A Scalable High-Performance Virus Detection Processor Against a Large Pattern Set for Embedded Network Security

Chieh-Jen Cheng, *Student Member, IEEE*, Chao-Ching Wang, *Member, IEEE*, Wei-Chun Ku, *Student Member, IEEE*, Tien-Fu Chen, *Member, IEEE*, and Jinn-Shyan Wang, *Member, IEEE*

***Abstract*—Contemporary network security applications generally require the ability to perform powerful pattern matching to protect against attacks such as viruses and spam. Traditional hardware solutions are intended for firewall routers. However, the solutions in the literature for firewalls are not scalable, and they do not address the difficulty of an antivirus with an ever-larger pattern set. The goal of this work is to provide a systematic virus detection hardware solution for network security for embedded systems. Instead of placing entire matching patterns on a chip, our solution is a two-phase dictionary-based antivirus processor that works by condensing as much of the important filtering information as possible onto a chip and infrequently accessing off-chip data to make the matching mechanism scalable to large pattern sets. In the first stage, the filtering engine can filter out more than 93.1% of data as safe, using a merged shift table. Only 6.9% or less of potentially unsafe data must be precisely checked in the second stage by the exact-matching engine from off-chip memory. To reduce the impact of the memory gap, we also propose three enhancement algorithms to improve performance: 1) a skipping algorithm; 2) a cache method; and 3) a prefetching mechanism.**

***Index Terms*—Algorithmic attacks, embedded system, memory gap, network security, virus detection.**

I. Introduction

NETWORK security has always been an important issue. End users are vulnerable to virus attacks, spam, and Trojan horses, for example. They may visit malicious websites or hackers may gain entry to their computers and use them as zombie computers to attack others. To ensure a secure network environment, firewalls were first introduced to block unauthorized Internet users from accessing resources in a private network by simply checking the packet head (MAC address/IP address/port number). This method significantly reduces the probability of being attacked. However, attacks such as spam, spyware, worms, viruses, and phishing target the application layer rather than the network layer. Therefore, traditional firewalls no longer provide enough protection. Many solutions, such as virus scanners, spam-mail filters, instant messaging protectors, network shields, content filters, and peer-to-peer protectors, have been effectively implemented. Initially, these solutions were implemented at the end-user side but tend to be merged into routers/firewalls to provide multi-layered protection. As a result, these routers stop threats on the network edge and keep them out of corporate networks.

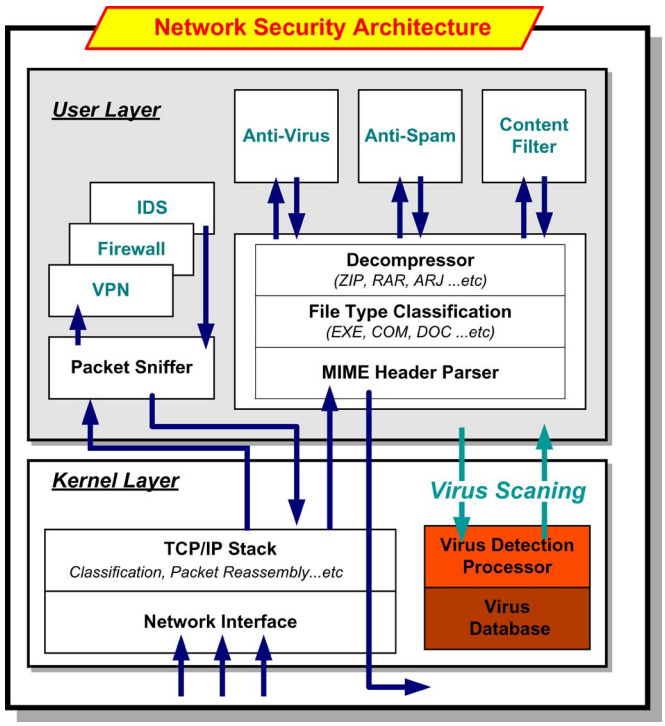


Fig. 1. Architecture of firewall router.

Fig. 1 shows a typical architecture of a firewall router. When a new connection is established, the firewall router scans the connection and forwards these packets to the host after confirming that the connection is secure. Because firewall routers focus on the application layer of the OSI model, they must reassemble incoming packets to restore the original connection and examine them through different application parsers to guarantee a secure network environment. For instance, suppose a user searches for

Manuscript received October 13, 2009; revised May 18, 2010 and November 20, 2010; accepted January 31, 2011. Date of publication March 28, 2011; date of current version April 06, 2012.

C.-J. Cheng and W.-C. Ku are with the Department of Computer Science and Information Engineering, National Chung Cheng University, Min-Hsiung Chia-Yi, Taiwan (e-mail: jjr88u@gmail.com; rogisterku@gmail.com).

C.-C. Wang and J.-S. Wang are with the Department of Electrical Engineering, National Chung Cheng University, Min-Hsiung Chia-Yi, Taiwan (e-mail: gaspard81@gmail.com; ieegsw@ccu.edu.tw).

T.-F. Chen is with the Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: tfchen@cs.nctu.edu.tw).

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/TVLSI.2011.2119382

information on web pages and then tries to download a compressed file from a web server. In this case, the firewall router might initially deny some connections from the firewall based on the target's IP address and the connection port. Then, the firewall router would monitor the content of the web pages to prevent the user from accessing any page that connects to malware links or inappropriate content, based on content filters. When the user wants to download a compressed file, to ensure that the file is not infected, the firewall router must decompress this file and check it using anti-virus programs. In summary, firewall routers require several time-consuming steps to provide a secure connection. However, even under numerous security constrictions, firewall routers are still required to provide high-speed transmission. Fortunately, most security-guaranteed programs use rule-based designs. Therefore, we have tried to develop a pattern matching processor to accelerate the detection speed. We call this design a virus detection processor because the database size it supports has reached the antivirus software level and is far greater than those of previous works.

The purpose of pattern matching is to check whether a text contains at least one of a given set of patterns. There are many algorithms [1]–[7] and accompanying hardware accelerators [4], [8]–[16] for fast pattern matching. One of the typical algorithms is the automation approach. This approach is based on Aho and Corasick's algorithm (AC) [1], which introduces a linear-time algorithm for multi-pattern search with a large finite-state machine. Its performance is not affected by the size of a given pattern set (the sum of all pattern lengths), but it requires a significant amount of memory due to state explosion. Experiments [17] have shown that the suboptimal AC algorithm requires 84.15 MB memory to represent Snort's rule set (4219 rules, as of December 2005). Even an Intel IXP2855 network processor (512 kB on-chip memory) must store such a pattern set in off-chip memory. Therefore, the memory hierarchy is the main factor in performance. Many previous studies have tried to lower memory requirements. In 2005, Lin Tan introduced a bit-split method [9] by splitting an 8-bit character into four 2-bit characters to construct the automaton. Their state machines are smaller than the original, and they have fewer fan-out states for each transaction. However, the bit-split method reads several memory blocks in parallel when matching patterns. Thus, it can only be implemented by on-chip memory because of its high memory read port requirements. Piti Piyachon and Yan Luo extended this concept [4] to the Intel IXP2855 network processor. For increasingly large pattern sets, an IBM team implemented an optimized AC algorithm [7] on the cell processor, and they discovered that the memory gap was the bottleneck. As a result, they modified the algorithm and used DMA to reduce the effect on the memory system.

In contrast, heuristic approaches are based on the Boyer-Moore [2] algorithm, which was introduced in 1977. Its key feature is the shift value, which shifts the algorithm's search window for multiple characters when it encounters a mismatch. The search window is a range of text exactly fetched by pattern matching algorithms for each examination. This algorithm performs better because it makes fewer comparisons than the naïve pattern-matching algorithm. At runtime, the Boyer-Moore algorithm uses a pattern pointer to locate a candidate position by assuming that a desired pattern exists at this position. The algorithm then shifts its search window to the right of this pattern. By default, desired patterns can exist in any position of a text; therefore, all positions in a text are candidate positions and must be examined. If the string of search windows does not appear in the pattern, the algorithm can shift the pattern pointer to the right and skip multiple characters from the candidate position to the end of the pattern without making comparisons. Based on this concept, Wu and Manber (WM) [18] modified the Boyer-Moore algorithm to search for multiple patterns. The WM algorithm is widely used in many applications, including Unix tools such as agrep and glimpse. However, the performance of both of these algorithms is bounded by the pattern length.

Software-based Bloom filters [3] were first described in 1970. These filters can determine whether an element is a non-member of a given set in a constant amount of time using several hash functions and a bit vector. The Bloom filter method is exceptionally space-efficient. In a typical case, the filter rate for 30 000 patterns reaches 90% and requires only 34.76 kB of memory. Although the Bloom filter rejects a non-member in constant time, it does not guarantee that an element is in a given set. False-positive problems necessitate a secondary method to verify the match. In brief, the Bloom filter does not perform pattern matching individually, except with an exact-matching method.

In 2004, Sarang *et al.* [4] presented a pattern-matching processor based on Bloom filters. They used multiple Bloom filters to check different-length prefixes of the pattern in parallel. This design needs 32 memory read ports because it uses 32 hash functions. However, most commonly used memory modules only have two ports: a read port and a write port. To lower the memory read port requirements, they divided a bit vector into several smaller vectors implemented by 140-block RAM of FPGA. The total memory size, then, is 70 kB for 10 038 patterns. The design also includes an analyzer that isolates false positives. The performance of this design can reach 2.46 Gb/s.

Some designs [4], [11]–[15], [19] take advantage of field-programmable gate array's (FPGA's) ability to be reconfigured to improve performance. Some of these designs [14], [15] are even based on non-deterministic finite automata (NFA) to handle complex regular expressions. These methods provide high throughput, but the maximum number of patterns they support is limited by the FPGA comparators. The Xilinx Virtex2-8000 FPGAs only support about 781 ClamAV rules. In 2004, to support an unlimited pattern count, Cho presented the idea [20], [21] of a two-phase architecture that implements a front-end filter with an FPGA and stores its full pattern database in a large memory. Later, Sourdi *et al.* implemented a perfect hashing function [19] on an FPGA to remove redundant memory accesses caused by address collisions. Some designs [10], [22] have used content-addressable memory (CAM) to improve engine filtering rates and to store the entire pattern database in a large external memory. Since then, pattern-matching designs have tended to use a two-phase architecture, in which one phase finds suspicious positions and the other phase precisely identifies patterns.

