SWIFT: Decoupled System-Wide Information Flow Tracking and its Optimizations

CHIWEI WANG AND SHIUHPYNG WINSTON SHIEH
Taiwan Information Security Center
National Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: cwwangabc@gmail.com; ssp@cs.nctu.edu.tw

Information flow analysis is a widely-adopted technique in software testing and malware analysis. For information flow analysis, a system-level emulator equipped with dynamic information flow tracking capability, DIFT, is needed. However, its effectiveness comes at a price of severe performance degradation due to interleaved system emulation and DIFT analysis. In this paper, a decoupled system-wide information flow tracking scheme, SWIFT, is proposed. Through decoupling system-wide information flow tracking from emulation, SWIFT regains the memory locality and code optimization. The proposed methods are able to aggressively eliminate dependency between the system-level emulator and the analysis thread. Our performance evaluation indicates that, under the same hardware specifications, SWIFT runs 2.74~7.48 times faster than the conventional interleaved design while being benchmarked by PassMark Performance Test 6.0. The performance improvement consequently makes the online analysis feasible in practice.

Keywords: binary translation, emulator optimization, information flow tracking, taint analysis, sandbox

1. INTRODUCTION

Dynamic information flow tracking, DIFT, has been a widely-adopted analysis technique for software testing, malware analysis, and intrusion detection [19-22]. Using emulation [1] or binary instrumentation [2, 3], executed instructions and accesses on memory or peripherals can therefore be monitored and analyzed. Although the effectiveness of DIFT and taint analysis has been demonstrated in much past research [4-7], it comes at the cost of high performance overhead. Although various research works toward software-based DIFT speeding-up were proposed in the last few years, they are limiting their scopes in one or several individual user-level process [11-17].

An ideal approach is to decouple the analysis task from the system emulation so that the two tasks can be performed in parallel. However, in reality the analysis task has heavy data and control-flow dependency on the outcome of the emulation process. Due to register-indirect addressing and virtual address translation, memory addresses are unpredictable and can only be acquired after being generated by emulator’s MMU. Consequently, causal relation and data dependency are introduced. Furthermore, to track information flows correctly, analysis must follow the execution path of the emulator, and therefore control-flow dependency is introduced. Delivering physical address to the analysis thread after each instruction execution is an intuitive but apparently inefficient
approach since the massive data exchange between the two threads could sabotage the benefit of decoupling. Things become even worse when control-flow dependency enters the picture. Since most IA-32 instructions could lead to exceptions such as page faults and privilege violation, control-flow transfer could happen for each instruction. If analysis thread wishes to follow execution path of the emulator, it must be informed of whether an instruction has been successfully executed, which introduces dependency on a per-instruction basis.

This paper presents SWIFT, an efficient system-wide information flow tracking platform. Two novel approaches were proposed to aggressively eliminate both data and control-flow dependency between emulator and analysis thread. For data (accessed physical address) dependency, we alleviate it with the fact that many memory accesses are EBP-based addressing and the value of EBP itself is changed less frequently. We leverage this observation to make the physical addresses of such memory accesses can be calculated by the analysis helper itself. On the other hand, to reduce control-flow dependency, we propose a communication mechanism where the analysis helper is informed of the execution path transfer on a per-block basis. Consequently, fewer message transfers are required. The proposed approach maintains correctness even if exceptions are introduced. Our evaluations indicate that SWIFT operates 2.74~7.48 times faster than conventional interleaved design while being benchmarked by PassMark Performance Test 6.0. Although the performance penalty on CPU-bound tasks is still high (12.74X~35.55X) in comparison with native execution, the overhead is mainly attributed to the inherent emulation nature. Below we highlight the major contributions of SWIFT proposed in this paper.

- A decoupled design is proposed to accelerate system-wide, instruction-grained information flow tracking by executing analysis task and system emulation in parallel.
- Approaches are proposed to aggressively eliminate the needs for the taint tracking thread to communicate with the emulator. The performance is hence accelerated due to less L2-cache confliction.
- Information flows incurred by IA-32 instructions are studied in detail. The result is used to propose a concise encoding format to preserve complete IA-32 instruction information flows. Meanwhile, the conciseness enables efficient processing.

