# PERG: A Scalable Pattern-Matching Accelerator

Johnny Tsung Lin Ho
Department of Electrical and Computer Engineering
University of British Columbia
johnnyho@ece.ubc.ca

Guy G. F. Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
lemieux@ece.ubc.ca

***Abstract*— PERG is an FPGA application for accelerating detection of computer virus signatures (patterns). A pattern consists of a sequence of one or more segments separated by gaps of fixed lengths. PERG preprocesses a database of these patterns into hardware. To our knowledge, PERG is the first pattern matching hardware targeting viruses, as well as the first among network intrusion detection systems (NIDS), which are similar in nature to PERG, to implement *Bloomier filters*. This makes guarding against false positives faster than traditional Bloom filters because verification requires checking against one pattern instead of several patterns. Using the ClamAV antivirus database, PERG fits 80,282 patterns containing over 8,224,848 characters into one modest FPGA chip with a small (4 MB) off-chip memory. The architecture achieves roughly 26x improved density (characters per memory bit) compared to the next-best NIDS pattern-matching engine which fits only 1/250th the characters. With an estimated throughput of about 200MB/s, PERG keeps up with most network or disk interfaces.**

***Keywords-Antivirus; Pattern Matching; FPGA; Bloomier Filter***

## I. Introduction

Computer virus protection is a ubiquitous part of every personal computer. However, the security of antivirus software comes at a hefty cost in system performance. A quantitative benchmark on antivirus overhead [1], conducted on a modern PC powered by an AMD Athlon 64 x2 4800+, reported over 40% slowdown in boot time and 1000% in disk I/O performance with antivirus software from popular vendors. Antivirus scanners utilize a plethora of different detection heuristics that vary from vendor to vendor, but matching a file against a large virus pattern database is a fundamental technique that is easily implemented in hardware. This pattern matching is the main performance bottleneck [2].

In this paper, we introduce PERG, a scalable hardware accelerator for pattern matching with the ClamAV virus database [3]. Sourcefire provides both ClamAV and Snort [4], a network intrusion detection system (NIDS), as open-source software. NIDS also uses pattern matching, but it is simpler than virus scanning. NIDS is compute-intensive, so it has been a popular target for FPGA accelerators to run at gigabit rates.

The PERG architecture, shown in Fig. 1, uses *Bloomier filters* [5] and a *fragment reassembly engine* to map 80,282 virus patterns, each with an average length of 102 bytes, into a single FPGA and a small off-chip memory. PERG requires about 0.324 memory bits per pattern character, which is over 26x the best density achieved with a NIDS hardware engine. Part of this density advantage is achieved due to *filter consolidation*. Traditionally, each pattern length gets its own filter unit. Virus patterns can be very long, resulting in many more filter units than NIDS and significantly more hardware. However, PERG packs several lengths into a single filter unit to save hardware by fragmenting a pattern, and uses the fragment reassembly engine to reconstruct the pattern later.

The authors thank CMC Microsystems/SOCRN for providing tools and the Amirix AP1000 design platform, and NSERC/Altera/Actel for funding.

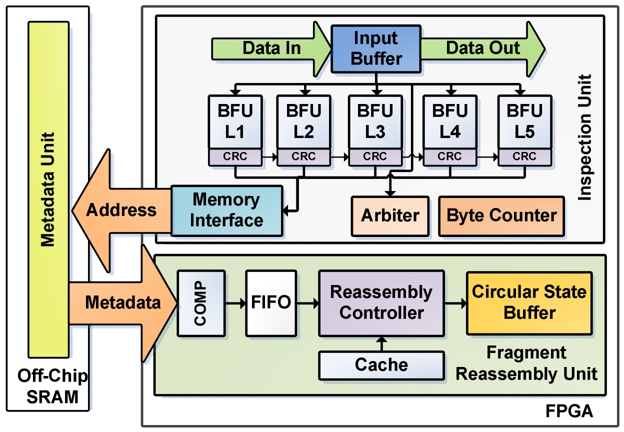

Figure 1. Top-level architectural diagram of PERG.

Another unique aspect of PERG is lower false positive rates and fast verification of a match. To our knowledge, PERG is the first NIDS/virus pattern matching application to employ Bloomier filters. Bloom and Bloomier filters do *approximate matching*, meaning there can be false positives but no false negatives. To eliminate false positives, each match detected by the filter must be further scrutinized by *exact matching* in software. While both filters use hash tables, Bloomier filters have a one-to-one mapping property which identifies just one rule for each detected match; in contrast, Bloom filters can only identify a set of rules. This property allows Bloomier filters to reduce false positives, eg, an 8-bit checksum gives a 1/256th reduction. In the rare case the checksum matches, software does an exact match against a single rule. If Bloom filters are used instead, there is no checksum to reduce false positives *and* each potential match must compare against the complete set of rules; in our database, the average set size is 4,832 rules.

