

Design and Implementation of a Fast Dynamic Packet Filter

Zhenyu Wu, Mengjun Xie, *Member, IEEE*, and Haining Wang, *Senior Member, IEEE*

Abstract—This paper presents *Swift*, a packet filter for high-performance packet capture on commercial off-the-shelf hardware. The key features of the Swift include: 1) extremely low filter update latency for dynamic packet filtering, and 2) gigabits-per-second high-speed packet processing. Based on complex instruction set computer (CISC) instruction set architecture (ISA), Swift achieves the former with an instruction set design that avoids the need for compilation and security checking, and the latter by mainly utilizing single instruction, multiple data (SIMD). We implement Swift in the Linux 2.6 kernel for both i386 and x86_64 architectures and extensively evaluate its dynamic and static filtering performance on multiple machines with different hardware setups. We compare Swift to BPF (the BSD packet filter)—the *de facto* standard for packet filtering in modern operating systems—and hand-coded optimized C filters that are used for demonstrating possible performance gains. For dynamic filtering tasks, Swift is at least three orders of magnitude faster than BPF in terms of filter update latency. For static filtering tasks, Swift outperforms BPF up to three times in terms of packet processing speed and achieves much closer performance to the optimized C filters. We also show that Swift can harness the processing power of hardware SIMD instructions by virtue of its SIMD-capable instruction set.

Index Terms—Linux kernel networking, packet filter.

I. INTRODUCTION

A PACKET filter is an operating system kernel facility that classifies network packets according to criteria given by user applications and conveys the accepted packets from a network interface directly to the designated applications without traversing kernel networking stack. Since the birth of the seminal BSD packet filter (BPF) [1], packet filters have become essential to a variety of network services ranging from traffic monitoring [2] [3] to network engineering [4] and intrusion detection [5]. In recent years, with escalating protocol complexity and dramatically increasing network speed, packet filtering has been facing intensified challenges posed by dynamic filtering tasks and stringent filtering requirements. However, existing packet filters have not yet fully addressed these challenges in an efficient and secure manner.

Manuscript received April 21, 2010; revised November 08, 2010; accepted January 18, 2011; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Crowley.

Z. Wu and H. Wang are with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23187 USA (e-mail: adamwu@cs.wm.edu; hnw@cs.wm.edu).

M. Xie is with the Department of Computer Science, University of Arkansas at Little Rock, Little Rock, AR 72204 USA (e-mail: mxie@ualr.edu).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TNET.2011.2111381

Dynamic filtering tasks refer to online packet filtering procedures in which filtering criteria frequently change over time. Typically, when a filtering task cannot fully specify its criteria *a priori* and the unknown part can only be determined at run-time, the filtering criteria have to be updated throughout the filtering process. For example, many protocols, such as FTP, Real Time Streaming Protocol (RTSP), and Session Initiation Protocol (SIP), establish connections with dynamically negotiated port numbers. Capturing the network traffic that uses such protocols requires dynamic filtering. Even with predetermined filter criteria, it is often necessary to use dynamic filtering. For instance, a network intrusion detection system (NIDS) needs to perform expensive deep traffic analyses on suspicious network flows. However, applying such a costly procedure to every packet in high-volume traffic would severely overload the system. Instead, an NIDS could first apply simple filtering criteria, such as only monitoring traffic to and from a honeypot or a darknet. When suspicious activities are detected, the NIDS can then update its filtering criteria to capture all the traffic related to the activities for deep inspection.

As the *de facto* packet filter on modern UNIX variants, BPF has shown insufficiency in handling both static and dynamic filtering tasks, particularly the latter [6]–[8]. A filter update in BPF must undergo three “preprocessing” phases: compilation, user-kernel copying, and security checking. In the compilation phase, the filtering criteria specified by the human-oriented *pcap* filter language [9] are translated and optimized into the machine-oriented BPF filter program. In the user-kernel copying phase, the compiled filter program is copied into the kernel. Finally, in the security checking phase, the kernel-resident BPF instruction interpreter scrutinizes the filter program copy to ensure that the program does not contain potentially dangerous operations such as backward branches. Consequently, an update to a BPF filter may induce prolonged latency, which ranges from milliseconds up to seconds depending on criterion complexity and system workload. In high-speed networks, hundreds or even thousands of packets of interest might be missed by BPF during each filter update, effectively leaving a “window of blindness.” Frequent filter updates, often required by a dynamic filtering task, exacerbate the degree of blindness. A “window of blindness” coinciding with the initialization of a new session can cause serious problems on certain applications such as NIDS since the beginning of a network connection normally is of particular interest for security analysis [10].

Recent packet filters such as xPF [8] and FFPF [6] move more packet processing capabilities from user space into the kernel, which reduces context switches and improves overall performance. However, neither filter directly addresses the issue of slow filter update. Because xPF uses the BPF-based filtering

engine, it offers no improvement on filter update latency. FFPF attempts to avoid filter update by introducing kernel space library functions, called external functions, which are precompiled binaries for specific functionalities such as parsing network protocols and updating filtering states. The use of external functions speeds up filtering and eases extensibility, but it also increases filter programming and updating complexity. External functions have full privileges to access the kernel, and thus they must be subjected to careful examination for potential security bugs at each loading, making them a poor fit for frequent updates.

In this paper, we propose *Swift*, a packet filter that takes an alternative approach to achieving high performance, especially for dynamic filtering tasks. Like BPF, Swift is based on a fixed set of instructions executed by an in-kernel interpreter. Unlike BPF, Swift is designed to optimize filtering performance with powerful instructions and a simplified computational model, and more importantly, to significantly reduce filter update latency. The Swift instruction set accomplishes common filtering tasks with a small number of fixed-size “complex” instruction set computer (CISC)-like instructions. In addition, the Swift instruction set architecture (ISA) enables support for more powerful optimization techniques, namely single instruction, multiple data (SIMD) extension, hierarchical execution optimization, and platform-specific hardware acceleration. Combining the powerful instructions with the simplified computational model, Swift removes filter compilation and security checking in filter update, and thereby significantly improves dynamic filtering performance in terms of filter update latency.

We implement Swift in the Linux 2.6 kernel for both i386 and x86_64 architectures. The kernel implementation of Swift is fully compatible and can coexist with Linux Socket Filter (LSF), “a BPF clone” in Linux. The Swift user-space libraries provide a BPF-friendly application programming interface (API) with “pcap-like” filter syntax for backward compatibility and an object-oriented API that simplifies filter coding. To validate the efficacy of Swift, we conduct extensive experiments on multiple machines with different hardware setups. We compare the performance of Swift to that of LSF and optimized C filters. The C filters are used to demonstrate the possible performance gains obtainable by optimized binary code. For dynamic filtering tasks, Swift achieves at least three orders of magnitude lower filter update latency than LSF and reduces the number of missing packets per connection by about two orders of magnitude in comparison to LSF. For static filtering tasks with simple filtering criteria, Swift runs as fast as LSF. However, with complex filtering criteria, Swift outperforms LSF up to three times in terms of packet processing speed and performs much closer to the optimized C filters than LSF.

We further augment Swift by leveraging the parallel processing power of hardware SIMD instructions. We first demonstrate Swift’s ability to transparently utilize the Streaming SIMD Extensions (SSE) instruction set—SIMD extensions that are now universally supported by modern x86 processors. Then, we analyze and measure how the SSE instructions can be used to accelerate filter execution speed. Finally, we present a comparison of filter execution times between our SSE enhanced Swift engine and the original Swift engine. Our results show

that, exploiting the hardware SIMD instructions, Swift can achieve significant speedups for large and complex filters.

The remainder of this paper is structured as follows. Section II surveys related work on packet filters. Section III details the design of Swift. Section IV describes the implementation of Swift. Section V evaluates the performance of Swift. Section VI discusses the performance boost of Swift from hardware SIMD support. Section VII concludes the paper.

II. RELATED WORK

Packet filters play a major role in network monitoring and measurement due to its simplicity, universal installation, high cost-effectiveness, and rich applications. The CMU/Stanford Packet Filter (CSPF) [11], a kernel-resident network packet demultiplexer, introduces the concept of packet filter. CSPF provides a fast path, instead of the normal layered/stacked path, for network packets to reach their destined user-space applications. Thus, the filtering in CSPF refers to *delayed demultiplexing* [12]. CSPF is designed to facilitate the implementation of network protocols such as TCP/IP at user space. The characteristic of kernel-resident and protocol-independent packet filtering in CSPF design has been widely adopted by successive packet filters.

BPF [1] aims to support high-speed network monitoring applications such as *tcpdump* [9]. Users inform the in-kernel filtering machine of their interests through a predicate-based filtering language [13], and then receive from BPF the packets that conform to filtering criteria. To achieve high performance, BPF introduces in-place packet filtering to reduce unnecessary cross-domain copies, a register-based filter machine to fix the mismatch between the filter and its underlying architecture, and a control flow graph (CFG) model to avoid redundant computations. BPF+ [14] further enhances the performance of BPF by exploiting global data-flow optimization to eliminate redundant predicates across filter criteria and employing just-in-time compilation to convert a filtering criterion to native machine code. xPF [8] increases the computation power of BPF by using persistent memory and allowing backward jumps.

mmdump [15] is a special monitoring tool developed to capture Internet multimedia traffic, in which dynamic ports are widely used. *mmdump* is based on *tcpdump* and reduces the cost of criterion compilation to efficiently handle dynamic ports. A customized function in *mmdump* assembles new filtering criteria by using pieces of precompiled criterion blocks preserved from the initial filter compilation. Swift’s high-level SIMD instructions and hierarchical instruction optimization can be viewed as a generalization of this technique, but Swift’s techniques apply to any type of filter and require no special compiler techniques.

Mach Packet Filter (MPF) [16], PathFinder [17], and Dynamic Packet Filter (DPF) [18] are filters designed to demultiplex packets for user-level networking. MPF extends the instruction set of BPF with an associative match function to efficiently demultiplex packets for multiple user-level applications. PathFinder abstracts the filtering process as a pattern matching process and adopts a special data structure for the abstraction. The abstraction makes PathFinder amenable to both software and hardware implementations and capable of handling gigabits-per-second network traffic. DPF utilizes

dynamic code generation technology, instead of a traditional interpreter-based filter engine, to compile packet filtering criteria into native machine code.

