count+1));
bcs->count++;
bc = &bcs->all_bcs[bcs->count-1];

cli_bytecode_load // read in the cbc file
load_oneldb // load logic signature in side the bytecode

if (bc->kind != BC_LOGICAL) // no for this case as kind=BC_LOGICAL
…

load and parse the bytecode – cli_bytecode_load

cli_bytecode_load // called by cli_loadcbc

parseHeader // parse line of bytecode head

```
ClamBCafhndcbn`ae|aeegcgefbg```c``a```|bjacflfafmfbfcfmb`cnbicgcnbccafmbecmbgfffecdfdfacdfcc``bcaaap`clamcoincidencejb:4096
```

state = PARSE_BC_LSIG;

case PARSE_BC_LSIG:
    parseLSig // parse line of logic signature

```
test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
```

    state = PARSE_BC_TYPES;

case PARSE_BC_TYPES:
    parseTypes // parse line of data type

```
Teddaaahdabahdacahdadahdaeahdafahdagahebbeebaeebcdebadaacb`bbadb`bdb`aahdb`db`b
```

    state = PARSE_BC_APIS;

case PARSE_BC_APIS:
    parseApis // parse line of api

```
Eaeaaaeb`e|amcgefdgfgifbgegcgnfafmfef``
```

    state = PARSE_BC_GLOBALS;

case PARSE_BC_GLOBALS:
    parseGlobals // parse line of global vars

```
Gd```hai`@`b`aC``a`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBad@`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBbd@`bcd@D```h`bad@Ab`bad@Ab`bad@Ac`bad@Ac`
```

    state = PARSE_MD_OPT_HEADER;

case PARSE_MD_OPT_HEADER:

if (buffer[0] == 'D') parseMD // no for this case

case PARSE_FUNC_HEADER:

parseFunctionHeader // parse line of function header, function structure details are set in this procedure

```
struct cli_bc_func {
    uint8_t numArgs;
    uint16_t numLocals;
    uint32_t numInsts;
    uint32_t numValues;/* without constants */
    uint32_t numConstants;
    uint32_t numBytes;/* stack size */
    uint16_t numBB;
    uint16_t returnType;
    uint16_t *types;
    uint32_t insn_idx;
    struct cli_bc_bb *BB;
    struct cli_bc_inst *allinsts;
    uint64_t *constants;
    unsigned *dbgnodes;
};
```

A`b`bLadb`b`aa`b`b`b`b`Fagac

state = PARSE_BB;

case PARSE_BB:

parseBB // parse line of basic block, bellow three lines will be checked

Bb`b`gbAd`aaaaeab`b`AcdTaaaaaaab

Bb`bababbaeAh`AodTcab`b@d

Bb`bacabbaeAf`AodTcab`b@dE

// if more functions – only one function in this case

if (bb >= bc->funcs[current_func].numBB)

state = PARSE_FUNC_HEADER; // do it all over again - PARSE_BB

current_func++;

case PARSE_SKIP:

if (buffer[0] == 'S') end = 1;

// line start with 'S' is source code line

Sfeidbeeecendadmdedoe`ebeedfdidhehbbbdgefcgdgoebfigdgefcfofdfefbbibSfeidbeeecendadmdedcehbbbadbblbbbbdbbibSdeadbegdeddehbgcibSceidgdndaddeeebeedceoeddedcdldoebdedgdidnddded cdldadbeedoeceidgdndaddeeebeedhbmfafgfifcfibSceidgdndaddeeebeedceoeddedcdldoeednddSSceidgdndaddeeebeedceoeddedfdoebdedgdidndSddedfdidndedoeceidgdndaddeeebeedhbmfafgfifcflbbbfcacfcacfcbcfcbcbbib`b`bobob`bdghfef`b`gafdgdgefbgnf`bafcg`bgbgbafafbfbfbb`bifnf`bhfefhgceidgdndaddeeebeedceoeednddd Sbfofoflf`blfofgfifcfaflfoedgbgifgfgfefbg`bhbfgofifdfibSkgSobob`b`d`bcflfafmfaffgmbbfigdgefcfofdfefmbcfofmf`giflfefbgobofbfjfobbeeflfefafcgefoblfifbfobcflfafnfgfobacnbacobifnfcflfegdfefobbfigdgefcfofdfefoelfofcfaflfnbhfbgefdgegbgnf`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfib`babmc`backcSmgSSifnfdg`befnfdgbgig`gofifnfdg`bhbfgofifdfibSkgSifnfdg`bcfofegnfdg`bmc`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfibkcifff`bhb`bcfofegnfdg`bmcmc`bccib`b`bffofegnfdffeifbgegcghbbbbdbbibkc`bobob`bcc`bmfafdgcfhfefcg`bofff`bblciafafbfbfbldilb`bffifnfdf`bfgifbgegcg`bbdSeflfcgef`bffofegnfdffeifbgegcg`bhbbbadbbibkc`bobob`bofdghfefbg`bcfafcgeflb`bffifnfdf`bfgifbgegcg`badbgefdgegbgnf`b`ckcSmgSS

enrich - struct cli_bc *bc

bc->trusted = trust;

parseHeader

bc->metadata.formatlevel = readNumber(buffer, &offset, len, &ok);

```
bc->metadata.timestamp = readNumber(buffer, &offset, len, &ok);
bc->metadata.sigmaker = readString(buffer, &offset, len, &ok);
bc->metadata.targetExclude = readNumber(buffer, &offset, len, &ok);
bc->kind = readNumber(buffer, &offset, len, &ok);
bc->metadata.minfunc = readNumber(buffer, &offset, len, &ok);
bc->metadata.maxfunc = readNumber(buffer, &offset, len, &ok);
bc->metadata.maxresource = readNumber(buffer, &offset, len, &ok);
bc->metadata.compiler = readString(buffer, &offset, len, &ok);
bc->num_types = readNumber(buffer, &offset, len, &ok);
bc->num_func = readNumber(buffer, &offset, len, &ok);
bc->state = bc_loaded;
bc->uses_apis = NULL;
bc->dbgnodes = NULL;
bc->dbgnode_cnt = 0;
bc->funcs = cli_calloc(bc->num_func, sizeof(*bc->funcs));
bc->types = cli_calloc(bc->num_types, sizeof(*bc->types));
```

parseLSig

```
bc->lsig = cli_strdup(buffer);
```

parseTypes

```
bc->start_tid = readFixedNumber(buffer, &offset, len, &ok, 2);
```

parseApis

```
bc->uses_apis = cli_bitset_init();
for (i=0;i < calls; i++) cli_bitset_set(bc->uses_apis, id); // set up api calls
```