The rest of the paper is organized as follows. Existing approaches in hardware pattern matching engines are introduced in Section 2. Details of ClamAV's signature database are covered in Section 3. Bloomier filter operation is explained in Section 4. The pattern compiler and hardware architecture are discussed in Sections 5 and 6, respectively. Simulation setup and results are presented in Section 7. Finally, conclusions and future work are given in Section 8.

## II. Background and Related Work

While hardware pattern matching for virus detection has not been studied much, in the context of NIDS, it has been studied

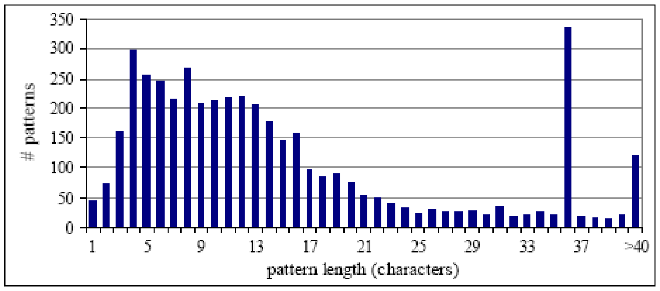

Figure 2. Pattern length distribution of Snort database [13]. Dec 2006.

extensively using the Snort database. Fig. 2 shows the length distribution of patterns in Snort [13]. In comparison, Fig. 3 shows the length distribution of the ClamAV database. Although similar to NIDS in concept, antivirus scanning is orders of magnitude more compute intensive.

Most modern pattern matching engines are FPGA-based and fall into two categories: finite state machine (FSM) [8] and Bloom filter [9]. FSM solutions are usually based on the Aho-Corasick (AC) algorithm [7]. AC converts a set of string patterns into a finite automaton, where each state in the automaton is a character of a pattern from the set. In general, FSM solutions share two drawbacks. First, they use more memory than Bloom filters. Second, daily updates to virus database can result in entirely different hardware structures, leading to lengthy delays to generate a new bitstream and periods of decreased protection.

A Bloom filter uses a hash table with *m* one-bit entries to store a match/no-match result. Due to limited space, please refer to [6] for a detailed explanation of Bloom filter. At a fixed false-positive probability, memory usage is independent of the pattern length and sub-linearly proportional to the number of rules, giving the Bloom filter a much higher density than the FSM approach. However, Bloom filters also have several disadvantages. First, each pattern length uses its own hash table. This leads to a large number of filter units because both Snort and ClamAV have a wide range of pattern lengths. Second, because of false positives, exact matching is required upon a hit, forcing additional computation. To make matters worse, Bloom filters can only determine membership; they do not indicate which particular rule is the potential match. The process of exact matching is still computationally intensive.

Another approach that relates closely with the Bloom filter is perfect hashing [10, 11]. While a Bloom filter can only detect membership, perfect hashing allows one-to-one association between hash keys and patterns, simplifying exact matching after each hit. However, these approaches need carefully chosen hash functions to avoid collisions. Daily updates to a large set of patterns make it increasingly difficult to generate a perfect hash function every time.

## III. CLAMAV DATABASE

Virus signatures in ClamAV can be divided to the three types: MD5 checksums, basic patterns, and regular expression (RegEx) patterns. MD5 checksums are ignored in this work because this accounts for only 0.64% of runtime [15]. A basic pattern is a continuous byte string. A RegEx pattern is an extension of the basic pattern with OR operators, displacement gaps, and wildcards. RegEx support is necessary for detection of polymorphic viruses, but the current implementation of PERG does not fully support them so they are mostly ignored. However, simple displacement gaps are supported; patterns can be made up of segments separated by gaps of a defined length.

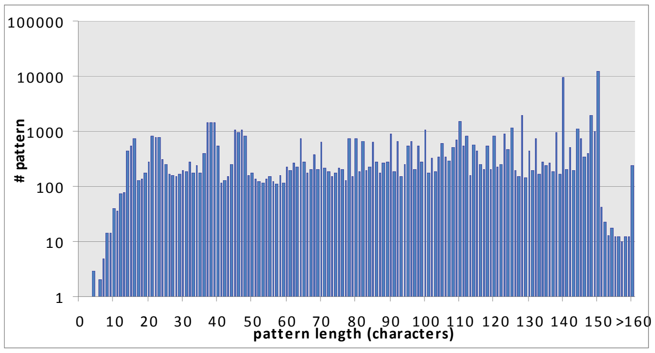

Figure 3. Segment length distribution of ClamAV database. Data from ClamAV main database version 0.93.1 on June, 2008, using basic patterns.