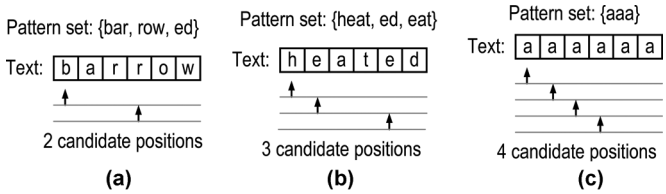


Fig. 2. Types of attack texts. (a) Deep-search attack; (b) sub-pattern attack; (c) an extreme case.

All of these designs provide more than 1-Gb/s performance, and some support even more than 10 Gb/s. However, with increasing pattern sets, it becomes more difficult to implement these designs in on-chip memory or dedicated circuits. Some designs attempt to store pattern sets in external memory, which is typically implemented by SDRAM or DDR, for their space requirements. Although DRAM technology has greatly improved over the last few decades, DRAM-based memories still require initial cycles before pumping out their first non-consecutive data. The gain only appears in consecutive readings of various sectors. A non-consecutive read operation of DDR memory still typically costs 25 $\sim$ 40 ns, compared to the 1 $\sim$ 3 ns working cycle of existing processors. Unfortunately, most pattern matching designs have irregular access to their memories. Thus, even though the kernels of these works are well designed, their performances are slowed dramatically by these long memory access processes [7], [11], [23], especially for filtering-based designs. For this reason, improving the filter rate and overlapping the access time are the two major trends. Some designs [5], [7], [17] try to use caches to overlap the access time.

Because the performance of the filtering engine of a two-phase architecture can be ten times better than the exact-matching, attackers can forge specific strings that trigger the architecture to launch a large number of false alarms by its front-end module, causing it to busy itself verifying these alarms precisely by its back-end module. Fig. 2 shows two typical types of attacks and one extreme case.

The first, shown in Fig. 2(a), is the most common and the easiest to build. This type of attack's text is constructed by patterns in the given pattern set, and therefore, these texts frequently cause the filtering engine to launch alarms and require verification of the exact-matching engine. Fig. 2(a) illustrates a typical attack text "barrow" built by the pattern set {bar, row, ed}. Two candidate positions cause exact-matching to take a long time to traverse its data structure. For the ClamAV, the average pattern length is 64 characters; thus, attack texts constructed with this method have a 1/64 chance of launching an exact match for each examination.

The second case shown in Fig. 2(b) is relatively rare, but it provides a higher-density attack text than the first. If the given pattern set includes a pattern that contains the prefix of another pattern, the attack texts built by these patterns provide more candidate positions in the same length. The patterns "heat" and "eat" in Fig. 2(b) are an example. For the same length, the text launches many more alarms than the text in Fig. 2(a).

Fig. 2(c) shows an extreme case that combines the first and second types: a pattern is constituted by a serial of characters "a"; the constitution of the attack text is also same as the pattern but is longer. As a result, the attack text in Fig. 2(c) can be considered to constitute a multiple of this pattern. In addition, this pattern contains the prefix of itself. Two of these patterns can compose the third. Therefore, this attack text can be considered to be a special case that combines by the first and second types and causes exact-matching for every position of itself. This extreme case dramatically lowers the performance. Although it can be avoided by choosing the pattern set well, the first type of attack text can still be built easily if the pattern sets are known by the attackers. Thus, a two-phase architecture is vulnerable to algorithmic attacks.

Related works have focused on algorithms and have even developed specialized circuits to increase the scanning speed. However, these works have not considered the interactions between algorithms and memory hierarchy. Because the number of attacks is increasing, pattern-matching processors require external memory to support an unlimited pattern set. This method makes the memory system the bottleneck. However, when the pattern set is already intractably large, a perfect solution is unattainable. A more realistic goal is to provide high performance in most cases while still performing reasonably well in the worst case. With an eye toward high performance, updatability, unlimited pattern sets and low memory requirements, we present a two-phase architecture that uses off-chip memory to support a large pattern set.

Our major contribution is to propose: 1) a shift-signature table and 2) a trie-skip mechanism to improve the performance and cushion the blow of the impact on memory gap for this two-phase architecture. First, we re-encode the shift table and Bloom filter to merge them into the same space, the shift-signature table. The new table not only maintains the shift value of properties but also avoids reducing the filter rate for a large scale pattern set. In the same space, 32 kB SRAM in this case, the filter rate of our approach is improved from 10.9% to 6.9% compared to the Bloom filter. Second, the trie-skip mechanism avoids performance reduction during malicious attacks. Our proposed trie-skip mechanism overcomes these two attacks by skip values and jump nodes. With these two fields, we use the new trie structure suitable for prefetched and cached to reduce the off-chip memory access just by rearranging the trie structure. Experiments show that our approach still provides reasonable 0.71-Gb/s performance, as compared to 0.33 Gb/s without optimization, for 30 000 patterns under heavy deep-search attacks.

The remainder of this paper is organized as follows. Section II presents the design of our two-phase pattern-matching processor. Sections III and IV detail the design of the proposed filtering and exact-matching engines. Section V evaluates the performance of our processor. Section VI demonstrates an FPGA development environment and the tapeout result. Finally, we compare our design with previous works in Section VII and present our conclusions in Section VIII.

II. Virus Detection Processor

Our design, shown on the right side of Fig. 3, is a two-phase pattern-matching architecture mostly comprising the filtering engine and the exact-matching engine. The filtering engine is

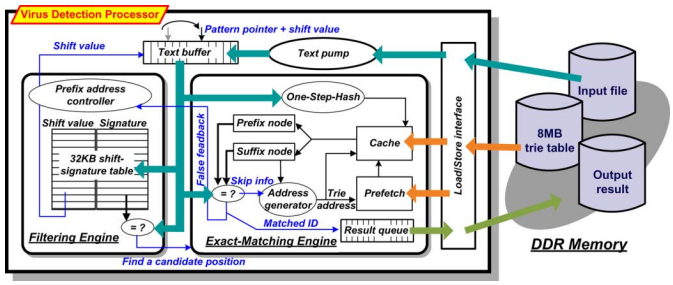


Fig. 3. Virus detection processor architecture.

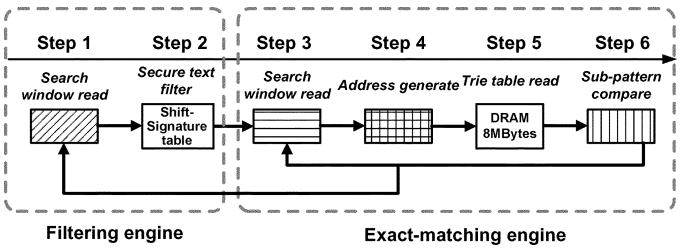


Fig. 4. Two-phase execution flow.

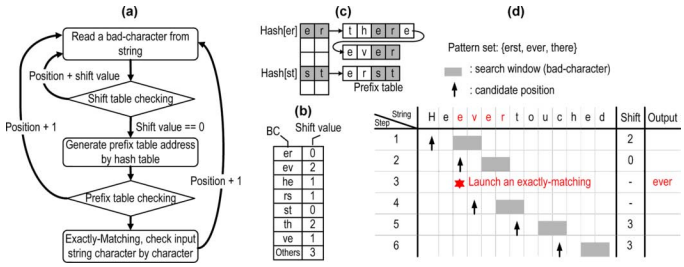


Fig. 5. Wu-Manber matching process. (a) Matching flow; (b) shift table; (c) hash table + prefix table; (d) matching process.

a front-end module responsible for filtering out secure data efficiently and indicating to candidate positions that patterns possibly exist at the first stage. The exact-matching engine is a back-end module responsible for verifying the alarms caused by the filtering engine. Only a few unsaved data need to be checked precisely by the exact-matching engine in the second stage.

Both engines have individual memories for storing significant information. For cost reasons, only a small amount of significant information regarding the patterns can be stored in the filtering engine's on-chip memory. In this case, we used a 32-kB on-chip memory for the ClamAV virus database, which contained more than 30 000 virus codes and localized most of the computing inside the chip.

Conversely, the exact-matching engine not only stores the entire pattern in external memory but also provides information to speed up the matching process. Our exact-matching engine is space-efficient and requires only four times the memory space of the original size pattern set. The size of a pattern set is the sum of the pattern length for each pattern in the given pattern set; in other words, it is the minimum size of the memory required to store the pattern set for the exact-matching engine. In this case, 8 MB of off-chip memory was required for the ClamAV virus database (2 MB).

The proposed exact-matching engine also supports data prefetching and caching techniques to hide the access latency of the off-chip memory by allocating its data structure well. The other modules include a text buffer and a text pump that prefetches text in streaming method to overlap the matching progress and text reading. A load/store interface was used to support bandwidth sharing.

This proposed architecture has six steps shown in Fig. 4 for finding patterns. Initially, a pattern pointer is assigned to point to the start of the given text at the filtering stage. Suppose the pattern matching processor examines the text from left to right. The filtering engine fetches a piece of text from the text buffer according to the pattern pointer and checks it by a shift-signature table. If the position indicated by the pattern pointer is not a candidate position, then the filtering engine skips this piece of text and shifts the pattern pointer right multiple characters to continue to check the next position. The shift-signature table combines two data structures used by two different filtering algorithms, the Wu-Manber algorithm and the Bloom filter algorithm, and it provides two-layer filtering. If both layers are missing their filter, the processor enters the exact-matching phase. The next section has details about the shift-signature table.

After an alarm caused by the filtering engine, the exact-matching engine precisely verifies this alarm by retrieving a trie structure [24]. This structure divides a pattern into multiple sub-patterns and systematically verifies it. The exact-matching engine generally has four steps for each check. First, the exact-matching engine gets a slice of the text and hashes it to generate the trie address. Then, the exact-matching engine fetches the trie node from memory. This step causes a long latency due to the access time of the off-chip memory. Finally, the exact-matching engine compares the trie node with this slice. When this node is matched, the exact-matching engine repeatedly executes the above steps until it matches or misses a pattern. The pattern matching processor then backs out to the filtering engine to search for the next candidate. The details of table generation and matching flow are explained in the following sections.

