

# Instruction Tracing and Application Profiling with QEMU

Radu VELEA and Mihai TOGAN

**Abstract**—This paper presents modifications done to the QEMU code base in order to enable instruction level tracing and profiling of individual applications or Linux systems. The data gathered can be used to calculate path length, memory behavior, cache hit-miss ratios and model application and system behavior in relation to a certain instruction set architecture (ISA). This is useful in comparing performance across different hardware architectures and exposing limitations of different execution models. The instruction flow can further be used to inspect the run-time behavior of the targeted applications and determine its impact on the system.

**Index Terms**—Binary analysis, characterization, emulation, instruction tracing, profiling, reverse engineering.

## I. INTRODUCTION

QEMU is an open source project aimed at enabling virtualization for binary and system execution. The current work uses some of the features integrated in QEMU to perform instruction tracing.

The term *tracing* refers to a field of computer science concerned with understanding the internal processes of a system. Tracing covers a broad spectrum and is applied in networking, software development and hardware design. Tracing involves obtaining granular information from a running system and then using that information for purposes such as performance analysis, characterization, debugging or forensics. The process of logging inside a software application is related to tracing. Internal events can be logged giving the user an idea about the logical flow and various states the application experiences at runtime. Using specialized tools such as debuggers or performance analyzers it is possible to track the sequence of library calls or system calls an application or operating system performs. Advanced tools can be used to analyze the assembly instructions executed.

All this information can be correlated with spikes in power consumption or high CPU load, to highlight performance bottlenecks. Tracing is mainly concerned with the following aspects:

- Behavioral analysis – for the system or the application
- Performance analysis – identifying where an application or system spends the most time
- Replayability of a process or application – useful for debugging in nondeterministic environments

- Characterization and projections for hardware platforms
- Forensics and support for binary analysis – software security-wise

Instruction tracing is usually performed through hardware instrumentation. Inspecting the flow of assembly instructions executed on the CPU can help model the behavior of a benchmark and detect areas of improvement in both software and hardware.

From a security perspective, instruction tracing can be a valuable asset in the process of malware detection. Malware signatures and detection patterns are often mapped against a well-known set of assembly instructions that are used to designate a piece of code as malicious. In the case of instruction tracing, the traces from an application or system can be used by security software to bypass obfuscation techniques. The problem with obfuscated code would be circumvented by the fact that the malware engine has access to the actual instructions that would be executed on the CPU, in a real-life scenario. Conventional methods like hardware instrumentation or debuggers could be hard to deploy alongside commercial solutions. In order to provide an alternative method of accessing the instruction flow, the current paper works out a proof of concept that uses QEMU to perform binary and system emulation that is able to perform instruction tracing in real time.

## II. RELATED WORK

QEMU is an open source virtualization solution [1]. The project encloses a set of tools and libraries that let the user run virtual machines or standalone executables across platforms. QEMU emulates a set of hardware architectures such as x86, x86-64, arm and aarch64. Cross-platform binaries (guest) are loaded into memory and translated into an intermediary language.

QEMU performs binary translation at runtime using an internal compiler (Tiny Code Generator or TCG) to transform original code into intermediary (TCG) code, and then into host code that will be executed. This provides a fast functional simulation of most instruction set architectures (ISA). Due to this flexibility it has been proposed that QEMU be used for dynamic program instrumentation and tracing [2]. Similar efforts include QEMU CPU Tracer (QCT), a solution that compares itself to other tools such as *oprofile* and *gprof* and emphasizes on instruction count in relation to various application elements (library functions, system calls, etc.) [3].

A TCG plugin was developed in order to implement program instrumentation with QEMU. The idea behind this design was to allow users to interfere in the process of code

R. Velea is with the Department of Computer Science, Technical Military Academy of Bucharest and a software engineer at Bitdefender, Romania (email: radu.velea@mta.ro, rvelea@bitdefender.com).

M. Togan is a professor with the Department of Computer Science, Technical Military Academy, 39-49 George Coşbuc Ave., Sector 5, 050141, Bucharest, Romania (e-mail: mihai.togan@mta.ro)

translation between emulated and TCG code. The user would have to write his own translation and extract desired information at this level. Most important counter available with existing plugins are instruction count, addresses of emulated basic blocks, bytes per instruction profile and Dinero IV-formatted data about load and stores – Dinero IV is a cache simulator for memory reference traces. TCG plugin is not supported in QEMU by default and cannot efficiently manipulate guest instructions using their TCG translation (one guest instruction is likely composed of several TCG opcodes) [4].

