Reverse Engineering Theology

make security great again by not building walls

by @sgniwx

About Me

Weekend CTF'er

With VN Security since year 2009

> CTF player

> Weekend gamer

Most of the time

Researcher at 360 Unicorn

> IOT Hacking

> IOT Hacking

> IOT Hacking

Once a year

Hack in the box die hard fans running CTF since year 2006

> Good friends

> Coffee Drinker

> 2008, Hack In The Box CTF Winner

> 2010, Hack In The Box Speaker, Malaysia

> 2012, Codegate Speaker, Korea

> 2015, VXRL Speaker, Hong Kong

> 2015, HITCON CTF, Prequal Top 10

> 2016, Codegate CTF, Prequal Top 5

> 2016, Qcon Speaker, Beijing

> 2016, Kcon Beijing, Speaker

> 2016, IAVC Tianjing, Speaker

> 2017, KCON Beijing, Trainner

> Research: OSX, Local Privilege Escalation

> Research: Code commit for metasploit 3

> Research: GDB Bug hunting

> Research: Metasploit module

> Research: Linux Randomization Bypass

> Blog: http://www.github.com/xwings

> Email: kj@reverser.io

> twitter: @sgniwx

vnsecurity.net

Who Are We

VN Security

- Active CTF Player (CLGT)
- Active speaker at conferences
  - Blackhat USA
  - Tetcon
  - Hack In The Box
  - Xcon

- Our Tools
  - PEDA
  - Unicorn/ Capstone/ Keystone
  - Xandora
  - OllyDbg, Catcha!
  - ROPEME

- Nations
  - Vietnamese
  - Malaysian
  - Singaporean

Nguyen Anh Quynh

- Security Researcher
- Active speaker at conferences
  - Blackhat USA
  - Syscan
  - Hack In The Box
  - Xcon

- Research Topics
  - Emulators
  - Virtualization
  - Binary Analysis
  - Tools for Malware Analysis

Who Are We

GDB

```
(gdb) disassemble
Dump of assembler code for function main:
0x000000000040058c <main+0>:     push   %rbp
0x000000000040058d <main+1>:     mov    %rsp,%rbp
0x0000000000400590 <main+4>:     sub    $0x10,%rsp
0x0000000000400594 <main+8>:     mov    $0x4,%edi
0x0000000000400599 <main+13>:    callq  0x4004a8 <_init+56>
0x000000000040059e <main+18>:    mov    %rax,0xfffffffffffffff0(%rbp)
0x00000000004005a2 <main+22>:    movl   $0x0,0xfffffffffffffffc(%rbp)
0x00000000004005a9 <main+29>:    mov    0xfffffffffffffffc(%rbp),%eax
0x00000000004005ac <main+32>:    cltq
0x00000000004005ae <main+34>:    shl    $0x2,%rax
0x00000000004005b2 <main+38>:    mov    %rax,%rdx
0x00000000004005b5 <main+41>:    add    0xfffffffffffffff0(%rbp),%rdx
0x00000000004005b9 <main+45>:    mov    0xfffffffffffffffc(%rbp),%eax
0x00000000004005bc <main+48>:    mov    %eax,(%rdx)
0x00000000004005be <main+50>:    mov    0xfffffffffffffffc(%rbp),%eax
0x00000000004005c1 <main+53>:    cltq
0x00000000004005c3 <main+55>:    shl    $0x2,%rax
0x00000000004005c7 <main+59>:    add    0xfffffffffffffff0(%rbp),%rax
0x00000000004005cb <main+63>:    mov    (%rax),%edx
0x00000000004005cd <main+65>:    mov    0xfffffffffffffffc(%rbp),%esi
0x00000000004005d0 <main+68>:    mov    $0x4006dc,%edi
0x00000000004005d5 <main+73>:    mov    $0x0,%eax
0x00000000004005da <main+78>:    callq  0x4004b8 <_init+72>
0x00000000004005df <main+83>:    addl   $0x1,0xfffffffffffffffc(%rbp)
0x00000000004005e3 <main+87>:    jmp    0x4005a9 <main+29>
End of assembler dump.
(gdb)
```

PEDA

```
gdb-peda$ start
[------------------------------registers------------------------------]
EAX: 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
EBX: 0xb7fcbff4 --> 0x155d7c
ECX: 0xd5eeaa03
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xbffff748 --> 0xbffff7c8 --> 0x0
ESP: 0xbffff748 --> 0xbffff7c8 --> 0x0
EIP: 0x80483e7 (<main+3>:        and     esp,0xfffffff0)
EFLAGS: 0x200246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------code---------------------------------]
   0x80483e3 <frame_dummy+35>:  nop
   0x80483e4 <main>:     push    ebp
   0x80483e5 <main+1>:   mov     ebp,esp
=> 0x80483e7 <main+3>:   and     esp,0xfffffff0
   0x80483ea <main+6>:   sub     esp,0x110
   0x80483f0 <main+12>: mov     eax,DWORD PTR [ebp+0xc]
   0x80483f3 <main+15>: add     eax,0x4
   0x80483f6 <main+18>: mov     eax,DWORD PTR [eax]
[-------------------------------stack--------------------------------]
0000| 0xbffff748 --> 0xbffff7c8 --> 0x0
0004| 0xbffff74c --> 0xb7e8cbd6 (<__libc_start_main+230>:        mov     DWORD PTR [e:
0008| 0xbffff750 --> 0x1
0012| 0xbffff754 --> 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
0016| 0xbffff758 --> 0xbffff7fc --> 0xbffff922 ("SHELL=/bin/bash")
0020| 0xbffff75c --> 0xb7fe1858 --> 0xb7e76000 --> 0x464c457f
0024| 0xbffff760 --> 0xbffff7b0 --> 0x0
0028| 0xbffff764 --> 0xffffffff
[-------------------------------------------------------------------]
Legend: code, data, rodata, value

Temporary breakpoint 1, 0x080483e7 in main ()
gdb-peda$
```

