PERG-Rx: A Hardware Pattern-Matching Engine Supporting Limited Regular Expressions

Johnny Tsung Lin Ho
University of British Columbia
johnnyho@ece.ubc.ca

Guy G. F. Lemieux
University of British Columbia
lemieux@ece.ubc.ca

ABSTRACT

PERG is a pattern matching engine designed for locating pre-defined byte string patterns (rules) from the ClamAV virus signature database in a data stream. This paper presents PERG-Rx, an extension of PERG that adds limited regular expression support for wildcard patterns used by rules that represent polymorphic viruses. To reduce the amount of state needed to track so many regular expressions, PERG-Rx employs a lossy scheme which increases the rate of false positives detected as the required state grows. The scalability and dynamic updatability of the PERG-Rx architecture to database updates are also evaluated.

Categories and Subject Descriptors
B.6.0 [**Logic Design**]: General – *FPGA, pattern-matching engine*

General Terms: Algorithms, Performance, Design, Security

Keywords
FPGA, Pattern Matching, Antivirus, Regular Expression

1. Introduction

Our previous work, PERG [1], is a hardware accelerator for pattern matching with the ClamAV virus database [3]. In this paper we present an extension to the original PERG architecture to handle patterns containing wildcards. Wildcard support is necessary for the detection of polymorphic viruses in ClamAV. To add such critical support, the extended architecture PERG-Rx, shown in Figure 1, features a new *Wildcard Table* unit. By trading off a slight increase in false-positive probability, the wildcard table is able to handle all types of wildcards found in the ClamAV database with low hardware overhead.

In addition to limited regular expression support, this paper also analyzes on scalability and dynamic updatability of PERG. Although a PERG instance of the ClamAV main database already has very high memory density per rule character (0.354 bit/char), it still has a theoretical capacity of 13.3% unused space for future rules. That is, up to 12,935 more rules might be added without changing the FPGA bitstream. Using the latest incremental (daily-update) database from ClamAV, we demonstrate that most new rules can be perfectly-hashed to fit directly into an existing instance without further hardware changes. To handle the cases where a perfect-hash cannot be achieved, we add to PERG-Rx the ability to alter the hash circuits with minimal hardware overhead and without the need to regenerate a new FPGA bitstream.

The rest of the paper is organized as follows. Background is in Section 2. Details of ClamAV's signature database are in Section 3. The pattern compiler and the PERG-Rx hardware architecture are discussed in Sections 4 and 5, respectively. Wildcard table and limited regular expression support are explained in Section 6. Scalability and dynamic updatability are analyzed in Section 7. Simulation results are presented in Section 8. Finally, conclusions and future work are given in Section 9.

FPGA'09, February 22–24, 2009, Monterey, California, USA.

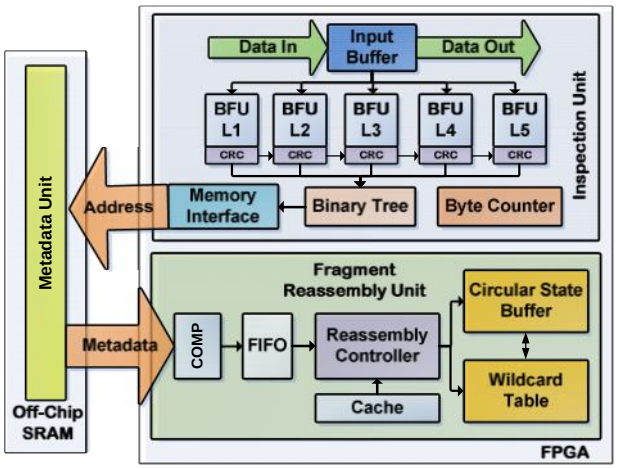

Figure 1. Top-level architectural diagram of PERG-Rx.

2. Background and Related Work

Most modern pattern matching engines are FPGA-based and fall into two categories: finite state machine (FSM) [8] and Bloom filter [6]. While each solution has its own advantages, Bloom filters offer much higher density than FSMs, making Bloom filters superior for large-scale databases like ClamAV.

A Bloom filter [6] uses a hash table with m 1-bit entries to store a match/no-match result. By using hash functions, Bloom filter memory consumption is independent of the pattern length, giving a Bloom filter a much higher density than the FSM approach. However, Bloom filter approaches also have several disadvantages. First, each pattern length uses its own hash table. This leads to a large number of filter units as both Snort and ClamAV have a wide range of pattern lengths. Second, because of false positives due to hash aliasing, verification is required upon a hit, forcing additional computation. To make matters worse, Bloom filters can only determine match/no-match; they do not indicate which particular rule is the potential match. Hence, the process of exact matching is fairly computationally intensive.

PERG [1] uses Bloomier filter [5], 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. Due to limited space, please refer to [5] for construction and proof of Bloomier filters.