2. RELATED WORK

Designs sped up with additional customized hardware are discussed widely. Hardware architectures with native taint propagation support are proposed in [8, 9]. However, the hardware extension requires corresponding modification on operating systems. Another effective approach is to decouple the analysis process from the program execution itself. There are two ways to realize the idea. One is to extend processors so that instructions executed on one core are recorded and enqueued in a hardware message queue, and processes running on other cores could “peek” logged instructions with a special dequeueing instruction. Processors with this capability are called Log-Based Architecture, LBA [10-13]. The most significant advantage of LBA-based CPUs is that the overhead of instruction tracing is eliminated by hardware. However, the mechanism mentioned above makes them suitable for process-level testing and monitoring, but not for sys-
tem-wide information flow tracking. The other decoupling methodology is to dynamically instrument instructions to execute so that instructions can be logged or monitored [14, 16, 17]. Since the binary instrumentation framework all limit their scope within one process, they cannot be applied in system-wide information flow tracking, either.

Using aggressive dynamic binary instrumentation and optimization, LIFT [15], Minemu [24], libdft [26], ShadowReplica [27] performs DIFT-based security checks on applications. However, there are malicious programs executing all their actions in the kernel space. In fact, Srizbi, which is a Trojan program responsible for 40% of all the spam on the Internet in 2008, hides its file and sends out spams without any user-space components. This kind of malware can only be analyzed at the system level. Demand emulation [18] and PTT [23] accelerate DIFT with virtualization. When accessing a tainted page, it will still fall back to emulation to track information flows. In malware analysis, a large part of the system will be tainted and the frequent switch between virtualization and emulation causes a 5.76X overhead on the emulator. SWIFT focuses on the acceleration on the emulation part, and the overhead imposed on the emulator is decreased to 1.28X–3.16X. PTT also adopted a decoupled design, but SWIFT proposes more optimization techniques, which are indispensable for a decoupled taint tracker to be efficient and correct. One is the per-block-basis delivering and the other one is the elimination of EBP (or ESP) base memory address delivering, and the other is the exception-aware control-flow delivering. Aftersight [25] decouples the execution and the analysis phase by only recording non-deterministic events, which is a different issue than the one SWIFT focuses. Pirate [28] supports DIFT under different architectures through dynamically compiling executables into LLVM intermediate representations (IR). Pirate analyzes the generated IR off-line like Aftersight, and it supports only process-level analysis.

3. BACKGROUND

We introduce certain preliminary knowledge and terminology about dynamic binary translation and QEMU which form the basis of our system. QEMU is system-wide emulator, which is capable of emulating the whole architecture instead of one process. To be specific and concise, we will refer to the emulated architecture in the rest of the paper as the guest and the machine running the emulation as the host. Any hardware devices and mechanisms existing in the architecture of guest machine such as registers, MMU, or peripherals are realized by software. Having the instruction “add eax, 1” executed inside the emulated guest machine, we expect that it is the emulated eax register, not the real one of the host machine, to be increased. Therefore, all instructions must be translated before execution so they can reflect expected behaviors. QEMU adopts dynamic binary translation to perform the translation. In addition, the translation is done “on the fly.” Namely, the binary translator would be invoked when the emulator encounters a code region which had not been translated. The translation is done on a per-block basis. Namely, the process of translating instructions continues until a branch or jump instruction is discovered. All guest instruction sequences (the branch or jump included) translated in the process above forms a basic block, and the binary code generated for actual execution on the host will be the corresponding code block. Code blocks are stored in a hash table for next time use, since the translation is computationally expensive.
Fig. 1. System architecture of SWIFT.

4. SYSTEM DESIGN

Previous emulation-based DIFT work inject desired binary routines directly into code blocks to propagate taint status of registers and physical memory addresses which the instruction accesses. This approach, which adopted in prior works, has been demonstrated to be effective yet inefficient because the injected code usually involve with complicated computation. This encumbers system emulation in both explicit and implicit ways. The injected analysis routines could perform tasks as complicated as the emulation itself. In addition, the alternation between system emulation and analysis makes software optimization much more difficult or even disables coherent hardware acceleration such as cache mechanism.