Why Greater China
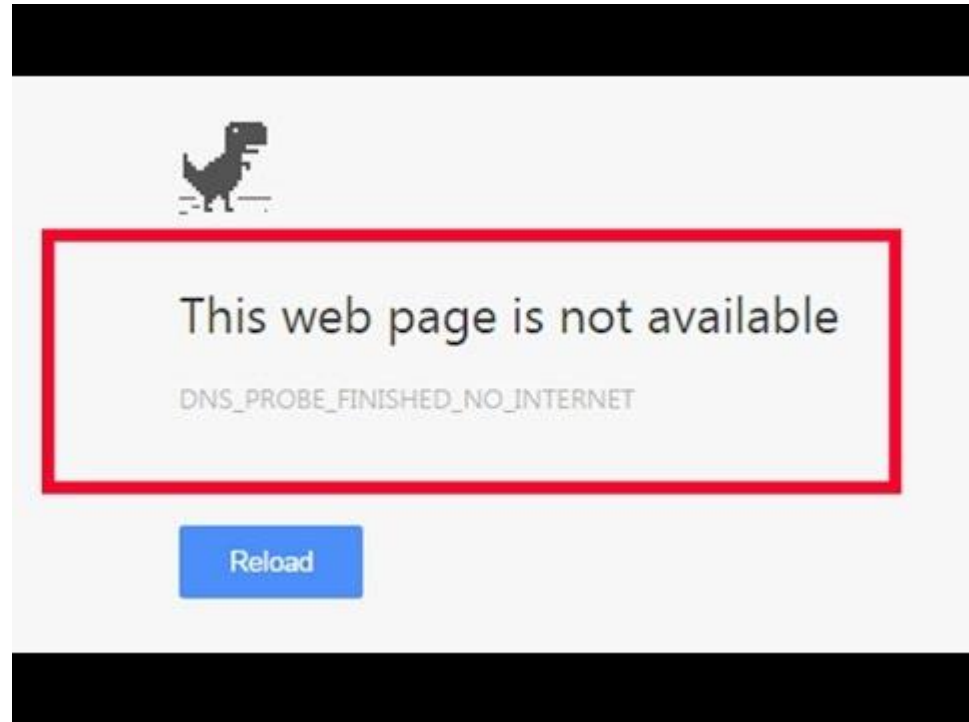
What Are These Things

What is Disassembler

› From binary to assembly code

› Core part of all binary analysis/ reverse engineering / debugger and exploit development

› Disassembly framework (engine/library) is a lower layer in stack of architecture

Example

- 01D8 = ADD EAX,EBX (x86)
- 1169 = STR R1,[R2] (ARM's Thumb)

Binary Analysis	Debugger	Exploit Development
Disassembler Engine	**CPU Emulator Engine**	**Assembler Engine**
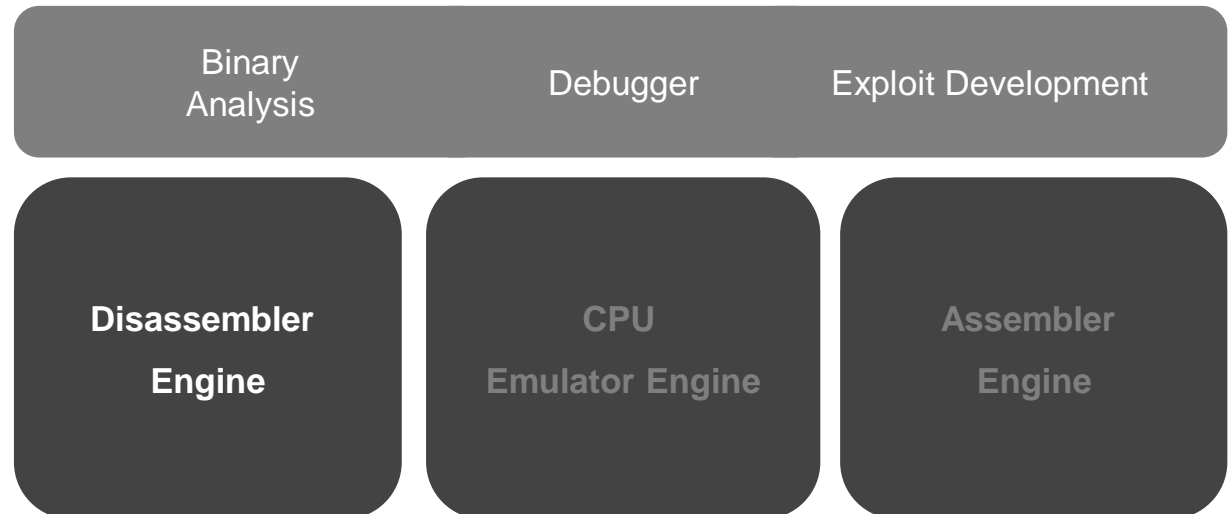
What is Emulator

- › Software only CPU Emulator

- › Core focus on CPU operations.

- › Design with no machine devices

- › Safe emulation environment

- › Where else can we see CPU emulator. Yes, Antivirus

Example

- 01D1 = add eax,ebx (x86)
  - Load eax & ebx register
  - Add value of eax & ebx then copy the result to eax
  - Update flag OF, SF, ZF, AF, CF, PF accordingly

Binary Analysis	Debugger	Exploit Development
Disassembler Engine	**CPU Emulator Engine**	**Assembler Engine**

What is Assembler

› From assembly to machine code

› Support high level concepts such as macro, functions and etc.

› Dynamic machine code generation

Example

- ADD EAX,EBX = 01D8 (x86)
- STR R1,[R2] = 1169 (ARM's Thumb)

Binary Analysis	Debugger	Exploit Development
Disassembler Engine	**CPU Emulator Engine**	**Assembler Engine**

Where Are We

Showcase

- CEnigma
- Unicorn
- CEbot
- Camal
- Radare2
- Pyew
- WinAppDbg
- PowerSploit
- MachOview
- RopShell
- ROPgadget
- Frida
- The-Backdoor-Factory
- Cuckoo

- Cerbero Profiler
- CryptoShark
- Ropper
- Snowman
- X86dbg
- Concolica
- Memtools Vita
- BARF
- rp++
- Binwalk
- MPRESS dumper
- Xipiter Toolkit
- Sonare
- PyDA

- Qira
- Rekall
- Inficere
- Pwntools
- Bokken
- Webkitties
- Malware_config_parsers
- Nightmare
- Catfish
- JSoS-Module-Dump
- Vitasploit
- PowerShellArsenal
- PyReil
- ARMSCGen

- Shwass
- Nrop
- Illdb-capstone-arm
- Capstone-js
- ELF Unstrip Tool
- Binjitsu
- Rop-tool
- JitAsm
- OllyCapstone
- PackerId
- Volatility Plugins
- Pwndbg
- Lisa.py
- Many Other More

Showcase

> UniDOS: Microsoft DOS emulator.

> Radare2: Unix-like reverse engineering framework and commandline tools.

> Usercorn: User-space system emulator.

> Unicorn-decoder: A shellcode decoder that can dump self-modifying-code.

> Univm: A plugin for x64dbg for x86 emulation.

> PyAna: Analyzing Windows shellcode.

> GEF: GDB Enhanced Features.

> Pwndbg: A Python plugin of GDB to assist exploit development.

> Eli.Decode: Decode obfuscated shellcodes.

> IdaEmu: an IDA Pro Plugin for code emulation.

> Roper: build ROP-chain attacks on a target binary using genetic algorithms.

> Sk3wlDbg: A plugin for IDA Pro for machine code emulation.

> Angr: A framework for static & dynamic concolic (symbolic) analysis.

> Cemu: Cheap EMUlator based on Keystone and Unicorn engines.

> ROPMEMU: Analyze ROP-based exploitation.

> BroIDS_Unicorn: Plugin to detect shellcode on Bro IDS with Unicorn.

> UniAna: Analysis PE file or Shellcode (Only Windows x86).

> ARMSCGen: ARM Shellcode Generator.

> TinyAntivirus: Open source Antivirus engine designed for detecting & disinfecting polymorphic virus.

> Patchkit: A powerful binary patching toolkit.