74 apis registered in the llvm system
these apis are defined in libclamav/bytecode_api.h

```
parseApis cli_apicalls[0].name=test1
parseApis cli_apicalls[1].name=read
parseApis cli_apicalls[2].name=write
parseApis cli_apicalls[3].name=seek
parseApis cli_apicalls[4].name=setvirusname
parseApis cli_apicalls[5].name=debug_print_str
parseApis cli_apicalls[6].name=debug_print_uint
parseApis cli_apicalls[7].name=disasm_x86
parseApis cli_apicalls[8].name=trace_directory
parseApis cli_apicalls[9].name=trace_scope
parseApis cli_apicalls[10].name=trace_source
parseApis cli_apicalls[11].name=trace_op
parseApis cli_apicalls[12].name=trace_value
parseApis cli_apicalls[13].name=trace_ptr
parseApis cli_apicalls[14].name=pe_rawaddr
```

```
parseApis cli_apicalls[15].name=file_find
parseApis cli_apicalls[16].name=file_byteat
parseApis cli_apicalls[17].name=malloc
parseApis cli_apicalls[18].name=test2
parseApis cli_apicalls[19].name=get_pe_section
parseApis cli_apicalls[20].name=fill_buffer
parseApis cli_apicalls[21].name=extract_new
parseApis cli_apicalls[22].name=read_number
parseApis cli_apicalls[23].name=hashset_new
parseApis cli_apicalls[24].name=hashset_add
parseApis cli_apicalls[25].name=hashset_remove
parseApis cli_apicalls[26].name=hashset_contains
parseApis cli_apicalls[27].name=hashset_done
parseApis cli_apicalls[28].name=hashset_empty
parseApis cli_apicalls[29].name=buffer_pipe_new
parseApis cli_apicalls[30].name=buffer_pipe_new_fromfile
parseApis cli_apicalls[31].name=buffer_pipe_read_avail
parseApis cli_apicalls[32].name=buffer_pipe_read_get
parseApis cli_apicalls[33].name=buffer_pipe_read_stopped
parseApis cli_apicalls[34].name=buffer_pipe_write_avail
parseApis cli_apicalls[35].name=buffer_pipe_write_get
parseApis cli_apicalls[36].name=buffer_pipe_write_stopped
parseApis cli_apicalls[37].name=buffer_pipe_done
parseApis cli_apicalls[38].name=inflate_init
parseApis cli_apicalls[39].name=inflate_process
parseApis cli_apicalls[40].name=inflate_done
parseApis cli_apicalls[41].name=bytecode_rt_error
parseApis cli_apicalls[42].name=jsnorm_init
parseApis cli_apicalls[43].name=jsnorm_process
parseApis cli_apicalls[44].name=jsnorm_done
parseApis cli_apicalls[45].name=ilog2
parseApis cli_apicalls[46].name=ipow
parseApis cli_apicalls[47].name=iexp
parseApis cli_apicalls[48].name=isin
parseApis cli_apicalls[49].name=icos
parseApis cli_apicalls[50].name=memstr
parseApis cli_apicalls[51].name=hex2ui
parseApis cli_apicalls[52].name=atoi
parseApis cli_apicalls[53].name=debug_print_str_start
parseApis cli_apicalls[54].name=debug_print_str_nonl
parseApis cli_apicalls[55].name=entropy_buffer
parseApis cli_apicalls[56].name=map_new
parseApis cli_apicalls[57].name=map_addkey
parseApis cli_apicalls[58].name=map_setvalue
```

```
parseApis cli_apicalls[59].name=map_remove
parseApis cli_apicalls[60].name=map_find
parseApis cli_apicalls[61].name=map_getvaluesize
parseApis cli_apicalls[62].name=map_getvalue
parseApis cli_apicalls[63].name=map_done
parseApis cli_apicalls[64].name=file_find_limit
parseApis cli_apicalls[65].name=engine_functionality_level
parseApis cli_apicalls[66].name=engine_dconf_level
parseApis cli_apicalls[67].name=engine_scan_options
parseApis cli_apicalls[68].name=engine_db_options
parseApis cli_apicalls[69].name=extract_set_container
parseApis cli_apicalls[70].name=input_switch
parseApis cli_apicalls[71].name=get_environment
parseApis cli_apicalls[72].name=disable_bytecode_if
parseApis cli_apicalls[73].name=disable_jit_if
```

9 apis called

```
LibClamAV info: DEBUG: in parseApis id=67, tid=98, name=engine_dconf_level
LibClamAV info: DEBUG: in parseApis id=66, tid=98, name=engine_functionality_level
LibClamAV info: DEBUG: in parseApis id=7, tid=99, name=debug_print_uint
LibClamAV info: DEBUG: in parseApis id=19, tid=99, name=test2
LibClamAV info: DEBUG: in parseApis id=6, tid=100, name=debug_print_str
LibClamAV info: DEBUG: in parseApis id=72, tid=101, name=get_environment
LibClamAV info: DEBUG: in parseApis id=1, tid=102, name=test1
LibClamAV info: DEBUG: in parseApis id=73, tid=103, name=disable_bytecode_if
LibClamAV info: DEBUG: in parseApis id=74, tid=103, name=disable_jit_if
```

```
parseGlobals //numglobals= 48
    bc->globals = cli_calloc(numglobals, sizeof(*bc->globals));
    bc->num_globals = numglobals;
    for (i=0;i<numglobals;i++)
        bc->globaltys[i] = readTypeID(bc, buffer, &offset, len, &ok);
        bc->globals[i] = cli_malloc(sizeof(*bc->globals[0])*comp);
        readConstant(bc, i, comp, buffer, &offset, len, &ok);
```

```
parseFunctionHeader //numLocals=156 numBB = 53
    func = &bc->funcs[fn]; // set a specific function
    func->numArgs = readFixedNumber(buffer, &offset, len, &ok, 1);
    func->returnType = readTypeID(bc, buffer, &offset, len, &ok);
    func->numLocals = readNumber(buffer, &offset, len, &ok);
    func->types = cli_calloc(all_locals, sizeof(*func->types));
    for (i=0;i<all_locals;i++)
```

```
        func->types[i] = readNumber(buffer, &offset, len, &ok);

    func->numInsts = readNumber(buffer, &offset, len, &ok);
    func->numValues = func->numArgs + func->numLocals;
    func->insn_idx = 0;
    func->numConstants=0;
    func->allinsts = cli_calloc(func->numInsts, sizeof(*func->allinsts));

    func->numBB = readNumber(buffer, &offset, len, &ok);
        func->BB = cli_calloc(func->numBB, sizeof(*func->BB));


parseBB //
    struct cli_bc_func *bcfunc = &bc->funcs[func];
    BB = &bcfunc->BB[bb];
    BB->numInsts = 0;
    BB->insts = &bcfunc->allinsts[bcfunc->insn_idx];
    BB->insts[BB->numInsts++] = inst;
    bcfunc->numBytes = 0;
    bcfunc->insn_idx += BB->numInsts;
```

compile the engine and bytecode testing run

call stack

```
==================== call stack ====================
cl_engine_compile
    cli_bytecode_prepare2
        run_builtin_or_loaded
            cli_bytecode_prepare_interpreter
            cli_bytecode_context_setfuncid
            cli_bytecode_run
        cli_bytecode_context_getresult_int
        selfcheck
            cli_bytecode_prepare_jit //libclamav/c++/bytecode2llvm.cpp
            cli_bytecode_prepare_interpreter
```

```
        run_selfcheck
    cli_bytecode_prepare_jit //libclamav/c++/bytecode2llvm.cpp
    cli_bytecode_done_jit
    cli_bytecode_prepare_interpreter
```