3. ClamAV Database

Virus signatures in ClamAV, shown in Table 1, 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 [2]. 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. Table 2 summarizes the various regular expression operators in ClamAV. Regular expression support is necessary for detection of polymorphic viruses in ClamAV. Note that PERG supports only the single-byte and displacement wildcards ? and {n}, which insert fixed-length gaps between string fragments. PERG-Rx adds support for the others which require arbitrary-length gaps.

4. PERG-Rx Compiler

The PERG-Rx system is divided to two sections: the pattern compiler, which acts as a preprocessor, and the hardware architecture. The compiler flow 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. In comparison with the compiler flow in PERG [1], the only difference is the addition of *OR-expansion* stage, added right after the *filter-mapping* stage. To handle Byte-Or regular expressions in ClamAV, each rule containing one or more Byte-Or operators is expanded to all the possible string combinations. A string with one Byte-Or operator would be expanded to two strings, a string with two Byte-Or operators would be expanded to four strings, and so on.

5. PERG-Rx Hardware

The PERG-Rx hardware architecture, shown in Figure 1, is divided into Inspection, Metadata, and Fragment Reassembly Units. Inspection Unit filters the input data stream through parallel Bloomier filter units (BFUs) and verifies primary (8-bit) checksums. 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. For details about operations of these units as well as circular speculative buffer (CSB), please refer to [1]. The main change since PERG is the addition of the Wildcard Table.

The Wildcard Table (WT) is accessed whenever a pattern fragment matches and the metadata indicates it is preceded and/or followed by a wildcard. Details about the internal operation of the WT will be discussed in Section 6; for now, we will treat WT as a black box that can somehow track the state a wildcard trace through the datastream. When the Reassembly Controller sends out new metadata, the same metadata is presented to the CSB and the WT in parallel. The metadata process, in concept, can be divided to two phases: a verification phase for determining if the incoming string fragment is *currently expected*, and a speculation phase for recording the *next expected* segment of the rule trace. As a result, it is possible that both WT and CSB are involved in the same metadata process. For instance, metadata expected by an ongoing trace in the CSB may indicate it is followed by a *within* wildcard. In such a scenario, the CSB is used to verify that the incoming fragment is expected by an ongoing trace. Upon verification, the CSB will send a control signal to the WT so the WT will record the expectation for the next segment.

Table 1. 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%)

Table 2. Regular Expression Operators in ClamAV

Symbol	Definition
(X|X)	Byte-Or
?	Single-Byte Wildcard
*	(Any-Number-of-Byte) Wildcard
{n}	n Byte Displacement
{n-}	At-Least (n-Byte) Wildcard
{-N}	Within (n-Byte) Wildcard
{n-N}	Range wildcard

6. Limited Regular Expressions

Before getting to the internal mechanism of the WT, it is appropriate to describe how we treat the various types of wildcard regular expressions in Table 2. We begin by converting the single-byte (?) and any-number-of-byte (*) wildcards. A single-byte wildcard is a displacement of fixed (1 byte) distance, and therefore can be handled by CSB alone. On the other hand, the any-number-of-byte wildcard is converted by the pattern compiler to an at-least wildcard, {n-}. As a hash-based solution, a *true* hit of any segment in PERG-Rx is only detected after the last byte of the pattern appears in the data stream. Take the pattern ABC*DEF as an example. First, the pattern would be split into two segments ABC and DEF. Once a hit of ABC is detected, the appearance of DEF is only possible after at least 3 more bytes have passed. Therefore the pattern ABC*DEF is converted to ABC{3-}DEF. After conversion, three types of wildcard remain: at-least {n-}, within {-N} and range {n-N}.

Unlike the others, the range wildcard contains two pieces of data: the lower and upper bound values. To minimize storage in both metadata and the WT, only the upper bound N is kept. As a result, range wildcards are converted to within wildcards ({-N}). *This step of the conversion is lossy*, meaning there will be some new false positives resulted from discarding the lower bound.

It is important to emphasize that we are not handling the entire set of regular expressions, but only those found in ClamAV. Most importantly, the WT is not a precise regular-expression handler but a lossy approximation that provides limited regular expression support using minimal hardware resources. While fundamentally very different in structure, the WT is similar in properties to a Bloomier filter as they both offer high storage

density, zero false-negative probability, and a small false positive probability. Any regular expression in ClamAV can be mapped into the PERG-Rx WT and remain detectable.

WT is essentially a one-dimensional table. Each entry is directly indexed by the rule ID of the incoming metadata and contains four attributes: *Valid*, *WildcardType*, *ByteRange*, and *LinkNumber*. With the exception of the valid bit, all the attributes are stored on-chip in BRAM. WildcardType is a flag that indicates whether the next expected segment is preceded by an at-least or within wildcard. ByteRange is the displacement value, *n* or *N*. Finally, LinkNumber is used to track the arrival sequence order among multiple segments in the rule.