Instead of being injected directly into basic blocks, analysis routines in SWIFT are executed by another helper thread. Additional code is injected only for delivering information flows and physical memory addresses for the helper thread to accomplish its analysis task. Decoupling the analysis from the system-wide emulation enables SWIFT to shift the analysis workload such as updating the taint map or security check onto a different core.

In Fig. 1 the basic system architecture of SWIFT is shown. On core 1, the system emulation is executed. As any hardware processor operating in a fetch-decode-execute loop, the software emulator also behaves similarly. The dispatcher always tries to search the code block pool with its instruction pointer to locate next block to execute. If the corresponding code block is found, it will be invoked. The emulation starts and then returns to the dispatcher after the code block finishes its task. Recall that all code blocks end with emulation of any jump or branch instructions. The dispatcher selects next code block to execute according to current target CPU status, and the process above repeats itself. However, if nothing is found in the search, the binary translator will be invoked to translate the basic block.

While generating code blocks for emulation, the binary translator of SWIFT in the meantime extracts information flow semantics of instructions. With the DIFT model we proposed later in section 4.1, extracted semantics are converted to the so-called IF-codes for delivery. The code blocks generated by the binary translator will deliver IF-codes to
the helper thread through inter-thread communication. The delivery is always done immediately after each instruction emulated. Through inter-thread communication the helper thread running on core 2 can therefore perform corresponding taint analysis or security checks.

At first glance, the improvement of this approach seems to have a 2X limitation. However, our close observation shows some interesting phenomenon when emulation and taint propagation are performed alternatively by threads. Consequently, we can take advantage of them to achieve performance over the seemingly 2X limitation. First of all, since an emulator simulates operations of a real machine and executes real programs on top of it, the caching behavior of the guest machine will be reflected on the host machine. Meanwhile, since taint analysis must update taint records of each byte in memory, the cache usage will be duplicated. The phenomenon will exhaust capacity of the hardware cache on the host machine rapidly. Secondly, optimization performed by dynamic binary translator becomes ineffective if complicated analysis routines are injected between translated instructions. For example, the complicated taint propagation may clobber all registers of the host CPU and disable register-allocation strategy of the binary translator. Due to the reasons above, executing the two tasks on separated cores not only benefit from parallelization but also recovers attenuated optimizations. Hence, a speed up by factor greater than 2 could be expected.

4.1 Encoding Information Flows of Instructions

Simply sending the helper thread raw instructions executed by the emulator, like LBA-based architecture does, is a viable option. This preserves most complete information for analysis, but the helper thread has to decode these raw instructions to perform corresponding analysis tasks. Another option is passing only data necessary for information flow tracking. This approach loses part of the information due to discarded instructions. However, it preserves the performance because the tedious decoding process can be removed from the helper thread. In SWIFT, we chose latter approach to meet the performance requirement.

No matter which instruction is executed and what values are involved in the calculation, two most essential elements are required to show the corresponding information flow: its source and destination. Since we are tracking system-wide dynamic information flow at byte-granularity, source and destination will always refer to a single byte of registers, memory, hard disks and network interface buffers.

In addition to the source and destination, another essential factor should be included in modeling information flows. Consider a data transfer instruction such as “mov”. The two operands obviously serve as the source and the destination in the information flow, and the original information in the destination operand is overwritten. For a binary arithmetic operation such as addition or bitwise exclusive-or, however, one of the operands will be used as both the input and the outcome variables. For example, the ebx register in the instruction “xor ebx, eax” is used as input in the exclusive-or operation and also storage for the output value. Instead of being overwritten, data of ebx is combined with the information flowing out of eax. Therefore, the flowing effect should be indicated when modeling an instruction. An effect of an information flow could be overwriting or appending. For concise expression, let’s denote overwriting information
flows and appending ones from \( B \) to \( A \) as \( A \leftarrow B \) and \( A \leftarrow (B) \) respectively.

However, complicated information flows could be generated by a single instruction. If we always encode the information flow at byte-granularity, the complication could lead to lengthy expressions. A few more observations can be levered to avoid this. Although our system tracks information flows at byte-granularity, IA-32 instructions always perform operations on specific widths such as byte, word (2 bytes), double-word (4 bytes), or quad-word (8 bytes). Specifying the width of operands directly in the encoding of information flows can therefore bring us a much more succinct processing.