For characterization purposes, QEMU has been used to design frameworks that allow the analysis of benchmarks and their specific workloads. This step is a key factor in the process of hardware design and can provide valuable insights on the future performance of the designed models. The data extracted in this manner can provide statistics to support the addition or removal of various components from the system [5]. QEMU can also be used to aid the simulation process when working with emergent technologies (for example: new hardware or new operating system) [6].

### III. ARCHITECTURE

The intermediary representation is optimized, compiled and then translated into native instructions that get executed on the real CPU. QEMU can use other virtualization support such as KVM<sup>2</sup> or Intel @VT (Virtualization Technologies). QEMU is able to emulate most modern operating systems: Linux, Windows, Android, and Chrome OS. Amongst other things this makes QEMU a valuable tool used for developing cross-platform applications, kernel and network programming, device emulation and integration with hypervisors like Xen.

To enable instruction tracing, the goal is to alter QEMU's code base to output a stream of instructions. The integrated runtime compiler (TCG) is, normally, responsible with the binary translation between guest code to host code. During the translation process the contents of the code blocks and their intermediate representation can be turned on by activating debug options. The emulated assembly code is organized into chained translation blocks (TB).

#### Part of the QEMU translation block.

```
struct TranslationBlock {
    target_ulong pc;
    target_ulong cs_base;
    uint64_t flags;
    uint16_t size;
    uint16_t cflags;
    void *tc_ptr; // pointer to translated code
    struct TranslationBlock *phys_hash_next;
    struct TranslationBlock *page_next[2];
    tb_page_addr_t page_addr[2];
    uint16_t tb_next_offset[2]; // offset of
original jump target
    uintptr_t tb_next[2]; // address of jump
generated code
    ...
    struct TranslationBlock *jmp_next[2];
    struct TranslationBlock *jmp_first;
    uint32_t icount;
};
```

Each block represents a zone from the executables text segment and is expected to be executed more than once. QEMU performs some optimizations in the form of caching and hashing code blocks (for better lookup). QEMU has an internal structure mimicking the characteristics of the emulated CPU. This structure is updated after the execution of every instruction or block and simulates asynchronous behavior like interrupt handling.

#### Part of the internal structure used by QEMU to simulate an ARM/AARCH64 processor.

```
typedef struct CPUARMState {
    /* Regs for current mode. */
    uint32_t regs[16];
    /* Regs for A64 mode. */
    uint64_t xregs[32];
    uint64_t pc;
    /* Banked registers. */
    uint64_t banked_spsr[8];
    uint32_t banked_r13[8];
    uint32_t banked_r14[8];

    /* These hold r8-r12. */
    uint32_t usr_regs[5];
    uint32_t fiq_regs[5];

    ...

    /* AArch64 exception link regs */
    uint64_t elr_el[4];
    /* AArch64 stack pointers */
    uint64_t sp_el[4];

    ...

    /* Internal CPU feature flags. */
    uint64_t features;

    void *nvic;
    struct arm_boot_info *boot_info;
} CPUARMState;
```

Going forward, the modifications presented in this paper are focused on cross-platform emulation (running ARM code on x86m or vice versa). Cross-platform emulation will push the guest code through the pipelined process described above and allow us to select, at each stage the information we want to save and output. For performance reasons, QEMU optimizes its internal representation of the guest code and streamlines the execution of previously translated blocks. This means that in the ideal case, the control will be passed to the emulated code for the most part of the process. This is good for emulation, but bad for intercepting instructions, because it takes away the real-time element in analyzing the instruction flow (and thus, the opportunity to analyze other elements such as memory contents is lost). The implementation described in this paper alters this default behavior in the detriment of performance and allows the inspection of every element of the emulated environment as execution progresses.

### IV. IMPLEMENTATION

The goal was to change QEMU's code base in order to create a multipurpose solution for instruction tracing that was easy to implement and extend. The following section is dedicated to modifications required for creating a tracing solution.

An optional step is to build the new framework around an existing functionality of QEMU, such as the debug option, which can be activated via command line. During the

<sup>2</sup> [https://www.linux-kvm.org/page/Main\\_Page](https://www.linux-kvm.org/page/Main_Page)

translation step the internal instruction cache was cleared. This modification had the effect of forcing each new block of code to undergo the same process and allowed the introduction of various callbacks and hooks at this stage. Rather than handling translated blocks and performing the reverse work of turning host code back into guest it was decided to let QEMU go through the whole process for every code block. An option that must be enabled is QEMU's singlestep option. This sets the number of maximum instructions per block to one. The change was required to achieve a granular control over the tracing process. The user may not be interested to perform a trace of the whole lifecycle of the targeted application and could use this feature to set limits (instruction start, end, amount of data captured, etc.). Also, malware signatures are not guaranteed to begin at the block start.