Scan

call stack

```
==================== the scan call stack ============================
scanfile
  cl_scandesc_callback
    scan_common
      cli_magic_scandesc
        magic_scandesc
          CL_TYPE_TEXT_ASCII //yes
              cli_scan_structured // no
          cli_scanraw //yes
              cli_fmap_scandesc //yes
                  matcher_run //yes
                      cli_bm_scanbuff
                      cli_ac_scanbuff
                  if(groot) cli_lsig_eval // yes but return clean
          cli_scanscript
              cli_scanbuff
                  matcher_run
                      cli_ac_scanbuff
              cli_lsig_eval
                  cli_bytecode_runlsig
                      cli_bytecode_context_setfuncid
                      cli_bytecode_context_setctx
                      cli_bytecode_context_setfile
                      cli_bytecode_run
                          cli_event_time_start
                          cli_vm_execute_jit //libclamav/c++/bytecode2llvm.cpp
                          cli_event_time_stop
                      cli_bytecode_context_getresult_int
```

from clamav to llvm

how loaded bytecode, scan environment and primary scan results are passed into llvm?

1. for bytecode, there are variables and functions to be passed in

bytecode will be called in follow fashion:

```
ret = cli_vm_execute_jit(bcs, ctx, &bc->funcs[ctx->funcid]);
```

the definition of the function is as bellow:

```
int cli_vm_execute_jit(const struct cli_all_bc *bcs, struct cli_bc_ctx *ctx,
                       const struct cli_bc_func *func)
```

all the info of the bytecode are loaded into cli_bc structure at cli_bytecode_load

three data structure are passed in:

cli_all_bc

```
struct cli_all_bc {
    struct cli_bc *all_bcs;
    unsigned count;
    struct cli_bcengine *engine;
    struct cli_environment env;
    int    inited;
};
```

cli_bc

```
struct cli_bc {
  struct bytecode_metadata metadata;
  unsigned id;
  unsigned kind;
  unsigned num_types;
  unsigned num_func;
  struct cli_bc_func *funcs;
  struct cli_bc_type *types;
  uint64_t **globals;
  uint16_t *globaltys;
  size_t num_globals;
  enum bc_state state;
  struct bitset_tag *uses_apis;
  char *lsig;
  char *vnameprefix;
  char **vnames;
  unsigned vnames_cnt;
  uint16_t start_tid;
  struct cli_bc_dbgnode *dbgnodes;
  unsigned dbgnode_cnt;
  unsigned hook_lsig_id;
  unsigned trusted;
  uint32_t numGlobalBytes;
  uint8_t *globalBytes;
};
```

engine - bytecode2llvm.cpp

```
struct cli_bcengine {
    ExecutionEngine *EE;
    JITEventListener *Listener;
    LLVMContext Context;
    FunctionMapTy compiledFunctions;
    union {
    unsigned char b[16];
    void* align;/* just to align field to ptr */
    } guard;
};
```

cli_bc_ctx

```
struct cli_bc_ctx {
    uint8_t timeout;/* must be first byte in struct! */
    uint16_t funcid;
    unsigned numParams;
    /* id and params of toplevel function called */
    const struct cli_bc *bc;
    const struct cli_bc_func *func;
    uint32_t bytecode_timeout;
    unsigned bytes;
    uint16_t *opsizes;
    char *values;
    operand_t *operands;
    uint32_t file_size;
    int outfd;
    off_t off;
    fmap_t *fmap;
    fmap_t *save_map;
    const char *virname;
    struct cli_bc_hooks hooks;
    struct cli_exe_info exeinfo;
    uint32_t lsigcnt[64];
    uint32_t lsigoff[64];
    uint32_t pdf_nobjs;
    struct pdf_obj *pdf_objs;
    uint32_t* pdf_flags;
    uint32_t pdf_size;
    uint32_t pdf_startoff;
    unsigned pdf_phase;
    int32_t pdf_dumpedid;
    const struct cli_exe_section *sections;
    uint32_t resaddr;
    char *tempfile;
    void *ctx;
    unsigned written;
    unsigned filewritten;
    unsigned found;
    unsigned ninflates;
    bc_dbg_callback_trace trace;
    bc_dbg_callback_trace_op trace_op;
    bc_dbg_callback_trace_val trace_val;
    bc_dbg_callback_trace_ptr trace_ptr;
```

```
    const char *directory;
    const char *file;
    const char *scope;
    unsigned trace_level;
    uint32_t scopeid;
    unsigned line;
    unsigned col;
    mpool_t *mpool;
    struct bc_inflate* inflates;
    struct bc_buffer *buffers;
    unsigned nbuffers;
    unsigned nhashsets;
    unsigned njsnorms;
    unsigned jsnormwritten;
    struct cli_hashset *hashsets;
    struct bc_jsnorm* jsnorms;
    char *jsnormdir;
    struct cli_map *maps;
    unsigned nmaps;
    unsigned containertype;
    unsigned extracted_file_input;
    const struct cli_environment *env;
    unsigned bytecode_disable_status;
    cli_events_t *bc_events;
    int on_jit;
    int no_diff;
};
```

cli_bc_func

```
struct cli_bc_func {
    uint8_t numArgs;
    uint16_t numLocals;
    uint32_t numInsts;
    uint32_t numValues;/* without constants */
    uint32_t numConstants;
    uint32_t numBytes;/* stack size */
    uint16_t numBB;
    uint16_t returnType;
    uint16_t *types;
    uint32_t insn_idx;
    struct cli_bc_bb *BB;
    struct cli_bc_inst *allinsts;
    uint64_t *constants;
    unsigned *dbgnodes;
};
```

2. for scan results

any scan results and other related info will be passed in via registered globals

the definition

in libclamav/bytecode_api.h

```
extern const uint32_t __clambc_match_counts[64];
extern const uint32_t __clambc_match_offsets[64];
extern const struct cli_pe_hook_data __clambc_pedata;
extern const uint32_t __clambc_filesize[1];
const uint16_t __clambc_kind;
```

in libclamav/bytecode_api_decl.c

```
const struct cli_apiglobal cli_globals[] = {
/* Bytecode globals BEGIN */
    {"__clambc_match_offsets", GLOBAL_MATCH_OFFSETS, 76,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.match_offsets - (char*)NULL)},
    {"__clambc_kind", GLOBAL_KIND, 16,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.kind - (char*)NULL)},
    {"__clambc_match_counts", GLOBAL_MATCH_COUNTS, 76,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.match_counts - (char*)NULL)},
    {"__clambc_filesize", GLOBAL_FILESIZE, 75,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.filesize - (char*)NULL)},
    {"__clambc_pedata", GLOBAL_PEDATA, 69,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.pedata - (char*)NULL)}
/* Bytecode globals END */
};
```

these globals have equivalents at cli_bc_hooks

```
struct cli_bc_hooks {
     const uint32_t* match_offsets;
     const uint16_t* kind;
     const uint32_t* match_counts;
     const uint32_t* filesize;
     const struct cli_pe_hook_data* pedata;
};
#endif
```