In Table 1 are listed typical information flows from \( B \) to \( A \) when the operand width and information flow effect of an instruction are considered simultaneously. In the formula, \( A \) and \( B \) are both variables of \( n \)-byte, and the lowest significant byte of \( A \) is denoted as \( A[0] \). For each category, a notation is also defined for succinct expression in later context. Note that these flows are successively ordered. We will refer to information flows falling into one of these categories as multi-byte information flows.

In Fig. 2, each category is demonstrated with a representative instruction. An edge in the figure represents the information flow from the sourcing byte to the destined byte in the calculation. One way to tell whether a variable \( A \) influences the other variable \( B \) is checking existence of a directed path from \( A \) to \( B \). In Fig. 2 (a), since a \texttt{mov} instruction copies data byte-wisely from the source to the destination as demonstrated, corresponding information flow will be overwriting ones and mutually independent. For (b), the information flows are appending because the \texttt{xor} instruction performs exclusive-or on the two sources yet uses \texttt{eax} as the output operand. Category (c) is used to depict the situation that all bytes are influenced by those with lower or equal significance. A good
example would be arithmetic addition or subtraction, in which higher bytes are influenced by all lower bytes due to the carry. The complicated information flows caused by multiplication are shown in Fig. 2 (d). Note that the entire output would be 8 bytes due to the multiplication on the two double-words. As it turns out, category (c) would suffice to describe the net information flowing effect on the four lower bytes (stored back in \texttt{eax}). However, we could observe that all four higher bytes (stored in \texttt{edx}) are influenced by all bytes of the two input operands. Therefore, we created category (d) to indicate such cases. It turns out that information flow of most IA-32 instructions can be totally described with these four categories and operand widths.

With all observations listed above, we are now ready to encode any information flow caused by IA-32 instructions with its destination, source, width, and effect. In Fig. 3 are shown the two encoding formats of information flows. In Fig. 3 (a), the format used to describe multi-byte ones is depicted. Fields \texttt{D} and \texttt{S} specify types of source and destination. \texttt{WTH} and \texttt{EFF} are used to specify the operand width \(n\) and the category in effect mentioned in Table 1. For the information flows that cannot be properly described by multi-byte rules, we leave them for the format depicted in Fig. 3 (b), which lacks \texttt{WTH} but provides two fields \texttt{D_OFF} and \texttt{S_OFF} to specify offsets of referred bytes.

![Fig. 3. Formats of IF-codes; (a) Format for describing multi-byte information flows; (b) Format for describing information flow from a single byte to another.](image)

<table>
<thead>
<tr>
<th>Category</th>
<th>Information Flow</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte-wise overwriting</td>
<td>(A[0] \rightarrow B[0])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A[1] \rightarrow B[1])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A[n] \rightarrow B[n])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td>Byte-wise appending</td>
<td>(A[0] \leftarrow B[0])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A[1] \leftarrow B[1])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A[n] \leftarrow B[n])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td>Incrementally mixed</td>
<td>(A[1] \leftarrow A[0], A[1] \leftarrow B[1])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A[n] \leftarrow A[n-1], A[n] \leftarrow B[n])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td>All mixed-up</td>
<td>(T \leftarrow \tilde{A}[0], \tilde{A}[1], \ldots, \tilde{A}[n])</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(T \leftarrow \tilde{A}[0], \tilde{A}[1], \ldots, \tilde{A}[n], \tilde{T})</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A[0] \leftarrow T, A[1] \leftarrow \tilde{T}, \ldots, A[n] \leftarrow \tilde{T})</td>
<td>(A \leftarrow B)</td>
</tr>
<tr>
<td></td>
<td>(A \leftarrow B)</td>
<td>(A \leftarrow B)</td>
</tr>
</tbody>
</table>

Table 1. Typical information flows on multibyte operand.
For registers, DST_REG and SRC_REG store their identities. The two fields cover general-purpose registers, segment registers, program counter (EIP), floating-point registers, control registers, and debug register. MMX registers, XMM registers, MTRRs, MSRs, and any other registers are viewed as a null source and sink for information flow, that is, information flowing out of them is always labeled as clean, and information flowing into them is not tracked. In addition, we ignore instructions of MMX, SSE, SSE2, and SSE3 extension. The simplification is simply an implementation issue. Tracking information flow for the enormous instructions and registers introduced by these extensions requires huge implementation efforts but brings less effectiveness since they are seldom used. Note that we leave 8 bits for the DST_REG and SRC_REG fields so that DIFT for these instructions and registers can always be implemented incrementally without modifying the current IF-code design. Note that these encoded information flows are simply 32-bit integers, and they could be generated when translating an instruction. We refer to these codes as IF-codes.

