PERG: A Scalable FPGA-based Pattern-matching Engine with Consolidated Bloomier Filters

Johnny Tsung Lin Ho and Guy G. F. Lemieux
University of British Columbia, Vancouver, Canada
johnnyho@ece.ubc.ca and lemieux@ece.ubc.a

Abstract

PERG is an FPGA application that accelerates the process of searching a stream of bytes against a large, fixed database of string patterns. The stream could be network, disk, or file traffic, while the pattern database may represent computer viruses, spam, keyword sequences, or watermarks. A full pattern, or rule, consists of a sequence of one or more segments separated by gaps. Each segment is an exact sequence of bytes, possibly 100s of bytes long. Each gap contains arbitrary bytes, but is a known length. PERG uses a pattern compiler to transform a database of these rules into a hardware implementation. To the authors' knowledge, this is the first pattern match engine hardware designed for large virus databases. It is also first among network intrusion detection systems (NIDS), which are similar in nature to PERG, to implement Bloomier filters. Like hash tables, Bloomier filters produce false positives due to aliasing, so all potential matches must be verified by exact matching. However, Bloomier filters are more powerful than their ancestral Bloom filters because they can identify the exact rule of a potential match. This enables two key advantages for PERG. First, it allows PERG to use a checksum to very efficiently reduce false positives. Second, exact matching with PERG filters is much faster than with Bloom filter systems because only one suspect pattern needs to be checked, not all patterns. To reduce memory requirements, PERG packs as many segments as possible into each Bloomier filter by consolidating several different segment lengths into the same filter unit. This is done by dividing each segment into two smaller but overlapping fragments of the same length. Dividing into non-overlapping fragments would create shorter fragments of uneven lengths, leading to higher false positives and differing lengths to consolidate later. Using the ClamAV antivirus database, PERG fits 80,282 patterns containing over 8,224,848 characters into a single modest FPGA chip with a small (4 MB) off-chip memory. It uses just 26 filter units, resulting in roughly 26x improved density (characters per memory bit) compared to the next-best NIDS pattern match engine which fits only 1/250th the characters. PERG can scan at roughly 200MB/s and match the speed of most network or disk interfaces.

1. Introduction

Computer virus protection is now 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 was recently posted online [1]. The test, 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. Experimental data from [2] shows a similar trend.

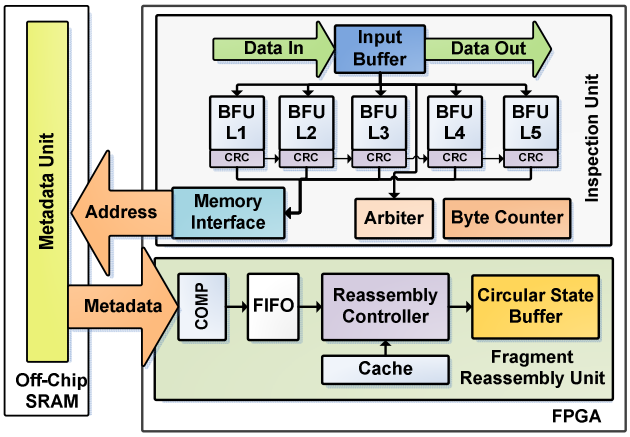

Figure 1. Top-level architectural diagram of PERG.

Antivirus scanners utilize a plethora of different detection heuristics that vary from vendor to vendor, but matching a file stream against a known virus pattern is a fundamental technique that is easily implemented in hardware. This pattern matching is a significant performance bottleneck [2]. With viruses growing faster than Moore's law, the performance penalty is getting worse. Private communication with one antivirus researcher indicated the current growth is roughly 1,000 *new* patterns per day. Scalable solutions are needed.

ClamAV [3] is open-source virus scanning software running on several operating systems including Linux, OS-X, and Windows. The project was acquired in 2007 by Sourcefire, the same company that provides Snort software [4], a popular network intrusion detection system (NIDS). It has been quite popular among researchers to accelerate Snort using hardware pattern match engines, but this is relatively "easy" with only 5,076 patterns that average 13 bytes each [17]. However, to detect viruses, the most up-to-date ClamAV database contains about 90,000 patterns; the majority of which are over 100 bytes long. Clearly, virus scanning is much more computationally intensive.