## IV. BLOOMIER FILTER

Bloomier filters [5] are an extension of the Bloom filter. Instead of providing a simple match/no-match answer, the Bloomier filter indicates *which pattern* resulted in a match, speeding up exact matching to verify against false positives. To look up a pattern *x* in a Bloomier filter, the input is hashed with all *k* hash functions, locating *k* entries in the hash table. All *k* entries are XORed together to form a hash select, *s(x)*, which identifies one of the hash functions. That hash function points to one table entry, which by construction of the filter is associated with a unique pattern. Due to limited space, please refer to [5] for construction and proof of Bloomier filters.

## V. PERG PATTERN COMPILER

The PERG system is divided to two sections: the pattern compiler and the hardware architecture. The compiler examines the pattern database and decides on several hardware parameters including the precise hash functions, number and size of Bloomier filter units, and the mappings of patterns to Bloomier filter table entries. A precise hardware instance is then generated from these parameters. Operation of the compiler consists of the following four main steps.

*Step 1, Segmentation and Special-case Removal*. Patterns are broken down into segments separated by displacement gaps. Any patterns containing segments shorter than 4 bytes or with displacement gap greater than the maximum supported displacement are removed. Since regular expressions other than simple displacement are not currently supported, any pattern containing a regular expression is removed as well. At the end of this stage, 80,282 unique patterns remain.

*Step 2, Filter Consolidation and Mapping*. Each Bloomier filter unit (BFU) works by hashing a specific string length into a dedicated hash table. However, it is unrealistic to create a BFU for every length due to the wide range of segment lengths (344 unique lengths) and limited FPGA resources. Instead, we select a *small number* (26) of distinct filter lengths and split segments of the wrong length into two *overlapping fragments* of the correct length. The sequence of segments or fragments are represented by *link numbers* which start at 0 for the initial (prefix) segment or fragment and are incremented by one for each next segment or fragment of the same pattern, ending with a suffix segment or fragment.

*Step 3, Metadata Generation.* This stage is responsible for the creation of rule ID's and collecting the "metadata" information, such as the link number, that links each fragment to their predecessor and successor fragments. When a segment or fragment is matched, the link number connects fragments together until they form the overall rule.

*Step 4, Bloomier Filter Generation.* The last stage generates the BFUs, starting from the shortest length to the longest. For each BFU, the construction creates two hash functions based on *shift-add-xor* (SAX) [13]. SAX hash functions can be fully unrolled in a hardware pipeline with overall length equal to the longest segment length. All BFUs share the same hashing pipeline by picking off the appropriate intermediate hash. Bloomier filter construction may fail because a unique one-to-one mapping cannot be produced; the failure probability is low due to the use of multiple hash functions [14]. Nevertheless, in such a case, new hash functions can be used by generating new hash parameters. Each entry in the Bloomier table also contains an 8-bit primary CRC checksum to reduce false positives. These are computed in this step as well. Entries in the table that do not map to a pattern are initialized to all 1's.

## VI. FPGA Hardware

The PERG hardware architecture, shown in Fig. 1, is divided into *Inspection*, *Metadata*, and *Fragment Reassembly Units*. Inspection Unit filters input data stream through parallel *Bloomier filter units* (BFUs) and performs the primary (8-bit) check. At each cycle, a new input byte is scanned in parallel by a set of BFUs to match different string lengths. A 32-bit *Byte Counter* counts the number of bytes in the current file stream.

When Inspection Unit detects a match, it determines which pattern caused the match and sends the information to Metadata Unit along with the current Byte Count and a 32-bit secondary checksum computed on the input data. The Metadata Unit retrieves pattern information, such as the expected checksum value, from an off-chip memory. Finally, the Fragment Reassembly Unit compares the secondary checksum and tracks the progress of searching for the overall rule/pattern.

### A. Inspection Unit

As shown in Fig. 1, the SAX hashing pipeline is fully unrolled and shared by all parallel BFUs in the same way as [13], resulting in a sliding-window hash. The primary and secondary checksums use the same pipelined structure. Multiple BFUs can report hits in the same cycle, so each BFU has a small *Filter FIFO*. Simultaneous requests are processed in Byte Count order by an *Arbiter* inspecting each Filter FIFO output, implemented as a pipelined sorting network.

### B. Metadata Unit

The Metadata Unit is a small external memory used to store connectivity information between fragments needed to form an entire pattern. After a fragment match in the Inspection Unit, the Arbiter passes requests to the Metadata Unit to access this memory. The address is derived from the BFU ID and the hash key, which is always unique for each fragment.