Although field DST_REG and SRC_REG seemed useless for memory operands, they keep important information for the next special case.

4.2 Delivering IF-codes and Memory Addresses

In SWIFT, binary code to assist the helper thread, instead of analysis itself, is injected. The major task of injected binary is delivering IF-codes of each emulated instruction to the helper thread, so the helper could analyze information flows of the original program and hence the overall performance could be improved since the analysis could be processed in parallel by another core.

However, since the delivering will be done for each information flow incurred by all emulated instructions, the mechanism must be efficient enough or it could impose a new overhead on the emulation. To be light-weight, the one-way communication is achieved through a circular queue residing on a shared memory region as depicted in Fig. 4. To utilize the queue more efficiently, entries in the circular queue are actually pointers to chunks of continuous space. In code blocks, enqueueing routines are injected so that IF-codes are enqueued after each instruction is emulated. Next chunk will be asked for once the current chunk is full. By selecting 4 KB as the chunk size and aligning all chunks on 4 KB boundary, in SWIFT the enqueueing is done with only 6 instructions.

So far we have introduced how an information flow is extracted and encoded. Nevertheless, the system-wide information flow tracking cannot be done unless addresses of
memory variables are also tracked. Recall that a memory access can be indirect. In an indirect memory access, the memory address depends on the value in a certain register, and it is hence impossible to predict these addresses in advance. Therefore, watching addresses of memory operands is postponed until runtime.

Prior DIFT works based on binary instrumentation framework such as PIN or StarDBT watch virtual addresses of memory operands. This approach seems intuitive and may even be the only feasible method since the binary instrumentation tool can only monitor a user-level process. Instead, SWIFT watches physical addresses of memory operands. Since QEMU provides software MMU for system-wide emulation, watching physical addresses of accessed memory is nothing harder than watching virtual addresses. We modify QEMU so that any physical memory addresses generated by the software MMU emulation will be recorded in global variables `last_phyaddr_written` and `last_phyaddr_read`, depending on whether operation is writing or not. The address delivering is implemented within 7 instructions.

5. OPTIMIZATIONS

Although approaches proposed so far successfully decouple the execution of system emulation and analysis task, the enqueuing operations still incur performance downgrade in three ways. First, extra instructions injected for enqueuing inherently introduce latency in the emulation process and the helper thread. Secondly, the massive memory accesses due to the enqueuing can consume hardware cache or causes cache misses generated by the producer-consumer communication. Thirdly, the more data enqueued, the faster the circular queue will be saturated. Therefore, reducing invocations on the enqueuing routine can accelerate the emulation operation in multiple ways. Two optimizations were proposed to aggressively remove invocations on enqueue in our design while preserving the correctness of information flow tracking.

5.1 OPT1: Delayed-Delivering on a Per-Block Basis

Avoiding enqueuing IF-codes frequently can bring us substantial performance improvement because they form the major part of messages delivered to the helper thread. One possible optimization toward this is to deliver IF-codes on a per-block basis. While translating a basic block, the translator could group all IF-codes generated into a special entity, called the IF-code block. In the IF-code block, IF-codes are stored in the order as corresponding instructions are arranged. Instead of delivering IF-codes between every emulated instruction, we only inject code in the beginning of the code block to inform the helper thread which code block is being emulated so the correct IF-code block can be traced. In doing so only one enqueuing operation is needed to deliver all IF-codes for a whole code block.

However, above optimization does not reflect correct information flows always. Consider the instruction “`mov eax, [ebp+8]`”. It accesses a memory address indirectly, and it can potentially lead to a page fault exception as long as EBP register contains an inappropriate value. To emulate such behaviors correctly, a code block must exit itself when things go wrong. Therefore, only first two IF-codes would be effective in such a
case because rest instructions had not been emulated. Note that since an exception is unpredictable, the amount of effective IF-codes may vary between each execution of the code block.