Showcase

- Keypatch: IDA Pro plugin for code assembling & binary patching.
- Radare2: Unix-like reverse engineering framework and commandline tools.
- GEF: GDB Enhanced Features.
- Ropper: Rop gadget and binary information tool.
- Cemu: Cheap EMUlator based on Keystone and Unicorn engines.
- Pwnypack: Certified Edible Dinosaurs official CTF toolkit.
- Keystone.JS: Emscripten-port of Keystone for JavaScript.
- Usercorn: Versatile kernel+system+userspace emulator.
- x64dbg: An open-source x64/x32 debugger for windows.
- Liberation: a next generation code injection library for iOS cheaters everywhere.

- Strongdb: GDB plugin for Android debugging.
- AssemblyBot: Telegram bot for assembling and disassembling on-the-go.
- demovfuscator: Deobfuscator for movfuscated binaries.
- Dash: A simple web based tool for working with assembly language.
- ARMSCGen: ARM Shellcode Generator.
- Asm_Ops: Assembler for IDA Pro (IDA Plugin).
- Binch: A lightweight ELF binary patch tool.
- Metame: Metamorphic code engine for arbitrary executables.
- Patchkit: A powerful binary patching toolkit.
- Pymetamorph: Metamorphic engine in Python for Windows executables.

Born of The Trinity

Fundamental Frameworks for Reversing

› Components for a complete RE framework

› Interchange between assembler and disassembler

› A full CPU emulator always help when comes with obfuscated code

Binary

Assembly

Capstone

Keystone

Unicorn

APP OS · APP OS · APP OS

Capstone Engine

NGUYEN Anh Quynh <aquynh -at- gmail.com>

http://www.capstone-engine.org
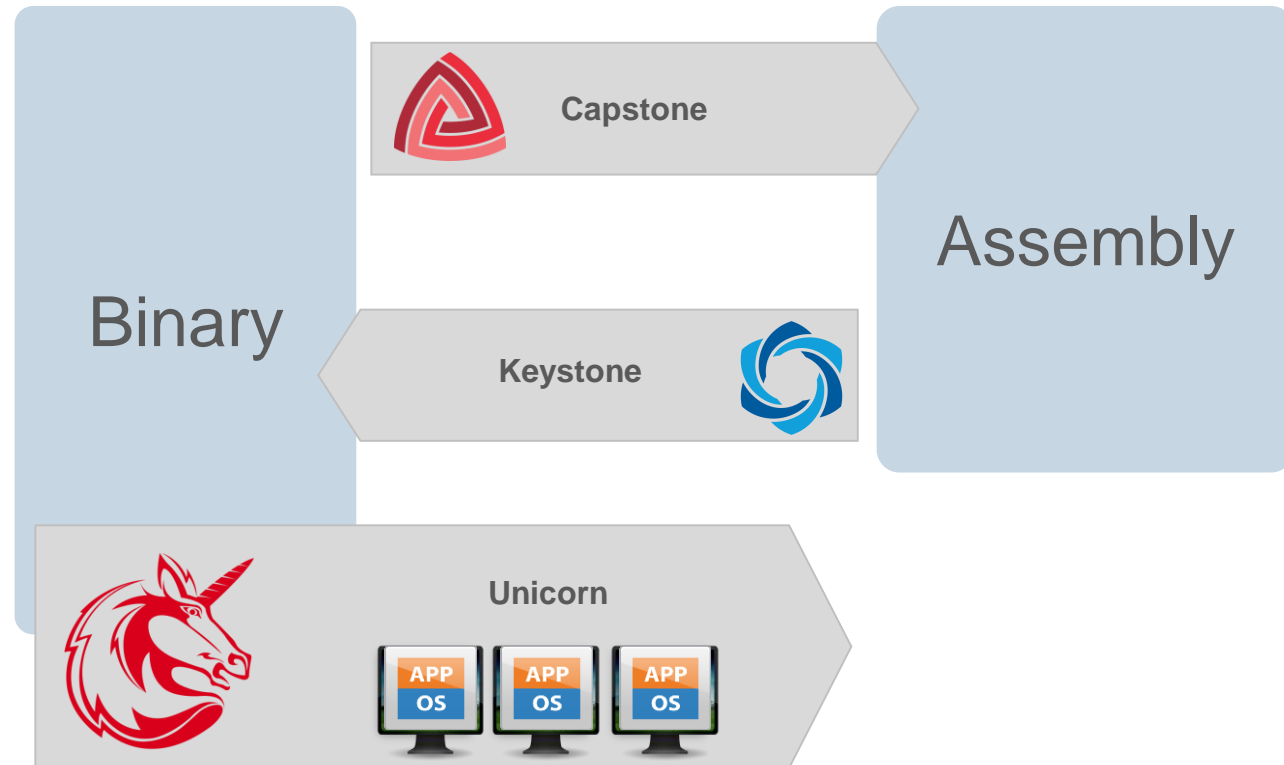
What's Wrong with Current Disassembler

Features	Distorm3	BeaEngine	Udis86	Libopcode
X86|Arm	✓|X	✓|X	✓|X	✓|✓ [1]
Linux|Windows	✓|✓	✓|✓	✓|✓	✓|X
Python|Ruby bindings	✓|X [2]	✓|X	✓|X	✓|X
Update	X	?	X	X
License	GPL	LGPL3	BSD	GPL

> ❯ Nothing works even up until 2013 (First release of Capstone Engine)
> ❯ Looks like no one take charge
> ❯ Industry stays in the dark side

What We Had In Mind

› Multiple archs: x86, ARM+ ARM64 + Mips + PPC and more

› Multiple platform: Windows, Linux, OSX and more

› Multiple binding: Python, Ruby, Java, C# and more

› Clean, simple, intuitive & architecture-neutral API

› Provide break-down details on instructions

› Friendly license: Not GPL

Lots of Work !