the enrichment

in cli_bytecode_runlsig

struct cli_bc_ctx ctx;

```
const struct cli_bc *bc = &bcs->all_bcs[bc_idx-1];
cli_bytecode_context_setfuncid(&ctx, bc, 0);
        func = ctx->func = &bc->funcs[funcid];
        ctx->bc = bc;
        ctx->numParams = func->numArgs;
        ctx->funcid = funcid;
ctx.hooks.match_counts = lsigcnt;
ctx.hooks.match_offsets = lsigsuboff;
cli_bytecode_context_setctx(&ctx, cctx);
cli_bytecode_context_setfile(&ctx, map);
        ctx->hooks.filesize = &ctx->file_size;
if (tinfo && tinfo->status == 1)
        ctx.sections = tinfo->exeinfo.section;
        pehookdata.offset = tinfo->exeinfo.offset;
        pehookdata.ep = tinfo->exeinfo.ep;
        pehookdata.nsections = tinfo->exeinfo.nsections;
        pehookdata.hdr_size = tinfo->exeinfo.hdr_size;
        ctx.hooks.pedata = &pehookdata;
        ctx.resaddr = tinfo->exeinfo.res_addr;

a detailed look into cli_bytecode_context_setfuncid(&ctx, bc, 0);
// how many functions are defined for a bytecode is logged via variable current_func defined inside cli_bytecode_load and used in
// rc = parseFunctionHeader(bc, current_func, (unsigned char*)buffer);
// and for now, suppose it's only support 1 function in bytecode as we always call cli_bytecode_context_setfuncid using funcid=0
cli_bytecode_context_setfuncid // this function will always be called with funcid=0
        func = ctx->func = &bc->funcs[funcid]; // funcid=0
        ctx->funcid = funcid;
```

cli_bytecode_runlsig

```
==================== cli_bytecode_runlsig ====================
// set functions defined in bytecode
cli_bytecode_context_setfuncid(&ctx, bc, 0);
    func = ctx->func = &bc->funcs[funcid]; // set functions defined by cbc code
    ctx->bc = bc; // set cli_bc of context
    ctx->numParams = func->numArgs; // set number of args of func, 0 in this case
    ctx->funcid = funcid; // set current function id
    if (func->numArgs) // set arg if necessary, no for this case
    s += 8;/* return value */
    ctx->bytes = s; // how many bytes are used for this func

/*
cli_bc_hooks
struct cli_bc_hooks {
    const uint32_t* match_offsets;
    const uint16_t* kind;
    const uint32_t* match_counts;
    const uint32_t* filesize;
    const struct cli_pe_hook_data* pedata;
};
*/

//Bytecode test_bytecode.cbc(1) has logical signature:
// test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
// lsigcnt points to an array stores match count of each logic sub sig
// so the match count would be retrieved by lsigcnt[id] where id is the sub logic sig id
// in this case, for logic sig's first pattern "61616262", the match count is 2
// so array lsigcnt will only have one element that is lsigcnt[0]=2
ctx.hooks.match_counts = lsigcnt;
// inited via lsigsuboff_first
// match position in the buffer
// for "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxaabbaabb"
// it's 36, i.e.: "a" in red
ctx.hooks.match_offsets = lsigsuboff;

// for pe info
if (tinfo && tinfo->status == 1) // no for this case


if (bc->hook_lsig_id) // no for this case
    /* this is a bytecode for a hook, defer running it until hook is
     * executed, so that it has all the info for the hook */
    if (cctx->hook_lsig_matches)
```

```
            cli_bitset_set(cctx->hook_lsig_matches, bc->hook_lsig_id-1);
        /* save match counts */
        memcpy(&ctx.lsigcnt, lsigcnt, 64*4);
        memcpy(&ctx.lsigoff, lsigsuboff, 64*4);


/* Running bytecode for logical signature match*/
ret = cli_bytecode_run(bcs, bc, &ctx);

// find a virus
if (ctx.virname)
      // doing PUA scan if we matching heuristics sig
      if (!strncmp(ctx.virname, "BC.Heuristics", 13))
            rc = cli_found_possibly_unwanted(cctx);
      else // report a virus
            rc = CL_VIRUS;

// no virus found
// bytecode return an result code and get it
ret = cli_bytecode_context_getresult_int(&ctx);
```

cli_bytecode_run

========== cli_bytecode_run ==========

cli_bc_inst

```
struct cli_bc_inst {
    enum bc_opcode opcode;
    uint16_t type;
    operand_t dest;
    interp_op_t interp_op;/* opcode for interpreter */
    union {
    operand_t unaryop;
    struct cli_bc_cast cast;
    operand_t binop[2];
    operand_t three[3];
    struct cli_bc_callop ops;
    struct branch branch;
    bbid_t jump;
    } u;
};
```

cli_bc_func

```
struct cli_bc_func {
    uint8_t numArgs;
    uint16_t numLocals;
    uint32_t numInsts;
    uint32_t numValues;/* without constants */
    uint32_t numConstants;
    uint32_t numBytes;/* stack size */
    uint16_t numBB;
    uint16_t returnType;
    uint16_t *types;
    uint32_t insn_idx;
    struct cli_bc_bb *BB;
    struct cli_bc_inst *allinsts;
    uint64_t *constants;
    unsigned *dbgnodes;
};
```

```
struct cli_events;
typedef struct cli_events cli_events_t;
struct cli_events {
    struct cli_event *events;
    struct cli_event errors;
    uint64_t oom_total;
    unsigned max;
    unsigned oom_count;
};
```

```
// some local vars before vm execute
struct cli_bc_inst inst;
struct cli_bc_func func;
cli_events_t *jit_ev = NULL, *interp_ev = NULL;

// get running env
ctx->env = &bcs->env;
context_safe(ctx); /* make sure some vars in ctx are never NULL */

if (test_mode) // not in test mode for this case
if (bc->state == bc_interp || test_mode) // no for this case
    cli_event_time_start(interp_ev, BCEV_EXEC_TIME);
    ret = cli_vm_execute(ctx->bc, ctx, &func, &inst);
    cli_event_time_stop(interp_ev, BCEV_EXEC_TIME);
    cli_event_int(interp_ev, BCEV_EXEC_RETURNVALUE, ret);
    cli_event_string(interp_ev, BCEV_VIRUSNAME, ctx->virname);


// deal with JIT mode as for this case
// bc->state == bc_jit
//bc_state
```

```
enum bc_state {
    bc_skip,
    bc_loaded,
    bc_jit,
    bc_interp,
    bc_disabled
};
```

```
if (bc->state == bc_jit || test_mode)
      ctx->bc_events = jit_ev;
      ctx->on_jit = 1;
      // execute
       cli_event_time_start(jit_ev, BCEV_EXEC_TIME);
       ret = cli_vm_execute_jit(bcs, ctx, &bc->funcs[ctx->funcid]);
            // called from libclamav/c++/bytecode2llvm.cpp
       cli_event_time_stop(jit_ev, BCEV_EXEC_TIME);
      // post execute
       cli_event_int(jit_ev, BCEV_EXEC_RETURNVALUE, ret);
       // get the virus name identified according bytecode executed by JIT vm
       // ctx->virname will be used after return as a judgment of if there's a match or
not
       cli_event_string(jit_ev, BCEV_VIRUSNAME, ctx->virname);
```