III. Filtering Engine (FE)

Designs that feature filters indicate that the action behind these filters is costly and necessary. In this work, the overall performance strongly depends on the filtering engine. Providing a high filter rate with limited space is the most important issue. We introduce two classical filtering algorithms for pattern matching in the following sections. We then show how to merge their structures in the same space to improve the filter rate.

A. Wu-Manber Algorithm

The Wu-Manber algorithm is a high-performance, multi-pattern matching algorithm based on the Boyer-Moore algorithm. It builds three tables in the preprocessing stage: a shift table, a hash table and a prefix table. The Wu-Manber algorithm is an exact-matching algorithm, but its shift table is an efficient filtering structure. The shift table is an extension of the bad-character concept in the Boyer-Moore algorithm, but they are not identical. The matching flow is shown in Fig. 5(a). The matching

flow matches patterns from the tail of the minimum pattern in the pattern set, and it takes a block B of characters from the text instead of taking them one-by-one. The shift table gives a shift value that skips several characters without comparing after a mismatch. After the shift table finds a candidate position, the Wu-Manber algorithm enters the exact-matching phase and is accelerated by the hash table and the prefix table. Therefore, its best performance is $O(BN/m)$ for the given text with length N and the pattern set, which has a minimum length of m. The performance of the Wu-Manber algorithm is not proportional to the size of the pattern set directly, but it is strongly dependent on the minimum length of the pattern in the pattern set. The minimum length of the pattern dominates the maximum shift distance $(m - B + 1)$ in its shift table. However, the Wu-Manber algorithm is still one of the algorithms with the best performance in the average case.

For the pattern set {erst, ever, there} shown in Fig. 5(d), the maximum shift value is three characters for $B = 2$ and $m = 4$. The related shift table, hash table and prefix are shown in Fig. 5(b) and Fig. 5(c). The Wu-Manber algorithm scans patterns from the head of a text, but it compares the tails of the shortest patterns. In step 1, the arrow indicates to a candidate position that a wanted pattern probably exists, but the search window (gray bar) is actually the character it fetches for comparison. According to $\mathrm{shift}[\mathrm{ev}] = 2$, the arrow and search window are shifted right by two characters. Then, the Wu-Manber algorithm finds a candidate position in step 2 due to $\mathrm{shift}[\mathrm{er}] = 0$. Consequently, it checks the prefix table and hash table to perform an exact-matching and then outputs the "ever" in step 3. After completing the exact match, the Wu-Manber algorithm returns to the shifting phase, and it shifts the search window to the right by one character to find the next candidate position in step 4. The algorithm keeps shifting the search window until touching the end of the string in step 6.

B. Bloom Filter Algorithm

A Bloom filter is a space-efficient data structure used to test whether an element exists in a given set. This algorithm is composed of k different hash functions and a long vector of v bits. Initially, all bits are set to 0 at the preprocessing stage. To add an element, the Bloom filter hashes the element by these hash functions and gets k positions of its vector. The Bloom filter then sets the bits at these positions to 1. The value of a vector that only contains an element is called the signature of an element. To check the membership of a particular element, the Bloom filter hashes this element by the same hash functions at run time, and it also generates k positions of the vector. If all of these k bits are set to 1, this query is claimed to be positive, otherwise it is claimed to be negative. The output of the Bloom filter can be a false positive but never a false negative. Therefore, some pattern matching algorithms based on the Bloom filter must operate with an extra exact-matching algorithm. However, the Bloom filter still features the following advantages: 1) it is a space-efficient data structure; 2) the computing time of the Bloom filter is scaled linearly with the number of patterns; and 3) the Bloom filter is independent of its pattern length.

Fig. 6(a) describes a typical flow of pattern matching by Bloom filters. This algorithm fetches the prefix of a pattern

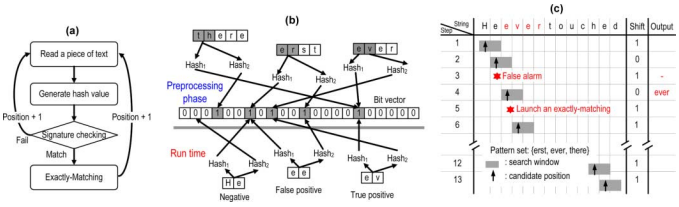


Fig. 6. Bloom filter matching process. (a) Matching flow; (b) bit-vector building; (c) matching process.

from the text and hashes it to generate a signature. Then, this algorithm checks whether the signature exists in the bit vector. If the answer is yes, it shifts the search window to the right by one character for each comparison and repeats the above step to filter out safe data until it finds a candidate position and launches exact-matching. Fig. 6(b) shows how a Bloom filter builds its bit vector for a pattern set {erst, ever, there} for two given hash functions. The filter only hashes all of the pattern prefixes at the preprocessing stage. Multiple patterns setting the same position of the bit vector are allowed. Fig. 6(c) shows an example of the matching process. The arrows indicate the candidate positions. The gray bars represent the search window that the Bloom filter actually fetches for comparison. Both the candidate position and search window are aligned together. Thus, the Bloom filter scans and compares patterns from the head rather than the tail, like the Wu-Manber algorithm. In step 1, the filter hashes "He" and mismatches the signature with the bit vector. The filter then shifts right 1 character and finds the next candidate position. For the search window "ee", the Bloom filter matches the signature and then causes a false alarm to perform an exact-matching in steps 2 and 3. The filter then returns to the filtering stage and shifts one character to the right in step 4, which launches a true alarm for the pattern "ever". Finally, the Bloom filter filters the rest of text and finds nothing.

C. Shift-Signature Algorithm

The proposed algorithm re-encodes the shift table to merge the signature table into a new table named the shift-signature table. The shift-signature table has the same size as the original shift table, as its width and length are the same as the original shift table. There are two fields, S-flag and carry, in the shift-signature table. The carry field has two types of data: a shift value and a signature. These two data types are used by two different algorithms. Thus, the S-flag is used to indicate the data type of a carry. The filtering engine can then filter the text using a different algorithm while providing a higher filter rate.

The method used to merge these two tables is described as follows. First, the algorithm generates two tables, a shift table and signature table, at the preprocessing stage. The generation of the shift table is the same as in the Wu-Manber algorithm. The shift table is used as the primary filter. The signature table could be considered a set of the bit vector of the Bloom filter, and it is used for the second-level filtering. The signature table's generation is similar to the Bloom filter but is not identical; it hashes the tail characters of patterns to generate their signatures instead of the prefix. Generated signatures are mapped onto the signature table and indexed by bad-characters, which have shift

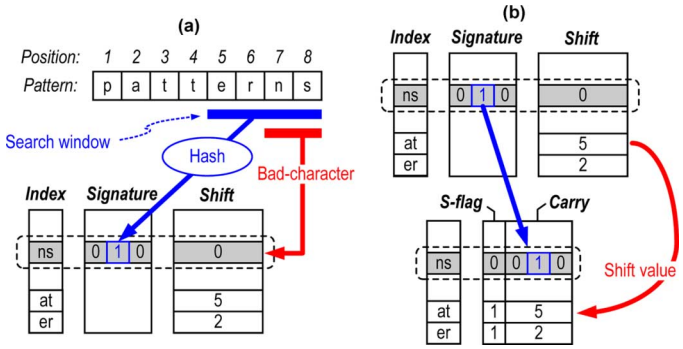


Fig. 7. Table generation and re-encoding of shift-signature algorithm. (a) Table generation; (b) table re-encoding.

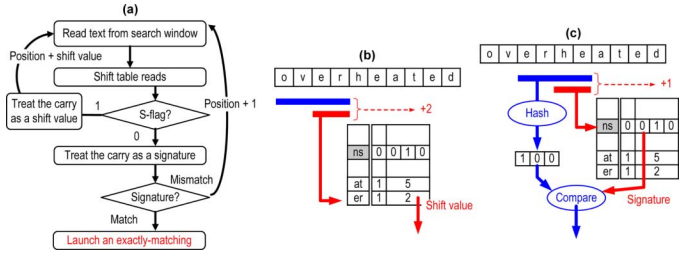


Fig. 8. Matching flow and filtering example. (a) Filtering flow; (b) shift filtering; (c) signature filtering.

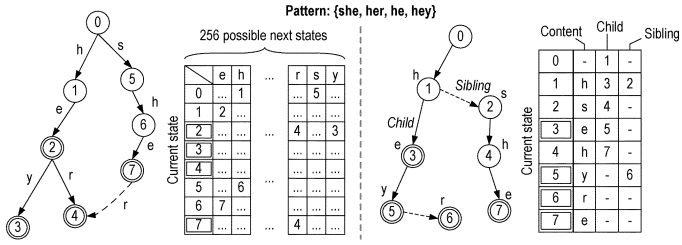


Fig. 9. (a) Compact Trie. (b) FSM of an AC algorithm.

values of zero in the shift table. In other words, a pattern is assigned a zero shift value in the shift table by its last characters, and it uses the same index to locate its signature in the signature table. After the shift table and signature table are generated, the algorithm re-encodes the shift value into two fields: an S-flag and a carry in the shift-signature table. The S-flag is a 1-bit field used to indicate the data type of the carry. Two data types, shift value or signature, are defined for a carry. The size and width of the shift-signature table are the same as those of the original shift table. To merge these two tables, the algorithm maps each entry in the shift table and signature table onto the shift-signature table. For the non-zero shift values, the S-flags are set, and their original shift values are cut out at 1-bit to fit their carries. Conversely, for the zero shift values, their S-flags are clear, and their carries are used to store their signatures. In this method, all of the entries in the shift-signature table contribute to the filtering rate at run time. Because of the address collision of bad-characters, most entries contain less than half of the maximum shift distance for a large pattern set. Therefore, although this method sacrifices the maximum shift distance, the filter rate is not reduced but rather improved.