After performing the changes listed above, the translated information inside the TB structure was saved, along with its context. The links between the TBs were broken to ensure control returned to QEMU after each instruction (otherwise we ran the risk of getting lost in loops). Besides instruction opcode, we included other data that could help rebuild the CPU and memory state by following the instruction flow:

- control register values
- flags
- program counter
- virtual and physical memory addresses.

At runtime the user would have the option to trace the entire application or enable tracing just for a certain interval. For example on aarch64 platforms, the trigger for starting the tracing process would be a magic value written in one of the general-purpose registers followed by a jump and return instruction.

*trigger\_start* and *trigger\_end* functions for ARM64 tracing.

```
.cpu generic
    .align 2
    .global trigger_start
    .type trigger_start, %function
trigger_start:
    ldr x0, =0xB16B00B5
    b block1
block1:
// We only need the value in x0 for 1
instruction.
    mov x0, 0x0
    ret
    .size trigger_start, .-trigger_start
// Stop tracing and revert to normal QEMU
emulation.
.cpu generic
    .align 2
    .global trigger_end
    .type trigger_end, %function
trigger_end:
    ldr x0, =0xDEADBEEF
    b block2
block2:
    mov x0, 0x0
    ret
    .size trigger_end, .-trigger_end
```

Under normal mode QEMU would decode and execute code blocks. The tracing code would simply check the first

instruction from each block for the magic value and then enter tracing mode (singlestep, cache flush, etc.). This method of doing things had two main advantages: speed and binary identification.

QEMU is optimized for speed and interfering in its internal translation process as described, incurred some performance penalties. This trick would ensure that code not worth capturing would be executed as fast as possible and that after the tracing process, the system would revert to its default behavior - enabling an endless cycle of trace and continue.

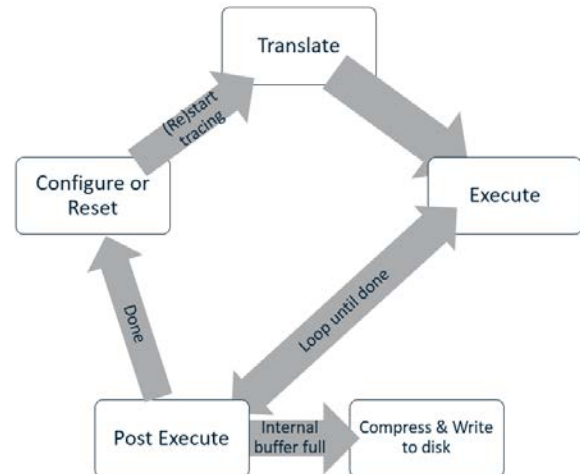


Figure 1. Instruction tracing stages with QEMU

Binary identification is important in scenarios where an entire operating system is emulated, rather than a single binary. The user would have a way of isolating instructions belonging to a single process from the other stuff executed on the virtual CPU. Other techniques to achieve this were tried such as disabling preemption and interrupts for critical code paths, but they were not successful.

The instruction flow was saved into a buffer in memory. The size of the buffer would determine how often I/O operations needed to be operated. The memory containing the instructions could also be shared with other applications. The design of the tracing code would simply allocate a new buffer and forward the old one to a dispatch module. The default behavior would use a new thread to asynchronously dump its contents to the disk and free the buffer. This way, the tracing process would be able to continue regardless of what operations were performed on the previously intercepted instructions.

## V. RESULTS

Experiments showed that the amount of data gathered for even simple binaries could be large: hundreds of thousands or millions of instructions). Complex benchmarks like SPEC CPU2006<sup>3</sup> could execute 40 \* 1012 instructions and run for a couple of hours on a real CPU. To handle the large amounts of data it made sense to use compression when storing the traces on the disk. The format of the human readable data was compression-friendly and I was able to achieve compression levels of about 97% using zlib<sup>4</sup>. The speed penalties were about 2x for opcode tracing and could

<sup>3</sup> <https://www.spec.org/cpu2006/>

<sup>4</sup> <https://zlib.net/>

reach up to 10x depending on the amount of information required per instruction. This penalty could nevertheless turn a runtime that lasts hours into an emulation and tracing that lasts days.

The current solution exposes the behavior an application/system would have if ran on actual hardware. Besides instruction count (feature which is present in similar works) the solution also gathers the information listed below.

*Instruction sampling*: file containing the program counters (PC) for executed instructions; it is possible to specify instruction intervals for this feature (ex: sample every 1000<sup>th</sup> instruction)

*Instruction trace (itrace) generation*: file containing detailed information about the whole run or only certain intervals of interest: PC, opcode, processor status register, physical and virtual memory accessed by the instruction (for load/store), instruction type (based on register size: word, extended, etc.), human readable representation of the instruction (for debug purposes).

*Instruction mix report (imix)*: end-of program file containing a breakdown of the assembly instructions used.