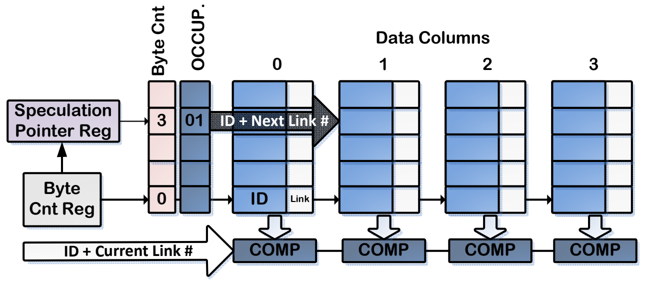

Figure 4. Circular Speculative Buffer.

### C. Fragment Reassembly Unit

The Fragment Reassembly Unit reconstructs full patterns from fragments or segments. First, secondary checksums from Metadata and Inspection Units are compared. A mismatch is ignored. A match with a single-segmented pattern generates a request to the host for exact matching. A match of a fragment of a multi-segmented pattern is sent to the *Reassembly Controller*, which performs final processing of the fragment. Matching a suffix fragment of a rule is insufficient; all prior fragments to the rule must also have matched in the correct order at their expected positions, forming a *trace* of the rule. The metadata indicates how fragments link together in a trace.

In some situations (236 cases), a segment string is *shared* by multiple rules. The *Cache* stores metadata for these shared fragments. Unlike typical caches, this cache is only used as a fast read-only memory. Shared fragments are shared by 2 rules on average, and most fragments are shared by less than 4 rules. For each rule sharing the fragment, one Cache access is needed to retrieve its metadata, and one update to the rule's overall progress is made in the Circular Speculative Buffer.

The *Circular Speculative Buffer* (CSB), shown in Fig. 4, tracks the progress of multiple traces for multi-segmented rules. CSB operates just like a time wheel in an event-driven simulator to schedule future events. In this case, a future event is a speculated match of the next fragment for this rule. This speculated match can be scheduled to occur at a specific Byte Count in the input stream. Like a time wheel, multiple events can be scheduled at the same Byte Count in the future using buckets or, in this case, Data Columns. The number of buckets used is called the *Occupancy*. Unlike a time wheel, however, CSB only schedule events up to N bytes into the future, where N is the number of entries in the circular buffer (512 here).

In our CSB implementation, we have four Data Columns. It is possible, though improbable, that more than 4 traces are expected to have a fragment match at exactly the same Byte Count. To handle this, the Occupancy is incremented once more (above 4) to signify this condition. On a fragment match at this Byte Count, we ignore the Data Column contents and always assume the fragment was expected. While this scheme does increase false positive probability, it guarantees detection of all rules without any false negative probability. In our experimental results, we find the probability of this happening is extremely low even with just four Data Columns.

## VII. Simulation Setup and Results

The PERG pattern compiler and a cycle-accurate C model of the hardware architecture have been completed. A hardware RTL implementation is underway targeting the Amirix AP1000 system which contains a Xilinx Virtex-II Pro VP100. The RTL is estimated to run at 200MHz; the expected critical path is the arbiter in the Inspection Unit. The entire datapath is feed-forward only and can be deeply pipelined to meet this clock frequency. For the Metadata Unit, we assume an external 4 MB SRAM with 64 bit data width running at 50 MHz. PERG usually accepts one byte every clock cycle from a continuous file stream. Due to occasional off-chip memory accesses and limited FIFO space, the PERG engine sometimes (~5%) stops accepting a new byte and applies backpressure flow control. This stalling can be reduced with deeper FIFOs.

Our test is based on ClamAV 0.93.1, containing 85,625 basic and regular expression rules in total. Patterns containing regular expressions, segments less than 4 bytes long, or displacement gaps larger than 512 bytes are ignored, resulting in removal of 5,343 patterns. We end up with a total of 80,282 rules containing a total of 125,078 fragments and 11,452,898 bytes (characters). A total of 236 segments/fragments are identified to be shared by multiple rules. File extensions are ignored and left to the host system for consideration.

Table I shows the estimated memory use for the main components (BFUs, FIFOs, Cache, and CSB) by the PERG engine. It does not include extra pipelining registers, special-purpose registers like the Byte Counter, the external memory interface, or any combinational logic. All of this additional logic is very simple and should easily fit in the FPGA fabric.