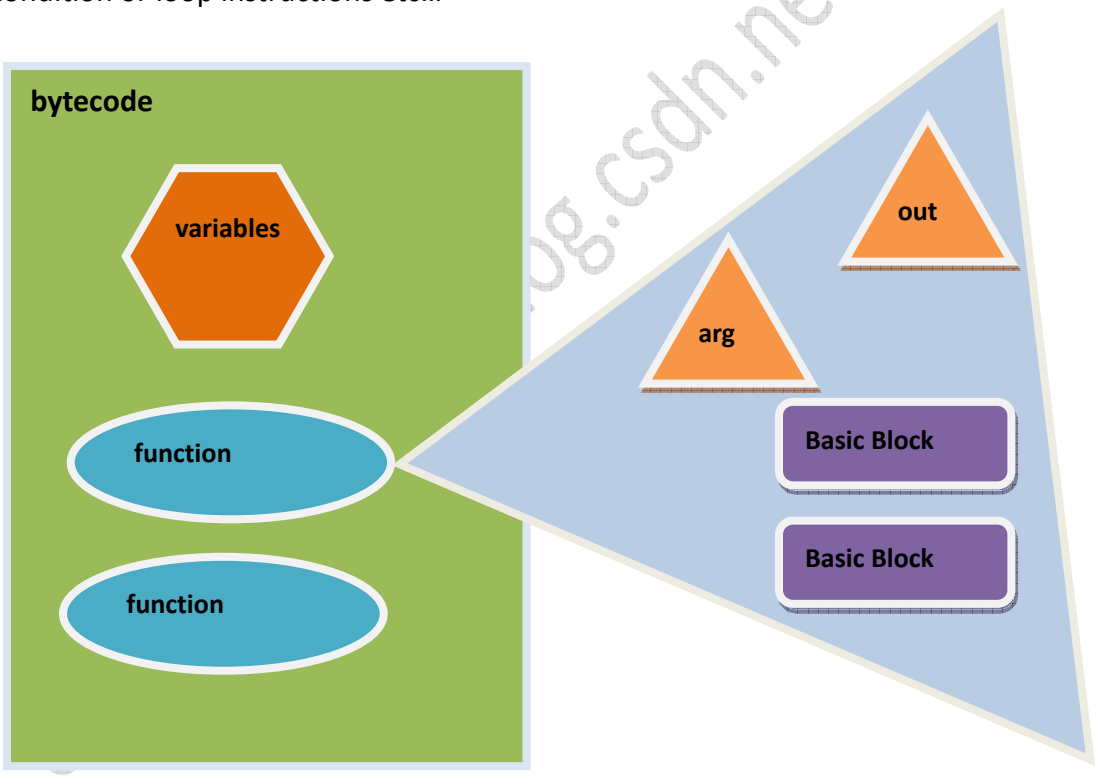
functions in libclamav/c++/bytecode2llvm.cpp

following functions are used in above process while doing scan in JIT mode

cli_bytecode_prepare_jit

Summary

in this function, bytecode functions will be translated to execution blocks.
bytecode consists of variables and functions, and a function consists of input arguments, return type and codes(instructions) and finally lines of codes form a BB(basic block). BB is a fundamental execution element that will be called by condition or loop instructions etc…



call stack

========== call stack of this function ==========
builder.create
addFunctionProtos
addFPasses
Function::Create

EE->addGlobalMapping(F, dest)
EE->getPointerToFunction(F)
Codegen.generate()
PM.add
EE->getPointerToFunction(Functions[i])

data structures

========== classes used in this function ==========
EngineBuilder
/// EngineBuilder - Builder class for ExecutionEngines. Use this by
/// stack-allocating a builder, chaining the various set* methods, and
/// terminating it with a .create() call.

ExecutionEngine

```
  const TargetData *TD;
  ExecutionEngineState EEState;
  bool CompilingLazily;
  bool GVCompilationDisabled;
  bool SymbolSearchingDisabled;
  friend class EngineBuilder;  // To allow access to JITCtor and InterpCtor.
protected:
  /// Modules - This is a list of Modules that we are JIT'ing from.  We use a
  /// smallvector to optimize for the case where there is only one module.
  SmallVector<Module*, 1> Modules;
```

ExecutionEngineState

```
public:
  struct AddressMapConfig : public ValueMapConfig<const GlobalValue*> {
    typedef ExecutionEngineState *ExtraData;
    static sys::Mutex *getMutex(ExecutionEngineState *EES);
    static void onDelete(ExecutionEngineState *EES, const GlobalValue *Old);
    static void onRAUW(ExecutionEngineState *, const GlobalValue *,
                       const GlobalValue *);
  };

  typedef ValueMap<const GlobalValue *, void *, AddressMapConfig>
      GlobalAddressMapTy;

private:
  ExecutionEngine &EE;

  /// GlobalAddressMap - A mapping between LLVM global values and their
  /// actualized version...
  GlobalAddressMapTy GlobalAddressMap;

  /// GlobalAddressReverseMap - This is the reverse mapping of GlobalAddressMap,
  /// used to convert raw addresses into the LLVM global value that is emitted
  /// at the address.  This map is not computed unless getGlobalValueAtAddress
  /// is called at some point.
  std::map<void *, AssertingVH<const GlobalValue> > GlobalAddressReverseMap;
```

TargetData

```
bool            LittleEndian;          ///< Defaults to false
unsigned        PointerMemSize;        ///< Pointer size in bytes
unsigned        PointerABIAlign;       ///< Pointer ABI alignment
unsigned        PointerPrefAlign;      ///< Pointer preferred alignment
SmallVector<unsigned char, 8> LegalIntWidths; ///< Legal Integers.
SmallVector<TargetAlignElem, 16> Alignments;
static const TargetAlignElem InvalidAlignmentElem;
mutable void *LayoutMap;
```

CommonFunctions

```
Function *FHandler;
Function *FMemset;
Function *FMemmove;
Function *FMemcpy;
Function *FRealmemset;
Function *FRealMemmove;
Function *FRealmemcmp;
Function *FRealmemcpy;
Function *FBSwap16;
Function *FBSwap32;
Function *FBSwap64;
```

Function : public GlobalValue, public ilist_node<Function>

```
public:
  typedef iplist<Argument> ArgumentListType;
  typedef iplist<BasicBlock> BasicBlockListType;
  typedef BasicBlockListType::iterator iterator;
  typedef BasicBlockListType::const_iterator const_iterator;
  typedef ArgumentListType::iterator arg_iterator;
  typedef ArgumentListType::const_iterator const_arg_iterator;
private:
  // Important things that make up a function!
  BasicBlockListType    BasicBlocks;          ///< The basic blocks
  mutable ArgumentListType ArgumentList;    ///< The formal arguments
  ValueSymbolTable *SymTab; ///< Symbol table of args/instructions
  AttrListPtr AttributeList;                  ///< Parameter attributes
```