The Fairly Fast Packet Filter (FFPF, later renamed as Streamline) [6] is the most recent research on packet filter. FFPF is a framework for network monitoring, which distinguishes itself from all other packet filters such as BPF. Multiple packet filtering programs can be simultaneously loaded into FFPF. The processing flow among these programs is organized as a directed graph. FFPF employs flow group, shared circular buffers, and even hardware to reduce the cost of packet copying among multiple concurrent filtering programs. FFPF expands filter capacity via external functions, which are essentially native code running in kernel space. In addition, FFPF features language-neutral design and provides backward compatibility with BPF.

FFPF and Swift are complementary as they target different problems. FFPF focuses on the packet-filtering framework, and its main contribution lies in the improvement of scalability for supporting multiple concurrent monitoring applications, while Swift aims at the packet-filtering engine and provides a fast, flexible, and safe filtering mechanism for individual applications. By virtue of the language-neutral design of FFPF, Swift can be implemented within the FFPF framework, taking advantages of both designs.

In addition to software-based packet capture solutions, hardware-based solutions such as [19] have been proposed to meet the challenge posed by extremely high-speed networks. Moreover, there exist many specialized-architecture monitoring systems such as Nprobe [20] and SCAMPI [21].

Besides packet filtering, packet classification and deep packet inspection (DPI) also play an important role in network monitoring. Although they share the same quest for high performance with packet filtering, they have quite different applications and usages, and hence take different and orthogonal optimization approaches. Packet classification needs to support massive simple matching rules lying in large search space, and thus its research mainly focuses on exploiting highly parallel data structures and hardware, such as HaRP [22], Hypersplit [23], and TCAM-based schemes [24], [25]. DPI requires using complex filtering rules, usually expressed in the form of regular expressions, and thus its research aims at accelerating regular expressing matching by utilizing various automata conversions, such as DFA [26], D2FA [27], and Tagged DFA [28].

III. DESIGN

In this section, we first present the motivation of Swift and its design overview, then we detail the design of Swift, including its unique ISA, and finally we analyze the characteristics of Swift in terms of performance and security.

A. Motivation

The inefficiency of BPF observed in our past experiences directly motivated Swift's design. The most significant performance degradation of BPF occurs in dynamic filtering tasks. This degradation is mainly caused by frequent filter updates. As mentioned, the unduly long filter update latency in BPF is

attributed to three filter preprocessing operations: filter recompilation, user-kernel copying, and security checking. While the latter two play nonnegligible roles in the long delay, the majority of the latency is introduced by filter recompilation [7], [15]. The duration of a filter update in BPF would be significantly shortened if the recompilation were selectively performed only on the changed primitive, or totally skipped, as *mmdump* accomplishes for selected filtering tasks. However, for general-purpose network monitoring tasks, recompiling the entire BPF filter is inevitable for each update because its instruction set architecture and filter program organization are unsuitable for fast update.

BPF uses a RISC-like instruction set for a low-level register machine abstraction. Thus, each *pcap* language primitive is translated into an instruction block that comprises a *variable* number of simple instructions. Changing a primitive in a filter often alters the size of the corresponding instruction block. Without recompiling, we need to modify code-offset-related instructions (e.g., conditional branch) throughout the entire compiled filter to accommodate the change. Control flow optimization, which is indispensable for BPF to speed up filter execution, makes the matter even worse. The BPF control flow optimization merges multiple identical instructions into one. This significantly reduces both filter program size and execution time, but complicates updates to instructions shared by several primitives.

In addition to filter update latency, we also find that the filter execution efficiency of BPF can be improved substantially. The RISC-like ISA in BPF induces high instruction interpretation overhead. Interpretation overhead refers to the operations an interpreter must perform before executing an actual filter instruction, such as program counter maintenance, instruction loading, operation decoding, and so forth. Those operations are unproductive toward evaluating filter criteria, but cannot be omitted. Because each BPF instruction accomplishes merely a very simple task, such as loading, arithmetic, and conditional branching, most of the CPU time in executing a BPF program is spent uselessly as interpretation overhead, and the CPU time spent in evaluating the actual packet filtering criteria only makes up a small fraction of the total. Our measurement results (detailed in Section V-B) show that the BPF interpretation overhead is about 5.2 ns on average in a machine with Intel Xeon EM64T 2.0 GHz CPU and makes up nearly 56% of the average instruction execution time.

BPF's continuing widespread use can be mainly attributed to: 1) the generic pseudomachine abstraction, which guarantees cross-platform compatibility; and 2) its natural-language-like, primitive-based filter language, which ensures ease of use to application developers and network administrators. Therefore, we decide to inherit from BPF the pseudomachine abstraction and language primitives while developing our own filtering model to achieve significant performance improvement.

B. Design Overview

The primary objective of Swift is to achieve low filter update latency. Our approach to reaching this goal is by reducing filter criterion preprocessing on filter update as much as possible. More specifically, we attempt to avoid filter recompilation

and optimization, allow “in-place” filter updating, and eliminate security checking.

To achieve “compilation-free” update, filtering criteria must map *directly* onto interpreter instructions. This makes a high-level, CISC-like instruction set architecture a natural choice for Swift. In addition to saving compilation time, the CISC-like instruction set also opens a door for performance optimization. A complex Swift instruction is able to accomplish the same task as several simple BPF instructions, thereby reducing instruction interpretation overhead.

Two design choices are made to enable in-place filter modification: fixing instruction length and removing filter optimization. By fixing the instruction length, we avoid the need to shift instructions on instruction replacement. By removing filter optimization, not only do we save precious time during a filter update, but also preserve one-to-one mapping between filtering primitives and filter program instructions: no instructions are shared. As a result, updates to a filter can be directly applied to the affected instructions without altering other instructions or filter program structure. This feature further helps to optimize filter update by reducing unnecessary user-kernel data copying. Only the updated part of a filter criterion is copied from user space to the kernel.

A simplified computational model ensures filter program safety for Swift without security checking. With the specialized ISA, each Swift instruction is able to perform a set of complex pattern-matching operations. The execution path of a filter program is determined by the Boolean evaluation result of each instruction: either continue (“true”) or abort (“false”). Therefore, Swift does not need storage or branch instructions to control the execution path of a filter program. With a fixed set of instructions, acyclic execution path, and zero data storage, any Swift filter program is safe to run in the kernel.

Our secondary objective is to increase filter execution efficiency. We achieve this goal by exploring the following two optimizations: SIMD expansion to the Swift instruction set and hierarchical execution optimization. SIMD allows an interpreter to perform a single instruction interpretation and apply the same operation on many sets of data, thereby significantly reducing the cost of instruction interpretation. SIMD has been widely used in contemporary high-performance processors, such as Intel Pentium series and IBM Power series processors. While Swift’s design ensures low filter update latency, it also forfeits the benefit associated with filter program optimization. To offset the possible performance loss, we introduce an alternative optimization method called hierarchical execution optimization. This optimization is based on our observation that during a dynamic filtering process, the newly added primitives are often related to some existing ones. For example, the new primitives quite often monitor the same host, but on different ports, or capture the same protocol traffic, but for different hosts. Therefore, the existing primitives can be viewed as the “parent” of the new primitives. Swift utilizes this hierarchical relationship among primitives to avoid redundant instruction executions. Instead of actively optimizing filter programs, i.e., performing automatic optimization in filter update, Swift makes the primitive hierarchy a hint for the filter execution engine and leaves applications responsible for constructing the hierarchy.

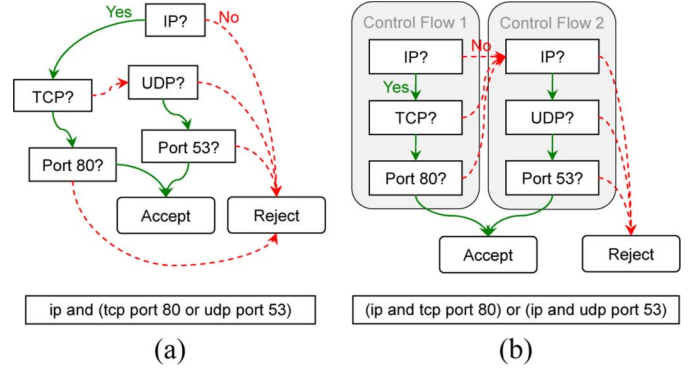


Fig. 1. Filter organizations of (a) BPF and (b) Swift for criteria matching HTTP and DNS traffic.

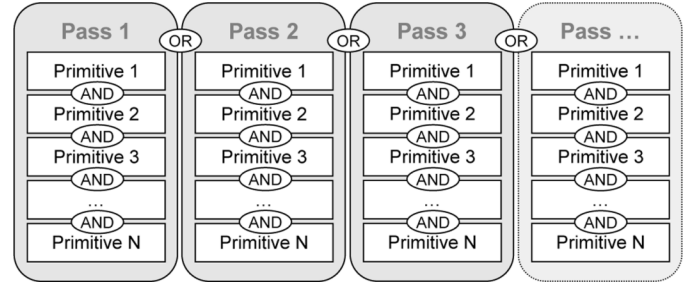


Fig. 2. Swift filter structure.

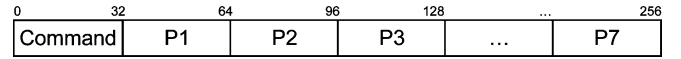


Fig. 3. Swift instruction format.

C. Detailed Design

The Boolean logic in a Swift filter is organized in disjunctive normal form. Fig. 1(b) illustrates the control flow organization of Swift, while the control flow graph of BPF, which is semantically equivalent, is shown in Fig. 1(a) for comparison. In the Swift control flow organization, each disjunct cluster—the rounded rectangle with shaded background—specifies a complete set of primitives that a packet must satisfy in order to be accepted by the Swift filter. In Swift, we name such a disjunct cluster a *Pass*, meaning a “passage” of packets.

A pass consists of one or more literals. In a BPF filter, a literal corresponds to a *pcap* language primitive. Swift inherits the primitives from BPF. However, instead of realizing a primitive with multiple simple instructions, Swift maps each type of primitive into a pseudomachine instruction—the basic building block of a filter program. Fig. 2 illustrates the structure of a Swift filter.

1) *Swift Instruction Set*: All Swift instructions have the same size, facilitating fast instruction modification on filter update. The Swift instruction layout is shown in Fig. 3, where one 32-bit command field is followed by seven 32-bit parameter fields. Such a nicely aligned 32-byte structure ensures efficient memory accesses.

