Unicorn: Next Generation CPU Emulator Framework

www.unicorn-engine.org

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

BlackHat USA, August 5th 2015
Self-introduction

- **Nguyen Anh Quynh (aquynh -at- gmail.com)**
  - PhD in Computer Science, security researcher
  - Operating System, Virtual Machine, Binary analysis, Forensic, etc
  - Capstone disassembly framework ([capstone-engine.org](http://capstone-engine.org))

- **Dang Hoang Vu (danghvu -at- gmail.com)**
  - PhD candidate in Computer Science at UIUC, security hobbyist
  - Member of VNSecurity.NET, casual CTF player, exploit writer
  - Capstone, Peda contributor
Algorithm of a Simplest Neural Network

1. **Input Layer**
   - Receive raw data

2. **Hidden Layer(s)**
   - Process data through weights and biases
   - Use activation functions

3. **Output Layer**
   - Generate predictions or classifications

4. **Backpropagation**
   - Calculate errors in the output layer
   - Adjust weights based on errors

Training the Model

1. **Initialization**
   - Set random weights and biases

2. **Forward Pass**
   - Propagate inputs through the network
   - Compute outputs

3. **Loss Calculation**
   - Compute the difference between predicted and actual outputs

4. **Backpropagation**
   - Propagate errors backward through the network
   - Update weights and biases

5. **Iteration**
   - Repeat steps 2 to 4 for multiple epochs

Evaluation

1. **Validation Set**
   - Test the model on unseen data
   - Calculate accuracy, precision, recall, etc.

2. **Hyperparameter Tuning**
   - Adjust learning rate, batch size, etc.
   - Use techniques like grid search or random search

3. **Final Testing**
   - Evaluate the model on an independent test set

Deployment

1. **Project Setup**
   - Environment setup
   - Dependencies installation

2. **Model Packaging**
   - Compile the model
   - Package for deployment

3. **Deployment**
   - Deploy the model in a server or cloud environment
   - Monitor performance and respond to errors
CPU Emulator

Definition
- Emulate physical CPU - using software only.
- Focus on CPU operations only, but ignore machine devices.

Applications
- Emulate the code without needing to have a real CPU.
  - Cross-architecture emulator for console game.
- Safely analyze malware code, detect virus signature.
- Verify code semantics in reversing.
Example

- Emulate to understand code semantics.

```assembly
mov eax, 0x30
mov esi, ecx
mov ebx, 0x45
add ecx, 0x78
sub ebx, 0x22
inc ecx
dec eax
mov ecx, eax
and ebx, 0x99
sub eax, 0x23
xor esi, esi
jz $l0

_l0:
shl ecx, 1
add eax, ebx
xor edx, edx
inc ebx
jmp $l1
nop

_l1:
shr ecx, 1
sub ecx, 0x11
inc eax
and esp, -0x10
dec eax
```
Internals of CPU emulator

Given input code in binary form

- Decode binary into separate instructions
- Emulate exactly what each instruction does
  - Instruction-Set-Architecture manual referenced is needed
  - Handle memory access & I/O upon requested
- Update CPU context (registors/memory/etc) after each step
Example of emulating X86 32bit instructions

- **Ex: 50 → push eax**
  - load `eax` register
  - copy `eax` value to stack bottom
  - decrease `esp` by 4, and update `esp`

- **Ex: 01D1 → add eax, ebx**
  - load `eax` & `ebx` registers
  - add values of `eax` & `ebx`, then copy result to `eax`
  - update flags OF, SF, ZF, AF, CF, PF accordingly
Challenges of building CPU emulator

Huge amount of works!

- Good understanding of CPU architecture
- Good understanding of instruction set
- Instructions with various side-effect (sometimes undocumented, like ex: Intel X86)
- Tough to support all kind of code existed
Good CPU emulator?

- Multi-arch?
  - X86, Arm, Arm64, Mips, PowerPC, Sparc, etc
- Multi-platform?
  - *nix, Windows, Android, iOS, etc
- Updated?
  - Keep up with latest CPU extensions
- Independent?
  - Support to build independent tools
- Good performance?
  - Just-In-Time (JIT) compiler technique vs Interpreter
## Existing CPU Emulators

<table>
<thead>
<tr>
<th>Features</th>
<th>libemu</th>
<th>PyEmu</th>
<th>IDA-x86emu</th>
<th>libCPU</th>
<th>Dream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-arch</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Updated</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Independent</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JIT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Multi-arch: existing tools only support X86
- Updated: existing tools do not support X86_64

---

1. Possible by design, but nothing actually works
2. Focus only on detecting Windows shellcode
3. Python only
4. For IDA only
Dream a good emulator

- Multi-architectures
  - Arm, Arm64, Mips, PowerPC, Sparc, X86 (+X86_64) + more
- Multi-platform: *nix, Windows, Android, iOS, etc
- Updated: latest extensions of all hardware architectures
- Independent with multiple bindings
  - Low-level framework to support all kind of OS and tools
  - Core in pure C, and support multiple binding languages
- Good performance with JIT compiler technique
  - Dynamic compilation vs Interpreter
- Allow instrumentation at various levels
  - Single-step/instruction/memory access
Problems