Our goal is to create an FPGA-based pattern matching engine to accelerate virus scanning in computers using a single FPGA. The form factor may take any number of shapes and sizes, ranging from an SDcard to a PCI slot. While we envision this could be useful for all personal computers, FPGAs are quite expensive, so early adopters

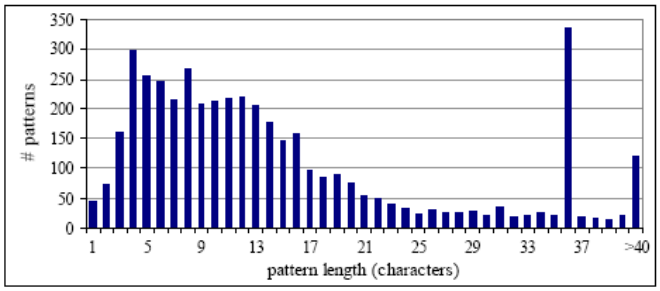

Figure 2. Pattern length distribution of Snort database [17]. Data collected on December 2006.

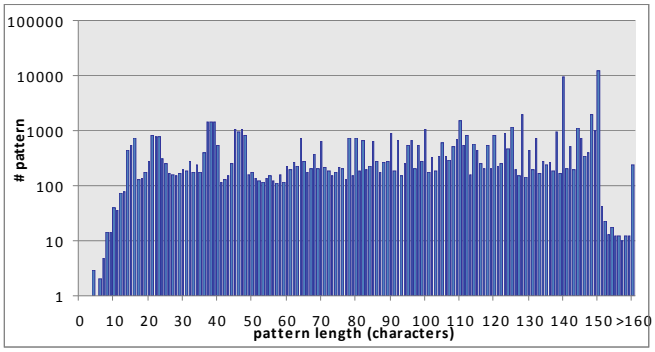

Figure 3. Segment length distribution of ClamAV database. Data from ClamAV 0.93.1 main database on June 9, 2008, using basic patterns only.

would likely be business file servers configured to scan each file upon every read (write scanning is also possible). This way, clients that work with only server files can dispense with virus scanning software altogether and see a performance boost. In a server, high throughput (disk/network speeds) is extremely important, so it is impractical to do server-side scanning in software. While pattern matching could be implemented in an ASIC, the frequent updates to virus databases make FPGAs significantly more suitable. Also, since this is a niche market, FPGAs are more cost-effective and risk-averse than ASICs.

In this paper, we introduce PERG, a scalable hardware accelerator for pattern matching with the ClamAV database. Our architecture, shown in Figure 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 0.335 memory bits per pattern character, which is over 26x the best density achieved with a NIDS hardware engine. To our knowledge, PERG is the first NIDS/virus pattern matching application to employ Bloomier filters. Bloomier filters only do *approximate matching*, meaning there can be false positives, which are reduced using checksums. Furthermore, by pinpointing the exact byte position and suspected rule, software can very quickly eliminate any remaining false positives.

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.

2. Background and Related Work

Hardware pattern matching for virus detection has not been studied much. However, in the context of NIDS, it has been studied extensively due to high-throughput requirements, a small database, and short patterns. NIDS protect a network by searching incoming packet headers against a set of suspicious string patterns, e.g., from the Snort database. Figure 2 shows the length distribution of patterns in Snort [17]. In comparison, Figure 3 shows the length distribution of the ClamAV database. Although similar to NIDS in concept, antivirus scanning is orders of magnitude more compute intensive.

Traditional pattern matching in NIDS is performed by a network processor or ternary content-addressable memory (CAM) [6]. However, neither approach meets modern network speeds and database sizes. Instead, the parallelism and reprogrammability of FPGAs makes them ideal. There are many FPGA-based NIDS pattern-matching engines, such as [7]-[14], but most fall into two categories: finite state machine (FSM) and Bloom filter [12]. These are discussed below.

FSM solutions are usually based on the Aho-Corasick (AC) algorithm [8]. 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. Scanning begins at the root of the automaton. Every character in the input string will cause a state transition. If a pattern from the rule set indeed is present in the input string, the state transition will eventually lead the current state to a state that indicates a complete match. FSM improvements include state partitioning [10] and bitmap compression [11]. The B-FSM approach [9] uses small transition tables for higher memory efficiency than traditional AC. In general, FSM solutions suffer from two drawbacks. First, they use more memory than Bloom filters. Second, the daily updates to the virus database can result in entirely different hardware structures, resulting in lengthy delays to generate a new bitstream and periods of decreased protection.

A Bloom filter is fast and space-efficient. It works like a hash table with m one-bit entries to store a match/no-match result. The desired pattern is hashed down to a value between 0 and m-1, and only that bit is set in the table. The input string is hashed with the same function; if the corresponding table entry is not set, there is no match. However, if the entry is set, there may be a false positive match due to aliasing. To reduce false positives, k different hash functions can be used to select k different bits, and all k bits must be set for a likely match. To save space even further, multiple patterns can share one table by ORing the bit vectors, but this increases false positives.