We formulate our instruction set based on BPF primitives. We first classify BPF primitives into two categories according to their addressing modes. “Direct addressing” primitives, such as

TABLE I
SAMPLE OF SWIFT INSTRUCTION SET

| Addressing mode | Opcode | Semantics | Corresponding BPF operation |
|-----------------------------|----------|---|---|
| Direct Addressing | D_EQ | Compare a 32bit integer at an absolute offset to a supplied integer operand | Compare host IP address |
| | D_MEQ | Similar to the above, but a bitmask is applied to both comparands before comparison | Compare packet protocol; Compare host IP netmask |
| | D_LEQ_M | Compare up to three continuous 32bit integers at an absolute offset to the supplied operands (each pair of comparands is associated to a bitmask) | Compare source and destination IP netmask; Tell if a packet is IP, non-fragment, and TCP/UDP |
| Layer 1 Indirect Addressing | L1_SEQ | Compare a 16bit shortint at an indirect offset to a supplied shortint operand | Compare IP protocol; Compare TCP / UDP port |
| | L1_SRNG | Test if a 16bit shortint at an indirect offset is within a numerical range specified in the parameter list | Capture TCP / UDP packets on a range of ports |
| Layer 2 Indirect Addressing | L2_LEQ | Compare up to five continuous 32bit integers at an indirect offset to the supplied operands | TCP payload content matching |
| | L2_LEQ_M | Compare up to two continuous 32bit integers at an indirect offset to the supplied operands (each pair of comparands is associated to a bitmask) | TCP payload content matching (bit precision) |

“ether proto” and “ip src host,” fetch data from an absolute offset in a packet. “L1 indirect addressing” primitives, such as “tcp dst port,” address data by calculating the variable header length of a protocol layer and adding a relative offset to it. We then generalize the manipulation and comparison operations used in the semantics of BPF primitives. There are three types of basic operations: 1) test if equal; 2) mask and test if equal; and 3) test if in range. Each type also has some variations on operand width (short or long integer). Finally, we design the complex instructions to accomplish the corresponding operations. We come up with 14 different operations that, alone or by combination, are able to perform equivalent operations of any BPF primitives except “expr,” which involves arbitrary arithmetic.

To further exploit the CISC architecture and enhance performance, we introduce a new addressing mode, “L2 indirect addressing,” with four additional instructions. In the new addressing mode, filtering operations address data by first performing “L1 indirect addressing” to retrieve the related information, which is used to calculate the variable header length of a deeper layer, and then adding the relative offset. While BPF does not provide such primitives, there are practical demands such as filtering based on TCP payload. Moreover, we add four more “power instructions” that perform equivalent operations of several frequently used BPF primitive combinations, such as “ip src and dst net” and “tcp src or dst port.” Therefore, in total Swift has 22 different types of instructions.

Table I lists a selection of Swift instructions, which captures the characteristics of Swift’s CISC-like ISA. The four columns from leftmost to rightmost refer to the addressing mode, the instruction type, the instruction functionality, and the equivalent BPF operation(s), respectively. Swift instructions are fairly generic in that given different parameters, one Swift instruction can function as several different *pcap* language primitives. Examples are given in the fourth column. Based on the Swift instruction set, we can derive alternative faster implementations for some BPF primitives. For instance, the “ip and tcp port” primitive in BPF requires three initial steps with six instructions to examine whether a packet is IP, nonfragment, and TCP. In Swift, we can take advantage of the “continuous masked

comparison” instruction (D_LEQ_M), to perform the same examination in a single instruction.

We add the SIMD feature into the Swift instruction set by packing additional operands into unused parameter fields. For instance, the “Direct addressing load, test if equal” instruction (D_EQ) uses only one 32-bit operand. In contrast, the SIMD version of this instruction can carry up to six additional operands, and the corresponding operation becomes “Direct addressing load, and test if equal to any of P[1] through P[7].”

2) *Swift Pass and Filter Program*: A series of instructions connected by logic “AND” form a pass. When a packet arrives, the instructions of a pass are evaluated one by one. If all evaluation results are “true,” the packet is accepted and copied to user space. Otherwise, if any evaluation result is “false,” the packet fails the current pass and will be tested by remaining passes or dropped if it fails all passes. Passes are thus effectively independent and are combined by logic “OR.”

We achieve the hierarchical execution optimization feature by establishing hierarchical relationships among passes. When a Swift filter is initially set, the passes it contains are created by the application from scratch. In subsequent changes, if a new pass is related to one of the existing passes, the application is entitled to add the new pass by duplicating an existing pass and modifying the copy, or “child” pass. Performing duplication, instead of creating a pass afresh, has two benefits. First, the application saves time in updating the criterion—only the difference between the old and new control flows needs to be updated. Second, the parent-child relationship is noted by the filtering engine and is used to optimize filter execution.

When a pass is added by duplication, Swift makes a bit-exact copy of the parent pass and then marks all the instructions of the child pass as “copied,” a hint for the filtering engine that the marked instruction is exactly the same as the corresponding one in its parent pass. When an instruction in the child pass is later modified, the associated “copied” mark is removed. To evaluate a Swift filter, the filtering engine traverses the passes according to their hierarchical relationships (if any) in a depth-first manner. A parent pass is evaluated before its children. If the parent pass succeeds, then, as for any pass, the filtering

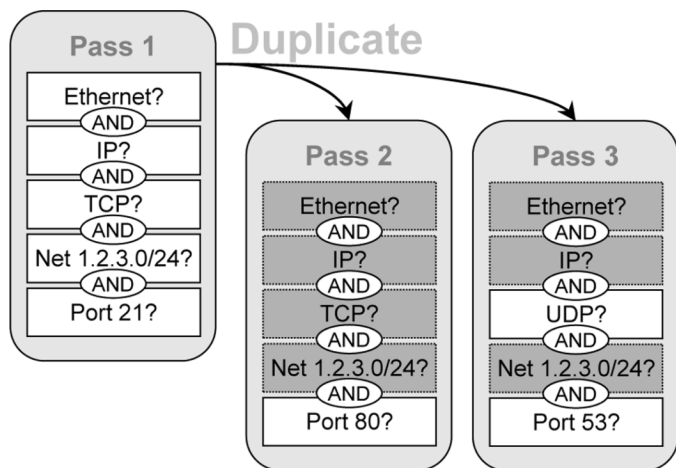


Fig. 4. Hierarchical pass relation diagram.

engine halts. Otherwise, Swift records those instructions that succeeded. When evaluating child passes, Swift need not re-execute any copied instruction that succeeded in the parent.

Fig. 4 illustrates an example of the pass relation in a Swift filter. Pass 1 is created from scratch, while the other two passes are added by duplicating pass 1. The instructions bearing the “copied” mark have shaded background, so the filtering engine may skip their evaluations.

D. Analysis

Before giving detailed analysis of Swift in terms of performance and security, we first summarize the shared design principles of Swift with other packet filters, especially BPF, as well as its unique design features that distinguish Swift from other packet filters. The shared features are marked with “+,” and the unique ones marked with “◇.”

- + Runs as a kernel module, filtering packets in place.
- + Uses architecture-independent pseudomachine.
- ◇ Utilizes CISC ISA with SIMD support.
- ◇ Enables compile-free, in-place filter modification.
- ◇ Ensures security with simplified computational model.

1) *Performance*: The performance superiority of Swift mainly originates from two aspects: high filter execution efficiency and low filter update latency.

Thanks to the capability of aggregating multiple simple operations into one instruction, Swift programs have much fewer instructions than their BPF counterpart. As a result, even though its per-instruction interpretation overhead is slightly higher than that of BPF, Swift achieves much lower interpretation overhead of an entire filter program. While the filter engine size of Swift (24 kB) is much larger than that of BPF (6 kB), our experimental results show that the larger code size has insignificant impact on performance. Even running Swift on the CPU with only 12 kB L1 cache (“PC1” in our experiment setup), there is still no observable performance degradation indicating cache thrashing.

Swift’s superior dynamic filtering performance is mainly attributed to its very low filter update latency. For a filtering program with N primitives and experiencing M changes per update, the three preprocessing phases in BPF—recompiling the

entire filter, copying the whole compiled filter code to the kernel, and security checking—all take $O(N)$ run-time. However, performing the same filter update in Swift involves neither compilation nor security checking. The only required operations, mapping the changed primitives into instruction opcodes and parameters, and copying the modified instructions into kernel, take $O(M)$ run-time. Because the filter update in Swift is only related to the number of changes per update (M), not to the complexity of the existing filter (N), its filter update latency can be substantially lower than that of BPF, especially when N is large (i.e., long and complex filtering criteria).

Although using fixed-length instructions increases the memory consumption of a Swift filtering program, the additional space overhead does not lead to a scalability issue. Each Swift instruction that corresponds to a filtering primitive only occupies 32 bytes of memory space. Thus, even for filter programs including thousands of filtering rules and hundreds of thousands of filtering primitives, their Swift filter code only consumes several megabytes of memory, which is insignificant on modern computer systems.

2) *Security*: Filter code safety has always been a concern in packet filter design. Since modern packet filters execute in kernel space, without proper code safety checking, a faulty filter program containing infinite loops, wild jumps, out-of-bound array indexes, etc., could lead to unpredictable results. In addition, a maliciously crafted filter program can bypass any user-level access protection and can seriously undermine system security.

Depending on the design model, different packet filters have different mechanisms to enforce the security of the filter programs. The FFPF filter languages allow memory allocation, and hence, FFPF has compile-time checks to control resource consumption and run-time checks to detect array boundary violations. In contrast, BPF only needs to perform a security check in the kernel just before the filter program is attached; any program containing backward or out-of-bound jumps or illegal instructions is rejected. However, Swift enforces security in its design and eliminates the necessity for run-time security checks. Swift trades off some of its computational power, i.e., arbitrary data manipulation, for simpler computational model. Because any Swift program is an acyclic deterministic finite-state automaton (DFA), the interpreter is always in a predetermined state, the execution of a finite-size filter program is always bounded, and Swift requires no security check at all.