- Multiple archs: x86, ARM

- Actively maintained & update within latest arch's change

- Multiple platform: Windows, Linux

- Understanding opcode, Intel x86 it self with 1500++ documented instructions

- Support python and ruby as binding languages

- Single man show

- Target finish within 12 months

Building A Good Disassembler

> Multiple archs: x86, ARM

> Actively maintained & update within latest arch's change

> Multiple platform: Windows, Linux

> Support python and ruby as binding languages

> Friendly license: BSD

> Easy to setup
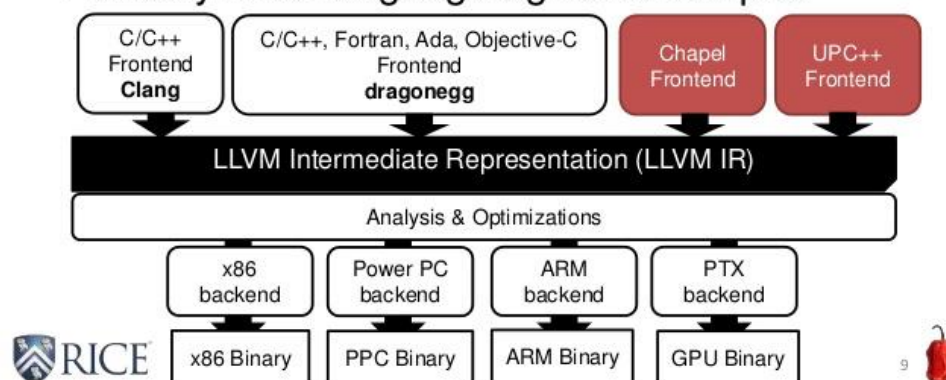
> ❯ Open source project compiler
> ❯ Sets of modules for machine code representing, compiling, optimizing
> ❯ Backed by many major players: AMD, Apple, Google, Intel, IBM, ARM, Imgtec, Nvidia, Qualcomm, Samsung, etc
> ❯ Incredibly huge (compiler) community around.
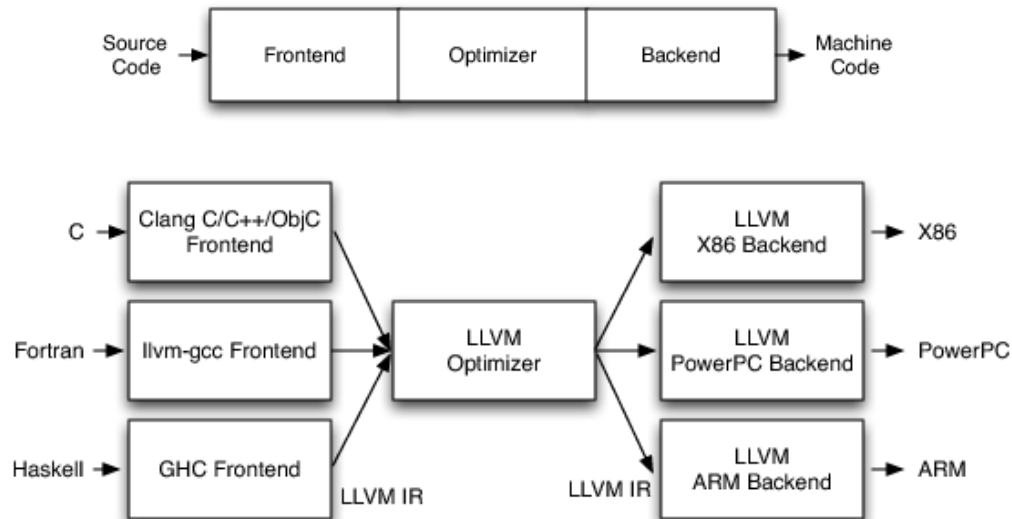
- › Multiple architectures ready
- › In-disassembler (MC module)
    - › Only, Only and Only build for LLVM
    - › actively maintained by the original vendor from the arch building company (eg, x86 from intel)
- › Very actively maintained & updated by a huge community

Not as Easy as Bitcoin CASH
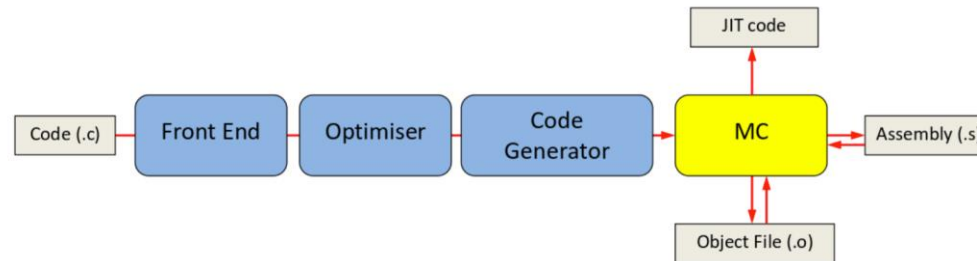
Issues

> Cannot just reuse MC as-is without huge efforts.

>> LLVM code is in C++, but we want C code.

>> Code mixed like spaghetti with lots of LLVM layers, not easy to take out

>> Need to build instruction breakdown-details ourselves.

>> Expose semantics to the API.

>> Not designed to be thread-safe.

>> Poor Windows support.

> Need to build all bindings ourselves.

> Keep up with upstream code once forking LLVM to maintain ourselves.

Solutions

> Fork LLVM but must remove everything we do not need

> Replicated LLVM's MC

>> Build around MC and not changing MC

>> Replace C++ with C

> Extend LLVM's MC

>> Isolate some global variable to make sure thread-safe

> Semantics information from TD file from LLVM

> cs_inn structure

>> Keep all information and group nicely

>> Make sure API are arch-independent

Capstone is not LLVM

Bitcoin Cash is not Bitcoin

More Superiors

- Zero dependency

- Compact in size

- More than assembly code

- Thread-safe design

- Able to embed into restricted firmware OS/ Environments

- Malware resistance (x86)

- Optimized for reverse engineers

- More hardware mode supported:- Big-Endian for ARM and ARM64

- More Instructions supported: 3DNow (x86)

More Robust

- Cannot always rely on LLVM to fix bugs
    - Disassembler is still conferred seconds-class LLVM, especially if does not affect code generation
    - May refuse to fix bugs if LLVM backed does not generate them (tricky x86 code)

- But handle all comer case properly is Capstone first priority
    - Handle all x86 malware ticks we aware of
    - LLVM could not care less

Demo

```c
1 /* test1.c */
2
3 #include <stdio.h>
4 #include <inttypes.h>
5
6 #include <capstone/capstone.h>
7
8 #define CODE "\x55\x48\x8b\x05\xb8\x13\x00\x00"
9
10 int main(void)
11 {
12   csh handle;
13   cs_insn *insn;
14   size_t count;
15
16   if (cs_open(CS_ARCH_X86, CS_MODE_64, &handle) != CS_ERR_OK)
17       return -1;
18   count = cs_disasm(handle, CODE, sizeof(CODE)-1, 0x1000, 0, &insn);
19   if (count > 0) {
20       size_t j;
21       for (j = 0; j < count; j++) {
22           printf("0x%"PRIx64":\t%s\t\t%s\n", insn[j].address, insn[j].mnemoni
23                   insn[j].op_str);
24       }
25
26       cs_free(insn, count);
27   } else
28       printf("ERROR: Failed to disassemble given code!\n");
29
30   cs_close(&handle);
31
32       return 0;
33 }
```

```python
1 # test1.py
2 from capstone import *
3
4 CODE = b"\x55\x48\x8b\x05\xb8\x13\x00\x00"
5
6 md = Cs(CS_ARCH_X86, CS_MODE_64)
7 for i in md.disasm(CODE, 0x1000):
8     print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
```

```
$ make
cc -c test1.c -o test1.o
cc test1.o -O3 -Wall -lcapstone -o test1

$ ./test1
0x1000: push        rbp
0x1001: mov     rax, qword ptr [rip + 0x13b8]
```

```
$ python test1.py

0x1000: push     rbp
0x1001: mov rax, qword ptr [rip + 0x13b8]
```

radare2

Unicorn Engine

NGUYEN Anh Quynh <aquynh -at- gmail.com>
DANG Hoang Vu <danghvu -at- gmail.com>

http://www.unicorn-engine.org

What is Wrong

Features	libemu	PyEmu	IDA-x86emu	libCPU
Multi-arch	X	X	X	X [1]
Updated	X	X	X	X
Independent	X [2]	X [3]	X [4]	✓
JIT	X	X	X	✓

> Nothing works even up until 2015 (First release of Unicorn Engine)
> Limited bindings
> Limited functions, limited architecture

What Do We Need ?