Fig. 7(a) shows an example of generating the shift and signature tables. Suppose the length of the shortest pattern "patterns" in the pattern set is 8 characters. The size of the bad-character is 2 characters, thus the maximum shift distance is $8 - 2 + 1 = 7$ characters. Seven possible bad-characters ("pa", "at", "tt", "te", "er", "rn", "ns") are defined according to the Wu-Manber algorithm, and their shift values are 6, 5, 4, 3, 2, 1, and 0. Before replacement, the algorithm first builds the signature table. For each pattern, the algorithm hashes the tail characters of a pattern (blue bar) to generate its signature. The signature is then assigned to the signature table indexed by the bad-character "ns". For multiple signatures mapped to the same entry, the entry stores the results of the OR operation of these signatures. In this work, we only use one hash function because of the space limitation of the signature table. The method of merging the shift table and signature table is shown in Fig. 7(b). The $\mathrm{shift}[\mathrm{ns}]$ is replaced by its signature ("010" in binary) because its shift value is zero. In contrast, the $\mathrm{shift}[\mathrm{at}] = 5$ and $\mathrm{shift}[\mathrm{er}] = 2$ keep their shift values in the shift-signature table.

The filtering flow is shown in Fig. 8(a). For the pattern set {patterns}, Fig. 8(b) and Fig. 8(c) illustrate how the filtering engine filters out the given text. The filtering engine fetches the text from the search window (blue bar), as shown in Fig. 8(c). One part of the fetched text (red bar), shown in Fig. 8(b), is used as a bad character to index the shift-signature table. If the S-flag is set, the carry is treated as a shift value. As a result, the filtering engine shifts the candidate position to the right by two characters for the text "overhead", as shown in Fig. 8(b). Conversely, if the S-flag is clear, the carry is treated as a signature. The filtering engine hashes the fetched text and matches it with the signature read from the shift-signature table. Fig. 8(c) indicates that the fetched text "he" has the same index as the bad-character "ns", but it fails to match the signature. Thus, the filtering engine shifts the candidate position to the right by one character to provide second-level filtering.

IV. Exact-Match Engine (EME)

The EME must verify the false positives when the filtering engine alerts. It also precisely identifies patterns for upper-layer applications. Most exact-match algorithms use the two kinds of trie structures shown in Fig. 9, loose and compact tries, to establish their pattern databases. Both trie structures have their merits. The AC algorithm uses loose tries, which check each input character in a constant amount of time because of their fan-out states for all possible input characters. Thus, the input data do not affect the AC-based algorithm's performance, but their memory requirements increase exponentially with pattern size. Unlike loose tries, compact tries construct pattern databases with two pointers, sibling and child, to reduce their memory requirements. However, this method has potential performance problems because it may redundantly search link lists formed by sibling pointers. Despite this limitation, compact tries are still highly practical because, in practice, attack texts are not easy to generate. Attacks can be avoided by removing patterns that cause attacks before constructing the pattern database. For this reason, we use compact tries as our exact-matching engine's algorithm, and we propose several solutions to mitigate the effect of algorithmic attacks.

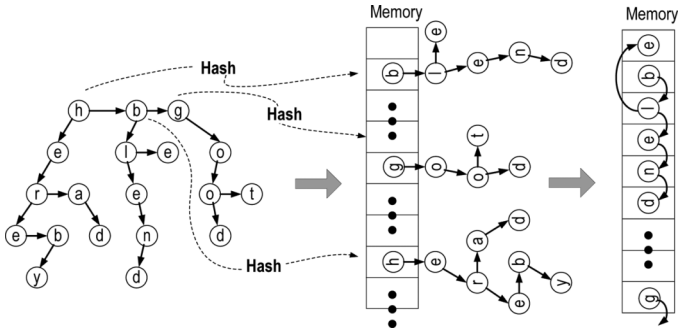


Fig. 10. One-step hash for a single-root problem.

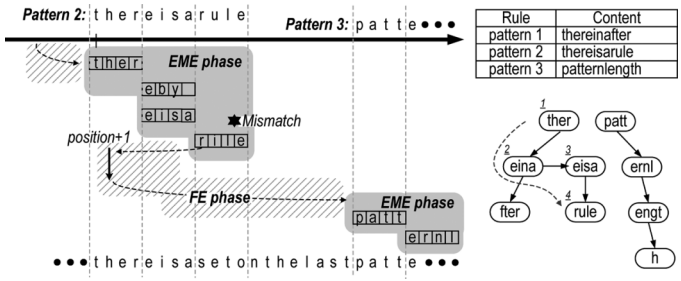


Fig. 12. Exact-matching with a multiple-character compact trie.

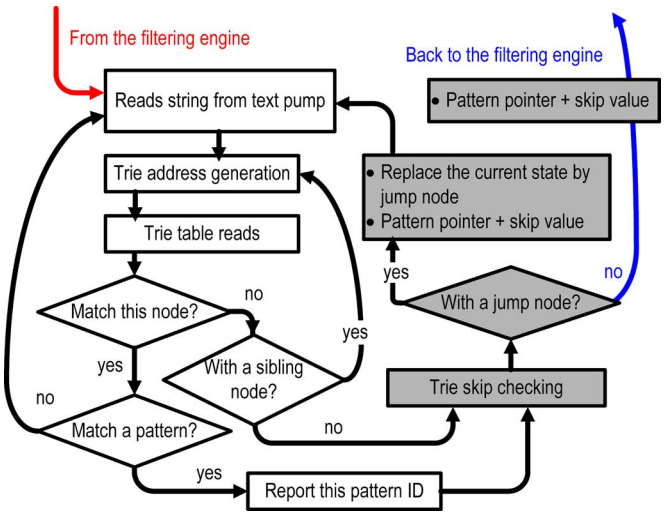


Fig. 11. Exact-matching flow.

A. Trie Table Generation and One-Step Hash

A traditional compact trie usually has only one entrance, as shown in Fig. 10. However, for multiple patterns, this method requires a significant amount of time to search the prefix node of a pattern in the entrance's sibling list. To reduce search time, we divide a huge trie into several lightweight tries to generate multiple entrances by hashing the root node of each lightweight trie. The generated hash values are root addresses for each lightweight trie tree. Therefore, the exact-matching engine can easily get more entrances and have first nodes with short sibling lists. In a DRAM, the read time for a consecutive access is shorter than that for non-consecutive access memory. Mapping nodes that belong to the same pattern in the neighborhood efficiently lower access time. In this way, prefetching helps the exact-matching engine overlap computation with memory access time. Therefore, simply hashing the trie to generate multiple entrances and carefully arranging the data in the memory efficiently solves the memory gap problem at the algorithmic level.

B. Exact-Matching Flow

Fig. 11 illustrates the flow that exact-matching engine verifies an alarm triggered by the filtering engine.

Step 1) This engine fetches a piece of text from the text pump according to the address given by the filter engine.

Step 2) If this is the first reading of the trie table for this alarm, then this engine hashes this text to generate the root address of its trie tree. Otherwise, it chooses the sibling pointer of the trie node that the engine last read as the new address.

Step 3) This engine fetches the trie node from memory according to the address provided by the above step.

Step 4) The engine compares this piece of text with the trie node. If the content of the trie node is the same as the piece of text, it jumps to Step 6. Otherwise, the engine continues and checks whether this node has a sibling pointer.

Step 5) If a sibling exists, the engine jumps to Step 3 and fetches its sibling node, according to the pointer. Otherwise, it jumps to Step 7 to execute the trie-skip mechanism.

Step 6) If a pattern exists at this node, the engine reports the pattern ID and goes to Step 7. Otherwise, it shifts the pattern pointer right and back to Step 1 to repeatedly examine the next piece of text.

Step 7) The pattern pointer shifts right several characters by the skip value. If the node has a jump node, the engine updates its state using this jump node and fixes its search window by the suffix offset. The engine then returns to Step 1. Otherwise, the engine finishes the verification and hands control back to the filtering engine.

Fig. 12 shows an example of exact matching without the trie-skip mechanism. After the filtering engine launches an alert, the exact-matching engine gets a slice of the input text "ther" and hashes this text to generate the root address. The engine then reads the root node from memory, compares it with the text and successfully matches the string at step 1. The exact-matching engine continues to compare the child node "eina", indicated by the child pointer of the root node at step 2, but it mismatches its child node. However, the child node "eina" has a sibling node; thus, it keeps comparing its sibling node at step 3. The engine then mismatches the node "rule" with the text "seto" at step 4. The exact-matching engine then returns control to the filtering engine to find the next candidate position. Finally, the pattern-matching matches pattern 3 at the tail of the text.

Observing the matching flow of the exact-matching engine in Fig. 12, we notice that the filtering engine can only shift one character right to the next candidate position after the exact-matching engine mismatches. This method may be vulnerable

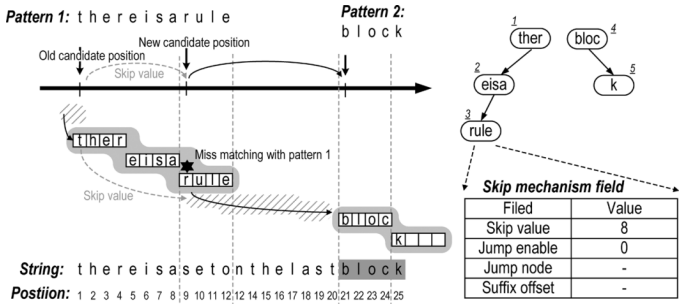


Fig. 13. Skip value of skip mechanism.

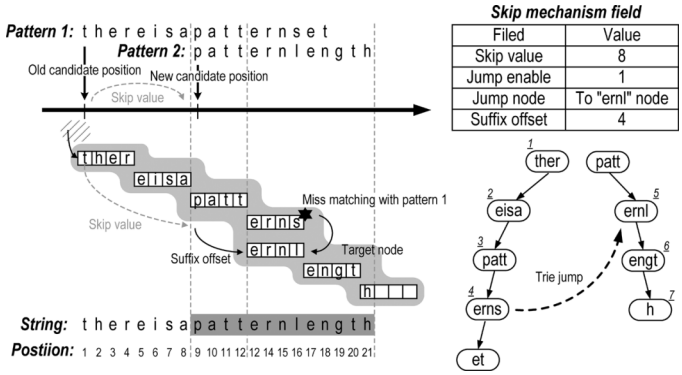


Fig. 14. Jump node of skip mechanism.

to algorithmic attacks. The concept of a failure state, an obvious solution to this problem, could not be implemented directly on the compact trie. Just like the node "rule" in Fig. 12, the exact-matching engine cannot be sure where to jump when the input string is not "rule." The engine cannot enter its failure state immediately when a mismatch occurs because the compact trie does not contain failure states for all possible input strings. However, the engine can still jump to its failure state based on its previously matched nodes: "ther" and "eisa". The following section describes cases for the failure state and how we implement it in the compact trie.