Two rationales justify this design tradeoff. First, the reduction of computational power is harmless in the context of packet filtering. A packet filter is a very specific system tool with a well-defined set of operations. PathFinder [17] shows that all operations in packet filtering can be generalized as pattern matching. The *pcap* filter language uses the special primitive “expr” to support arbitrary data manipulation. However, this primitive is rarely used in practice because its main usage is to specify uncommon filtering criteria that are not covered by regular primitives. Second, BPF’s support for arbitrary data manipulation comes with a high performance penalty. BPF does not differentiate predefined and arbitrary data manipulation operations. Instead, all data manipulations are broken down to multiple elementary instructions. As a result, a significant amount of time is

wasted in interpretation, and sometimes BPF even takes a longer time to interpret an instruction than to execute it.

Different from BPF, we apply the strategy of “*optimize for the common case and prepare for the worst*” in the design of Swift. Swift supports well-defined and commonly used data manipulations by incorporating each variant in a single instruction and integrating their implementations into the filtering engine. Since those operations are carried out by native binary code, Swift achieves very high execution efficiency. Swift cannot perform data manipulations that are not defined in its instruction set. Instead, the user applications need to carry out the custom data manipulations by themselves. However, in case an unsupported data manipulation is desperately needed (for example, a new layer-2 or layer-3 protocol appears requiring different data parsing operations), we can always add new instructions to Swift.

IV. IMPLEMENTATION

We have implemented the Swift kernel engine and user-space libraries in Linux 2.6. Currently, we provide implementations for both i386 and x86_64 architectures, and we plan to port Swift to other open-source UNIX variants such as FreeBSD in the future.

A. Kernel Implementation

Swift coexists with the Linux kernel’s LSF. LSF is the module equivalent to BPF in BSD UNIX and the default packet filtering module for the widely used *libpcap* library. Our implementation requires little modification to the existing kernel code and is compatible with the existing packet-filtering framework. Swift’s user-kernel communication mechanism uses the `setsockopt()` system call. Swift filter programs are attached to the same kernel data structure `sk_filter` as LSF filter programs, with a flag set to tell two kinds of programs apart. Packets captured by Swift and LSF share the same delivery path no matter which packet filter is being used.

B. Userland Libraries

The *libpcap* library provides a set of well-designed routines for setting filter programs and processing packets, as well as utility functions for handling devices and dumping captured packets. Instead of hacking *libpcap* to incorporate Swift, we developed a set of complementary libraries. Applications based on Swift can seamlessly use those *libpcap* functions that are unrelated to filter setting, but must invoke a separate mechanism to communicate with the Swift filter engine for filter program installation and update.

As shown in Table II, the C library *libswift* provides a set of function APIs for the convenient manipulation of Swift filter programs. We also implement a C++ library *ooswift*, providing object-oriented filter program control and manipulation and improved debugging support. Table III shows a common filtering criterion expressed in *pcap* primitives, in Swift using *ooswift*, and in compiled LSF code. The table illustrates the clear logical connection and easy transformation between the Swift filter program and the *pcap* language primitives.

TABLE II
SELECTION OF *libswift* APIs

| Routine | Functionality |
|--------------|---|
| SPF_Open | Create and attach an empty Swift filter to a socket |
| SPF_NewPass | Create a new pass in the Swift filter |
| SPF_DelPass | Remove a pass from the Swift filter |
| SPF_DupPass | Create a copy of a given pass |
| SPF_SelectOp | Assign a predefined operation (equivalent to a <i>pcap</i> language primitive) to an <i>instruction</i> of a given pass |
| SPF_AddParam | Add an additional (SIMD) parameter to an <i>instruction</i> in the given pass |
| SPF_DelParam | Remove a given parameter from an <i>instruction</i> in the given pass |
| SPF_PokeInst | Change an arbitrary part of an <i>instruction</i> in the given pass |

TABLE III
FILTER EXPRESSED BY *pcap*, SWIFT, AND LSF

| Filtering criterion by <i>pcap</i> | | | | |
|---|------|---------------|-------|-------|
| ip src net 192.168.254.0/24 and tcp dst port 23 | | | | |
| Swift filter program | | | | |
| Pass.Inst(0)→EtherIP_TCP_NonFrag() Pass.Inst(1)→Ether_IPSrc(0xFFFFFFFF00, 0xC0A8FE00) Pass.Inst(2)→EtherIP_TCUDPDst(23) | | | | |
| LSF filter program | | | | |
| (00) | ldh | [12] | | |
| (01) | jeq | #0x800 | jt 2 | jf 13 |
| (02) | ld | [26] | | |
| (03) | and | #0xffffffff00 | | |
| (04) | jeq | #0xc0a8fe00 | jt 5 | jf 13 |
| (05) | ldb | [23] | | |
| (06) | jeq | #0x6 | jt 7 | jf 13 |
| (07) | ldh | [20] | | |
| (08) | jseq | #0x1fff | jt 13 | jf 9 |
| (09) | ldxb | 4*([14]&0xf) | | |
| (10) | ldh | [x + 16] | | |
| (11) | jeq | #0x17 | jt 12 | jf 13 |
| (12) | ret | #96 | | |
| (13) | ret | #0 | | |

C. Implementation Optimizations

In addition to the Swift architecture, we have employed several implementation optimization techniques, which help Swift to gain further performance improvement.

First, we introduce computational cache to reduce computation redundancies. Swift does not provide any storage to the filter program, and thus no information can be explicitly exchanged between instructions. As a result, the execution of instructions incurs computation redundancies, such as repeated calculation of protocol header lengths. Computational cache provides an implicit channel to share information across instructions. When protocol header parsing on a packet is performed for the first time, the header length is stored in the computational cache, and therefore future header-length computations on the same packet can be saved by directly retrieving the cached results.

Second, we integrate the SIMD indicator with the instruction identifier. This optimization essentially unrolls evaluation loops

for multiple operands into linear instructions. It avoids performance penalties of using loops with small code segments. Moreover, it also eliminates the need to allocate and check a separate SIMD indicator for each instruction, saving additional decode time.

Finally, we make use of hardware stack in filter evaluation. Because recursive function calls are impractical for in-kernel code, Swift must evaluate hierarchical filter passes iteratively, which requires simple local storage (a stack or queue). Using high-level data structures, such as an indexed array, is too slow in that every storage access involves many operations: accessing, modifying, and storing the array index, loading the array base address, and finally, dereferencing and loading the desired element. Instead, we directly use the local stack provided by the processor and avoid interfering accesses to function local variables by instructing the C compiler to keep the frame-pointer of the evaluation engine function.

V. EVALUATION

In this section, we evaluate the performance of Swift on both dynamic and static filtering tasks and compare it to that of LSF and C kernel filters. The C kernel filters (“Opt-C” for short in the following) are hand-coded, compiled (using gcc “-O2” option) C programs that provide some indication of possible performance gains obtainable by noninterpreted binary code. Each C kernel filter is coded for a specific filtering task. We use the performance of Opt-C filters as an approximation to the performance of FFPF filters. An FFPF filter written in FPL-3, which is FFPF’s native language, is first transformed into a C program and compiled into native code by gcc, and then loaded as a kernel module for use. Since both FPL-3 filters and our Opt-C filters run inside the kernel natively and only a single filter program runs in each experiment, the performance difference between FFPF filters and Opt-C filters should be minimal.

Swift, LSF, and Opt-C filters share the same filtering framework and only differ in filtering engine. Therefore, we use the number of CPU cycles spent by the filtering engine as the microbenchmark metric. This metric is computed by taking the difference of the x86 Time-Stamp Counter (TSC) just before and right after a specific filter operation. For dynamic filtering tasks, the operation is filter update, while for static filtering tasks, the operation is filter evaluation. In order to compare filter performance across different platforms, we further convert CPU cycles into clock time based on the corresponding machine’s processor frequency.

To evaluate the filters in a realistic but controllable environment, we set up a test bed using a gigabits-per-second SMC managed switch to connect four different machines. We use the mirror function of the switch to redirect the traffic on the specified source port to the mirror port. The packet generator machine (PCS) connected to the source port replays traces, and one of the other three machines (PC1–3) connected to the mirror port captures the replayed traffic as a monitoring device. The four machines (PCS and PC1–3) have different generations of processors ranging from Pentium 4 x86 to the latest Xeon dual-core EM64T. The configurations of these machines are listed in Table IV.

TABLE IV
TEST BED MACHINE CONFIGURATIONS

| | CPU | L2 | FSB |
|-----|--|-------|---------|
| PC1 | 1 × Intel Pentium 4 2.8GHz (32-bit) | 1MB | 533MHz |
| PC2 | 2 × Intel Xeon 2.8GHz (32-bit w/ HyperThreading) | 512KB | 800MHz |
| PC3 | 2 × Intel Xeon 2.0GHz (EM64T DualCore) | 4MB | 1333MHz |
| PCS | 1 × Intel Pentium D 2.8GHz (EM64T DualCore) | 2MB | 800MHz |

A. Dynamic Filtering Performance

We use the task of capturing FTP passive-mode traffic, a typical dynamic filtering task, to measure the performances among Swift, LSF, and Opt-C filters. We developed an application called *FTPCap* to monitor FTP traffic and collect performance statistics. Three variants that use Swift, LSF, or Opt-C are called *FTPCap-Swift*, *FTPCap-LSF*, and *FTPCap-Opt-C*, respectively.

1) *Experimental Setup*: In passive-mode FTP, the server port of a control connection is fixed (usually 21), but the server ports of data connections are dynamically assigned. *FTPCap-LSF* initially employs “(ip and tcp port ftp)” to capture FTP control packets. When a control packet containing the port number for a new data connection is captured, the server IP address and port number for the new connection will be recorded, and *FTPCap-LSF* will generate a new criterion similar to “(ip and tcp port ftp) or (ip x1 and (tcp port y1 or tcp port y2)) or (ip x2 and (tcp port y3 or tcp port y4)),” in which “x1” and “x2” refer to the server IP addresses and “y1 . . . y4” refer to the port numbers. The LSF optimizer performs better when the port numbers of the same server are grouped together. Correspondingly, *FTPCap-Swift* initializes the first pass with the criterion “(ip and tcp port ftp)” to capture FTP control packets. When a data connection setup event is detected, *FTPCap-Swift* either simply includes the new port number in the corresponding pass if the server is already observed, or adds a pass using hierarchical execution optimization otherwise. *FTPCap-Opt-C*, unlike the previous two, has no code for filter setting and updating because the work is already taken by the Opt-C filter. It simply turns on/off the Opt-C filter.