GlobalValue

```
public:
  /// @brief An enumeration for the kinds of linkage for global values.
  enum LinkageTypes {
    ExternalLinkage = 0,///< Externally visible function
    AvailableExternallyLinkage, ///< Available for inspection, not emission.
    LinkOnceAnyLinkage, ///< Keep one copy of function when linking (inline)
    LinkOnceODRLinkage, ///< Same, but only replaced by something equivalent.
    WeakAnyLinkage,     ///< Keep one copy of named function when linking (weak)
    WeakODRLinkage,     ///< Same, but only replaced by something equivalent.
    AppendingLinkage,   ///< Special purpose, only applies to global arrays
    InternalLinkage,    ///< Rename collisions when linking (static functions).
    PrivateLinkage,     ///< Like Internal, but omit from symbol table.
    LinkerPrivateLinkage, ///< Like Private, but linker removes.
    LinkerPrivateWeakLinkage, ///< Like LinkerPrivate, but weak.
    LinkerPrivateWeakDefAutoLinkage, ///< Like LinkerPrivateWeak, but possibly
                                     ///  hidden.
    DLLImportLinkage,   ///< Function to be imported from DLL
    DLLExportLinkage,   ///< Function to be accessible from DLL.
    ExternalWeakLinkage,///< ExternalWeak linkage description.
    CommonLinkage       ///< Tentative definitions.
  };

  /// @brief An enumeration for the kinds of visibility of global values.
  enum VisibilityTypes {
    DefaultVisibility = 0,  ///< The GV is visible
    HiddenVisibility,       ///< The GV is hidden
    ProtectedVisibility     ///< The GV is protected
  };
```

protected:

```
  Module *Parent;
  // Note: VC++ treats enums as signed, so an extra bit is required to prevent
  // Linkage and Visibility from turning into negative values.
  LinkageTypes Linkage : 5;   // The linkage of this global
  unsigned Visibility : 2;    // The visibility style of this global
  unsigned Alignment : 16;    // Alignment of this symbol, must be power of two
  std::string Section;        // Section to emit this into, empty mean default
```

class GlobalVariable : public GlobalValue, public ilist_node<GlobalVariable>

bool isConstantGlobal : 1; // Is this a global constant?

bool isThreadLocalSymbol : 1; // Is this symbol "Thread Local"?

LLVMCodegen

```
private:
    const struct cli_bc *bc;
    Module *M;
    LLVMContext &Context;
    ExecutionEngine *EE;
    FunctionPassManager &PM, &PMUnsigned;
    LLVMTypeMapper *TypeMap;

    Function **apiFuncs;
    LLVMTypeMapper &apiMap;
    FunctionMapTy &compiledFunctions;
    Twine BytecodeID;

    TargetFolder Folder;
    IRBuilder<false, TargetFolder> Builder;

    std::vector<Value*> globals;
    DenseMap<unsigned, unsigned> GVoffsetMap;
    DenseMap<unsigned, constType*> GVtypeMap;
    Value **Values;
    unsigned numLocals;
    unsigned numArgs;
    std::vector<MDNode*> mdnodes;

    struct CommonFunctions *CF;
```

functions

========== functions used in this function ==========

addFunctionProtos

CF->FHandler = Function::Create(FTy, Function::ExternalLinkage,
"clamjit.fail", M);

CF->FMemset = Function::Create(FuncTy_3, GlobalValue::ExternalLinkage,
"llvm.memset.i32", M);

CF->FMemmove = Function::Create(FuncTy_4, GlobalValue::ExternalLinkage,
"llvm.memmove.i32", M);

CF->FMemcpy = Function::Create(FuncTy_4, GlobalValue::ExternalLinkage,
"llvm.memcpy.i32", M)
CF->FBSwap16 = Function::Create(FuncTy_5, GlobalValue::ExternalLinkage,
"llvm.bswap.i16", M);

CF->FBSwap32 = Function::Create(FuncTy_6, GlobalValue::ExternalLinkage,
"llvm.bswap.i32", M);

CF->FBSwap64 = Function::Create(FuncTy_7, GlobalValue::ExternalLinkage,

"llvm.bswap.i64", M);

CF->FRealmemset = Function::Create(DummyTy, GlobalValue::ExternalLinkage,
"memset", M);

CF->FRealMemmove = Function::Create(DummyTy, GlobalValue::ExternalLinkage,
"memmove", M);

CF->FRealmemcpy = Function::Create(DummyTy, GlobalValue::ExternalLinkage,
"memcpy", M);

CF->FRealmemcmp = Function::Create(FuncTy_5, GlobalValue::ExternalLinkage,
"memcmp", M);

EE->addGlobalMapping(F, dest)

/// addGlobalMapping - Tell the execution engine that the specified global is
/// at the specified location. This is used internally as functions are JIT'd
/// and as global variables are laid out in memory. It can and should also be
/// used by clients of the EE that want to have an LLVM global overlay
/// existing data in memory.

```
void ExecutionEngine::addGlobalMapping(const GlobalValue *GV, void *Addr) {
  MutexGuard locked(lock);

  DEBUG(dbgs() << "JIT: Map \'" << GV->getName()
        << "\' to [" << Addr << "]\n";);
  void *&CurVal = EEState.getGlobalAddressMap(locked)[GV];
  assert((CurVal == 0 || Addr == 0) && "GlobalMapping already established!");
  CurVal = Addr;

  // If we are using the reverse mapping, add it too
  if (!EEState.getGlobalAddressReverseMap(locked).empty()) {
    AssertingVH<const GlobalValue> &V =
      EEState.getGlobalAddressReverseMap(locked)[Addr];
    assert((V == 0 || GV == 0) && "GlobalMapping already established!");
    V = GV;
  }
}
```

EE->getPointerToFunction(F);

/// getPointerToFunction - This method is used to get the address of the
/// specified function, compiling it if neccesary.

```
  if (void *Addr = getPointerToGlobalIfAvailable(F))
    return Addr;      // Check if function already code gen'd
```

```
// Now that this thread owns the lock, make sure we read in the function if it
// exists in this Module.
F->Materialize(&ErrorMsg))

// ... and check if another thread has already code gen'd the function.
if (void *Addr = getPointerToGlobalIfAvailable(F))
    return Addr;

if (F->isDeclaration() || F->hasAvailableExternallyLinkage()) {
    bool AbortOnFailure = !F->hasExternalWeakLinkage();
    void *Addr = getPointerToNamedFunction(F->getName(), AbortOnFailure);
    addGlobalMapping(F, Addr);
    return Addr;
}
```