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. In addition, because the hash tables used in Bloom filters are stored in on-chip memory, dynamic update of the rule database is possible without complete reprogramming of the FPGA bitstream.

Despite this advantage over FSMs, it has 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, the Bloom filter can only determine membership; it does not indicate which particular rule is a potential match, making the process of exact matching more compute intensive. Finally, the Bloom filter does not blend easily with regular expressions like FSM-based solutions.

The use of Bloom filters in hardware pattern matching was first proposed in [13]. Most incoming packets are presumably safe, and no malicious activity will be overlooked. Only upon a hit in the Bloom filter is the expensive process of *exact matching* performed.

Some drawbacks of the Bloom filter can be addressed by combining it with AC. In [14], patterns are split into *s*-byte long segments, which are then mapped to *s* Bloom filters of length *s* down to one byte. To link segments back to a complete pattern, AC automaton is constructed from these segments. The Bloom filter tables are stored in on-chip memory, but state transitions and the string pattern are stored in an off-chip memory. Whenever a hit is reported by a Bloom Filter, exact matching is performed by loading the segment information from off-chip memory. In [14], 2,259 strings required several megabytes of QDRII-SRAM. This works with small databases of short patterns like Snort, but it does not scale for large databases of long patterns like ClamAV.

Another approach that relates closely with the Bloom filter is perfect hashing [15, 16, 22]. 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.

An alternative to perfect hashing is Cuckoo hashing [17, 21]. Cuckoo hashing relies on k hash functions like the Bloom filter, but it uses k distinct hash tables. When preparing the tables and inserting a new pattern R, aliasing with a previously mapped rule S in table 0 may occur, i.e., $H_0(R) = H_0(S)$. In this case, S is kicked out and moved to table 1 at position $H_1(S)$. If a collision occurs there as well, the previous occupant T is kicked out and moved back to table 0 at position $H_0(T)$, and so on. It is possible to get caught in an infinite loop; in such a case, the setup process fails and a different set of hash functions need to be used. During lookups, the input string R is hashed, using $H_0(R)$ to access table 0 and $H_1(R)$ to access table 1. In [17], the hash functions and table lookups proceed in parallel using BRAM.

3. ClamAV Database

As of June 2008, the entire ClamAV database contains over 300,000 virus signatures. The database consists of two parts, the *main* database that is released with the software, and the *daily* database that is regularly updated. This work uses only the main database from the latest stable version 0.93.1 (June 9, 2008), with about 231,800 signatures. We ignore the daily databases for scientific benchmarking simply because they may be more difficult to obtain in the future.

Table 1. Number of different signatures in ClamAV 0.93.1 main database.

MD5 Checksums	Basic Patterns	Regular Expression Patterns
146,214 (63.1%)	80,262 (34.6%)	5,363 (2.3%)

Virus signatures in ClamAV, shown in Table I, can be divided to three types: MD5 checksums, basic patterns, and regular expression patterns. MD5 checksums are ignored in this work because this accounts for only 0.64% of runtime [18]. A basic pattern is a continuous byte string. A regular expression pattern is an extension of the basic pattern with OR operators, displacement gaps, and wildcards. Regular expression 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.

In this paper only basic and simple displacement gap patterns are supported. The segment length distribution of our database is shown in Figure 3. The average segment length is 102 bytes.

4. Bloomier Filter

The Bloomier filter [5] is 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.

A Bloomier filter is constructed in careful hash input order to achieve one-to-one association between a pattern rule and each table entry (hash key). The algorithm uses a table with $m \geq k*n$ entries, where k is the number of hash functions used and n is the total number of patterns. Each table entry is $\log_2(k)$ bits wide, and is used to select a specific hash function which, ultimately, determines which pattern was just matched.

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 k hash functions. That hash function points to one table entry, which by construction of the filter is associated with a unique rule/pattern.

Setting up a Bloomier filter requires some trial-and-error preprocessing of the patterns to determine the right hash functions, which patterns are associated with which table entries, and the precise table entry values. Please refer to [5] for details on how to construct a Bloomier filter.