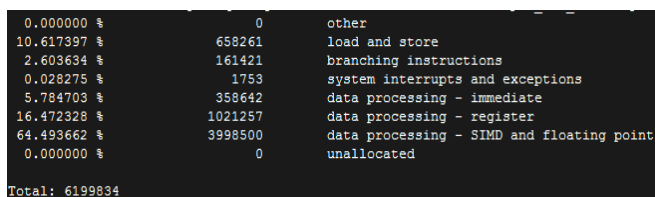
The contents of the trace file can also be used for cross-platform differentiation (comparing code generated by different compilers), instruction level debugging and hardware simulation.

## VI. CASE STUDIES

### A. Benchmarking on AARCH64

The workflow described in this section has been used for performance analysis and characterization of arm and aarch64 traces in support of benchmarking, system analysis; computation of cache hit-miss ratios, path length, memory behavior and application modeling.

Instruction tracing and application profiling can be done using *qemu-linux-user* (single process binary translation) or *qemu-softmmu* (inside a virtual machine). The following images show a runtime report generated for an AARCH64 micro-benchmark and an *itrace* snippet example for SPECCPU2006 run in an Ubuntu AARCH64 virtual machine – with both kernel and user-space code present:



0.000000 %	0	other
10.617397 %	658261	load and store
2.603634 %	161421	branching instructions
0.028275 %	1753	system interrupts and exceptions
5.784703 %	358642	data processing - immediate
16.472328 %	1021257	data processing - register
64.493662 %	3998500	data processing - SIMD and floating point
0.000000 %	0	unallocated
<b>Total: 6199834</b>		

Figure 2. Instruction tracing report for a benchmark

The performance overhead for sampling a benchmark is 2x, while full *itrace* profile can incur a penalty of approximately 7x, depending on the type of instructions executed. These values are considered acceptable when compared to other tracing and instruction analysis tools [7],[8].

### B. Analyzing Packed Executables

Code obfuscation is the reason hackers often resort to packaging (to hide malicious code from detection software). There is an estimate that about 80% of malware samples are packed. Differentiating legitimate packed applications from

malware and recognizing threats in their compressed form is a complex task for any security solution. Using this framework, we were able to unpack ELF and Mach-O binaries that had been compressed with UPX, MPRESS or custom packers, and dump traces of their full instruction flow (including the obfuscated payload). This technique might be useful in the future in dealing with the struggle of detecting ever-evolving forms of packed malware.

## VII. CONCLUSION

QEMU allows applications to run on systems that are not yet available on the market and provides a lot of hardware virtualization options. The current modifications make it possible to gather data for benchmark and system analysis and perform instruction level debugging.

The framework presented in this paper is consistent with the upstream version of QEMU and can be toggled on or off with a combination of command line options and code macro definitions. The framework is useful in the absence of a hardware tracing mechanism or if the user wants to integrate into his application a module that is capable of outputting the instruction flow of an executable. Such an application would, most likely be a performance analyzer or forensics tool.

The key elements that can be achieved with this way of performing instruction tracing is speed (building on the performance of QEMU) and versatility: keeping a simple configuration that allows users to select the information they want displayed at runtime and the execution interval they are interested in.

## REFERENCES

- [1] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, Anaheim, CA, US, 2005.
- [2] Y. Nakamoto, T. Osaki, and I. Abe, "Proposing universal execution trace framework for embedded software using QEMU," in *Future Dependable Distributed Systems, Software Technologies for, IEEE*, Tokyo, Japan, 2009.
- [3] S. Chylek, "QEMU CPU Tracer—an exact profiling tool," *Metody Informatyki Stosowanej*, vol. 5, no. 30, pp. 167-172, 2011.
- [4] C. G. Cedric Vincent, "qemu," 2011. [Online]. Available: <https://github.com/cedric-vincent/qemu/tree/master/tcg/plugins>.
- [5] N. Puzovic, S. A. McKee, R. Eres, A. Zaks, P. Gai, S. Wong, and R. Giorgi, "A multi-pronged approach to benchmark characterization," in *2010 IEEE Int. Conf. on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, Heraklion, Crete, Greece, 2010.
- [6] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Portland, OR, 2013.
- [7] X. Gao, M. Laurenzano, B. Simon, and A. Snavely, "Reducing overheads for acquiring dynamic memory traces," in *Proc. of the IEEE International Workload Characterization Symposium*, Austin, TX, 2005.
- [8] B. Scherer and G. Horvath, "Measurement based software execution tracing in HIL (Hardware In the Loop) tests," in *2014 IEEE/ASME 10th Int. Conf. on Mechatronic and Embedded Systems and Applications (MESA)*, Senigallia, Italy, 2014.