To amend the problem, a counter should be added to accumulate instructions emulated for each code block. Between each emulated instruction, we inject code to increase the counter so it reflects the amount of emulated instruction. The accumulation will continue until the code block exits. The value of the counter will be delivered to the helper thread in the very beginning of next code block.

OPT1 eliminates enqueueing operations effectively with the observation that once a basic block is generated, its IF-codes would be fixed also. However, large amount of enqueueing operation are still needed to pass physical memory addresses. As stated earlier, these physical memory addresses can only be watched when the code block is being executed. To reduce the overhead incurred by memory address delivery, another optimization is proposed below.

5.2 OPT2: EBP or ESP-based Indirect Accessing

The foundation of the second optimization relies on several phenomena observed on the frame pointer register and stack pointer register, namely EBP and ESP, in IA-32 architecture. Due to the conventional design of common compilers, this register stores the beginning address of an activation record and top of the stack in EBP and ESP respectively. In addition, their values usually change only when a new activation record is created. Referencing memory indirectly with these two registers is a common way to access local variables and function arguments. The third emulated instruction “mov eax, [ebp+8]” listed in Fig. 5 is a representative instance. More, in more than 90 percent of EBP or ESP-based memory accesses, their offsets distribute over the range from −1024 to +512 bytes. The clustering phenomenon is understandable since local variables and arguments usually locate near the beginning of the frame and occupy little space.

In Fig. 5 we depict the three scenarios could occur when adding offsets within the range above to EBP. It is easy to see that two pages at most could be cross by the ranging offset. Note that although the two pages are continuous in virtual address space, their physical locations may be not due to the virtual memory mapping. For the two physical pages, we refer to the page pointed by EBP as the base page and the other one as the siding page. The physical address generated by EBP-based addressing with such offsets could be acquired using the algorithm listed in Fig. 6. Note that the optimization above also applies on ESP-based indirect addressing.

![Fig. 5. Scenarios of EBP-based addressing with offsets within range −1024→+512.](image-url)
Using the algorithm the helper thread could calculate the physical address generated by EBP (or ESP)-based addressing instructions as long as the instruction has a proper offset. Let’s consider the four inputs needed for the algorithm. Since the offset is encoded in the instruction, it could be determined in translation phase and stored as part of the IF-code. This observation saves us from enqueueing memory address for each EBP (or ESP)-based memory access.

6. EVALUATION

Benchmark tools and workloads in real world are used to evaluate how SWIFT imposes performance downgrade on the original QEMU. The design and optimizations proposed in this study focus on alleviating the additional overhead on emulators imposed by DIFT analysis. The experiments are performed in following configurations.

(a) Native QEMU
(b) SWIFT (decoupled design)
(c) SWIFT w/ only OPT1 enabled
(d) SWIFT w/ both OPT1 and OPT2
(e) QEMU with inlined taint propagation
(f) TEMU (Based on QEMU Ver. 0.9.1)

To evaluate performance gain provided by each approach proposed in this paper, benchmarks are performed under different optimization configurations. To set up a baseline for our evaluations, a native version of QEMU, which our system base on, is tested in configuration (a). Note that neither KQEMU nor KVM was activated because we want to benchmark the performance of the pure emulation. In (b), solely the decoupling mechanism is enabled so that its performance advantage could be measured. Configuration (c) and (d) operate with the decoupled design as well as (b), yet OPT1 and OPT2 are enabled respectively. To understand how much performance gain could be achieved with the proposed schemes, we also set up a configuration (e), which inlines taint propagation routines of SWIFT directly in code blocks generated by original QEMU. We also included TEMU, a well-known system-wide taint analysis system, as configuration (f) in

Input
- offset : offset encoded in EBP-based accessing instruction.
- ebp : current value of the register.
- paddr_base : physical address of base page.
- paddr_siding : physical address of siding page.