The features offered by a Bloomier filter are similar to Cuckoo hashing. However, one striking difference is the Bloomier filter is a true perfect-hashing scheme, making it easy to verify false positives. It is more difficult to provide this feature with Cuckoo hashing. Due to collisions, Cuckoo hashing requires k comparisons to determine the matching pattern. These comparisons can be done in parallel, but readout of k full pattern strings requires extensive on-chip storage and very wide, parallel memories. Instead, a pattern checksum can be used to narrow the potential match, but the checksum function

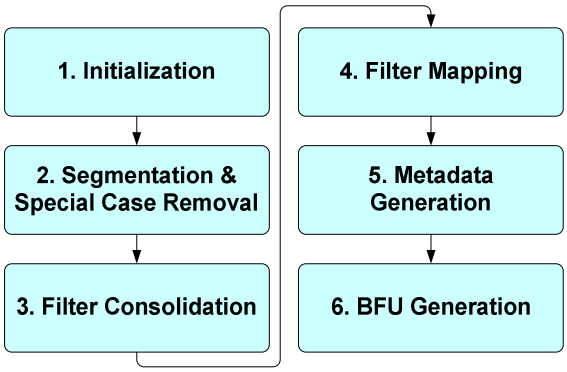

Figure 4. Flow diagram of the PERG pattern compiler.

must be chosen to ensure the stored checksums are unique across all *k* entries. In summary, the Bloomier filter solves several problems associated with Cuckoo hashing.

5. PERG Pattern Compiler

The PERG system is divided to two sections: the pattern compiler, which acts as a preprocessor, and the hardware architecture. The compiler flow, shown in Figure 4, 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. The operation of the compiler proceeds according to the following steps.

Step 1, Initialization. The compiler begins by merging the database files (main.db and main.ndb) and eliminating duplicate patterns. Most signatures in ClamAV apply to *all* file extensions, but sometimes certain extensions (such as **.exe* and **.dll* files) share identical pattern strings. In many environments, PERG may not be aware of the file extension or the purpose of the file access, so it checks all files regardless of extension. As a result, duplicate patterns that apply to different file extensions can be removed (111 out of 85,626 rules). When a potential match is reported, host software can decide what action to take and whether file extensions matter.

Step 2, Segmentation and Special-case Removal. Following initialization is the segmentation and special-case removal stage. 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 (512 bytes) are removed (5 cases). Since regular expressions other than simple displacement are not currently supported, any pattern containing a regular expression is removed as well (5228 cases). At the end of this stage, 80,282 unique patterns remain.

Step 3, Filter Consolidation. The next stage determines the number of Bloomier filter units (BFUs) and the segment length for each one. Like Bloom filters, each 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 and limited FPGA resources. Instead, we select a *small number* of distinct filter lengths and split segments of the wrong length into two *overlapping fragments* of the correct length. The actual process of splitting fragments is performed in Step 4.

```
for( i=longest_segment_or_fragment_length; i>=4; i--) {
    new_segments = basic_patterns[i] + segments[i];
    if( new_segments > 0 ) {
        accum_fragments += new_segments;
        if( (accum_fragments%BFU_LEN) > (0.9*BFU_LEN/2)
            || accum_fragments > (MAX_BRAMS-1)*BFU_LEN/2
            || (filter_cnt>0 && filter_length[filter_cnt-1]+1>=2*i) ) {
                // create a new Bloomier filter unit for length i
                filter_length[filter_cnt++] = i;
                accum_fragments = 0; // don't carry forward fragments
        } else {
            // carry forward all new segments by splitting in 2 fragments
            // (i.e., count the second fragment here)
            accum_fragments += new_segments;
        }
    }
}
```

Figure 5. Simplified C code for the filter consolidation step.

The C code to select filter lengths is shown in Figure 5. In the code, *basic_patterns[i]* is the number of basic patterns with just one segment of length *i*, and *segments[i]* is the number of segments of length *i* produced in Step 2. The algorithm starts from the longest segment length down to the shortest length supported. Xilinx's 18kb BRAM can be configured to 9-bit mode with 2048 entries, so PERG uses 8 bits for a primary CRC checksum (see Step 6) and 1 bit for the Bloomier hash selection data. Consequently, BFU_LEN is set to 2048. A Bloomier filter table requires *n*k* entries for *n* patterns and *k* hash functions. The algorithm tries to accumulate enough segments (>90%) without over-filling (<1024) a Bloomier filter table of 2048 entries. The modulo (% operator) in the code implies that a larger table, a multiple of BFU_LEN, will be created if needed to reach 90% utilization. In our actual implementation, this multiple is forced to be a power of 2 to simplify address decoding for the consolidated filter table. Also, creation of a new filter is forced if the last filter created used a length that is almost twice the current length – this ensures split fragments always overlap.

Step 4, Filter Mapping. The next stage maps all the patterns to BFUs selected during consolidation. This involves the actual splitting of segments into two overlapping fragments to match BFU lengths. The sequence of segments or fragments are represented by *link numbers* which start at 0 for the first (prefix) segment or fragment and are incremented by one for each next segment or fragment within the same pattern, ending with the last (suffix) segment or fragment.

