DragonFFI

*Foreign Function Interface and JIT for C code*

https://github.com/aguinet/dragonffi

EuroLLVM 2018 - Adrien Guinet (@adriengnt)
2018/04/17
Content of this talk

- whoami
- FFI? (and related work)
- FFI for C with Clang/LLVM
- What's next
Whoami

Adrien Guinet (@adriengnt)

- Quarkslab
- Working on an LLVM-based obfuscator
**FFI?**

**Wikipedia:** A foreign function interface (FFI) is a mechanism by which a program written in one programming language can call routines or make use of services written in another.

**In our case:** (compiling and) calling C functions from any language

```python
import pydffi
CU = pydffi.FFI().cdef("int puts(const char* s);")
CU.funcs.puts("hello world!")
```
What's the big deal?

C functions are usually called from "higher" level languages for performances...

- ...but C functions are compiled for a specific ABI
- There isn't *one* ABI, this is system/arch dependant
- It's a huge mess!

=> We don't want to deal with it, we want a library that makes this for us!
Related work

- **libffi**: reference library, implements a lot of existing ABI and provides an interface to call a C function

```c
ffi_cif cif;
ffi_type *args[] = {&ffi_type_pointer};
void* values[] = &s;

ffi_prep_cif(&cif, FFI_DEFAULT_ABI, 1,
             &ffi_type_sint, args);
s = "Hello World!";
ffi_call(&cif, puts, &rc, values);
```

- **cffi**: uses **libffi** to provide this interface to Python, and uses **pycparser** to let the user define C functions/types easily
Why another one?

- **libffi**: far from trivial to insert a new ABI (hand-written assembly); the `ms_abi` calling convention under Linux isn't supported.
- **cffi**: does not support a lot of C construction:
  ```python
  cffi.FFI().cdef("#include <stdio.h>"
  CDefError: cannot parse "#include <stdio.h>"
  :2: Directives not supported yet
  </stdio.h></stdio.h>

  cffi.FFI().cdef("__attribute__((ms_abi)) int foo(int a, int b) { return a+b; }")
  CDefError: cannot parse "__attribute__((ms_abi)) int foo(int a, int b) { return a+b;"
  :2:15: before: (

- I want to be able to use my libraries' headers out-of-the-box!
FFI for C with Clang/LLVM

Why Clang/LLVM?

• Clang can parse C code: parse headers to gather definitions (types/functions/attributes...)
• Clang support lots of these ABIs, and LLVM can compile the whole thing
• So let's put all of this together \o/
FFI for C with Clang/LLVM

Gather C types

Using DWARF debug information from the LLVM IR:

typedef struct {
    short a;
    int b;
} A;

void print_A(A s) {
    printf("%d %d\n", s.a, s.b);
}

$ clang -S -emit-llvm -o - -m32 a.c -g
!11 = distinct !DICompositeType(tag: DW_TAG_structure_type, size: 64, elements: !12)
!12 = ![!13, !15]
!13 = !DIDerivedType(tag: DW_TAG_member, name: "a", baseType: !14, size: 16)
!14 = !DIBasicType(name: "short", size: 16, encoding: DW_ATE_signed)
!15 = !DIDerivedType(tag: DW_TAG_member, name: "b", baseType: !16, size: 32, offset: 32)
!16 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
FFI for C with Clang/LLVM

DragonFFI type system

DWARF metadata are parsed to create **DFFI** types:

- All basic C types (w/ non standards like (u)int128_t)
- Arrays, pointers
- Structures, unions, enums (w/ field offsets)
- Function types

Every type can be const-qualified!
FFI for C with Clang/LLVM

Calling a C function

A DFFI function type is parsed to create a function call wrapper:

```c
// For this function declaration
int puts(const char* s);

// We generate this wrapper
void __dffi_wrapper_0(int32_t ( __attribute__((cdecl)) *__FPtr)(char *),
                      int32_t *__Ret,
                      void** __Args) {
  __Ret = (__FPtr)(*(((char**)__Args)[0]));
}
```

- Clang handle all the ABI issues here!
- Clang emits the associated LLVM IR, that can be jitted, and there we go!
Usage examples

```python
# Use opendir/readdir from python
import pydffi
F = pydffi.FFI()
CU = F.cdef("#include <dirent.h>"

dir_ = CU.funcs.opendir("/home/aguiinet/dev/dragonffi/")
if not dir_:
    print("error reading directory")
    sys.exit(1)
readdir = CU.funcs.readdir
while True:
    dirent = readdir(dir_)
    if not dirent:
        break
    print(dirent.obj.d_name.cast(F.CharPtrTy).cstr.tobytes())
assert(CU.funcs.closedir(dir_) == 0)
```

b'.git'
b'tests'
b'build_release_static_clang'
b'README.rst'
b'include'
```
What's next

Support parsing of debug informations from shared libraries directly!

```python
import pydffi
import sys

F=pydffi.FFI()
CU=F.from_dwarf("/home/agunet/dev/libarchive-3.3.2/_build_reewithdebinfo/libarchive/libarchive.so")

funcs = CU.funs
archive_read_next_header = funcarchive_read_next_header
archive_entry_pathname_utf8 = funcarchive_entry_pathname_utf8
archive_read_data_skip = funcarchive_read_data_skip

a = funcarchive_read_new()
funcs.archive_read_support_filter_all(a)
funcs.archive_read_support_format_all(a)
r = funcarchive_read_open_filename(a, "/home/agunet/Downloads/z3-460-x64-debian-810.zip", 10240)
if r != 0:
    raise RuntimeError("unable to open archive")
entry = F.ptr(CU.types.archive_entry)()
while archive_read_next_header(a, F.ptr(entry)) == 0:
    pathname = archive_entry_pathname_utf8(entry)
    print(pathname.cstr.tobytes().decode("utf8"))
    archive_read_data_skip(a)
funcs.archive_read_free(a)
```

z3-460-x64-debian-810/LICENSE.txt
z3-460-x64-debian-810/include/z3_rcf.h
z3-460-x64-debian-810/include/z3.h
z3-460-x64-debian-810/include/z3_macros.h
z3-460-x64-debian-810/include/z3++h
What's next

Support parsing of debug informations from shared libraries directly!

Work in progress:

- Debug information can be huged: [https://github.com/aguinet/llvm-lightdwarf](https://github.com/aguinet/llvm-lightdwarf): experimental LLVM pass that reduces debug info to the things we need (from 1.8Mb to 536KB for libarchive)
- Merge all the compilation units into one
- Idea: static FFI compiler: generate a mylibrary-ffi.so that contains wrappers and reduced DWARF informations!
What's next

• Reducing binary size: pydffi.cpython-36m-x86_64-linux-gnu.so is 55Mb. Two versions:
  ■ "core": w/o clang, only the ABI-related part. Very close to what libffi does!
  ■ "full": optional module w/ clang
• JIT and optimize the full glue from Python/Ruby/... to the C function call (easy::jit?)
Thanks for your attention!

https://github.com/aguinet/dragonffi

pip install pydffi

For Linux/OSX/Windows users!

Twitter: @adriengnt
Mail: adrien@guinet.me