Output
- Physical address accessed by this instruction

1. PAGEMASK = 0xFFFFF000
2. if (ebp & PAGEMASK) != (ebp+offset) & PAGEMASK
3. return paddr_siding + ((ebp+offset) & ~PAGEMASK)
4. else
5. return paddr_base + ((ebp+offset) & ~PAGEMASK)

Fig. 6. Method to calculate physical address accessed by EBP-based memory access.
this benchmark evaluation. A little more explanation is needed to elaborate the goal of this evaluation. For configuration (b), (c), (d), and (e) we label all data in guest memory or received from network as tainted. Although this leads to considerable memory usage since the companied shadow memory becomes as large as the allocated RAM size of the guest machine, it is necessary for measuring the performance gain under the worst case. Moreover, it is difficult to fairly compare TEMU with our scheme directly. TEMU is designed with extremely high flexibility, and it thus contains large taint record for each byte and many callbacks for additional plug-ins. These features inherently incur severe overhead on performance of TEMU. However, it is also difficult to port its code into SWIFT for direct comparison because TEMU is based on an older version of QEMU. Due to reasons above, we only activate the taint propagation of TEMU and remove any other plug-ins in configuration (f). In addition, no taint data is introduced in configuration (f) among all experiments.

All the evaluations are performed on an IBM System x3650, with one unit of Intel Xeon E5430 2.66 GHz Quad-Core Processor, 8GB DDR2 RAM, and a 150GB SATA-II hard disk installed. In each configuration we load one identical virtual machine snapshot into the emulator to ascertain fairness. The virtual machine is allocated with 512 MB RAM and a 10GB hard disk. Windows XP with service pack 3 is installed and booted in the snapshot. In addition, we allocate 512 MB for the IF-code delivering circular queue.

To perform information flow tracking SWIFT consumes more memory than the original emulator does. First of all, an extra 512 MB space was allocated to construct the circular queue for IF-code delivering. In addition, we augment each byte in the guest memory with an extra shadow byte to preserve its taint status. Since in our evaluation we label every byte in memory as tainted, the shadow memory occupies the same size as the physical memory size of the guest. Also, a shared 128MB memory region is pre-allocated to store IF-code blocks generated in the translation phase. We also force the emulator to flush all the code blocks when this region is full. Therefore, all these extra memory usage can be statically calculated to be 512+512+128 = 1152MB.

The first result of performance evaluation is acquired with PassMark Performance Test 6.0, which is an off-the-shelf commercial test suites adopted extensively in CPU and system benchmarking. Benchmark items could be categorized into CPU-intense jobs and memory-intensive ones. To present overhead imposed by each configuration more clearly, benchmark scores of configuration (b), (c), (d), (e), and (f) are divided by the score of baseline configuration (a). As indicated in Fig. 7, although the design with decoupled DIFT still imposes high overhead (1.43X-5.00X) among all test items, it already outperforms configuration (e) significantly. When OPT1 and OPT2 are enabled, the overall performance downgrade can be reduced to 1.28X-3.16X, which are 2.74X-7.48X times faster than the interleaved design (f). The result demonstrates effectiveness of optimizations proposed in this paper. In addition, close scores between (e) and (f) give us faith on the representativeness of configuration (f).

Next, we benchmark same configurations with common workload such as file transferring or source code compiling to further investigate the analysis overhead in real applications. Details of these workloads are explained below. All measurements are repeated certain times and average values are calculated. The number of repetition of each item is listed after the name of the workload.
System Booting (50) – We measure the time elapsed from powering on the emulator until Windows loads Graphical Identification and Authentication, GINA, which brings up the Windows Security dialog for users to log on.

Web Browsing (50) – The experiment is carried out by measuring the time needed for sequentially browsing top 50 websites, which are ranked by Alexa Internet, an authoritative Internet information provider. The sequential browsing mechanism is implemented with a Firefox plugin, which automatically visits next website once it receives an event of page loading complete.

Communication over SCP (20) – In this benchmark a large file is downloaded into the emulator through Secure Copy, a file transfer mechanism based on SSH protocol to provide confidentiality and authentication. The file consists of 120 MB random binary sequence and resides on a host locating in the same 100BASE-TX local area network.

Kernel Compiling (5) – Compiling the kernel of an operating system is a common workload used to benchmark the overall performance of a computer. This is carried out by building the Windows Research Kernel in the emulated machine.