Step 5, Metadata Generation. The next stage is responsible for the creation of rule ID numbers and collecting the "metadata" information, such as the link number, that links fragments to their predecessor and successor fragments. When a segment or fragment is matched, the Metadata Unit provides the information needed to link that segment back to the overall rule number. This way, fragments and segments can be strung together to detect an overall rule.

Most metadata is stored in an off-chip memory called the Metadata Unit. This memory is 64-bits wide and is addressed by an identifier formed by the {BFU number, hash key} pair. This pair is guaranteed to uniquely identify exactly one rule.

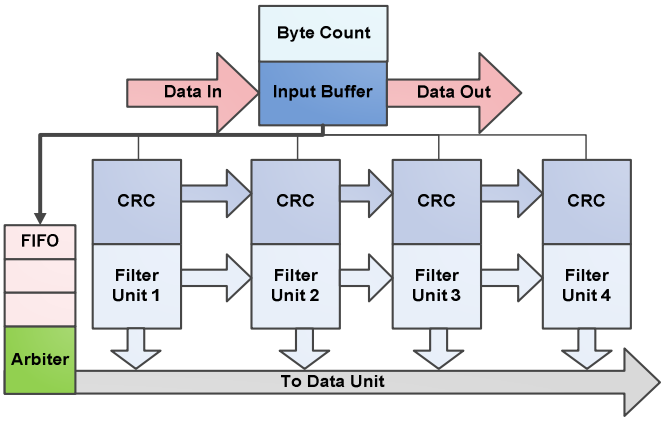

Figure 6. Inspection Unit.

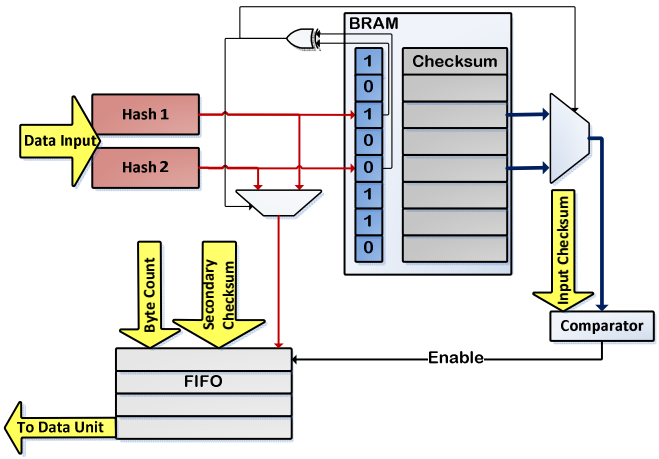

Figure 7. Bloomier Filter Unit.

The contents of the metadata record fill 64 bits:

- Rule identification number (ID)
- Link number of this segment or fragment
- Displacement gap: byte distance between the last byte of this segment and the last byte of the next segment
- Flag indicating a suffix fragment
- Secondary checksum (32 bits)

The secondary checksum filters more false positives. The primary checksum offers some filtering, but it is only 8 bits; a 32-bit secondary checksum provides even more filtering.

In some situations (236 cases), a segment string is *shared* by multiple rules. In each case, more than one metadata record is needed (one for each rule) because multiple rules can make progress towards their match. For such rules, the secondary checksum is still stored in the Metadata Unit because it is common to all rules. However, unique per-rule data is stored on-chip in a structure called the Cache. For these shared segments, the Metadata Unit record contains:

- Pointer to Cache location
- Cache burst length: # of rules sharing segment
- Secondary checksum (32 bits)

The metadata records stored in the Cache (one for each rule) follow the standard Metadata Unit format, with the difference that no secondary checksum is needed. The Cache will be accessed once for each rule that shares the segment.

Step 6, 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) [20] shown in Equation 1. In this equation, H_i is the partially-formed hash value after seeing i bytes. C_i is the i^{th} byte of the input string, S_{Li} is the amount to shift left, and S_{Ri} is the amount to shift right. This is 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. The initial hash H_0 as well as S_{Li} and S_{Ri} at each stage uniquely determine the two hash functions needed.

$$H_i = H_{i-1} \wedge ((H_{i-1} << S_{Li}) + (H_{i-1} >> S_{Ri}) + C_i) \quad (1)$$

As mentioned earlier, 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 [19]. Nevertheless, when this happens, new hash functions have to be used. If the failure occurs while the number of constructed BFUs is low, a new H_0 is generated and all constructed filters are reconstructed. This ensures good H_0 selection. However, if the failure occurs later after many filters have been constructed, a complete reconstruction of all filters is impractical. Instead, we modify the shift values for that filter length without disturbing the previous generations. This incremental hashing significantly reduces the time required for filter generation. The entire process is completely automated and takes only a few minutes to complete.