The FTP traffic trace is obtained in a LAN environment. We set up 10 FTP servers with different IP addresses. For each server, we make 20 concurrent passive-mode file transfer connections, which are initiated one by one. In other words, at maximum there are 200 concurrent passive FTP data connections. This trace lasts 45 s with 3948 packets per second (pps) on average. In addition, we emulate the scenario of monitoring FTP packets under high-rate background traffic by mixing the captured FTP traffic with a constant high-rate (500 Kpps) non-FTP background traffic. The background traffic is generated by using *tcpreplay* [29] to play back a large trace file, which is captured at the edge gateway of our campus network.

Besides using filter update latency as the microbenchmark performance metric, we also use the number of missing packets per data connection as the macrobenchmark performance metric. The missing packets refer to those packets that are not captured by *FTPCap* at the beginning of a newly established

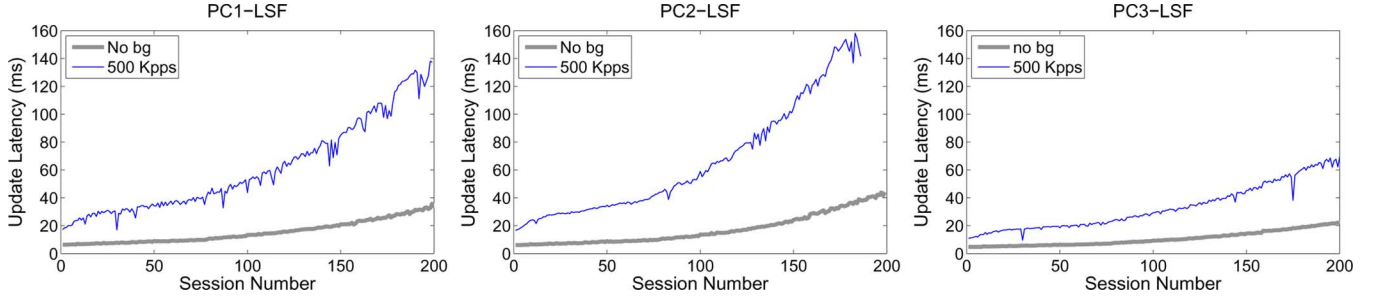


Fig. 5. LSF filter update latency.

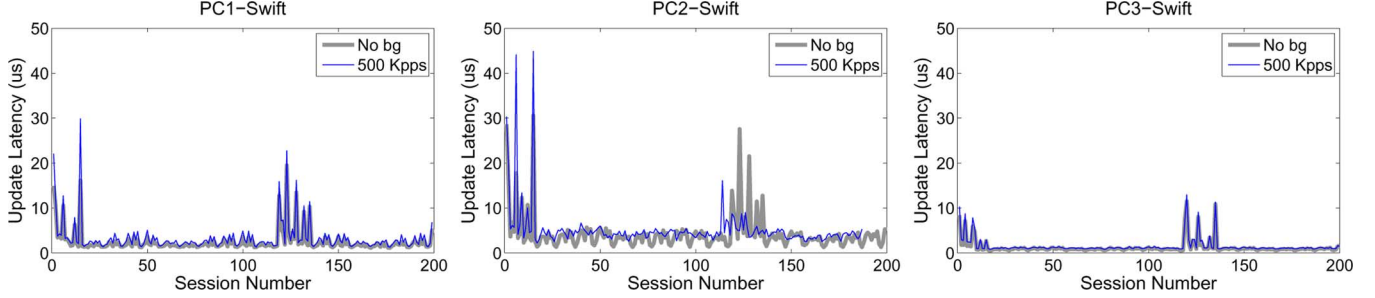


Fig. 6. Swift filter update latency.

data connection. The packet miss is caused by the filter update latency being larger than the FTP client acknowledgment delay—the interval between the time when the client receives the port assignment message and the time when the client starts to communicate with the server on that port. The metric is derived by counting the number of the transmitted packets (including TCP control packets) prior to the first packet captured by FTPCap, based on the offline analysis of the replayed trace.

2) *Experimental Results:* We run FTPCap-LSF and FTPCap-Swift 20 times each on PC1–3, 10 times with the FTP traffic trace replayed and the other 10 times with the mixed traffic trace replayed. We take the median of 10 experimental results as the final result. FTPCap-Opt-C is also tested. Because there is no filter update at user space, its filter update latency is zero, and no packet is missed by FTPCap-Opt-C for either FTP traffic or mixed traffic. Therefore, we focus on the performance comparison between LSF and Swift.

Figs. 5 and 6 show how filter update latency changes with an increase in concurrent data connections for LSF and Swift, respectively. The thick and thin curves show the filter update latencies for the traces with no background traffic and with 500-Kpps background traffic, respectively. The most significant difference between Figs. 5 and 6 lies in the scale of the y -axis. While the filter update latency for LSF is on the order of milliseconds (ms), the filter update latency for Swift is only on the order of microseconds (μ s). By eliminating filter compilation and security checking, Swift gains at least three orders of magnitude speedup against LSF in filter update. Over 99% of LSF’s latency is caused by user-level filter recompilation, but the remaining user-kernel copy and security check latency is still much larger than Swift’s entire update latency. For example, the user-kernel copy and security check latency on PC3 grows from 10 to 20 μ s in the experiment. FTPCap running on PC2 does not capture all control packets that carry dynamic port information under mixed traffic, which results

in incomplete thin curves in “PC2-LSF” and “PC2-Swift.” The missing critical control packets are mainly due to PC2’s insufficient processing capacity.

As shown in Fig. 5, both concurrent connections and background traffic affect the filter update latency of LSF. When the number of concurrent connections increases, the filtering criterion expressed in *pcap* language becomes longer, and the compilation procedure and security checking consume more CPU cycles. In contrast, the filter update latency of Swift is basically insusceptible to changes in concurrent connections and background traffic: Although all Swift curves in Fig. 6 fluctuate slightly, the thin curves overlap with the thick curves to a great extent. This is because Swift filter updates are incremental and adding filter instructions for a new connection takes almost constant time. The large spikes of Swift curves, which occur at the beginning and around the addition of the 120th connection, are attributed to the relatively large overheads caused by pass duplication.

Figs. 7 and 8 illustrate the average number of missing packets per data connection by LSF and Swift, respectively. The y -axis scales are again significantly different. The average numbers of missing packets per connection for LSF range from 30 to 160, while those for Swift are only one or two at maximum. Without background traffic, Swift does not miss any packet no matter how many concurrent connections exist. With background traffic, the average levels of the “500 Kpps” curves slightly lift after around 120 concurrent connections, which coincides with the occurrence of the second group of large spikes in Fig. 6. The lift of fluctuation level may be attributed to the added passes and related pass duplication. The addition of more passes extends the filtering path for non-FTP packets and results in more CPU time spent on non-FTP traffic filtering. Even so, Swift only misses one or two packets per connection.

There are two additional issues associated with the LSF curves in Fig. 7. First, the ceiling phenomena—both “No bg”

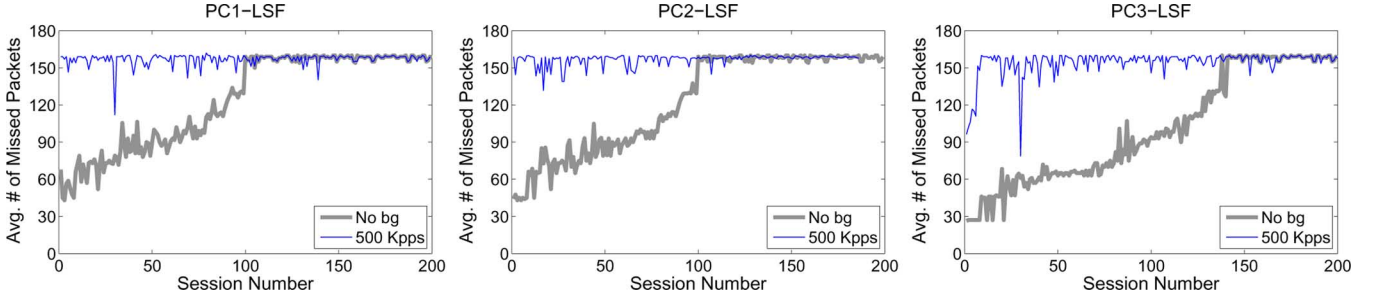


Fig. 7. Missing packets per data connection by LSF.

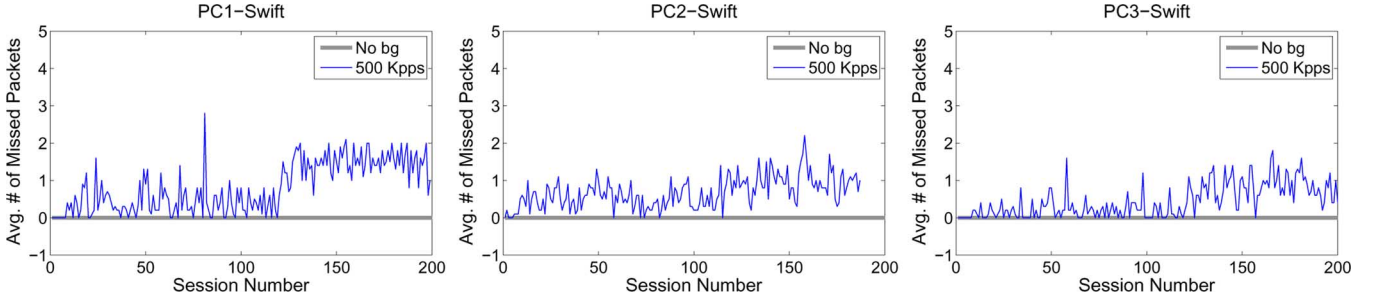


Fig. 8. Missing packets per data connection by Swift.