- No reasonable CPU emulator even in 2015!
- Apparently nobody wants to fix the issues
- No light at the end of the dark tunnel
- Until **Unicorn** was born!
Unicorn == Next Generation CPU Emulator
Goals of Unicorn

- Multi-architectures
  - Arm, Arm64, Mips, PowerPC, Sparc, X86 (+X86_64) + more
- Multi-platform: *nix, Windows, Android, iOS, etc
- Updated: latest extensions of all hardware architectures
- Core in pure C, and support multiple binding languages
- Good performance with JIT compiler technique
- Allow instrumentation at various levels
  - Single-step/instruction/memory access
Unicorn vs others

<table>
<thead>
<tr>
<th>Features</th>
<th>libemu</th>
<th>PyEmu</th>
<th>IDA-x86emu</th>
<th>libCPU</th>
<th>Unicorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-arch</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Updated</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Independent</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JIT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Multi-arch: existing tools only support X86
- Updated: existing tools do not supports X86_64
Challenges to build Unicorn engine

Huge amount of works!
- Too many hardware architectures
- Too many instructions
- Instructions with various side-effect (sometimes undocumented, like Intel X86)
- Hard to support all kind of code existed
- Limited resource
  - Started as a personal for-fun in-spare-time project
Unicorn design
Ambitions & ideas

- Have all features in months, not years!
- Stand on the shoulders of the giants at the initial phase.
- Open source project to get community involved & contributed.
- Idea: Qemu!
Introduction on Qemu

Qemu project

- Open source project (GPL license) on system emulator: http://www.qemu.org
- Huge community & highly active
- Multi-arch
  - X86, Arm, Arm64, Mips, PowerPC, Sparc, etc (18 architectures)
- Multi-platform
  - Compile on *nix + cross-compile for Windows
Qemu architecture

 Courtesy of cmchao
Why Qemu?

- 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
Are we done?

FORK ALL THE THINGS

DONE!

imgflip.com
Challenges to build Unicorn (1)

Qemu codebase is a challenge
- 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

Unicorn job
- 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)
Challenges to build Unicorn (2)

Qemu is set of emulators

- 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 :-(

Unicorn job

- 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
Challenges to build Unicorn (3)

Qemu has no instrumentation

- Instrumentation for static compilation only
- JIT optimizes for performance with lots of fast-path tricks, making code instrumenting extremely hard :-(

Unicorn job

- 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 or memory during emulation.
  - Handle exception, interrupt, syscall (arch-level) through user provided callback.
Challenges to build Unicorn (4)

Qemu is leaking memory

- Objects is open (malloc) without closing (freeing) properly everywhere
- Fine for a tool, but unacceptable for a framework

Unicorn job

- Find and fix all the memory leak issues
- Refactor various subsystems to keep track and cleanup dangling pointers.
Unicorn vs Qemu

Forked Qemu, but go far beyond it

- 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.
### Qemu vulnerabilities

<table>
<thead>
<tr>
<th>CVE-2015-5165</th>
<th>QEMU leak of uninitialized heap memory in rtl8139 device model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2015-5166</td>
<td>Use after free in QEMU/Xen block unplug protocol</td>
</tr>
<tr>
<td>CVE-2015-5154</td>
<td>QEMU heap overflow flaw while processing certain ATAPI commands.</td>
</tr>
<tr>
<td>CVE-2015-3209</td>
<td>Heap overflow in QEMU PCNET controller, allowing guest→host escape</td>
</tr>
<tr>
<td>CVE-2015-4106</td>
<td>Unmediated PCI register access in qemu</td>
</tr>
<tr>
<td>CVE-2015-4105</td>
<td>Guest triggerable qemu MSI-X pass-through error messages</td>
</tr>
<tr>
<td>CVE-2015-4103</td>
<td>Potential unintended writes to host MSI message data field via qemu</td>
</tr>
<tr>
<td>CVE-2015-2756</td>
<td>Unmediated PCI command register access in qemu</td>
</tr>
<tr>
<td>CVE-2015-2152</td>
<td>HVM qemu unexpectedly enabling emulated VGA graphics backends</td>
</tr>
<tr>
<td>CVE-2013-4375</td>
<td>qemu disk backend (qdisk) resource leak</td>
</tr>
<tr>
<td>CVE-2013-4344</td>
<td>qemu SCSI REPORT LUNS buffer overflow</td>
</tr>
<tr>
<td>CVE-2013-2007</td>
<td>qemu guest agent (qga) insecure file permissions</td>
</tr>
<tr>
<td>CVE-2013-1922</td>
<td>qemu-nbd format-guessing due to missing format specification</td>
</tr>
<tr>
<td>CVE-2012-6075</td>
<td>qemu (e1000 device driver): Buffer overflow when processing large packets</td>
</tr>
</tbody>
</table>
Write applications with Unicorn
**Introduce Unicorn API**