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.

6. FPGA Hardware

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

When the Inspection Unit detects a match, it determines which pattern caused the match and sends the information to the Metadata Unit along with the current Byte Count and the secondary checksum of the data. The Metadata Unit retrieves the metadata information about the suspected pattern from a small off-chip memory and transfers all data to the Fragment Reassembly Unit. Finally, the Fragment Reassembly Unit uses the metadata information for the secondary checksum and reconstruction of the pattern.

A. Inspection Unit

The Inspection Unit structure is shown in Figure 6. As mentioned earlier, the SAX hashing pipeline is fully unrolled and shared by all parallel BFUs in the same way as [17], resulting in a sliding-window hash. The primary and secondary checksums use the same pipelined structure.

Figure 7 shows the internals of each BFU. Each BFU has a unique BFU ID which is used to partially form the memory address for the Metadata Unit. The BFU contains

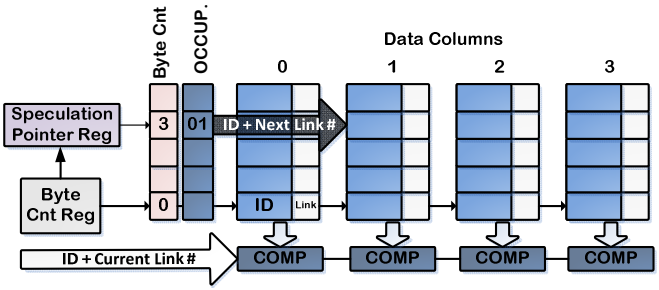

Figure 8. Circular Speculative Buffer.

a hash filter table of two columns stored in BRAM. The first 1-bit column is the Bloomier hash select data. On a match, the XOR gate in the figure selects a hash function to determine which segment matched. The second column stores the primary checksum of this segment to reduce false positives. In our experiments, we use CRC-8 checksums, but others can be used as well.

We can reduce false positives in two other ways: introducing more hash functions in the Bloomier filter, and adding a 1-bit column for Bloom (not Bloomier) filtering. The increased number of hash functions reduces the setup failures during Bloomier filter construction, but it lowers memory utilization and increases table size and external memory by a factor of k_{new}/k [5]. In this paper, we rely mostly on the CRC checksums for false positive reduction and use a minimal number of hash functions (k=2). Moreover, two hash functions map well to the dual ports of each BRAM.

To handle the possibility of multiple BFUs reporting hits in the same cycle, each BFU has a small *Filter FIFO* implemented with LUTs operating in SRL16 shift register mode. The Filter FIFO stores the hash key (table entry number), Byte Counter value, and secondary checksum. The Filter FIFO does not need to store the full 32-bit Byte Counter value; the number of bits depends on the depth of the Filter FIFO and the number of BFUs in the system according to Equation 2.

$$\#\text{ of bits} = \text{ceiling}(\log_2(\#\text{ of entries in filter FIFO} \times \#\text{ of BFUs})) \quad (2)$$

Simultaneous requests are processed in Byte Count order by an *Arbiter* inspecting each Filter FIFO output, implemented as a pipelined sorting network. The *Arbiter FIFO* stores the complete Byte Count whenever one or more match is reported.

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. Only read access is needed, so the memory can be RAM, E^2PROM, or Flash; faster memory reduces the size of FIFOs needed inside the FPGA. The address is derived from the BFU ID and the hash key, which is always unique for each fragment. We considered using a pointer table stored on-chip as in [19] to improve utilization of the Metadata Unit memory. However, this increases latency, adds complexity, and uses on-chip memory to save a small amount of external memory.

C. Fragment Reassembly Unit

The Fragment Reassembly Unit, shown in Figure 1, reconstructs full patterns from fragments or segments. First, the *Comparator* compares the secondary checksum from the metadata to reduce false positives. A mismatch is ignored. A match with a complete pattern (i.e., single-segmented pattern), generates an interrupt to the host for exact matching. A match with one fragment of a multi-segmented pattern enters the *Reassembly FIFO* and reaches the *Reassembly Controller*. The Reassembly Controller performs final processing of the fragment. Just 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* for the rule. The metadata provides the required information about how fragments link together. Usually this data is provided by the Metadata Unit, but sometimes it is provided by the Cache. The order and precise locations in which fragments of a rule are discovered is tracked by the *Circular Speculative Buffer*.