C. Trie-Skip Mechanism for Algorithmic Attack Avoidance

There are two cases that trie-skip mechanism can remove redundant comparison. The first case, shown in Fig. 13, has two wanted patterns in its database. Pattern 2 does not exist in any part of pattern 1. When the exact-matching engine mismatches pattern 1, the filtering engine should restart to find the next candidate from the point where the exact-matching engine mismatched because the string (nodes 1 and 2) that has already been compared cannot contain another pattern. To implement this concept, our trie node contains a precalculated skip value that lets the filtering engine find the next candidate position by skipping after a mismatch occurs. In this case, the exact-matching engine can skip eight characters to search for the next candidate after a mismatch.

The second case contains two wanted patterns such that one is a prefix of the other. In this case, the exact-matching engine does not go back to the filtering engine after a mismatch occurs. Instead, the engine launches another exact match because the mismatched point might contain another pattern. Take Fig. 14 as an example. First, the filtering engine finds a suspicious position and launches an exact match, but the exact-matching engine finds a fault alert for pattern 1 after mismatching at node 4. In the traditional approach, the exact-matching engine triggers the filtering engine to find the next candidate position. However, the string from node 1 to 3 contains the prefix of pattern 2 in its tail. Therefore, exact matching would begin again at position 9 after the exact-matching engine backs out to the filtering engine. According to the failure state, however, the exact-matching engine can continue to search for pattern 2 after mismatching pattern 1. To implement this concept, we propose a trie-skip mechanism. Each state in our mechanism has only a single failure state pointing to the most suitable node rather than unrolling all possible failure states. Therefore, our method provides a failure-state mechanism while preserving space efficiency.

The trie-skip mechanism is implemented by four major fields (shown in Fig. 14) for each trie node. The skip value is eight, meaning that the closest candidate pattern is behind the current candidate by eight characters. However, because the exact-matching engine does not always start from the beginning of a pattern, the jump node field indicates the first node that the exact-matching engine should compare for the new candidate pattern after a mismatch occurs. The exact-matching engine can continue to compare from the node "ernl" of pattern 2 because the node "patt" of pattern 2 has already been checked. The suffix offset fixes the search window and notifies the exact-matching engine, which fetches characters behind the new pattern pointer with four characters. The jump-enable bit and jump node are used to implement this jumping idea.

V. Experiments

We assume that the trie table is stored in DDR memory. The latency for a random access, comprising row access (RAS), column access (CAS) and precharge time, is typically from 25 to 40 ns [23]. We take 40 ns as our worst-case memory gap. Our goal is to provide a pattern matching processor with performance better than 1 Gb/s using memory with 40 ns access time for the 30 000 ClamAV virus codes. The input text file is an 8.7 MB file merged from two Windows EXE files, a PDF file, a JPEG file, a WMA file, a MP3 file, a Word DOC file, and a randomly generated file. To evaluate our proposed architecture, we first analyze the filtering rates of the Bloom filter, Wu-Manber algorithm, and our proposed shift-signature table to determine the minimum on-chip memory required to obtain this performance. We then assess their average performance on different numbers of patterns using this memory size. The last experiment demonstrates their performance for two types of algorithmic attacks. Based on the results of the chip implemented by the TSMC.13-μm 1P8M process, we assume pattern processors in the following experiments operate at a clock speed of 534 MHz.

A. Analysis of Shift Rate

Because the ClamAV pattern set only has about 30 000 examples and does not have many patterns of sufficient length for

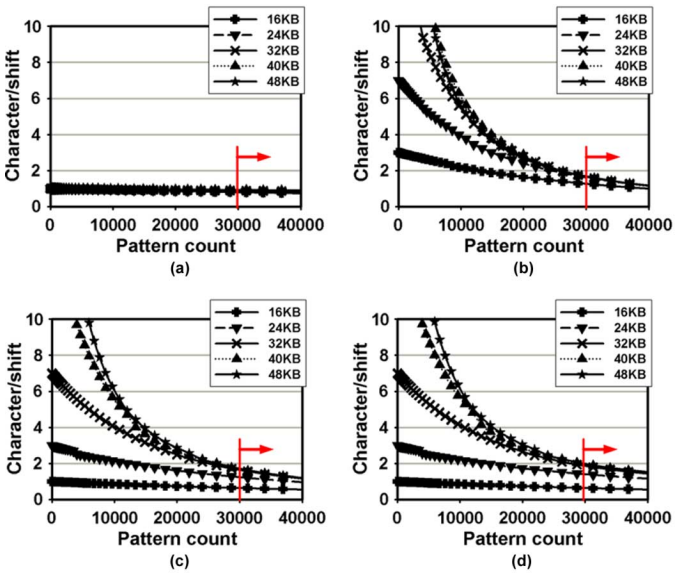
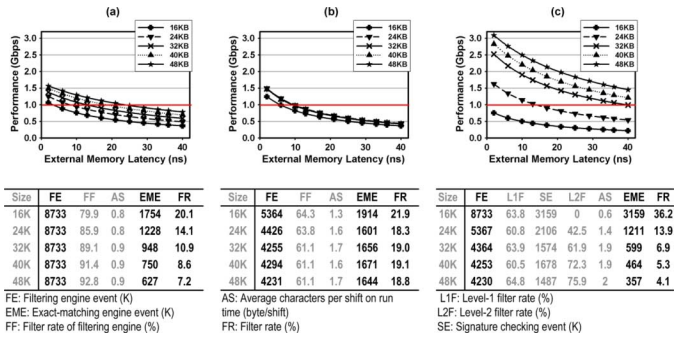


Fig. 15. Analysis of shift rate for random patterns. (a) Bloom filter; (b) Wu-Manber; (c) shift-signature Table (L1); (d) shift-signature Table $(\text{total} = L1 + L2)$.

the Wu-Manber algorithm, we use a random pattern set in this experiment. The x-axis of Fig. 15 is the number of patterns that we used. The y-axis is the average number of characters that were shifted for each check during the run. Fig. 15(a) indicates that the shift rate of the Bloom filter algorithm is very stable. Its average shift value is less than 1 character because the Bloom filter algorithm filters texts character by character.

The results of the Wu-Manber algorithm with different table sizes, 16 kB (2×2^{16} bits), 24 kB (3×2^{16} bits), 32 kB (4×2^{16} bits), 40 kB (5×2^{16} bits), and 48 kB (6×2^{16} bits), are shown in Fig. 15(b). Ideally, they would be able to shift up to 3, 7, 15, 31, and 63 characters, respectively, for each examination, but their average shift values decrease dramatically with the number of supplied patterns. When there are more than 30 000 patterns, their average shift values are almost the same. Bad-character collision limits the maximum shift value actually recorded in the shift table. Therefore, increasing the table size does not benefit the average shift value for a large pattern set. However, the average shift value remains higher than the Bloom filter.

Our proposed shift-signature table, shown in Fig. 15(c) and (d), also has this feature. Fig. 15(c) only uses the shift concept, but Fig. 15(d) uses both the shift concept and signature concept. Notably, the average shift value for 16 kB is lower than that in the 16-kB table of the Wu-Manber algorithm. The field width for the shift value in our proposed method is 1 bit shorter than that in the Wu-Manber algorithm, and therefore, the ideal maximum shift value is half that of the Wu-Mamber algorithm. The maximum shift values are 1, 3, 7, 15, and 31 characters for table sizes of 16, 24, 32, 40, and 48 kB, respectively. Otherwise, most of their average shifted characters are similar to the Wu-Mamber algorithm. We concentrate on their performance for 30 000 patterns in the following experiment.



Size	FE	FF	AS	EME	FR
16K	8733	79.9	0.8	1754	20.1
24K	8733	85.9	0.8	1228	14.1
32K	8733	89.1	0.9	948	10.9
40K	8733	91.4	0.9	750	8.6
48K	8733	92.8	0.9	627	7.2

FE: Filtering engine event (K)
EME: Exact-matching engine event (K)
FF: Filter rate of filtering engine (%)

Size	FE	FF	AS	EME	FR
16K	5364	64.3	1.3	1914	21.9
24K	4426	63.8	1.6	1601	18.3
32K	4255	61.1	1.7	1656	19.0
40K	4294	61.1	1.6	1671	19.1
48K	4231	61.1	1.7	1644	18.8

AS: Average characters per shift on run time (byte/shift)
FR: Filter rate (%)

Size	FE	L1F	SE	L2F	AS	EME	FR
16K	8733	63.8	3159	0	0.6	3159	36.2
24K	5367	60.8	2106	42.5	1.4	1211	13.9
32K	4364	63.9	1574	61.9	1.9	599	6.9
40K	4253	60.5	1678	72.3	1.9	464	5.3
48K	4230	64.8	1487	75.9	2	357	4.1

L1F: Level-1 filter rate (%)
L2F: Level-2 filter rate (%)
SE: Signature checking event (K)

Fig. 16. On-chip memory requirement for 30 000 random rules. (a) Bloom filter; (b) Wu-Manber; (c) shift-signature Table.

B. Requirements for On-Chip Memory

The experiment demonstrates why the average shift of our proposed method is better than that in the Wu-Manber algorithm and determines the table size required for good performance on the ClamAV pattern set. Fig. 16(a) indicates that the Bloom filter only requires a 16-kB table to reach 1-Gb/s scanning speed with ideal memory. Performance improves monotonically with table size, and performance degrades when the latency of external memory increases.

The performance of the Wu-Manber algorithm shown in Fig. 16(b) does not increase with table size but rather remains almost unchanged. The experiment indicates that only 1.3 to 1.7 characters can be shifted for each check during the run. However, the filtering engine based on the Wu-Manber algorithm still has fewer checks (FE events) than the bloom filter because of its shifting feature. Conversely, the Wu-Manber launches more exact-matching events (EME events) than the Bloom filter because of the collision of bad characters. Finally, the performance of the Wu-Manber algorithm is worse than that of the Bloom filter because of the long access time of the external memory.