Codegen.generate() - Function* generate()

```
Generate LLVM IR functions
in this function:
for variables in bytecode, will be added to maps
for functions in bytecode, Create function called will actually generate the code
Function::Create
BasicBlock::Create
Builder.CreateAlloca
Builder.CreateInBoundsGEP
Builder.CreateBitCast
Builder.CreateLoad
Builder.CreateAdd
.....
CallInst::Create
ReturnInst::Create
```

globals of clamav

```
================== deal with globals passed in via clamav ==================
/* loading all the globals' type
*/
// 5 global in total
// and the map will be
DenseMap<unsigned, constType*> GVtypeMap;
```

```
struct cli_bc_hooks {
      const uint32_t* match_offsets;
      const uint16_t* kind;
      const uint32_t* match_counts;
      const uint32_t* filesize;
      const struct cli_pe_hook_data* pedata;
};
```

globalid	Type id	type	type variable
GLOBAL_MATCH_OFFSETS	76	uint32_t*	match_offsets
GLOBAL_KIND	16	uint16_t*	kind
GLOBAL_MATCH_COUNTS	76	uint32_t*	match_counts
GLOBAL_FILESIZE	75	uint32_t*	filesize
GLOBAL_PEDATA	69	struct cli_pe_hook_data*	pedata

```
for (unsigned i=0;i<cli_apicall_maxglobal - _FIRST_GLOBAL;i++)
      unsigned id = cli_globals[i].globalid;
      // apiMap stores are the function calls and global from clamav
      // which is defined in libclamav/bytecode_api_decl.c
      constType *Ty = apiMap.get(cli_globals[i].type, NULL, NULL);
      GVtypeMap[id] = Ty;
/*
globalid is a type of cli_apiglobal
struct cli_apiglobal {
    const char *name;
    enum bc_global globalid;
    uint16_t type;
    unsigned offset;
};
const struct cli_apiglobal cli_globals[] = {
/* Bytecode globals BEGIN */
    {"__clambc_match_offsets", GLOBAL_MATCH_OFFSETS, 76,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.match_offsets - (char*)NULL)},
    {"__clambc_kind", GLOBAL_KIND, 16,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.kind - (char*)NULL)},
    {"__clambc_match_counts", GLOBAL_MATCH_COUNTS, 76,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.match_counts - (char*)NULL)},
    {"__clambc_filesize", GLOBAL_FILESIZE, 75,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.filesize - (char*)NULL)},
    {"__clambc_pedata", GLOBAL_PEDATA, 69,
     ((char*)&((struct cli_bc_ctx*)0)->hooks.pedata - (char*)NULL)}
/* Bytecode globals END */
};

enum bc_global {
  _FIRST_GLOBAL = 0x8000,
  GLOBAL_MATCH_COUNTS = 0x8000,
  GLOBAL_KIND,
  GLOBAL_VIRUSNAMES,
  GLOBAL_PEDATA,
  GLOBAL_FILESIZE,
  GLOBAL_MATCH_OFFSETS,
  _LAST_GLOBAL
};

*/
```

```
// The hidden ctx param to all functions – sth behind cli_bc_hooks that is passed in
unsigned maxh = cli_globals[0].offset + sizeof(struct cli_bc_hooks);
/// PointerType::getUnqual - This constructs a pointer to an object of the
/// specified type in the generic address space (address space zero).
constType *HiddenCtx =
      PointerType::getUnqual(ArrayType::get(Type::getInt8Ty(Context), maxh));
```

globals of bytecode

```
================== deal with globals in bytecode =================
// reserve space for globals in bytecode
globals.reserve(bc->num_globals);
BitVector FakeGVs;
FakeGVs.resize(bc->num_globals);
globals.push_back(0);
/* load globals from bytecode to global map at JIT
*/
for (unsigned i=1;i<bc->num_globals;i++)
      constType *Ty = mapType(bc->globaltys[i]);
      if (isa<PointerType>(Ty)) // if is a pointer type
            unsigned g = bc->globals[i][1];
            if (GVoffsetMap.count(g)) // complex type??? deal with it later
                  FakeGVs.set(i); // used in FakeGVs.any()
                  globals.push_back(0);
                  continue;
      // for simple type, create constant type in llvm
      Constant *C = buildConstant(Ty, bc->globals[i], c);
      // create the variable
/// GlobalVariable class represents a single global variable (or constant) in the VM.
// Global variables are constant pointers that refer to hunks of space that are
// allocated by either the VM, or by the linker in a static compiler.    A global
// variable may have an intial value, which is copied into the executables .data
// area.    Global Constants are required to have initializers.
/// GlobalVariable ctor - This creates a global and inserts it before the
/// specified other global.

/// Twine - A lightweight data structure for efficiently representing the
/// concatenation of temporary values as strings.
      GV = new GlobalVariable(*M, Ty, true, // create global data
                              GlobalValue::InternalLinkage,
                              C, "glob"+Twine(i));
```

```
    globals.push_back(GV); // add to global stack
```

functions of bytecode

```
================== deal with functions in bytecode =================

/* load function from bytecode to global map at JIT
*/
// Create LLVM IR Function
```

arg/ret/func

```
================== function - arguments/return value/function
// allocate memory to store functions
Function **Functions = new Function*[bc->num_func];
// deal with each function in bytecode
// deal with arg/ret/BB
for (unsigned j=0;j<bc->num_func;j++)
    const struct cli_bc_func *func = &bc->funcs[j];
    // add HiddenCtx    passed in from clamav to global map
    argTypes.push_back(HiddenCtx);
    // deal with each argument in one function
    for (unsigned a=0;a<func->numArgs;a++)
        argTypes.push_back(mapType(func->types[a]));
    // create function return type
    constType *RetTy = mapType(func->returnType);
    /// FunctionType - Class to represent function types
    FunctionType *FTy =    FunctionType::get(RetTy, argTypes, false);
    // create the function
    /// will call Function ctor - If the (optional) Module argument is specified, the
    /// function is automatically inserted into the end of the function list for
    /// the module.
    /// generate code of function
    Functions[j] = Function::Create(FTy, Function::InternalLinkage,
                                    BytecodeID+"f"+Twine(j), M);
```

Basic Block

```
================== function - Basic Block
    // Generate LLVM IR for functions
    // deal with BB and arguments
    /// Basic Block represents a single basic block in LLVM. A basic block is simply a
```

```
/// container of instructions that execute sequentially. Basic blocks are Values
/// because they are referenced by instructions such as branches and switch
/// tables. The type of a BasicBlock is "Type::LabelTy" because the basic block
/// represents a label to which a branch can jump.
///
/// A well formed basic block is formed of a list of non-terminating
/// instructions followed by a single TerminatorInst instruction.
/// TerminatorInst's may not occur in the middle of basic blocks, and must
/// terminate the blocks. The BasicBlock class allows malformed basic blocks to
/// occur because it may be useful in the intermediate stage of constructing or
/// modifying a program. However, the verifier will ensure that basic blocks
/// are "well formed".

for (unsigned j=0;j<bc->num_func;j++) // 1 for this case
    const struct cli_bc_func *func = &bc->funcs[j];
    // Create all BasicBlocks
    Function *F = Functions[j];
    BasicBlock **BB = new BasicBlock*[func->numBB];
    /// Create - Creates a new BasicBlock. If the Parent parameter is specified,
    /// the basic block is automatically inserted at either the end of the
    /// function (if InsertBefore is 0), or before the specified basic block.
    /// generate code of BB
    for (unsigned i=0;i<func->numBB;i++) BB[i] = BasicBlock::Create(Context, "",
F);

    // load in arguments of function – 0 for this case
    Values = new Value*[func->numValues];
    Builder.SetInsertPoint(BB[0]);
    Function::arg_iterator I = F->arg_begin();
    for (unsigned i=0;i<func->numArgs; i++)    Values[i] = &*I; ++I;
    // allocate space for the local variable to store value, numValues=4 for this
case
    for (unsigned i=func->numArgs;i<func->numValues;i++)
        Values[i] = Builder.CreateAlloca(mapType(func->types[i]));

    numLocals = func->numLocals; // 4 for this case
    numArgs = func->numArgs; // 0 for this case

    // deal with complex globals in bytecode
    if (FakeGVs.any()) // yes for this case
        //bc->globals[i][1]
        //bc->globals[i][0]
        unsigned g = bc->globals[i][1];
        constType *Ty = GVtypeMap[g];
```

```
Ty = PointerType::getUnqual(PointerType::getUnqual(Ty));
globals[i] = createGEP(SpecialGV, 0, ARRAYREF(Value*,C, C+1));
globals[i] = Builder.CreateBitCast(globals[i], Ty);
GetElementPtrInst *GI = dyn_cast<GetElementPtrInst>(globals[i])
```