Features	libemu	PyEmu	IDA-x86emu	libCPU	Unicorn
Multi-arch	X	X	X	X	✓
Updated	X	X	X	X	✓
Independent	X	X	X	✓	✓
JIT	X	X	X	✓	✓

> Multiple archs: x86, x86_64, ARM+ ARM64 + Mips + PPC

> Multiple platform: Windows, Linux, OSX, Android and more

> Multiple binding: Python, Ruby, Java, C# and more

> Pure C implementation

> Latest and updated architecture

> With JIT compiler technique

> Instrumentation eg. F7, F8

Lots of Work !

> Multiple archs: x86, ARM

> Actively maintained & update within latest arch's change

> Multiple platform: Windows, Linux

> Understanding opcode, Intel x86 it self with 1500++ documented instructions

> Support python and ruby as binding languages

> Single man show

> Target finish within 12 months

Good Emulator

A Good Emulator

- Multiple archs: x86, x86_64, ARM, ARM64, Mips and more

- Actively maintained & update within latest arch's change

- Multiple platform: Windows, Linux, OSX, Android and more

- Code in pure C

- Support python and ruby as binding languages

- JIT compiler technique

- Instrumentation at various level
  - Single step
  - Instruction
  - Memory Access

QEMU, Maybe
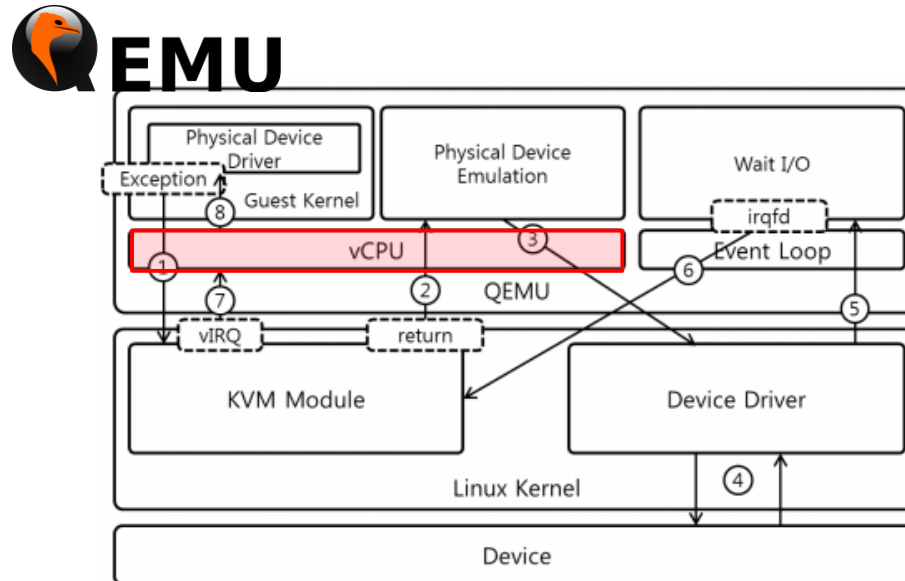
> Open source project on system emulator

> Very huge community and highly active

> Multiple architecture: x86, ARM, ARM64, Mips, PowerPC, Sparc, etc (18 architectures)

> Multiple platform: *nix and Windows

Bitcoin Gold. Oh Yeah
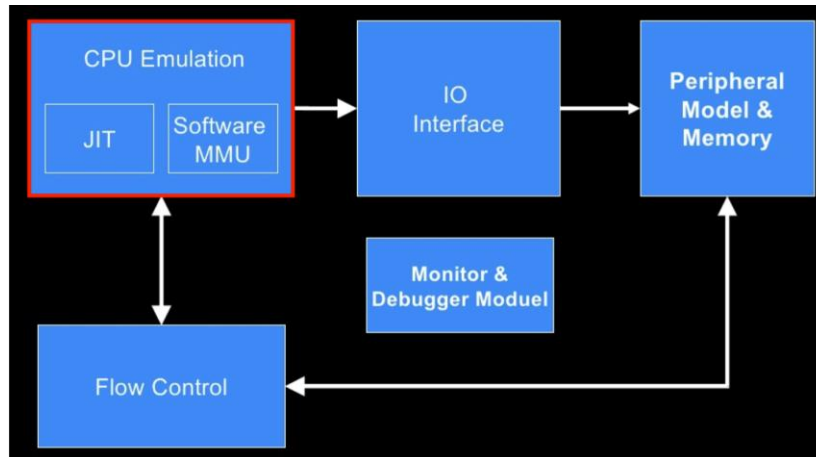
> ❯ Support all kind of architectures and very updated
> ❯ Already implemented in pure C, so easy to implement Unicorn core on top
> ❯ Already supported JIT in CPU emulation, optimization on of of JIT
> ❯ Are we done ?

Done ?

Issues 1

> Not just emulate CPU, but also device models & ROM/BIOS to fully emulate physical machines

> Qemu codebase is huge and mixed like spaghetti

> Difficult to read, as contributed by many different people

Solutions

> Keep only CPU emulation code & remove everything else (devices, ROM/BIOS, migration, etc)

> Keep supported subsystems like Qobject, Qom

> Rewrites some components but keep CPU emulation code intact (so easy to sync with Qemu in future)

Issues 2

> Set of emulators for individual architecture

>> Independently built at compile time

>> All archs code share a lot of internal data structures and global variables

> Unicorn wants a single emulator that supports all archs

Solutions

> Isolated common variables & structures

>> Ensured thread-safe by design

> Refactored to allow multiple instances of Unicorn at the same time Modified the build system to support multiple archs on demand

Not Yet?

Issues 3

> Instrumentation for static compilation only

> JIT optimizes for performance with lots of fast-path tricks, making code instrumenting extremely hard

Solutions

> Build dynamic fine-grained instrumentation layer from scratch Support various levels of instrumentation

>> Single-step or on particular instruction (TCG level)

>> Instrumentation of memory accesses (TLB level)

>> Dynamically read and write register

>> Handle exception, interrupt, syscall (arch-level) through user provided callback.