At the same table size, Fig. 16(c) shows that most cases perform better with our method than with the Bloom filter or Wu-Manber algorithm, with the exception of the 16-kB table. The 16-kB table is a 2-bit wide table, where one bit is the S-flag and the other is a shift value. In this case, it can only shift a maximum of one character, and therefore, its effect is roughly the same as an 8-kB Bloom filter but with degraded performance. However, when the table size is increased, the filtering engine is given the opportunity to shift multiple characters. The experimental results show that the re-encoded shift-signature table has almost the same average run-time shift value and filter rate (L1F) as the Wu-Manber algorithm. Although more SE alarms in Fig. 16(c) are caused by the first level in our method than the EME alarms in Fig. 16(b) caused by the Wu-Manber algorithm, the signature-checking mechanism filters out most alarms. As a result, the shift-signature table generates fewer exact-matching events than the others. In addition, it also produces fewer filtering events than the Bloom filter. Therefore, its performance is better than that of the other methods. Our goal is to produce a pattern matching processor that performs at a rate better than 1 Gb/s using external memory with 40 ns

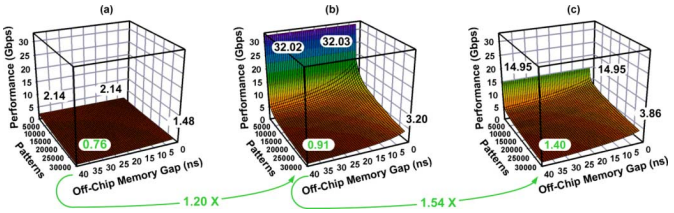


Fig. 17. Performance and load distribution for different filtering engines. (a) Bloom filter; (b) Wu-Manber; (c) shift-signature Table.

average access time. Therefore, we chose a 32-kB table size for the subsequent experiments.

C. Average Performance Analysis

In this experiment, we scan the previous test file with different filtering engines to measure the effect of the memory gap on performance. All of the engines use the same exact-match engine on the back-end. The X-axis of Fig. 17 gives the number of ClamAV virus signatures, and the Y-axis is the latency of each memory access. We compared Bloom filters, shift tables, and our shift-signature table, all implemented with a 32-kB on-chip memory. Fig. 17 shows that the performance of Bloom filters is not strongly related to the pattern count or the memory gap. Even the best performance is only 2.14 Gb/s at 534 MHz. In contrast, the ideal performance of the shift table reaches 32.03 Gb/s, but its performance heavily depends on memory latency and pattern count. However, the performance of the shift table is better than that of the Bloom filter in the ideal cases (3.20 Gb/s versus 1.48 Gb/s) for 30 000 patterns. Because the width of a shift value of our proposed shift-signature table is 1 bit shorter than in the Wu-Manber algorithm, the maximum performance of our proposed algorithm is half that of the Wu-Manber algorithm, but this special case only occurs for a few patterns. For the large-scale pattern set, our proposed shift-signature table improves the performance of the Wu-Manber algorithm by 54%, from 0.91 to 1.40 Gb/s, when using a second-level filter.

D. Performance Analysis Under Algorithmic Attacks

Because the size of the pattern sets increases and is already greater than the capacity of on-chip memory, most designs store their pattern sets in off-chip memory. Therefore, the memory gap becomes a bottleneck. Malicious attackers usually exploit this weakness to attack systems. For this reason, the following experiments address two types of attacks mentioned in Section I: 1) deep-search attacks and 2) sub-pattern attacks. Because performance is dramatically affected by attack strings, we use different ratios to test this design. The X-axis of Fig. 18 is the percentage of attack strings in input data, and the Y-axis is the latency of memory. The following experiments prove that although the heuristic design has a worst-case problem, it also achieves acceptable performance when enhanced by our proposed methods.

1) Deep Search Attacks: Simply, the deep search attack takes the rule sets as its attack strings. Each attack string comprises several complete or partial rules that trigger numerous searches. These searches make the exact-matching engine traverse into the depths of the trie tree, and the exact-matching engine has

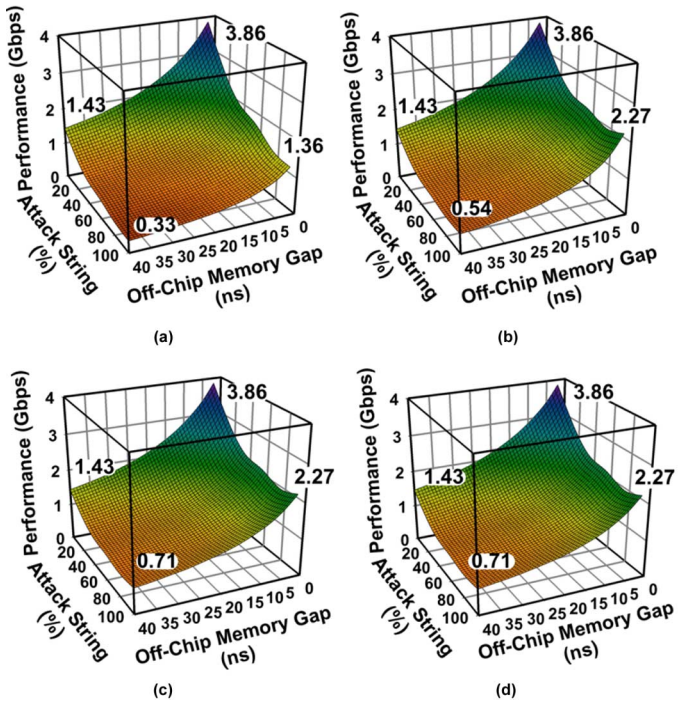


Fig. 18. Algorithmic attacks. Case 1: deep-search attacks. (a) Without optimization; (b) skip; (c) skip + prefetch; (d) skip + prefetch + cache.

to begin a new search from the next character after finishing the previous search if it does not skip. Thus, the performance shown in Fig. 18(a) drops off dramatically with the proportion of attack strings that result from a large number of memory accesses.

However, our proposed skip method removes these unnecessary memory accesses using skipping rules. Fig. 18(b) shows that the exact-matching engine triggered the most skip events and reduced memory access by 40%, and the attack string is entirely composed of the rule set because most rules do not contain any prefixes of other rules. Thus, while finishing a check, the exact-matching engine can skip its checked part to find the next candidate without redundant checking. Notably, the jump events, compared to skip events, do not increase massively. In this case, most rules do not contain any other rules, and therefore, the concept of a failure state does not work well (although it is not suitable for this case, we will describe how it works for the other case later). However, the prefetch mechanism further raises the performance from 0.54 to 0.71 Gb/s, even when the memory gap is very large, 40 ns. Because the access time of DDR memory only requires a column access (CAS) time for each non-first continuous address access in burst mode, simply relocating those trie nodes that correspond to the same rule into neighboring addresses improves performance. Although this method does not actually reduce the number of memory accesses, it effectively overlaps memory access time and computing time in prefetching.

2) Sub-Pattern Attacks: Although sub-pattern attacks are the worst case in pattern matching and can be avoided through the selection of rule sets, we experimented with this type of attack to discuss its effect on performance. We also took ClamAV as our benchmark in this experiment, but we added a new rule "aaa. . ." whose length is 66 characters, the same as the average length of

TABLE I
CASE 1: OFF-CHIP MEMORY ACCESS OF DEEP SEARCH ATTACKS

Attack pattern (%)	(a) Original	(b) Skip mechanism						(c) Skip + prefetch				(d) Skip + prefetch + cache			
	Memory access (M)	Skip event (K)	Skipped bytes (K)	Jump event (K)	Jumped bytes (K)	Memory access (M)	**Reduce (%)**	Prefetch hit (K)	Prefetch hit rate (%)	Memory access (M)	**Reduce (%)**	Cache hit (K)	Catch hit rate (%)	Memory access (M)	**Reduce (%)**
0	1.1	3	7	0.0	0.0	1.1	**0.3**	0.6	0.06	1.1	**0.3**	31.6	3.00	1.0	**3.3**
20	1.9	25	304	0.3	4.0	1.7	**8.4**	70.7	4.15	1.6	**12.2**	20.7	1.21	1.6	**13.3**
40	2.9	61	872	0.8	11.4	2.5	**15.6**	204.6	8.16	2.3	**22.5**	22.4	0.89	2.3	**23.3**
60	3.7	102	1,477	1.4	18.3	2.9	**21.7**	347.9	11.98	2.6	**31.0**	21.3	0.73	2.5	**31.6**
80	5.2	159	2,559	2.4	35.3	3.6	**31.6**	603.5	16.90	3.0	**43.2**	19.4	0.54	2.9	**43.5**
100	6.0	198	3,510	3.0	44.1	3.6	**40.2**	827.5	22.97	2.8	**53.9**	18.1	0.50	2.8	**54.2**

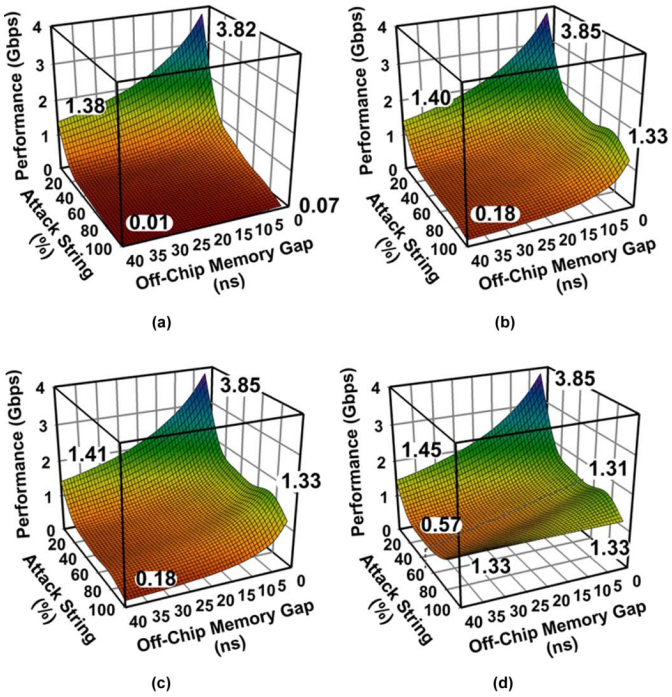


Fig. 19. Algorithmic attacks. Case 2: sub-pattern attacks. (a) Without optimization; (b) skip; (c) skip + prefetch; (d) skip + prefetch + cache.

