Emulator Design, Traps and Pitfalls

Paul Irofti
paul@irofti.net

Analysis of Executables: Benefits and Challenges
Dagstuhl Seminar, 2012

Outline

1. Intermediate Representation
2. Translator
3. Compiler
4. Memory Management Unit
5. Virtual File System
6. System Calls
7. Conclusions

ISA

Objectives

- Sane
- Orthogonal
- Small

Sane – Things to Avoid

IA-32 and friends

- variable instruction size
- ambiguity
- unaligned access

Example

Before

```
407F1A E834000000 CALL sample.407F53
...
407F4F 20978CEAF873 AND BYTE PTR DS:[EDI+73F8EA8C],DL
407F55 020F ADD CL, BYTE PTR DS:[EDI]
```

After

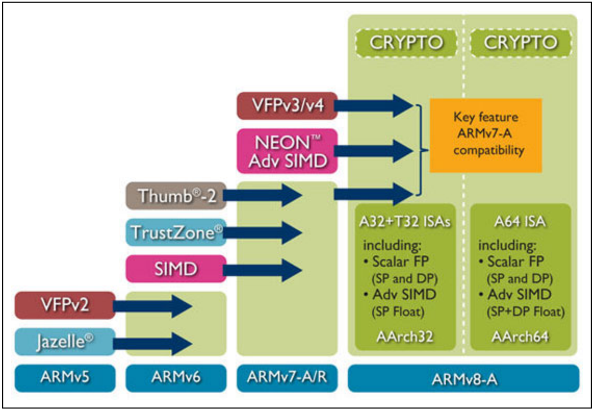
```
407F53 F8 CLC
407F54 7302 JNB SHORT sample.407F67
```

NOTE: Could've been worse, could've been a RET.

Sane – Things to Avoid (2)

ARM

ISA revamp every 6 months



Orthogonal

Definition

All instruction types can use all addressing modes.

Example

- VAX
- ARM11

Small

When designing the ISA:

- keep a simple minimal set of instructions
- make sure you can reduce other CISC instructions to it

When Picking Registers

There's not much to consider except:

- default size (32/64/128)
- granularity (RAX/EAX/AX/AH/AL or NONE)
- how many (why make billions when you can make millions?)

Choosing a stack

Only a few choices really:

- have a machine word-sized stack
- have a granularity-challenged stack (see IA-32)
- have no stack at all

Chaos – Going after ISA extensions

This is the deal breaker. Options:

- design only for a specific platform (e.g. IA-32-based only)
- sprinkle hacks throughout the codebase
- create a dog-slow pseudo-syscall system specially for them
- ignore them and hope for the best

Disassembler

For each ISA implement a disassembler that:

- tokenizes the instructions
- fetches the implicit or explicit opcode arguments
- dispatches for translation

Handlers

For each opcode have a translating function that:

- receives its arguments
- writes out the equivalent functionality in IR opcode(s)

Once everything is translated in IR one can:

- compile
- interpret
- do a mixture of the two

Accessing memory

What happens when any of the following needs to be emulated?

```
MOV EAX, [1000]
JMP [EDX]
STOS DWORD PTR ES:[EDI]
```

Resolving memory access requests

Keeping track of memory writes and reads. Requires:

- Initial memory state – OS dependent
- Stack state – partially OS dependent
- Doing writes and reads on an internally stored memory map

Other optimizations depending on design choice.

Accessing imports

But what if the following code pops-up?

```
01002E8D PUSH ESI
01002E8E LEA EAX, [EBP-0x8]
01002E91 PUSH EAX
01002E92 CALL DWORD [0x1001074]
7DD85AB0 CALL DWORD 0x7dd85ab5
```

An API call to kernel32.dll!GetSystemTimeAsFileTime.

File System Access

Or what if the sample wants to:

- create, read, or write a file?
- touch magical things like the registry?
- have a gentoo-ish peek at /proc and optimize itself?

The Need for a Virtual File System

It is obvious that you need to create a sort of fake fs that:

- stores created or modified files throughout the emulation
- provides a minimal fs environment resembling the expected OS
- takes care of special features such as registry
- mimics special files such as the ones found in /proc and /dev

Low Level System Specific Functions

The API call problem still exists.
A connection between the sample and the library needs to be made.

Solution

Write a loader for each expected filetype (e.g. PE, ELF).

Loaders

A loader should:

- setup the virtual address space for the sample
- resolve link to external libraries

Function Implementations

Options:

- emulate the real functions
- roll your own and run them outside emulation
- a mix of the two

Native Implementations

Advantages:

- speed – ran outside emulation
- trusted – code you wrote
- usually smaller code size

Native Implementations(2)

Disadvantages:

- crashing brings everything down
- harder to debug

Emulating Implementations

Advantages:

- better control
- crashing doesn't affect the emulator
- you can feed the original binary from VFS
- less time spent in development

Emulating Implementations(2)

Disadvantages:

- very slow due to emulation
- slow due to complexity you might not need
- binary redistribution rights

Mostly Harmless

Writing an emulator is juggling with trade-offs

- speed
- generalization
- information retrieval
- hair loss

So Long, and Thanks for All the Fish

Questions?