instructions

========== function - Instructions

```
for (unsigned i=0;i<func->numBB;i++) // for each BB
    const struct cli_bc_func *func = &bc->funcs[j];
        for (unsigned j=0;j<bb->numInsts;j++) // for each Instruction
            const struct cli_bc_inst *inst = &bb->insts[j];
```

// cli_bc_inst

```
struct cli_bc_inst {
    enum bc_opcode opcode;
    uint16_t type;
    operand_t dest;
    interp_op_t interp_op;/* opcode for interpreter */
    union {
    operand_t unaryop;
    struct cli_bc_cast cast;
    operand_t binop[2];
    operand_t three[3];
    struct cli_bc_callop ops;
    struct branch branch;
    bbid_t jump;
    } u;
};
```

```
... ...
switch (inst->opcode)
case OP_BC_ADD: // add instruction
   Store(inst->dest, Builder.CreateAdd(Op0, Op1));
... ...
```

call function and ret instruction

========== warp it up

```
// add protection for stack smash
for (unsigned j=0;j<bc->num_func;j++)
    Function *F = Functions[j];
    AddStackProtect(F);

// deal with entry point function
// first need to define a callable function type which should match the enrtypoint function which is function[0]
FunctionType *Callable = FunctionType::get(Type::getInt32Ty(Context),
                                           args, false);
```

```
// All functions have the Fast calling convention, however
// entrypoint can only be C, emit wrapper
Function *F = Function::Create(Functions[0]->getFunctionType(),
                               Function::ExternalLinkage,
                               Functions[0]->getName()+"_wrap", M);
// generate code for BB of entry point
BasicBlock *BB = BasicBlock::Create(Context, "", F);
/// CallInst - This class represents a function call, abstracting a target
/// machine's calling convention.  This class uses low bit of the SubClassData
/// field to indicate whether or not this is a tail call.  The rest of the bits
/// hold the calling convention of the call.
// generate code for call instruction
CallInst *CI = CallInst::Create(Functions[0], ARRAYREFVECTOR(Value*, Args), "", BB);
CI->setCallingConv(CallingConv::Fast);
/// ReturnInst - Return a value (possibly void), from a function.  Execution
/// does not continue in this function any longer.
// generate code for return instruction
ReturnInst::Create(Context, CI, BB);
```

details codes

```
==================== the details for cli_bytecode_prepare_jit ==============
ScopedExceptionHandler handler;
// setup exception handler to longjmp back here
HANDLER_TRY(handler)
// LLVM itself never throws exceptions, but operator new may throw bad_alloc
try{
Module *M = new Module("ClamAV jit module", bcs->engine->Context);

// Create the JIT.
EngineBuilder builder(M);
ExecutionEngine *EE = bcs->engine->EE = builder.create(); // the JIT engine
bcs->engine->Listener    = new NotifyListener();
EE->RegisterJITEventListener(bcs->engine->Listener);

struct CommonFunctions CF;
addFunctionProtos(&CF, EE, M); // set some common functions

FunctionPassManager OurFPM(M), OurFPMUnsigned(M); // function pass manager
addFPasses(OurFPM, true, EE->getTargetData());// add function pass
addFPasses(OurFPMUnsigned, false, EE->getTargetData());// add function pass
```

```
LLVMTypeMapper apiMap(bcs->engine->Context, cli_apicall_types,
cli_apicall_maxtypes, HiddenCtx);

// dealing with APIs defined by clamav
Function **apiFuncs = new Function *[cli_apicall_maxapi]; // function array for apis
for (unsigned i=0;i<cli_apicall_maxapi;i++)
    const struct cli_apicall *api = &cli_apicalls[i]; // the api array defined by clamav
    Function *F = Function::Create(FTy, Function::ExternalLinkage,
                                   api->name, M); // create Function structure for api
    void *dest;
    // get api dest inside the memory
    EE->addGlobalMapping(F, dest); // add it to the map in JIT
    EE->getPointerToFunction(F); // get function address
    apiFuncs[i] = F; // log down the function

// dealing with functions defined in bytecode
llvm::Function **Functions = new Function*[bcs->count];
for (unsigned i=0;i<bcs->count;i++)
    // structure for code generate
    LLVMCodegen Codegen(bc, M, &CF, bcs->engine->compiledFunctions, EE,
                        OurFPM, OurFPMUnsigned, apiFuncs, apiMap);
    Function *F = Codegen.generate(); // generate the function
    Functions[i] = F; // log down the functions

PassManager PM; // pass mamager
PM.add(new TargetData(*EE->getTargetData()));
// add passes
PM.add(createSCCPPass());
PM.add(createCFGSimplificationPass());
PM.add(createGlobalOptimizerPass());
PM.add(createConstantMergePass());

// compile all functions now - not during runtime, that is not lazily!
for (Module::iterator I = M->begin(), E = M->end(); I != E; ++I)
    Function *Fn = &*I;
    // compile the functions
    if (!Fn->isDeclaration())    EE->getPointerToFunction(Fn);

for (unsigned i=0;i<bcs->count;i++)
    const struct cli_bc_func *func = &bcs->all_bcs[i].funcs[0];
    bcs->engine->compiledFunctions[func] =
        EE->getPointerToFunction(Functions[i]); // get functions compiled above
    bcs->all_bcs[i].state = bc_jit;
```

```
    return CL_SUCCESS; // good to return
} catch{
    …
}
```

cli_bytecode_done_jit

```
LLVMApiScopedLock scopedLock; // get lock
// release resources
delete bcs->engine->EE;
bcs->engine->EE = 0;
delete bcs->engine->Listener;
bcs->engine->Listener = 0;
delete bcs->engine;
bcs->engine = 0;
```

cli_vm_execute_jit

```
// when called from clamav, func will be 0
// so entry point will always be called first as a start
void *code = bcs->engine->compiledFunctions[func];
// execute
ret = bytecode_execute((intptr_t)code, ctx);
```

bytecode_execute

```
ScopedExceptionHandler handler;
    // real execute;
    HANDLER_TRY(handler) {
    // setup exception handler to longjmp back here
        uint32_t result = ((uint32_t (*)(struct cli_bc_ctx *))(intptr_t)code)(ctx);
        *(uint32_t*)ctx->values = result;
        return 0; // success and return
    }
    HANDLER_END(handler);

   // a failure
    return CL_EBYTECODE;
```