ClamAV virus codes, which comprises repetitions of the same character. We used this rule as the attack string "aaa..." to evaluate performance. Because this rule "aaa..." appears at the head of the attack string "aaa..." and can also appear in any position of the attack string, the pattern matching processor must launch an exact match for each position of the attack string, which requires a larger number of memory accesses. The performance of pattern matching processors without any optimization, shown in Fig. 19(a), dramatically drops off from 3.82 Gb/s to close to 0 Gb/s because of the larger number of memory accesses.

Table II (b) shows that our skip mechanism eliminates 95% of memory accesses when the attack string consists entirely of the "aaa..." string due to its failure state concept. With an increase in the proportion of attacks, the numbers of skip events and jump events also increases, as shown in Table II (b). In the extreme case, the attack string occurs in numbers equivalent to the length of the input string (the total size of the input string is 8733 kB). This means that the skip mechanism and jump mechanism occur in every position of the attack string. Notably, the worst performance does not occur here but rather when dealing with a string that contains about 40% of the attack string. The blend of attack string and normal string keeps the skip mechanism from skipping the examined part. In addition, although the skip mechanism can remove 95% of memory access and improve the ideal performance from 0.07 to 1.33 Gb/s, the number of memory accesses is still not small enough to avoid terrible performance when considering the memory gap. Furthermore, the prefetch mechanism does not work well in this case because the number of memory accesses caused by algorithmic attacks is considerably more than the prefetch saves. However, this case is highly suitable for cache mechanisms because only a few trie nodes are referenced. The cache lets the pattern matching processor preserve reasonable performance with high-latency memory.

VI. IMPLEMENTATION

We integrate this design on an FPGA development board with a general personal computer to replace the pattern-matching kernel of Snort/ClamAV. Thus, we not only prove the feasibility of this design, but we also discuss some integration issues. The Altera EP1S60-5 FPGA board we chose is plugged into the PCI interface at the south bridge and has private DDR RAM, as shown in Fig. 20. This system runs on Mandrake Linux 8.1, and we use WinDriver to write the required PCI driver. The ISE synthesis tool reports that this design can run maximally at 148.59 MHz, but because of the limits of the PCI interface, this design only runs at 66 MHz on the FPGA board. We implement two modes: 1) application mode and 2) kernel mode.

For demonstration purposes, the application mode executes the target application in the host in an x86-linux environment and replaces pattern-matching kernel with our PME. In theory, the system can provide a 477-Mb/s scan rate, but its performance is reduced substantially when we replace the Snort's kernel with this design to scan real connections at run time. This mode is not effective because of significant data movement and driver overhead. Thus, both the FPGA board and network adapter are on the south bridge while the main memory is on the north bridge. When a packet comes in, it triggers a processor to switch into the system mode to execute the driver program of the network adapter and to set up a DMA to move data from the south side to the north side. However, after the processor completes the network protocol, it moves the data from the north bridge to the south bridge again for virus scanning. Redundant data movement and frequent context switches hinder performance. We suggest that PME should be implemented at the north bridge or be a coprocessor. Because of the limitations of PC architecture, we build this mode as a prototype.

TABLE II
CASE 2: OFF-CHIP MEMORY ACCESS OF DEEP-SEARCH ATTACKS

Attack pattern (%)	(a) Original	(b) Skip mechanism						(c) Skip + prefetch				(d) Skip + prefetch + cache			
	Memory access (M)	Skip event (K)	Skipped bytes (K)	Jump event (K)	Jumped bytes (K)	Memory access (M)	Reduce (%)	Prefetch hit (K)	Prefetch hit rate (%)	Memory access (M)	Reduce (%)	Cache hit (K)	Catch hit rate (%)	Memory access (M)	Reduce (%)
0	1.0	24	43	16	1,027	1.0	**3.1**	9.8	1.01	1.0	**4.1**	44.1	4.5	0.9	**7.6**
20	5.9	335	1,238	282	17,793	3.1	**47.2**	263.1	8.43	2.9	**51.7**	657.3	21.0	2.5	**58.4**
40	14.2	779	2,809	668	42,134	5.0	**64.8**	588.2	11.80	4.4	**68.9**	1,501.0	30.1	3.5	**75.4**
60	26.0	1,342	4,697	1,168	73,631	6.5	**74.8**	965.6	14.77	5.6	**78.5**	2,503.6	38.3	4.0	**84.5**
80	42.3	2,023	6,733	1,807	113,858	7.4	**82.5**	1,327.7	17.97	6.1	**85.6**	3,596.4	48.6	3.8	**91.0**
100	209.6	8,733	8,733	8,733	550,232	8.7	**95.8**	0.0	0.00	8.7	**95.8**	8,733.8	100.0	0.0	**100.0**

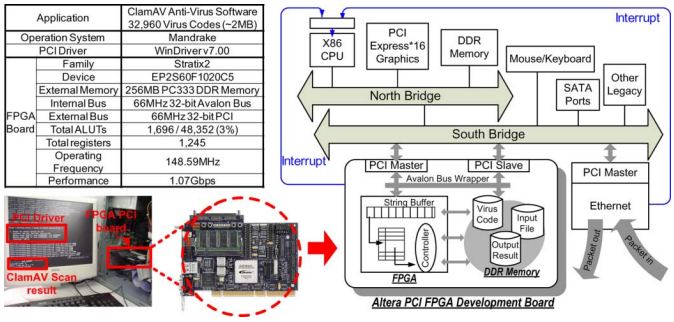
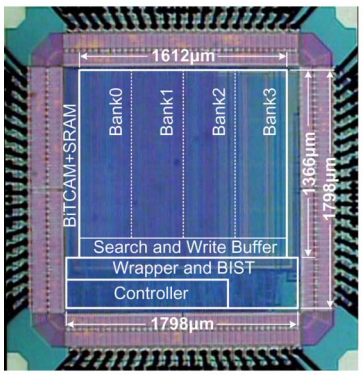


Fig. 20. Integrated FPGA experiment environment.



Technology		0.13μm 1P8M Mixed Signal CMOS Process
Core Supply Voltage		1.2V
I/O Supply Voltage		3.3V
Rule Count		**ClamAV 30,000**
On-chip Memory Size		**8KB BiTCAM+ 8.5KB SRAM**
CAM cell area/bit		6.238μm×2.440μm
On-chip Memory Area		1.366mm×1.612mm
Core area		1.798mm×1.798mm
Power	**Processor**	**131.22mW**
	Controller	28.04mW
	Wrapper	20.72mW
	Memory	**82.46mW**
Simulated Clock Freq.		380MHz
Measured Clock Freq.		**380MHz**
Throughput		**3.04Gbps**

Fig. 21. Chip implementation.

The kernel mode is used to model the scenario with off-chip memory and investigate the pure performance of the pattern matching processor. We modified the testing flow. First, we download a test file and pattern set (32 960 ClamAV rules) onto the local DDR memory on the FPGA. The PME then scans the file and accesses the pattern set directly from the FPGA's local memory. Therefore, there is no redundant data movement or context switch overhead. The performance is improved to approximately 477 Mb/s. This result shows that PME is helpful and suitable for network security systems.

Additionally, we also implemented the PME using a TSMC 13-μm process. Fig. 21 summarizes its features and shows the die photo. Unlike the design in the previous chapter, this chip uses a specialized BiTCAM [25] to implement the shift-signature table to further reduce the size of on-chip memory. The detailed algorithm for reducing the requirement of on-chip memory is not discussed in this paper. Although both versions are slightly different, they use the same control flow to match a string; therefore, both versions have nearly the same complexity.

According to the measured result, the controller of the PME maximally reaches 534 MHz. This version performs at 3.86 Gb/s for 32 960 ClamAV virus codes by 32-kB SRAM. If the shift table is implemented by BiTCAMs, then the maximum working frequency falls to 380 MHz due to the BiTCAM's limitations, but it still performs better than 3 Gb/s. By observing the chip layout, we find that most of the area of this chip (> 90%) is used for the shift table. Because using on-chip memory to store an entire virus database is very costly, a two-layer design is more reasonable for virus detection requirements. The key design issues of such approaches are: 1) reducing the chip area; 2) increasing the filter rate of the filter engine; and 3) reducing memory latency or access counts. This design attempts a case-study approach for algorithm improvement, chip implementation, and systems integration to explore the problems faced by the PME.

VII. COMPARISON

We have collected previous designs and have categorized them into two types: automata-based and filtering/heuristic-based architectures, as shown in Table III. Most automata-based approaches are based on the algorithm [8] proposed by Aho and Corasick in 1975. Although the original algorithm guarantees linear performance at the algorithmic level, its state table grows dramatically with its pattern set. Many designs [7] based on the AC algorithm try to reduce memory requirements. Lin Tan proposed a successful design by splitting large automaton matching into several lightweight machines at the bit level. However, a large number of read ports makes this design implementable only with on-chip memory because each lightweight machine requires a dedicated memory port. Piti Piyachon and Yan Luo extended this concept to the Intel IXP2855 network processor, and they minimized the requirements of on-chip memory. In 2007, an IBM team implemented an optimized AC algorithm [7] on a cell processor for large pattern sets, but the large pattern sets and irregular access patterns caused the memory gap to become the bottleneck. Therefore, they provided a DMA-based communication mechanism. In summary, the AC algorithm guarantees linear performance at the algorithmic level, but in a real implementation, the performance is affected by memory pressure caused by the irregular accessing of its large state table.