- Clean/simple/lightweight/intuitive architecture-neutral API.
- The core provides API in C
  - open & close Unicorn instance
  - start & stop emulation (based on end-address, time or instructions count)
  - read & write memory
  - read & write registers
  - memory management: hook memory events, dynamically map memory at runtime
    - hook memory events for invalid memory access
    - dynamically map memory at runtime (handle invalid/missing memory)
  - instrument with user-defined callbacks for instructions/single-step/memory event, etc
- Python binding built around the core
Sample code in C

```c
#define X86_CODE32 "\x41\x4a" // INC ecx; DEC dex
#define ADDRESS 0x18000000 // memory address where emulation starts

static void test_i386(void) {
    uc_handle;
    uc_err err;
    uc_errno err2;
    uc_regs uc_regs;
    int r_ecx = 0x1234; // ECX register
    int r_edx = 0x7890; // EDX register

    // Initialize emulator in X86-32bit mode
    err = uc_open(UC_ARCH_X86, UC_MODE_32, &handle);
    if (err) {
        printf("Failed on uc_open() with error returned: %s", err);
        return;
    }

    // read from memory
    if (uc_mem_read(handle, ADDRESS, (uint8_t *)&tmp, 4)) {
        printf(">>> Read 4 bytes from [0x%x] = 0x%x\n", ADDRESS, tmp);
    } else {
        printf(">>> Failed to read 4 bytes from [0x%x]\n", ADDRESS);
    }

    uc_close(&handle);
}
```
Sample code in Python

```
x86_CODE32 = b"\x41\x4a"  # INC ecx; DEC dex
ADDRESS = 0x1000000  # memory address where emulation starts

print("Emulate i386 code")
try:
    # Initialize emulator in X86-32bit mode
    mu = Uc(UC_ARCH_X86, UC_MODE_32)

    # map 2MB memory for this emulation
    mu.mem_map(ADDRESS, 2 * 1024 * 1024)

    # write machine code to be emulated to memory
    mu.mem_write(ADDRESS, X86_CODE32)

    # initialize machine registers
    mu.reg_write(X86_REG_ECX, 0x1234)
    mu.reg_write(X86_REG_EDX, 0x7890)

    # emulate machine code in infinite time
    mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))

    # done. now print out some registers
    r_ecx = mu.reg_read(X86_REG_ECX)
    r_edx = mu.reg_read(X86_REG_EDX)
    print(">> ECX = 0x%0x" % r_ecx)
    print(">> EDX = 0x%0x" % r_edx)

    # read from memory
    tmp = mu.mem_read(ADDRESS, 2)
    print(">> Read 2 bytes from [0x%0x] = " % (ADDRESS), end="")
    for i in tmp:
        print("%02x" % i, end="")
    print("\n")

except UcError as e:
    print("ERROR: %s" % e)
```
Live demo
Status & future works

Status

- Support Arm, Arm64, Mips, M68K, PowerPC, Sparc, X86 (+X86_64)
- Python binding available
- Based on Qemu 2.3

Future works

- Support all the rest architectures of Qemu (alpha/s360x/microblaze/sh4/etc - totally 18)
- Stripping more utility code from Qemu e.g. improve the disassembler (with potential integration with Capstone).
- More bindings promised by community!
- Synchronize with Qemu 2.4 (released soon)
  - Future of Unicorn is guaranteed by Qemu active development!
Conclusions

- **Unicorn** is an innovative next generation CPU emulator
  - Multi-arch + multi-platform
  - Clean/simple/lightweight/intuitive architecture-neutral API
  - Implemented in pure C language, with bindings for Python available.
  - High performance with JIT compiler technique
  - Support fine-grained instrumentation at various levels.
  - Thread-safe by design.
  - Open source GPL license.
  - Future update guaranteed for all archs.

- We are seriously committed to this project to make it the best CPU emulator.
Call for beta testers

- Run beta test before official release
- Willing to help? If you can code, contact us!
  - Unicorn homepage: http://www.unicorn-engine.org
  - Unicorn twitter: @unicorn_engine
  - Unicorn mailing list: http://www.freelists.org/list/unicorn-engine

- First public version to be released after the beta phase - in GPL license.
Questions and answers

Unicorn: Next Generation CPU Emulator Framework

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

- Qemu: http://www.qemu.org
- libemu: http://libemu.carnivore.it
- PyEmu: http://code.google.com/p/pyemu
- libcpu: https://github.com/libcpu/libcpu
- IDA-x86emu: http://www.idabook.com/x86emu/index.html
- Unicorn engine
  - Homepage: http://www.unicorn-engine.org
  - Mailing list: http://www.freelists.org/list/unicorn-engine
  - Twitter: @unicorn_engine
Acknowledgement

- Nguyen Tan Cong for helped with the shellcode demo!
- Other beta testers helped to improve our code!