Issues 4

> Objects is open (malloc) without closing (freeing) properly everywhere

> Fine for a tool, but unacceptable for a framework

Solutions

> Find and fix all the memory leak issues

> Refactor various subsystems to keep track and cleanup dangling pointers

Unicorn is not QEMU
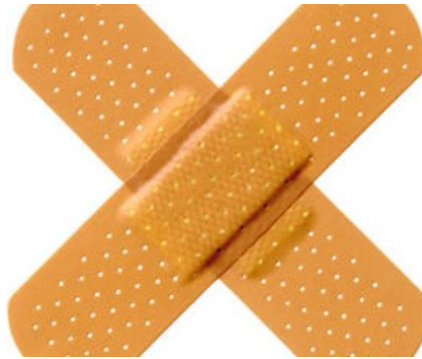
- › Independent framework
- › Much more compact in size, lightweight in memory
- › Thread-safe with multiple architectures supported in a single binary Provide interface for dynamic instrumentation
- › More resistant to exploitation (more secure)
  - › CPU emulation component is never exploited!
  - › Easy to test and fuzz as an API.

```c
1 #include <unicorn/unicorn.h>
2
3 // code to be emulated
4 #define X86_CODE32 "\x41\x4a" // INC ecx; DEC edx
5
6 // memory address where emulation starts
7 #define ADDRESS 0x1000000
8
9 int main(int argc, char **argv, char **envp)
10 {
11   uc_engine *uc;
12   uc_err err;
13   int r_ecx = 0x1234;     // ECX register
14   int r_edx = 0x7890;     // EDX register
15
16   printf("Emulate i386 code\n");
17
18   // Initialize emulator in X86-32bit mode
19   err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
20   if (err != UC_ERR_OK) {
21     printf("Failed on uc_open() with error returned: %u\n", err);
22     return -1;
23   }
24
25   // map 2MB memory for this emulation
26   uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);
27
28   // write machine code to be emulated to memory
29   if (uc_mem_write(uc, ADDRESS, X86_CODE32, sizeof(X86_CODE32) - 1)) {
30     printf("Failed to write emulation code to memory, quit!\n");
31     return -1;
32   }
33
34   // initialize machine registers
35   uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
36   uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);
37
38   // emulate code in infinite time & unlimited instructions
39   err=uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32) - 1, 0, 0);
40   if (err) {
41     printf("Failed on uc_emu_start() with error returned %u: %s\n",
42       err, uc_strerror(err));
43   }
44
45   // now print out some registers
46   printf("Emulation done. Below is the CPU context\n");
47
48   uc_reg_read(uc, UC_X86_REG_ECX, &r_ecx);
49   uc_reg_read(uc, UC_X86_REG_EDX, &r_edx);
50   printf(">>> ECX = 0x%x\n", r_ecx);
51   printf(">>> EDX = 0x%x\n", r_edx);
52
53   uc_close(uc);
54
55   return 0;
56 }
```

```
$ make
cc  test1.c -L/usr/local/Cellar/glib/2.44.1/lib -L/usr/local/opt/gettext/...

$ ./test1
Emulate i386 code
Emulation done. Below is the CPU context
>>> ECX = 0x1235
>>> EDX = 0x788f
```

```python
1 from __future__ import print_function
2 from unicorn import *
3 from unicorn.x86_const import *
4
5 # code to be emulated
6 X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
7
8 # memory address where emulation starts
9 ADDRESS = 0x1000000
10
11 print("Emulate i386 code")
12 try:
13     # Initialize emulator in X86-32bit mode
14     mu = Uc(UC_ARCH_X86, UC_MODE_32)
15
16     # map 2MB memory for this emulation
17     mu.mem_map(ADDRESS, 2 * 1024 * 1024)
18
19     # write machine code to be emulated to memory
20     mu.mem_write(ADDRESS, X86_CODE32)
21
22     # initialize machine registers
23     mu.reg_write(UC_X86_REG_ECX, 0x1234)
24     mu.reg_write(UC_X86_REG_EDX, 0x7890)
25
26     # emulate code in infinite time & unlimited instructions
27     mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
28
29     # now print out some registers
30     print("Emulation done. Below is the CPU context")
31
32     r_ecx = mu.reg_read(UC_X86_REG_ECX)
33     r_edx = mu.reg_read(UC_X86_REG_EDX)
34     print(">>> ECX = 0x%x" %r_ecx)
35     print(">>> EDX = 0x%x" %r_edx)
36
37 except UcError as e:
38     print("ERROR: %s" % e)
```

```
$ python test1.py

Emulate i386 code
Emulation done. Below is the CPU context
>>> ECX = 0x1235
>>> EDX = 0x788f
```

Keystone Engine

NGUYEN Anh Quynh <aquynh -at- gmail.com>

http://www.keystone-engine.org

What Is Wrong

› Nothing is up to our standard, even in 2016!

  › Yasm: X86 only, no longer updated

  › Intel XED: X86 only, miss many instructions & closed-source

  › Use assembler to generate object files

  › Other important archs: Arm, Arm64, Mips, PPC, Sparc, etc?

What Do We Need

> Multiple archs: x86, ARM+ ARM64 + Mips + PPC and more

> Multiple platform: Windows, Linux, OSX and more

> Multiple binding: Python, Ruby, Java, C# and more

> Clean, simple, intuitive & architecture-neutral API

> Provide break-down details on instructions

> Friendly license: BSD

Lots of Work !

- Multiple archs: x86, ARM

- Actively maintained & update within latest arch's change

- Multiple platform: Windows, Linux

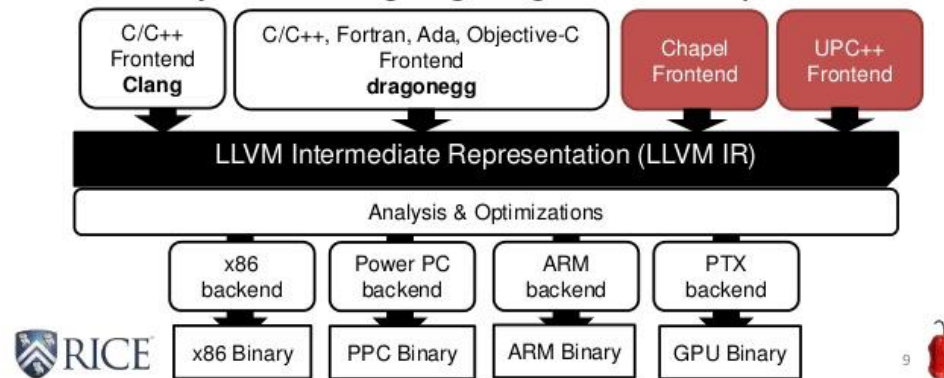- Understanding opcode, Intel x86 it self with 1500++ documented instructions

- Support python and ruby as binding languages

- Single man show

- Target finish within 12 months

A Good Assembler

> Multiple archs: x86, ARM

> Actively maintained & update within latest arch's change

> Multiple platform: Windows, Linux

> Support python and ruby as binding languages

> Friendly license (BSD)

> Easy to setup

Why LLVM?

❑ Widely used language-agnostic compiler

C/C++ Frontend **Clang**	C/C++, Fortran, Ada, Objective-C Frontend **dragonegg**	Chapel Frontend	UPC++ Frontend

LLVM Intermediate Representation (LLVM IR)

Analysis & Optimizations

x86 backend	Power PC backend	ARM backend	PTX backend
x86 Binary	PPC Binary	ARM Binary	GPU Binary

RICE

9

› Open source project compiler

› Sets of modules for machine code representing, compiling, optimizing

› Backed by many major players: AMD, Apple, Google, Intel, IBM, ARM, Imgtec, Nvidia, Qualcomm, Samsung, etc

› Incredibly huge (compiler) community around.