Each rule may contain multiple wildcards. While searching the datastream, this can lead to tracking the progress of multiple states or traces for each rule. Since there is only one entry per rule, *the WT can only keep one trace* per rule. To do this, the trace only progresses forward (i.e., advances in link number); each entry remains static until only the next expected segment arrives, thus any arrival of current and previous segments of the same rule are ignored. The link numbers provide the information needed to only make forward progress. The Valid bit, stored in a flip-flop, is used to indicate the relevance of the entry information and reset upon the arrival of a new file stream.

Clearly, the above method works for at-least wildcards without any increase in false positive probability. Take the example of the pattern ABC{3-}DEF{3-}EFG and assume the WT entry has already detected up to the second fragment DEF. Regardless of when and how many instances of ABC and DEF appear later, our only concern is the appearance of EFG, which can appear any time after three bytes have passed after the first DEF is detected. In fact, which fragments and combinations of wildcards precede the fragment EFG is irrelevant at this point as the trace only progresses forward. In the case where a fragment is preceded by a wildcard and followed by a regular fragment (as the DEF in the pattern ABC{-3}DEF{3}EFG), its corresponding WT entry is left unchanged as the speculation phase is performed in the CSB not in the WT. As a result, the WT will still expect the fragment DEF while the CSB expects the next fragment EFG. Hence, the concurrent use of CSB and WT follows two traces.

For within wildcard, everything works the same as at-least wildcard with one exception: an update to the byte range. Unlike the at-least condition, the within condition expires after the given number of bytes has passed since detection of the last fragment. One way to get around such complexity is to simply remove the upper limit by converting the within wildcard back to an at-least wildcard by using the byte length of the second fragment as lower bound for *n*. This approach works at the cost of increased false positive probability due to loss of the upper bound.

Instead, PERG-Rx keeps the upper bound by refreshing the ByteRange condition every time the current fragment is matched again. Whenever the current fragment (whose next link number is the same as the link number in the WT entry) re-appears in the datastream, the upper bound condition ByteRange is increased accordingly. To do this, WT is essentially monitoring two traces: both the current and next link numbers of the incoming metadata.

In any true (non-lossy) trace of a within wildcard, the current fragment must always arrive before the appearance of the next fragment at a byte position satisfying the ByteRange. In our lossy WT trace, the range attribute will always be refreshed just in time in a true trace. However, additional false positives are introduced when the ByteRange is refreshed by a current fragment that appears but does not satisfy all of the proper preceding fragments in the overall rule. The number false positives introduced by this scheme should be much lower than the alternative discussed earlier of removing the upper limit; we have not investigated the difference, but expect it is significant.

One issue with WT is that forward-only lossy progress may lead to a large number of verifications. For example, for the rule ABC{-3}DEF{-3}EFG, consider that once progress is made to the last fragment EFG, a verification must be done whenever another EFG segment is matched in the datastream. The only time this history state is cleared is when a new file begins. However, our experiments show that with a Bloomier filter and two levels of checksum, it is very unlikely a trace would progress that far in the first place unless such a pattern truly does appear in the data stream. Alternatively, host software may be able to roll back the WT state for a rule if verification by the host fails.

7. Scalability and Dynamic Updateability

Given the rapid growth and frequent updates to virus databases, it is important to verify PERG/PERG-Rx scalability and updateability. By scalability, we mean the architecture can achieve high density/utilization of patterns in each BFU without reporting a large number of false positives. By a dynamic update, we mean the ability to add new rules to previously generated hardware without the need to recompile the RTL. Because recompiling and verifying RTL can take days, this feature allows immediate virus protection during that time period.

After compiling the main database, the utilization and configuration of each BFU is analyzed. Recall from [1] that the filter consolidation algorithm attempts to pack each BFU above 90%. The exceptions to this are: the utilization bar is lowered greatly for short lengths, and a new BFU length is forced whenever the current length is equal to half of the last-assigned BFU length (32/16 split) or if the current length has an excess number of fragments that is much greater than the BFU_TABLE size. There is still room available in each BFU to map new rules.

The maximum number of patterns that can be mapped onto a Bloomier hash table of *m* entries with two hash functions is *m/2*. Consider a specific PERG hardware instance holding the main ClamAV database. Theoretically, in this instance with 1.7 fragments per rule on average and 86.7% utilization in filter table entries, the system is still capable of accepting 12,935 more rules with the exact same hardware resource usage. This estimation does not account for new rules that contain common fragments – these utilize the on-chip cache, which has 87% utilization in this instance.