The *Cache* stores metadata for fragments shared by multiple rules. Unlike typical caches, which remember recently-accessed information, the data in this cache never changes; it is only used as a fast read-only memory. Shared fragments are shared by 2 rules on average, and most fragments are shared by 4 or fewer rules. For each rule sharing the fragment, one Cache access is needed to retrieve the linking metadata, and one update to the rule's overall progress is made in the Circular Speculative Buffer.

The *Circular Speculative Buffer* (CSB) shown in Figure 8 tracks the progress of *multiple traces* for multi-segmented rules. The 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, we simplify our CSB implementation to only schedule events up to N bytes into the future, where N is the number of entries (rows) in the circular buffer (512 here).

Operation of the CSB is best explained by example. Initially, all entries are empty. Suppose a single-byte fragment is immediately matched, making the current Byte Count equal to 0, the position of the last byte in the fragment. The metadata indicates the *displacement gap* is 3 bytes, meaning the last byte of the next fragment for this rule will appear 3 bytes into the future. This 3 is added to the current Byte Counter to form the Speculation Pointer (SP), which is used to index the CSB. If necessary, the SP wraps around the end of the table, creating a circular buffer. The occupancy of the SP row is 0, so a new entry is added in Data Column 0, the Occupancy is incremented, and the future Byte Count for that row is written as 3. The Data Column stores 2 pieces of information: the link number of the next fragment and the current rule ID. As additional input is scanned, additional fragments may be matched and the Byte Count is incremented. If a matching fragment is found at a Byte Count of 3, that row of the

CSB is checked to ensure the Byte Count matches; it will not match if the row holds only stale-dated speculations. The Occupancy indicates how many rule traces are in-flight and expecting a fragment match at this time. The rule ID and link numbers of the current matched fragment are compared to those active traces in Data Columns 0 through Occupancy-1. For each Data Column (active trace) that matches the current fragment, action is taken. The type of action can be either (1) adding a new entry to the CSB for the next expected fragment, or (2) if the matched fragment is a suffix, then the entire rule ID is matched and an interrupt is sent to the host for exact matching.

In our CSB implementation, the number of Data Columns is fixed (4 here). 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. This is not harmful because it merely means a match may be generated when no match should have occurred, meaning a false positive is created. If the entire rule matches, the host will do an exact match to verify the pattern. While this scheme does introduce more false positives, 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.

It is possible to modify the CSB to behave more like a true time wheel and schedule events far into the future. To do this, each Data Column must include a tag field to store the higher-order bits of the Byte Count. This obsoletes the CSB Byte Count column, but complicates the addition of new events.

7. Simulation Setup and Results

We evaluated PERG by developing a cycle-accurate RTL model in C. The design assumptions (e.g., memory block sizes) target a Virtex-II Pro VP100 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 ¼ clock speed (50 MHz). PERG usually accepts one byte every clock cycle from a continuous file stream. However, due to internal multi-cycle operations and limited FIFO space, the PERG engine sometimes (~5%) stops accepting a new byte and applies backpressure flow control. This stalling can be reduced using larger FIFOs.

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.

Our test starts with 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 two or

Table 2. PERG System Parameters

Pre-Processor Parameters	**User Defined**	**Value**
Minimum Segment Length (bytes)	Yes	4
Maximum Segment Length (bytes)	Yes	150
# of Hash Functions	Yes	2
# of Data Columns in CSB	Yes	4
Maximum Byte Displacement	Yes	512
Hardware System Parameters		
Number of Bloomier Filter Units	Generated	26
Maximum Hash Table Size (entries)	Generated	32,768
Filter Unit FIFO Depth	Yes	32
Maximum # of Cache Entries	Generated	512
Maximum # of Entries in State FIFO	Yes	64
Maximum # of Entries in Arbiter FIFO	Yes	64
Secondary Checksum (bits)	Yes	32

Table 3. 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 4. Simulation Results

	Single File (Ubuntu7_10_x86.iso)	Extracted Files (274)
Num of Bytes Scanned	729,608,192	727,677,929
Number of False Positives	4	4
False Positive Probability for Each Byte Scanned	0.0000005%	0.0000005%
Number of Off-chip Memory Requests	73,739,161	73,666,748
Probability of Off-chip Memory Request for Each Byte Scanned	10.11%	10.12%
Off-chip Memory Throughput	19.27 MB/s	19.31 MB/s
Number of Secondary Check Hits	9,770	9,737
Probability of Secondary Check Hits for Each Byte Scanned	0.13 %	0.13 %
Average Throughput	190.7 MB/s	190.7 MB/s