TABLE V
STATIC FILTERS IN *pcap* LANGUAGE AND THEIR INSTRUCTION COUNTS IN LSF AND SWIFT

| Filter | Description | LSF Inst.# | Swift Inst.# |
|--------|--|------------|--------------|
| 1 | "" (Accept all packets) | 1 | 0 |
| 2 | "ip" | 3 | 1 |
| 3 | "ip src net 192.168.2.0/24 and dst net 10.0.0.0/8" | 10 | 2 |
| 4 | "ip src or dst net 192.168.2.0/24" | 10 | 2 |
| 5 | "ip and tcp port (ssh or http or imap or smtp or pop3 or ftp)" | 23 | 2 |
| 6 | "ip and (not tcp port (80 or 25 or 143) and not ip host (...))" (The ellipsis mark stands for 38 IP addresses ORed together.) | 95 | 10 |

and “500 Kpps” curves bounded by 160—are caused by the rate-limiting of the FTP servers. In the initial period of a data connection, the servers first transmit about 160 packets in tens of milliseconds, and then stay idle for the next several hundred milliseconds (not all shown in figure) to limit downloading rate. Such “bounding” behavior occurs for a wide range of rate-limit settings (e.g., from 100 kB/s to 2 MB/s). Since an LSF filter update latency is always shorter than the duration of the idle phase, the number of missing packets in each update is bounded. Second, the round-trip time (RTT) of the FTP trace is small, varying from tens of microseconds to hundreds of microseconds, as the trace is collected in a LAN environment. A larger RTT would cause fewer packets to be transmitted during the time window of a filter update, thus reducing the impact of filter updates on packet missing. Compared to LSF, Swift is almost insensitive to the variation of RTT, and hence can support applications that require high-fidelity data capture in diverse network environments.

B. Static Filtering Performance

We use six sets of filters with increasing complexity, as shown in Table V, for static filtering performance evaluation. The instruction numbers of these filters in LSF and Swift are also listed

for comparison. The Opt-C filter programs show performance gains that could potentially be achieved by improving LSF and Swift to native code speeds.

The trace for static filtering is captured at the gateway of our campus network. It contains over 14 million packets (75 bytes snap length), and its size is around 1.1 GB. We play back the trace file at 250 Kpps rate using *tcpreplay*. Assuming an average of 500 bytes per packet, the playback rate represents a fully utilized 1-Gb/s link bandwidth. We record the average time spent in accepting and rejecting packets separately and select the larger value of the two as the filter performance data. We choose the larger value, instead of the smaller one or the average, because the worse-case runtime is much less affected by network traffic conditions, such as traffic speed and composition.

Fig. 9 illustrates the per-packet processing time of LSF, Swift, and Opt-C on all machines for each filter. In addition, Table VI details the breakdown of the per-packet processing time for both LSF and Swift on PC3. The “Exec.” column shows the average execution time per instruction and the average number of instructions executed per packet, in the format of (time/instruction) \times (instruction count). The “Aux.” column shows the auxiliary processing time spent on filter engine setup and shut-down operations, such as call/ret instructions and local stack maintenance.

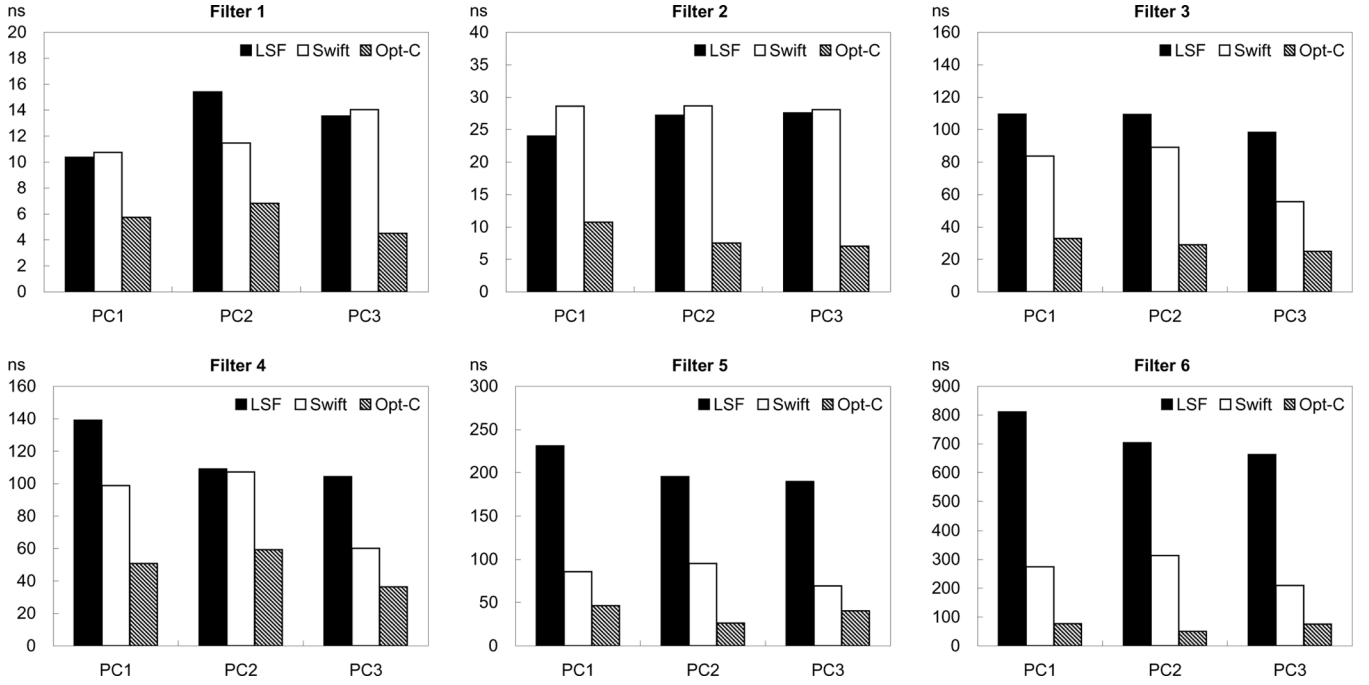


Fig. 9. Per-packet processing time of each static filter (nanoseconds).

TABLE VI
PROCESSING TIME BREAKDOWN FOR LSF AND SWIFT

| Filter | LSF | | Swift | |
|--------|-------------|--------|------------|--------|
| | Exec. | Aux. | Exec. | Aux. |
| 1 | 5.2ns×1.0 | 13.8ns | (N/A)×0 | 23.1ns |
| 2 | 10.3ns×3.0 | 14.1ns | 14.5ns×1.0 | 22.1ns |
| 3 | 9.4ns×9.0 | 12.9ns | 13.8ns×2.0 | 20.9ns |
| 4 | 8.4ns×6.0 | 12.5ns | 13.7ns×2.0 | 21.2ns |
| 5 | 10.3ns×19.8 | 14.8ns | 25.4ns×1.9 | 21.3ns |
| 6 | 8.0ns×78.6 | 13.4ns | 29.5ns×7.4 | 21.5ns |

Filters 1 and 2 are the simplest criteria designed to show the minimum overhead induced by the filtering engine. The corresponding results in Fig. 9 demonstrate that Swift and LSF have approximately the same processing speed with these two simple filters. Both Swift and LSF run slower than Opt-C. Table VI further sheds some light on the performances of both LSF and Swift filter engines. For filter 1, the LSF filter program only consists of a simple “ret” instruction, and thus the 5.2 ns per-instruction execution time is mainly determined by LSF’s interpretation overhead. In contrast, the Swift filter engine is designed to accept all packets by default. Therefore, the Swift filter program does not contain any code, and its processing time is spent entirely on the filter engine setup and shutdown. By adding a “nop” instruction for Swift to execute before accepting a packet, we estimate Swift’s interpretation overhead to be about 8.2 ns. For filter 2, although the per-instruction execution time of LSF is 29% shorter than that of Swift, its overall execution time is longer than that of Swift. This is because the instruction count ratio between LSF and Swift is 3:1.

Filters 3 and 4 are light-load criteria designed to demonstrate filtering engine performance on basic packet classification. The corresponding results in Fig. 9 indicate that Swift has a moderate performance advantage over LSF on all machines.

For filter 3, compared to Opt-C, LSF takes 232%–292% more time to process a packet, with an average slowdown of 267%; Swift takes 122%–207% more time to process a packet, with an average slowdown of 161%. The average speedup of Swift over LSF is 143%. For filter 4, compared to Opt-C, LSF takes 84%–187% more time to process a packet, with an average slowdown of 148%; Swift takes 65%–95% more time to process a packet, with an average slowdown of 80% times. The average speedup of Swift over LSF is 139%. Similar to the cases of filters 1 and 2, Table VI shows that for filters 3 and 4, the per-instruction execution time of Swift is about 50% longer than that of LSF, but the much larger instruction count makes LSF slower than Swift in packet processing.

Filter 5 is a moderate-load criterion designed to test the filtering engine’s capability of handling highly specific operation. The corresponding results in Fig. 9 show that Swift outperforms LSF by a significant amount on all machines. Compared to Opt-C, LSF takes 368%–638% more time to process a packet, with an average slowdown of 468%; Swift takes 70%–260% more time to process a packet, with an average slowdown of 139%. The average speedup of Swift over LSF is 250%. The significant speedup of Swift is due to its architectural advantages and specifically SIMD instructions. The ability to pack many operands (12 for TCP/DUP ports) in one instruction and batch the execution of comparison operations within a single filter engine “cycle” enables many-fold reduction at the cost of instruction interpretation and improves the performance of Swift close to that of Opt-C. As shown in Table VI, compared to previous filters, LSF maintains its per-instruction execution time, but executes much more instructions. By contrast, Swift maintains its instruction count and packs more operations in each instruction.

Filter 6 is a heavy-load, “real-life” criterion obtained from the campus network administrator. This filter is used by an application to detect suspicious IRC traffic. The filter is sufficiently complex for Swift to utilize the optimizations discussed earlier,

TABLE VII
OPTIMIZATION EFFECTS OF SWIFT FILTER 5

| Filter 5 | Original | Less-optimized | Unoptimized |
|------------|----------|----------------|-------------|
| Exec. Time | 69.7ns | 178.4ns | 204.5ns |

namely SIMD instructions and hierarchical execution optimization. The corresponding results in Fig. 9 show even higher performance increase of Swift against LSF. Compared to Opt-C, LSF takes 783%–1294% more time to process a packet, with an average slowdown of 1009%; Swift takes 179%–521% more time to process a packet, with an average slowdown of 318%. The average speedup of Swift over LSF is 279%. According to Table VI, even though the per-instruction execution time of LSF is less than one-third of Swift, the instruction count ratio between LSF and Swift is 10:1.