Among all experiments shown in Fig. 8, configurations with analysis decoupled showed enormous performance advantage over those with inline analysis routines. In addition, effectiveness of the two optimizations are also demonstrated in configuration (c) and (d). The result shows that our system remains 50%~85% performance of a native emulator when both optimizations are enabled. Moreover, compared with configuration (e), a greater than 2X performance advantage is given by configuration (e) in nearly all commercial benchmark items and workload tests. The observation justifies that the investment of utilizing an addition CPU core is paid off.

On the other hand, the performance may be also affected by the size of the circular queue used for IF-code delivering. The overhead can be contributed by a saturated IF-code delivering queue. In current implementation, the analysis thread has not been optimized. Therefore, the speed of the analysis thread, which consumes IF-codes, cannot keep up with the emulator. Once the queue is saturated, the emulator must wait for the analysis thread.
To verify this conjecture, we set up an experiment with two additional profilers in SWIFT. One is installed in the emulation thread to measure how many code blocks are emulated per second. The other is installed in the DIFT thread to measure the distance between the enqueuing pointer and the de-queuing pointer periodically. The distance indicates the usage of the message-delivering queue. By plotting the two kinds of measurements versus the system clock time on the same graph, we can study the correlation between the emulation speed and the queue usage. It is worth noting that in our experiment we allocated a 512 MB queue. This assures that the saturation (if any) can be observed even if an impractically large queue is given. The result is shown in Fig. 9.

Fig. 9 contains two plots, which are acquired in the process of running Passmask CPU integer operation and encryption benchmark, respectively. In each plot, the emulation speed versus time graph and queue usage versus time graph are plotted. The patterns circled by dashed boxes indicate that the emulation performance drop when the queue becomes saturated. Namely, the emulation thread will still be cumbered by the DIFT thread eventually. The data indicates an interesting fact worth noting. In both plots, the drop is not negligible. Namely, the current taint tracking implementation in SWIFT is not able to keep up with the emulation part, and it indeed eventually incur significant overhead on long CPU-bound tasks.
7. CONCLUSION

In this paper, a decoupled design of system-wide information flow tracking, SWIFT, is presented to shift the heavy overhead imposed by the analysis process onto another processor core. Unlike previous DIFT-capable system emulators injecting analysis routines directly into generated code blocks, our design extracts information flows only at translation phase, and therefore analysis to be performed on extracted information flows can be carried out by different threads. To further improve the analysis performance, two optimization techniques are proposed herein to eliminate unnecessary message exchanges between the emulator and the helper executing analysis routines. Each of the proposed techniques is shown effective in our performance evaluation. Compared with conventional interleaved design, our method operates 1.82~3.22 times faster on common workloads. SWIFT runs 2.74~7.48 times faster than the interleaved design does in PassMark Performance Test 6.0.

REFERENCES


Chi-Wei Wang (王繼偉) is currently a Ph.D. candidate in the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. He has been very active in the malicious software analysis community, and has received many awards. Recently, he led his team and achieved 1st place in the Wargame Contest held by Hacks in Taiwan Conference 2010 and 2011. He also received the 1st place in the Microsoft Cross-Strait Innovation Contest in 2007. His research interests include network security, software security, and operating systems.

Shiuhyung Winston Shieh (謝緯平) received the M.S. and Ph.D. degrees in Electrical and Computer Engineering from the University of Maryland, College Park, respectively. He is a Professor of the Department of Computer Science, National Chiao Tung University (NCTU), and the Director of Taiwan Information Security Center at NCTU. Dr. Shieh currently serves as the Chair of IEEE Reliability Society Taipei and Tainan Chapter, and an ACM SIGSAC Awards Committee member. He is also the Editor-in-Chief of IEEE Reliability Society Newsletter, an Associate Editor of IEEE Transactions on Reliability, IEEE Transactions on Dependable and Secure Computing, former editor of ACM Transactions on Information and System Security, Journal of Computer Security, Journal of Information Science and Engineering, Journal of Computers, and guest editor of IEEE Internet Computing, respectively. He was on the organizing committees of numerous conferences, such as Steering Committee Chair and Program Chair of ACM Symposium on Information, Computer and Communications Security. He received the ACM Distinguished Scientist Award among the 41 recipients worldwide in 2010. He also received ACM Service Award for his contribution to ACM, and Distinguished Information Technology Award (presented by Vice President of Taiwan). He is also an IEEE Fellow. His research interest includes reliability and security mechanisms, network and system security.