Bitcoin Silver !
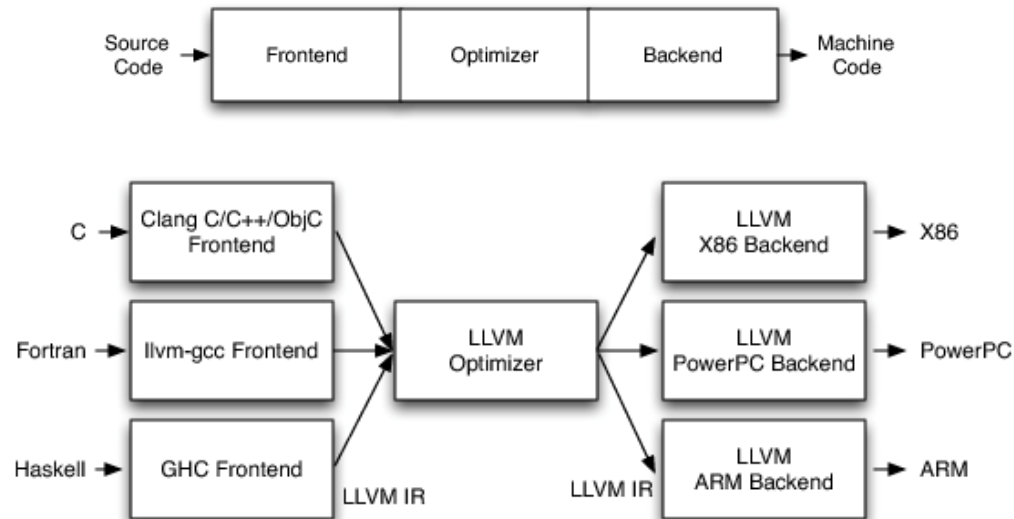
- ❯ Multiple architectures ready
- ❯ In-build assembler (MC module)
  - ❯ Only, Only and Only build for LLVM
  - ❯ actively maintained
- ❯ Very actively maintained & updated by a huge community

Easy Job !

Issue 1

> LLVM not just assembler, but also disassembler, bitcode, InstPrinter, Linker Optimization, etc

> LLVM codebase is huge and mixed like spaghetti

Solutions

> Keep only assembler code & remove everything else unrelated

> Rewrites some components but keep AsmParser, CodeEmitter & AsmBackend code intact (so easy to sync with LLVM in future, e.g. update)

> Keep all the code in C++ to ease the job (unlike Capstone)

>> No need to rewrite complicated parsers

>> No need to fork llvm-tblgen

Issue 2

> LLVM compiled into multiple libraries

>> Supported libs

>> Parser

>> TableGen and etc

> Keystone needs to be a single library

Solutions

> Modify linking setup to generate a single library

>> libkeystone.[so, dylib] + libkeystone.a

>> keystone.dll + keystone.lib

Not So Easy

Issue 3

> Relocation object code generated for linking in the final code generation phase of compiler

> Ex on X86:
> > inc [_var1] → 0xff, 0x04, 0x25, A, A, A, A

Solutions

> Make fixup phase to detect & report missing symbols

> Propagate this error back to the top level API ks_asm()

Issue 4

> Ex on ARM: blx 0x86535200 → 0x35, 0xf1, 0x00, 0xe1

Solutions

> ks_asm() allows to specify address of first instruction

> Change the core to retain address for each statement

> Find all relative branch instruction to fix the encoding according to current & target address

More Problems

Issue 5

- Ex on X86: vaddpd zmm1, zmm1, zmm1, x → "this is not an immediate"
- Returned llvm_unreachable() on input it cannot handle

Solutions

- Fix all exits & propagate errors back to ks_asm()
  - Parse phase
  - Code emit phase

Issue 6

- LLVM does not support non-LLVM syntax
  - We want other syntaxes like Nasm, Masm, etc
- Bindings must be built from scratch
- Keep up with upstream code once forking LLVM to maintain ourselves

Solutions

- Extend X86 parser for new syntaxes: Nasm, Masm, etc
- Built Python binding
- Extra bindings came later, by community: NodeJS, Ruby, Go, Rust, Haskell & OCaml
- Keep syncing with LLVM upstream for important changes & bug-fixes

Keystone is not LLVM

Fork and Beyond

- Independent & truly a framework
    - Do not give up on bad-formed assembly
- Aware of current code position (for relative branches)
- Much more compact in size, lightweight in memory
- Thread-safe with multiple architectures supported in a single binary More flexible: support X86 Nasm syntax
- Support undocumented instructions: X86
- Provide bindings (Python, NodeJS, Ruby, Go, Rust, Haskell, OCaml as of August 2016)
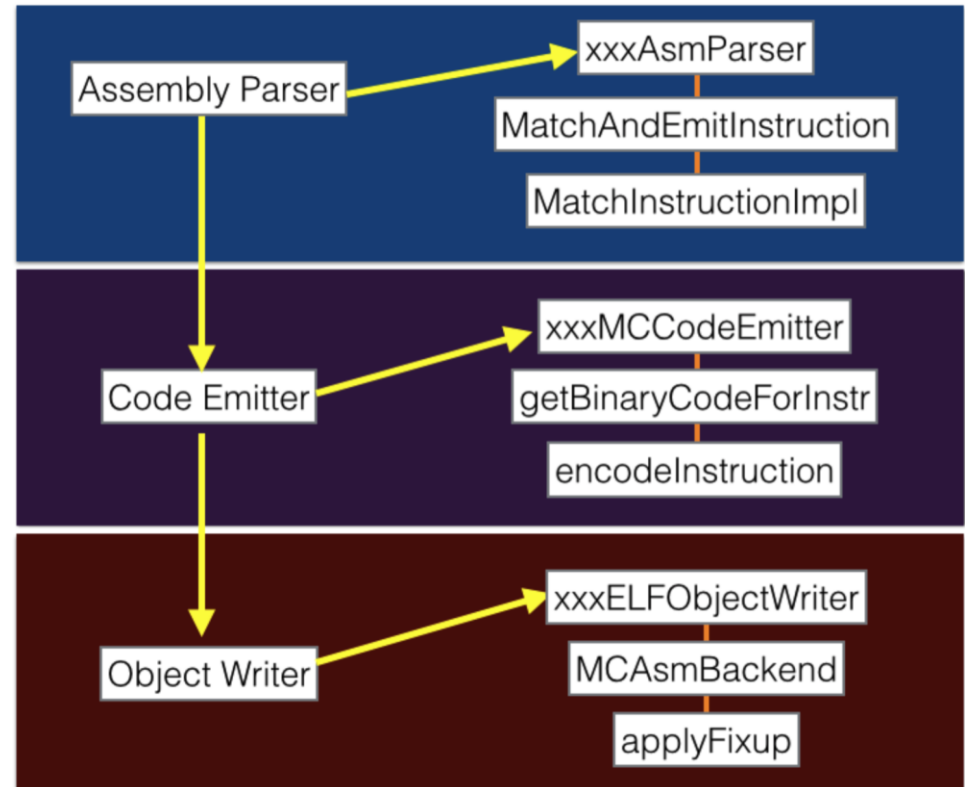
```c
1  /* test1.c */
2  #include <stdio.h>
3  #include <keystone/keystone.h>
4
5  // separate assembly instructions by ; or \n
6  #define CODE "INC ecx; DEC edx"
7
8  int main(int argc, char **argv)
9  {
10     ks_engine *ks;
11     ks_err err;
12     size_t count;
13     unsigned char *encode;
14     size_t size;
15
16     err = ks_open(KS_ARCH_X86, KS_MODE_32, &ks);
17     if (err != KS_ERR_OK) {
18         printf("ERROR: failed on ks_open(), quit\n");
19         return -1;
20     }
21
22     if (ks_asm(ks, CODE, 0, &encode, &size, &count) != KS_ERR_OK) {
23         printf("ERROR: ks_asm() failed & count = %lu, error = %u\n",
24                     count, ks_errno(ks));
25     } else {
26         size_t i;
27
28         printf("%s = ", CODE);
29         for (i = 0; i < size; i++) {
30             printf("%02x ", encode[i]);
31         }
32         printf("\n");
33         printf("Compiled: %lu bytes, statements: %lu\n", size, count);
34     }
35
36     // NOTE: free encode after usage to avoid leaking memory
37     ks_free(encode);
38
39     // close Keystone instance when done
40     ks_close(ks);
41
42     return 0;
43  }
```

```
$ make
cc -o test1 test1.c -lkeystone -lstdc++ -lm

$ ./test1
INC ecx; DEC edx = 41 4a
Compiled: 2 bytes, statements: 2
```

```python
1  from keystone import *
2
3  # separate assembly instructions by ; or \n
4  CODE = b"INC ecx; DEC edx"
5
6  try:
7      # Initialize engine in X86-32bit mode
8      ks = Ks(KS_ARCH_X86, KS_MODE_32)
9      encoding, count = ks.asm(CODE)
10     print("%s = %s (number of statements: %u)" %(CODE, encoding, count))
11 except KsError as e:
12     print("ERROR: %s" %e)
```

```
$ ./test1.py
INC ecx; DEC edx = [65, 74] (number of statements: 2)
```