For all these filters, the auxiliary processing time for both LSF and Swift is fairly steady, as shown in Table VI. Although the auxiliary cost of Swift is about 8–9 ns more than that of LSF, the extra cost is insignificant as the filter becomes more complex and requires more time to execute.

We further measure the average execution time of Swift filter 5 with no SIMD extension (“Less-optimized”) and with neither SIMD extension nor hierarchical execution (“Unoptimized”) to provide more insights into the effect of Swift optimizations on performance improvement. The corresponding results are listed in Table VII. The removal of SIMD instructions exerts a great impact on the performance of the Swift filter, resulting in a slowdown of 156%. The comparatively small increase of execution time after the removal of hierarchical execution indicates that the hierarchical execution has a minor effect on the performance of the Swift filter. For Swift filter 6, due to filter program organization, we remove hierarchical execution optimization for the “Less-optimized” experiment and remove both hierarchical execution and SIMD extension for the “Unoptimized” experiment. Again, the impact of SIMD extension is far greater than that of hierarchical execution on the average execution time.

Overall, we find that: 1) the SIMD extension plays a very important role in speeding up Swift filter execution; and 2) the hierarchical execution also helps the speedup, but its effect is much smaller than that of SIMD extension, especially with a large instruction count. Without the SIMD extension and hierarchical execution, Swift can only perform comparably to optimized LSF for static filter tasks. These results are consistent with our observation from Fig. 9 and Table VI: The speedup of Swift is mainly attributed to its use of much fewer instructions than LSF.

Fig. 10 presents a comprehensive picture of filter execution time for LSF, Swift, and Opt-C filters among all machines categorized by six filtering criteria. It provides a good overview of the static filtering performance for cross comparison. When the filtering criteria are simple, LSF, Swift and Opt-C have nearly indistinguishable performance. As the criteria become more complex, the differences of filter execution time among the three filtering engines grow. Although both Swift and LSF run slower than Opt-C, the filter execution time of Swift grows at a much slower rate than that of LSF, and thus Swift achieves much closer performance to Opt-C than LSF.

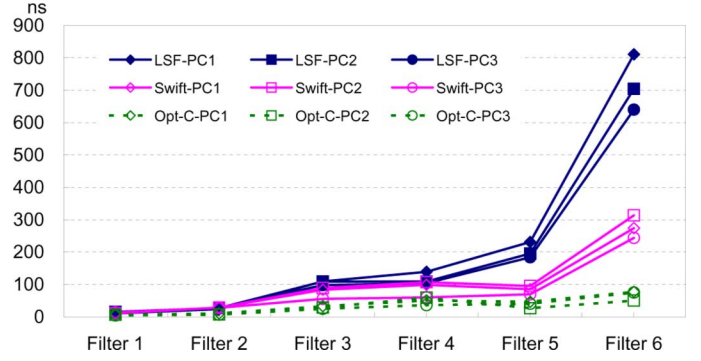


Fig. 10. Per-packet processing time on all machines.

VI. HARDWARE ENHANCED IMPLEMENTATION

Enhanced instructions such as SIMD instructions have been available on modern CPUs. Among those instructions, the family of SSE instruction sets on the x86 architecture is most notable and widely supported. However, these hardware SIMD instructions have not been leveraged by previous packet filters or network traffic monitoring programs. Thanks to the special instruction design, it is easy and straightforward to map Swift’s SIMD instructions to hardware SIMD instructions. Therefore, Swift is capable of harnessing the parallel processing power provided by modern processors to accelerate packet filtering.

A. SSE Conversion

The main feature as well as the key to the speedup of SIMD instructions is their ability to process multiple data fields in parallel. Most of the Swift instructions match well with this feature, as described in Section III-C. Consequently, the conversion of Swift execution engine to using SSE instructions is not only natural, but also transparent. In other words, the binary representations of all Swift instructions remain the same for non-SSE and SSE implementations of the execution engine.

We use two typical Swift instructions, EQ_4 and SRNG_4, as the examples to illustrate the implementation conversion. The EQ_4 instruction compares a 32-bit integer against four 32-bit comparands. The execution of the non-SSE implementation, shown in Fig. 11(a), involves sequential comparisons of the data to each comparand. Thus, four comparisons are made in the worst case. In contrast, for the execution of the SSE implementation, shown in Fig. 11(b), four copies of the data are first prepared in a single XMM register, and then only one comparison is performed. The SRNG_4 instruction examines if a 16-bit integer is within the four given ranges. Its executions of the non-SSE and SSE implementations are shown in Fig. 12(a) and (b), respectively. The non-SSE implementation requires eight sequential comparisons in the worst case, while the SSE implementation needs only one subtract-with-saturation operation and one comparison operation.

B. Implementation Detail

Our enhanced implementation of the Swift filtering engine utilizes SSE instructions to interpret most of Swift instructions except for layer header length calculation, which cannot be converted to any currently available SSE instructions.

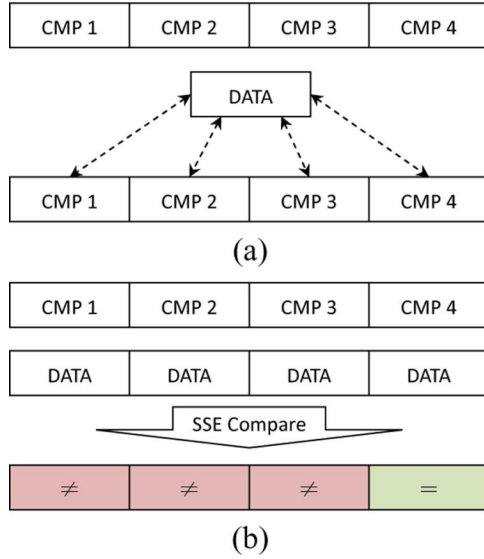
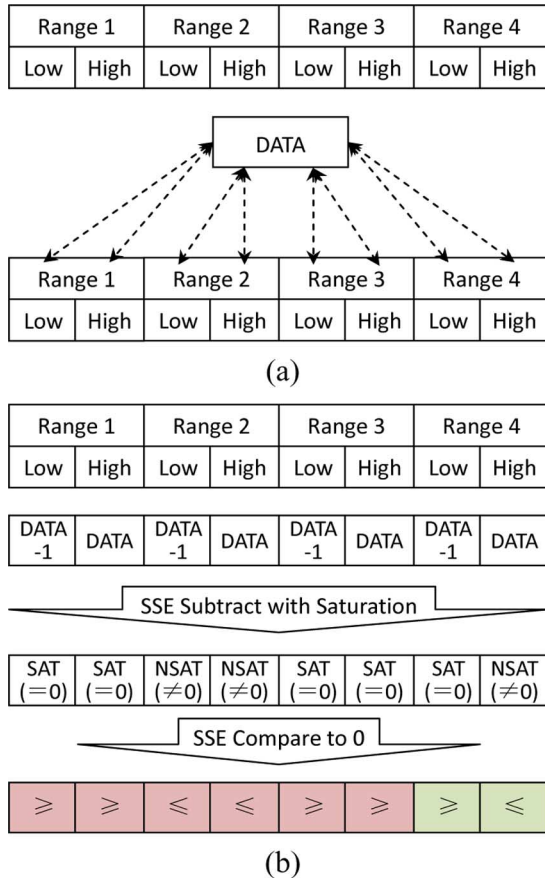
Fig. 11. Illustrations of two EQ₄ implementations. (a) Non-SSE. (b) SSE.Fig. 12. Illustrations of two SRNG₄ implementations. (a) Non-SSE. (b) SSE.

Table VIII lists the collection of all SSE instructions used in our implementation.

Theoretically, the use of SSE instructions can accelerate filter execution. The speedup, however, does not come without cost. The execution of SSE instructions also incurs overhead, due mainly to the use of complex operations and the involvement of additional processor hardware such as registers. Therefore, the

TABLE VIII
LIST OF SSE INSTRUCTIONS USED IN SWIFT

| Intel Intrinsics | x86 Instruction |
|-------------------|-----------------|
| _mm_cvtsi32_si128 | MOVQ |
| _mm_shuffle_epi32 | PSHUFD |
| _mm_cmpeq_epi32 | PCMPEQD |
| _mm_cmpeq_epi16 | PCMPEQW |
| _mm_movemask_epi8 | PMOVBMSKB |
| _mm_and_si128 | PAND |
| _mm_load_si128 | MOVDQA |
| _mm_subs_epu16 | PSUBUSW |

TABLE IX
INSTRUCTION EXECUTION TIME (NANOSECONDS)

| EQ Instruction | | | | | | | |
|----------------|------|------|------|------|------|------|------|
| Operands | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| non-SSE | 37.6 | 39.4 | 40.2 | 41.6 | 42.5 | 43.3 | 44.7 |
| SSE | 38.0 | 38.5 | 38.0 | 38.0 | 39.8 | 40.7 | 40.7 |

| SRNG Instruction | | | | | | | |
|------------------|------|------|------|------|------|------|------|
| Operands | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| non-SSE | 39.8 | 42.5 | 45.1 | 47.8 | 49.5 | 50.9 | 52.2 |
| SSE | 45.1 | 45.6 | 45.6 | 45.6 | 46.0 | 46.4 | 46.0 |

performance of the SSE implementation on simple or nonparallelized operations might not be as good as that of the non-SSE implementation. To overcome this limitation, our enhanced Swift engine incorporates both SSE and non-SSE implementations, and allows a user application to mix the use of both implementations to create “hybrid” filters that can take full advantage of Swift. This is achieved by utilizing a reserved bit in the Swift binary instruction header as the selector, thanks to the binary compatibility between SSE and non-SSE implementations.

C. Performance Evaluation

We first measure the performance of individual instructions with SSE enhancement. Table IX presents the execution times of the two example instructions with different number of operands. From this table, we can observe that, with one or two operands, non-SSE implementations are slightly faster than SSE implementations. However, the execution time of a non-SSE implementation tends to increase linearly with the number of operands, while that of the corresponding SSE implementation tends to remain constant when multiple operands can be handled in parallel. Consequently, with a large number of comparands, the performance of SSE is evidently superior than that of non-SSE, as shown in Fig. 13.