For sample data input, we use the Ubuntu 7.10 ISO image and summarize results in Table II. In the Single File test, the entire ISO image file is scanned. In the Extracted Files test, the ISO image is extracted into 274 files (files shorter than 4 bytes are excluded). At the end of each file, the internal state is reset in a single cycle prior to beginning the next file; to do this, all reset-critical state is stored in flip-flops.

PERG is the first hardware engine for antivirus pattern matching, so we cannot compare to similar work. Instead, Table III roughly compares PERG to NIDS systems using Snort. Only on-chip memory is included in the comparison, since this is what limits the scalability of the engine to larger databases. In terms of logic cell density, we expect PERG to be slightly higher than [13] due to the checksum calculations.

## VIII. Conclusions and Future Work

Novel aspects of PERG include the use of Bloomier filters to simplify exact matching, consolidation to reduce the number of filter units and improve the density of rules fit into each filter, and the FRU used to track the state of patterns that have fixed-length gaps or that have been divided into multiple fragments. In terms of memory density, PERG shows a promising 26x improvement over the best NIDS pattern match engine. We are now implementing our design on a Xilinx Virtex-II Pro. Early synthesis results of sub-modules comply with our simulator's estimates. We are also verifying an extension that supports wildcards and regular expressions.

TABLE I. Estimated On-Chip Memory Usage.

| | LUTs (16b each) | BRAMs (18kb each) |
|---|---|---|
| BFUs | 1,716 | 137 |
| FIFOs | 480 | 0 |
| Cache | 0 | 1 |
| Circular Spec. Buffer | 1,600 | 4 |
| Hash Pipeline | 319 | 0 |
| Prim. Chksum Pipe | 75 | 0 |
| Second. Chksum Pipe | 75 | 0 |
| Estimated Total | 4,265 | 142 |

TABLE II. Simulation Results.

| | Single File | Extracted |
|---|---|---|
| # of Bytes Scanned | 729,608,192 | 727,677,929 |
| # of False Positives | 4 | 4 |
| False Positive Prob. Per Scan | 0.0000005% | 0.0000005% |
| # of Off-chip Mem. Req. | 73,739,161 | 73,666,748 |
| Prob. of Off-chip Mem. Req. per ByteScan | 10.11% | 10.12% |
| Off-chip Memory Throughput | 19.27 MB/s | 19.31 MB/s |
| # of Secondary Check Hits | 9,770 | 9,737 |
| Prob. of Secondary Check Hits per ByteScan | 0.13% | 0.13 % |
| Average Throughput | 190.7 MB/s | 190.7 MB/s |

TABLE III. Memory Density Comparison.

| | # of Chars | Memory (kb) | Mem. density (bits/char) |
|---|---|---|---|
| PERG (raw) | 8,224,848 | 2,612 | 0.335 |
| PERG (post compiler) | 11,452,898 | 2,612 | 0.238 |
| Cuckoo Hashing [13] | 68,266 | 1,116 | 16.7 |
| PH-Mem [10] | 20,911 | 288 | 14.1 |
| HashMem [11] | 18,636 | 630 | 34.6 |
| ROM+Coproc [12] | 32,384 | 276 | 8.73 |

## References

[1] What Really Slows Windows Down. http://www.thepcspy.com/read/what_really_slows_windows_down.

[2] D. Uluski, M. Moffie, and D. Kaeli, "Characterizing antivirus workload execution," *SIGARCH Comp. Arch. News, ACM,* 2005*, 33*, 90-98.

[3] Clam Antivirus. http://www.clamav.net.

[4] Snort. http://www.snort.org.

[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," *Society for Industrial and Applied Mathematics,* 2004, 30-39.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Comm. ACM,* 1970*, 13*, 422-426.

[7] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Comm. ACM,* 1975*, 18*, 333-340.

[8] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching Using FPGAs," *IEEE Symp. on FCCM*, 2001, 227-238.

[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel Bloom filters," *Symposium on High Performance Interconnects,* 2003, 44-51.

[10] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," *Int'l. Conf. on Field Programmable Logic and Applications,* 2005, 644-647.

[11] G. Papadopoulos and D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," *Proc. International Conference on Field Programmable Logic and Applications,* 2005, 39-4.

[12] Y. H. Cho and W. H. M-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005, pp. 215–224.

[13] T. N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," *Int'l. Conf. on Field-Programmable Technology ICFPT 2007,* 2007, 121-128.

[14] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A storage-efficient, collision-free hash-based network processing architecture," *Int'l Symp. on Computer Architecture*, 2006, 203-215.

[15] X. Zhou, B. Xu, Y. Qi, and J. Li, "MRSI: A fast pattern matching algorithm for anti-virus applications," *Int'l. Conf. on Networking,* 2008, 256-261.