metame

Before

After

One More Thing

Instant Patch

IDA Pro
- RE Standard
- Patching on the fly is always a must
- Broken "Edit\Patch Program\ Assembler" is always giving us problem

Assemble instruction

Previous line:

Address : 0x3 : 0x15860

Instruction: `push   rbp|`

Help   Cancel   OK

PUSH RAX

Invalid operand

OK

ARM

Sorry, this processor module doesn't support the assembler.

☐ Don't display this message again (for this session only)

OK
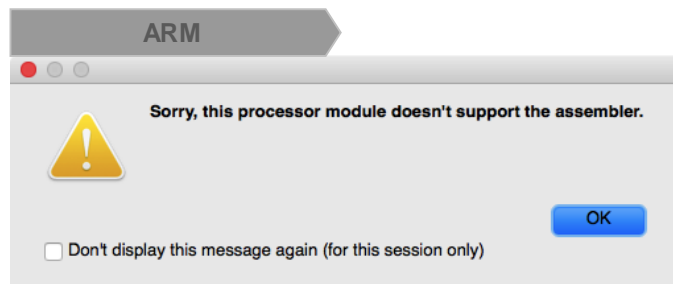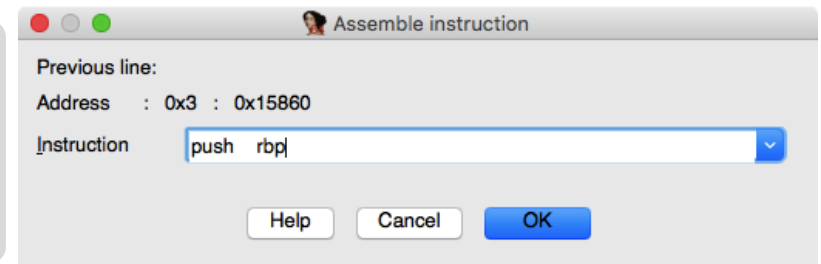
PUSH ESI

```
15860 55                      push    rbp
15861 48 8D 57 10             lea     rdx, [rdi+10h]
15865 56                      push    rsi
15866 48 89 FB                mov     rbx, rdi
15869 50                      push    rax
```
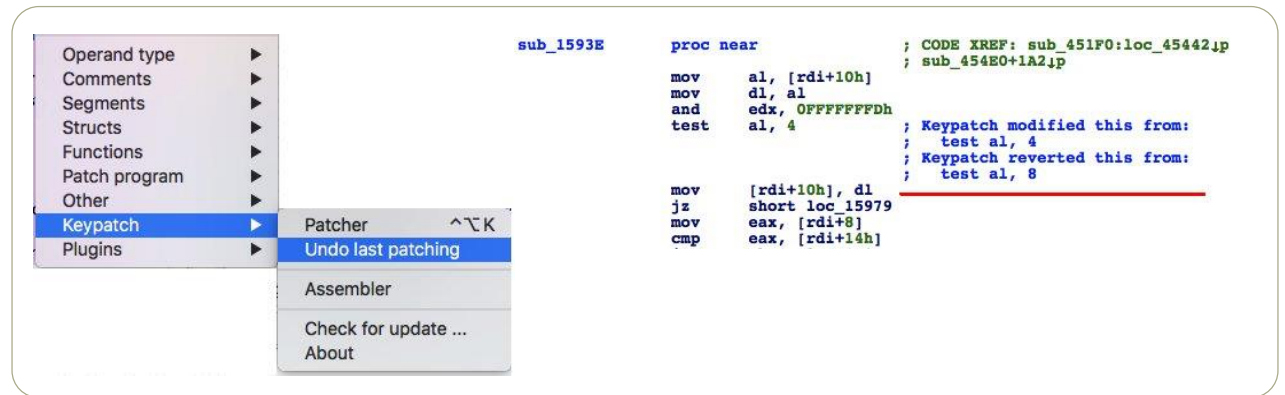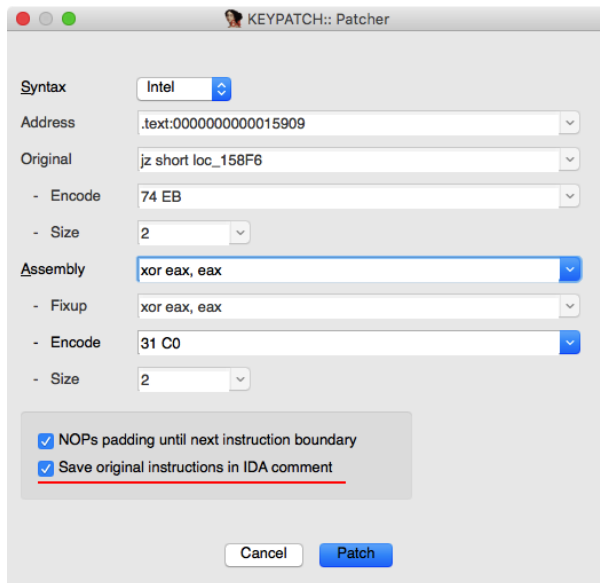
Keypatch

A binary editor plugin for IDA Pro

- Fully open source @ https://keystone-engine.org/keypatch
- On the fly patching in IDA Pro with Multi Arch
- Base on Keystone Engine
- By Nguyen Anh Quynh & Thanh Nguyen (rd) from vnsecurity.net

Latest Version works with IDA Pro 7

Fill Range
- Select Start, End range and patch with bytes
- Goto: Edit | Keypatch | Fill Range
- QQ: 2880139049

Thank You

kaijern <kj -at- reverser.io>
twitter: @sgniwx
weibo: 爱吃饺子的程序员

https://github.com/xwings