We then measure the “composite” speedup of the SSE implementation by recording the execution time of each static filter used in Section V.¹ Filters 1 and 2 are excluded due to their simplicity. The results of filters 4–6 are presented in Fig. 14, which illustrates the comparison on the execution time of each filter between the SSE and the non-SSE implementations. Filters 3 and 4 consist of all nonparallelized Swift instructions, and their execution times in SSE are 10.0% and 15.6% slower than those in non-SSE, respectively, due to the overhead of SSE.

¹The measurements are performed on a more recently purchased machine than PC3. The configurations of the new machine are: dual Xeon 5400 processors with 8-MB L2 cache, integrated memory controller (no FSB), and 6-GB DDR3 SDRAM.

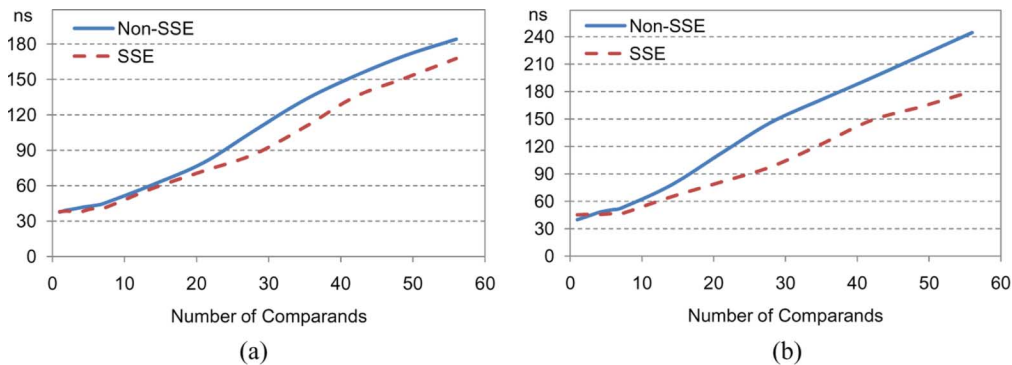


Fig. 13. Time of execution versus number of comparands. (a) EQ instructions. (b) SRNG instructions.

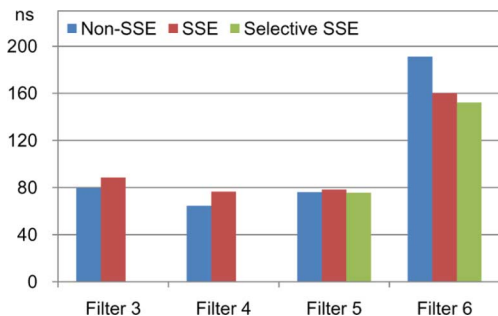


Fig. 14. Execution times of filters 3–5.

However, with increased filter complexity, more instructions are parallelizable in filters 5 and 6. Thus, SSE runs only 2.8% slower than non-SSE in filter 5, while SSE runs 19.3% faster than non-SSE in filter 6. To assess the maximal potential of the enhanced Swift engine, we create “hybrid” versions of filters 5 and 6 by employing both SSE and non-SSE implementations. In particular, we identify those Swift instructions that execute slower with SSE implementation in filters 5 and 6 and revert them to non-SSE implementation. The hybrid filters 5 and 6, presented as “Selective SSE” in Fig. 14, run 0.6% and 25.6% faster than their non-SSE counterparts, respectively.

D. Future Extensibility

Swift can be further enhanced with the upcoming addition of newer SSE instructions. For example, the “fused multiply-add” instruction can be applied to speed up packet layer header calculation; various integer arithmetic and logical instructions can be utilized to provide Swift programs with new features such as high performance checksum verification and hash generation, etc.

VII. CONCLUSION

This paper presents the design and implementation of the Swift packet filter. Swift provides an elegant, fast, and efficient packet filtering technique to handle the challenge of high-speed network monitoring with dynamic filter updates. The key features of Swift lie in its low filter update latency and high execution efficiency. Swift attains these performance advantages by embracing several major design innovations: 1) a specialized CISC instruction set increases filter execution efficiency

and eliminates filter recompilation, resulting in significantly reduced filter update latency; 2) a simple computational model removes the necessity of security checking and improves filter update latency; and 3) SIMD extensions further boost filter execution efficiency.

Our extensive experiments have validated Swift’s efficacy and demonstrated the superiority of Swift against the *de facto* packet filter, BPF. For dynamic filtering tasks, the filter update latency of Swift is three orders of magnitude lower than that of BPF, and on each filter update, the number of packets missed by Swift is about two orders of magnitude less than that by BPF. For static filtering tasks, Swift runs as fast as BPF on simple filtering criteria, but is up to three times as fast as BPF on complex filtering criteria. Swift also performs much closer to optimized C filters than BPF.

The parallel processing nature of Swift instruction enables further performance improvement by leveraging the power of hardware SIMD instructions. We have studied the use of SSE instructions for accelerating filter execution. We have developed an enhanced Swift filtering engine and shown that it can significantly improve the performance of complex filters.

REFERENCES

- [1] S. McCanne and V. Jacobson, “The BSD packet filter: A new architecture for user-level packet capture,” in *Proc. Winter USENIX Tech. Conf.*, 1993, pp. 259–269.
- [2] S. Kornel, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, “Building a time machine for efficient recording and retrieval of high-volume network traffic,” in *Proc. ACM/USENIX IMC*, 2005, pp. 267–272.
- [3] S. Saroiu, S. D. Gribble, and H. M. Levy, “Measurement and analysis of spyware in a university environment,” in *Proc. USENIX NSDI*, 2004, pp. 141–153.
- [4] C. Partridge, A. C. Snoeren, W. T. Strayer, B. Schwartz, M. Condell, and I. Castineyra, “FIRE: Flexible intra-AS routing environment,” in *Proc. ACM SIGCOMM*, 2000, pp. 191–203.
- [5] V. Paxson, “Bro: A system for detecting network intruders in real-time,” *Comput. Netw.*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.
- [6] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, “FFPF: Fairly fast packet filters,” in *Proc. USENIX OSDI*, 2004, pp. 347–363.
- [7] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, “Operational experiences with high-volume network intrusion detection,” in *Proc. ACM CCS*, 2004, pp. 2–11.
- [8] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis, “xPF: Packet filtering for low-cost network monitoring,” in *Proc. IEEE HPSR*, 2002, pp. 121–126.
- [9] V. Jacobson, C. Leres, and S. McCanne, “Tcpdump(1),” *Unix Manual Page*, 1990.

- [10] J. M. Gonzalez, V. Paxson, and N. Weaver, "Shunting: A hardware/software architecture for flexible, high-performance network intrusion prevention," in *Proc. ACM CCS*, 2007, pp. 139–149.
- [11] J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The packet filter: An efficient mechanism for user-level network code," in *Proc. 11th ACM SOSP*, 1987, pp. 39–51.
- [12] G. Varghese, *Network Algorithmics—An Interdisciplinary Approach to Designing Fast Networked Devices*. San Mateo, CA: Morgan Kaufmann, 2005.
- [13] S. McCanne, C. Leres, and V. Jacobson, "Libpcap," Lawrence Berkeley Laboratory, Berkeley, CA, Feb. 2011 [Online]. Available: <http://www.tcpdump.org>
- [14] A. Biegel, S. McCanne, and S. L. Graham, "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *Proc. ACM SIGCOMM*, 1999, pp. 123–134.
- [15] J. van der Merwe, R. Caceres, Y. hua Chu, and C. Sreenan, "mmdump—A tool for monitoring Internet multimedia traffic," *Comput. Commun. Rev.*, vol. 30, no. 5, pp. 48–59, 2000.
- [16] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Proc. Winter USENIX Tech. Conf.*, 1994, pp. 153–165.
- [17] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "Pathfinder: A pattern-based packet classifier," in *Proc. USENIX OSDI*, 1994, pp. 115–123.
- [18] D. R. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. ACM SIGCOMM*, 1996, pp. 53–59.
- [19] WAND, "The DAG project," Univ. Waikato, Hamilton, NZ, Feb. 2011 [Online]. Available: <http://dag.cs.waikato.ac.nz/>
- [20] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt, "Architecture of a network monitor," in *Proc. IEEE PAM*, 2003, pp. 77–86.
- [21] J. Coppens, E. Markatos, J. Novotny, M. Polychronakis, V. Smotlacha, and S. Ubik, "Scampi—A scalable monitoring platform for the Internet," in *Proc. 2nd Int. Workshop Inter-Domain Perform. Simul.*, 2004, pp. 121–127.
- [22] F. Pong and N.-F. Tzeng, "Hashing round-down prefixes for rapid packet classification," in *Proc. USENIX Annu. Tech. Conf.*, San Diego, CA, 2009, pp. 71–85.
- [23] L. X. Yxuan Qi and B. Yang, "Packet classification algorithms: From theory to practice," in *Proc. IEEE INFOCOM*, 2009, pp. 648–656.
- [24] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, Jan.–Feb. 2005.
- [25] D. Pao, Y. K. Li, and P. Zhou, "Efficient packet classification using TCAMs," *Comput. Netw.*, vol. 50, pp. 3523–3535, 2006.
- [26] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE ANCS*, 2006, pp. 93–102.
- [27] S. Kumar and P. Crowley, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. ACM SIGCOMM*, 2006, pp. 339–350.
- [28] T. Hruby, K. van Reeuwijk, and H. Bos, "Ruler: High-speed packet matching and rewriting on NPUs," in *Proc. ACM/IEEE ANCS*, 2007, pp. 1–10.
- [29] A. Turner, "Tcpreplay," Feb. 2011 [Online]. Available: <http://tcpreplay.synfin.net/trac/>



Zhenyu Wu received the M.Sc. degree in computer science from the College of William and Mary, Williamsburg, VA, in 2005, and is currently pursuing the Ph.D. degree in computer science at the College of William and Mary.

His current research area focuses on data-center resource management and network optimization. His research interest also lies in system and network security, including but not limited to malware analysis, packet filters, and Internet chat and online game security.



Mengjun Xie (SM'08–M'10) received the Ph.D. degree in computer science from the College of William and Mary, Williamsburg, VA, in 2009.

He is an Assistant Professor of computer science with the University of Arkansas at Little Rock. His research interests include network security, information security, network systems, and operating systems.



Haining Wang (S'97–M'03–SM'09) received the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2003.

He is an Associate Professor of computer science with the College of William and Mary, Williamsburg, VA. His research interests lie in the areas of networking systems, security, and distributed computing.