TABLE III
COMPARISONS WITH PREVIOUS WORK

Types	Automata-based			Filtering/heuristic-based				
Year	2004	2006	2008	2004	2005	2005	This work	
Architecture	Bit-split [11]	Bit-byte AC [19]	AC+IBM Cell processor [9]	Bloom [6, 25]	Reconfigurable perfect-hashing [9, 21]	Hash + SRAM [9, 23]	Re-encoded shift table + skip mechanism	
Application set, # of rules	Snort, ~1,000	Snort, 4,219	English words, 20,000	No description 10,000	Snort, ~1742	Snort, 1,729	ClamAV 3,000	ClamAV 32,960
Internal memory	0.4 MB	271.97 KB	256 KB/SPE	70 KB	76.5 KB	108 KB	32 KB	
External memory	NA	NA	>64 MB	No data	NA	NA	8 MB	
Operating frequency	1.26 GHz	1.40 GHz	3.20 GHz	81 MHz	716 MHz	893 MHz	534 MHz	
Total throughput	10.1 Gbps	5 Gbps	2.18 Gbps	2.46 Gbps	5.73 Gbps	7.144 Gbps	9.06 Gbps	3.86 Gbps
Gbps/engine	10.1 Gbps	0.31 Gbps	0.27 Gbps	0.6 Gbps	5.73 Gbps	7.144 Gbps	9.06 Gbps	3.86 Gbps
# parallel engine	many	16	8	4	1	1	1	
Feature	1. Has a linear performance at the algorithmic level. 2. Provide one-pass examination. 3. Space requirements could increase exponentially with pattern set 4. Irregular access causes memory pressure, layout issues and hot spots for large number of pattern set.			1. Two-phase architecture: a fast front-end with slower back-end. 2. The front-end with a small table could be implemented in internal memory and provide efficient filtering. In some cases, these designs can handle more than one character per operation 3. The back-end is usually implemented by a compact trie structure and is more memory efficient than AC-based designs. 4. Provide sub-linear throughput, but has worst cases while under algorithmic attacks.				

Filtering-based architectures [4], [19], [21] do not guarantee linear performance, but the front-end engine provides an efficient filtering rate using a filtering table. The size of the filtering table is generally between 10 and 100 kB and can therefore be implemented by internal memory. Localizing computation on the chip makes the average performance of filtering-based architectures attractive. Even these architectures have a worst-case scenario, but some solutions can address this scenario and still provide reasonable performance. In addition, some front-ends, such as "shift table," operate on multiple characters, which further improves performance. As a result, filtering-based architectures have the potential to provide higher performance than automata-based architectures. Furthermore, the back-end of a filtering-based architecture is generally based on a compact trie structure, which is more memory-efficient than automata-based architectures. According to our design, this work only requires 8 MB of external memory for 30 000 patterns (2 MB pattern bytes) rather than 64 MB for 20 000 patterns (151 kB pattern bytes). Under the same conditions, this design provides a 9.06 Gb/s scan rate compared to previous designs [4], [7], [9], [17], [19], [21], [23] when given 3000 pattern, due to numerous optimizations at the algorithm level. Even for 32 960 given patterns, this design still performs at 3.86 Gb/s. In brief, filtering-based architectures are more suitable for embedded security systems in both performance and cost.

VIII. CONCLUSION

Many previous designs have claimed to provide high performance, but the memory gap created by using external memory decreases performance because of the increasing size of virus databases. Furthermore, limited resources restrict the practicality of these algorithms for embedded network security systems. Two-phase heuristic algorithms are a solution with a tradeoff between performance and cost due to an efficient filter table existing in internal memory; however, their performance is easily threatened by malicious attacks. This work analyzes two scenarios of malicious attacks and provides two methods for keeping performance within a reasonable range. First, we re-encoded the shift table to make it provide a bad-character heuristic feature and high filter rates for large pattern sets at the same time. Second, the proposed skip mechanism cushions the blow to performance under algorithmic attacks.

The operating frequency of this design, when implemented by a TSMC .13-μm 1P8M process, is above 534 MHz. Its kernel provides 3.86-Gb/s average performance and 1.31-Gb/s performance under attacks by 30 000 ClamAV virus codes. When considering the memory gap (40 ns per memory access), this design still has 1.43-Gb/s average performance with 0.57-Gb/s performance under algorithmic attacks. The proposed re-encoding stage removes 95.42% of the exact-matching events in the front-end for normal cases, and the skip mechanism eliminates 95.8% of the external memory accesses for attack cases with just 32 kB internal memory and 8 MB external memory.

REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, pp. 333–340, 1975.
[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, pp. 762–772, 1977.
[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, 1970.
[4] S. Dharmapurikar, P. Krishnamurthy, and T. S. Sproull, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, Jan. 2004.
[5] D. P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the Cell processor," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2007, pp. 1–8.
[6] O. Villa, D. P. Scarpazza, and F. Petrini, "Accelerating real-time string searching with multicore processors," *Computer*, vol. 41, pp. 42–50, 2008.
[7] D. P. Scarpazza, O. Villa, and F. Petrini, "High-speed string searching against large dictionaries on the Cell/B.E. processor," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–8.
[8] R.-T. Liu, N.-F. Huang, C.-N. Kao, and C.-H. Chen, "A fast string-matching algorithm for network processor-based intrusion detection system," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 614–633, 2004.
[9] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. 32nd Annu. Int. Symp. Comput. Arch.*, 2005, pp. 112–122.
[10] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proc. 12th IEEE Int. Conf. Netw. Protocols*, 2004, pp. 174–183.

[11] G. Memik, S. O. Memik, and W. H. Mangione-Smith, "Design and analysis of a layer seven network processor accelerator using reconfigurable logic," in *Proc. 10th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2002, pp. 131–140.
[12] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *Proc. Reconfig. Comput. Going Mainstream, 12th Int. Conf. Field-Program. Logic Appl.*, 2002, pp. 452–461.
[13] Z. K. Baker and V. K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," presented at the ACM Symp. Arch. for Netw. Commun. Syst., Princeton, NJ, 2005.
[14] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *Proc. 12th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 249–257.
[15] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Annu. IEEE Symp. Field-Program. Custom Comput. Mach*, 2001, pp. 227–238.
[16] S. Yi, B.-K. Kim, J. Oh, J. Jang, G. Kesidis, and C. R. Das, "Memory-efficient content filtering hardware for high-speed intrusion detection systems," presented at the ACM Symp. Appl. Comput., Seoul, Korea, 2007.
[17] P. Piyachon and Y. Luo, "Efficient memory utilization on network processors for deep packet inspection," presented at the ACM/IEEE Symp. Arch. for Netw. Commun. Syst., San Jose, CA, 2006.
[18] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Univ. Arizona, Tucson, Report TR-94-17, 1994.
[19] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *Proc. 15th Int. Conf. Field Program. Logic Appl.*, 2005, pp. 644–647.
[20] Y. H. Cho and W. H. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," in *Proc. 12th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 125–134.
[21] Y. H. Cho and W. H. Mangione-Smith, "A pattern matching coprocessor for network security," presented at the 42nd Annu. Des. Autom. Conf., Anaheim, CA, 2005.
[22] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *Proc. 12th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2004, pp. 258–267.
[23] Micron Technology, Inc., Boise, ID, "256 MB DDR2 SDRAM datasheet," 2003.
[24] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, pp. 490–499, 1960.
[25] C.-C. Wang, C.-J. Cheng, T.-F. Chen, and J.-S. Wang, "An adaptively dividable dual-port BiTCAM for virus-detection processors in mobile devices," *IEEE J. Solid-State Circuits*, vol. 44, no. 5, pp. 1571–1581, May 2009.

Chieh-Jen Cheng (S'06) was born in Taiwan, in 1980. He received the B.S. and M.S. degrees in computer science from the National Chung Cheng University, Taiwan, in 2003 and 2006. He is currently pursuing the Ph.D. degree from the Institute of Computer Science, National Chung Cheng University, Taiwan.

His research interest lies in microprocessor architecture, system simulation, and memory hierarchy.

Chao-Ching Wang (S'06–M'08) was born in Taiwan, in 1981. He received the B.S. and Ph.D. degrees in electrical engineering from the National Chung Cheng University, Taiwan, in 2003 and 2008, respectively.

Since then, he has been with Himax Media Solutions, Inc., Tainan Tree Valley Park, Taiwan, where he is currently a Senior Engineer. His research interests include high speed, low-leakage, and low-power memory designs, high-performance digital integrated circuit designs.

Wei-Chun Ku (S'06) was born in Miaoli, Taiwan, in 1981. He received the B.S. degree in Department of Computer Science and Information Engineering from Tamkang University, Taipei, Taiwan, in 2004, respectively. He is currently pursuing the Ph.D. degree from the Department of Computer Science and Information Engineering, National Chung Cheng University, Chia-Yi, Taiwan.

His research interests include embedded system design, computer architecture, and SOC design.

Tien-Fu Chen (S'90–M'93) received the B.S. degree in computer science from National Taiwan University, Taiwan, in 1983, and the M.S. degree and Ph.D. degrees in computer science and engineering from the University of Washington, Seattle, in 1991 and 1993, respectively

He joined Wang Computer Ltd., Taiwan, as a System Software Engineer for three years. He is currently a Professor with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. In the past, he had published several widely-cited papers on dynamic hardware prefetching algorithms and designs. In recent years, he has made contributions to processor design and SOC design methodology. His recent research results include multithreading/multicore media processors, on-chip networks, and low-power architecture techniques as well as related software support tools and SoC design environment. His current research interests include computer architectures, system-on-chip design, design automation, and embedded software systems.

Jinn-Shyan Wang (S'85–M'88) was born in Taiwan, in 1959. He received the B.S. degree in electrical engineering from the National Cheng-Kung University, Tainan, Taiwan, in 1982 and the M.S. and Ph.D. degrees from the Institute of Electronics, National Chiao-Tung University, Hsinchu, Taiwan, in 1984 and 1988, respectively.

He was with Industrial Technology Research Institute (ITRI) from 1988–1995, engaged in ASIC circuit and system design, and became the Manager of the Department of VLSI Design. He joined the Department of Electrical Engineering, National Chung-Cheng University, Chia-Yi, Taiwan, in 1995, where he is currently a full Professor. His research interests include low-power and high-speed digital integrated circuits and systems, analog integrated circuits, IP and SOC design, and CMOS image sensors. He has published over 20 journal papers and 40 conference papers and holds over 20 patents on VLSI circuits and architectures.