Table 5. Memory Density Comparison

	# of Chars	Memory (kb)	Memory per Character (bits/char)
PERG (raw)	8,224,848	2,612	0.335
PERG (after processing)	11,452,898	2,612	0.238
B-FSM* [9]	25,200	656	26.4
Cuckoo Hashing [17]	68,266	1,116	16.7
PH-Mem [16]	20,911	288	14.1
HashMem [15]	18,636	630	34.6
ROM+Coproc [22]	32,384	276	8.73

Due to PERG's use of a pattern compiler, two sets of data are presented. "Raw" data is calculated with the original rule database after the special-case removal stage. "After processing" is calculated at the memory-mapping stage, after going through filter-mapping.

*: Data for 20 BFSM's using case-insensitive patterns

more rules. During matching, file extensions are ignored and left to the host system for consideration.

Table III 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.

To evaluate the performance of our design, we use the Ubuntu 7.10 ISO image as a data source and summarize results in Table IV. As expected, ClamAV software finds no virus matches in this data. Hence, all matches found by PERG are false positives. In the Single File test, the entire ISO image file (696 MB) is scanned and 4 matches are detected by PERG. In the Extracted Files test, the ISO image is extracted into 274 individual files totaling a size of 694MB (files shorter than 4 bytes are excluded) and 4 matches are also detected. Three of these are true pattern matches, but ClamAV considers them safe due to mismatched file extensions. We also manually injected some virus patterns into the data and PERG was able to find all of them.

PERG is the first hardware engine for anti-virus pattern matching, so we cannot compare to similar work. Instead, Table V 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 (RAM is cheap, FPGAs are not).

As shown, the results are quite promising. The primary and secondary checksums together ensure a low number of false-positives. The experimental throughput is slightly worse than the maximum throughput of 200 MB/s due to backpressure flow control; flow control issues can be reduced using deeper FIFOs. Also, we anticipate the final clock frequency can be increased, if needed, by very deep pipelining.

8. Conclusions and Future Work

PERG is a novel FPGA-based pattern matching engine designed to pack a large number of patterns into a small amount of resources. Through the use of Bloomier filters and the pre-processing compiler, PERG can fit over 80,000 patterns ranging up to hundreds bytes long in a single FPGA with a single 4 MB off-chip memory.

Novel aspects of PERG include the use of Bloomier filters to make exact matching easier, checksums to reduce false positives, consolidation to reduce the number of Bloomier filters and improve the density of rules fit into each filter, and the use of a Circular Speculative Buffer (CSB) as a means to track the state of patterns that have fixed-length gaps or that have been divided into multiple fragments due to consolidation. If the number of patterns being tracked accurately is too large for the CSB, it won't miss viruses but will instead compensate by reducing accuracy and indicating more false positives.

In terms of memory density, PERG shows a promising 26x improvement over the best NIDS pattern match engine. To fit most of the ClamAV database, PERG uses slightly over 2.5Mb. This amount of memory can be found in mid-range Stratix III or Virtex-4 devices. Among low-cost FPGAs, it can also be found in high-end Cyclone III devices and perhaps the next generation of Xilinx Spartan devices. This shows that it is possible to build a low-cost virus scanning accelerator, potentially putting an FPGA in every personal computer.

We are currently implementing our design on a Xilinx Virtex-II Pro and testing the approach against more data streams. The simulator takes about 1 day to scan the single large Ubuntu file. We plan to investigate the scalability of the approach and are looking for a good way to generate larger pattern databases. In addition, we are currently verifying an extension that supports wildcards and regular expressions.

Acknowledgments

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

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., 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] F. Yu, R. Katz, and T. Lakshman, "Gigabit rate packet pattern-matching using TCAM," *IEEE Int'l. Conf. on Network Protocols,* 2004, 174-183.

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

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

[9] J. van Lunteren, "High-performance pattern-matching for intrusion detection," *IEEE Int'l. Conf. on Comp. Comm.*, 2006, 1-13.

[10] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," *International Symposium on Computer Architecture*, 2005, 112-122.

[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *Proc. INFOCOM 2004,* 2004*, 4*, 2628-2639 vol.4.

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

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

[14] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE Journal on Selected Areas in Communications,* 2006*, 24*, 1781-1792.

[15] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," *FPL,* 2005, 644-647.

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

[17] T. N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," *ICFPT 2007,* 2007, 121-128.

[18] 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.

[19] 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.

[20] M. V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," *Int'l Conf. on Database Systems for Advanced Applications, World Scientific Press,* 1997, 215-224.

[21] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol 51, 2004, pp. 122-144.

[22] 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.