One possible counter argument to the scalability claim is that as a derivative of a Bloom Filter, the false positive probability in each Bloomier filter will increase with higher levels of utilization; as a result, performance will degrade due to more frequent off-chip Metadata Unit accesses. In reality, however, the hit rate for each BFU is relatively evenly distributed with bursts at a couple of filter lengths. While due to these bursts we cannot conclude the hits in BFUs are evenly distributed, it does indicate that there is no obvious correlation between the hit rate and utilization. PERG-Rx relies mostly on CRC8 checksum for false positive hits.

Another possible argument against our scalability claim is that the probability of perfect-hashing a new pattern into the BFUs would be low. To determine this behavior, we evaluated the dynamic updatability of PERG-Rx experimentally using the daily signature update (29-Sept-2008) as a test subject. After preprocessing through the compiler flow, 1033 rules were identified and broken down into 2040 unique fragments.

We test the probability of successful insertion to the BFUs as follows. BFUs were generated with the original main database. Two tests were then run. In Test 1, each fragment is inserted one by one and removed before insertion of the next fragment so they do not interfere each other. In Test 2, the fragments are inserted cumulatively and never removed unless it fails to map; the order they are inserted is arbitrary without any optimization. In both tests, only one fragment from one rule cannot be perfectly-hashed by the hard-coded hash functions. Mapping this requires new hash functions which can be altered without a new bitstream.

8. Simulation Results

PERG-Rx has been implemented in both C as a cycle accurate simulator and synthesizable Verilog. According to Xilinx ISE, the design can operate at 180MHz on a Virtex-II Pro VP100 FPGA. For the Metadata Unit, we use 4 MB SRAM with 64 bit data width operating at one-quarter of the core frequency. While the Verilog source code is functional under Modelsim, we do not have a complete platform to evaluate PERG-Rx in hardware yet. Hence, results regarding false-positive probability, BFU hit rates, etc. are determined from the C simulator.

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 1,246 patterns. We end up with a total of 84,380 rules containing a total of 164,864 fragments and 12,049,565 bytes (characters). A total of 1,230 segments/fragments are identified to be shared by two or more rules. During matching, file extensions are ignored and left to the host system for consideration.

Table 3 shows the resource utilization as well as comparison with other pattern matching designs. The resource usage in PERG-Rx includes the SRAM memory controller used. Note with PERG-Rx as the first hardware engine for anti-virus pattern matching, so we cannot compare to similar work. Instead, Table 3 roughly compares PERG-Rx to NIDS systems using Snort. Performance Efficiency Metric (PEM) is a common metric used by pattern-matching engine designs and defined as the ratio of throughput (Gbps) to logics cells per character.

9. Conclusion and Future Work

PERG-Rx is an extension of the PERG pattern matching engine designed to support limited regular expressions, namely the various types of wildcards found in the ClamAV database. The support is added at a very small hardware cost. Despite the lossy state tracking mechanism which maps multiple traces for each rule with wildcard operators to a single trace, experimental results show that the false positive rate remains as low as the original PERG design.

Table 4. Simulation Results

	Single File (Ubuntu7_10_x86.iso)	Extracted Files (274)
# of Bytes Scanned	729,608,192	727,677,929
# of False Positives	4	4
False Positive Prob. for Each Byte Scanned	0.0000005%	0.0000005%
# of Off-chip Mem. Req.	82,499,591	80,500,329
Prob. of Off-chip Mem. Req. for Each Byte Scanned	11.31%	11.07%
Off-chip Mem. Throughput	21.56 MB/s	21.10 MB/s
# of Secondary Check Hits	5,931,478	4,005,953
Prob. of Secondary Check Hits for Each Byte Scanned	0.81 %	0.55 %

Acknowledgements

The authors thank CMC Microsystems/SOCRN for providing equipments, as well as NSERC, Altera, and Actel for funding.

10. References

[1] J. Ho and G. Lemieux, "PERG: A Scalable FPGA-based Pattern-matching Engine with Consolidated Bloomier Filters," *ICFPT*, 2008, 73-80.

[2] 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.

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

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

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

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

[7] Y. Cho and W. M-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network," *FCCM*, 2005, 215–224.

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

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

Table 3. Resource Utilization and Comparison

System	Device (Xilinx)	Freq. (MHz)	# of Chars	# of LCs	Mem[*1] (kb)	LCs per Char	Mem.per Char (bits/char)	Throughput (Gbps)	PEM
PERG-Rx	XC2VP100	180	8,645,488	42,809	3,024	0.00495	0.354	1.3[*2]	262.62
Cuckoo Hashing [4]	XC4VLX25	285	68,266	2,982	1,116	0.043	16.7	2.28	53.02
HashMem [9]	XC2V1000	338	18,636	2,570	630	0.140	34.6	2.70	19.60
ROM+Coproc [7]	XC4VLX15	260	32,384	8,480	276	0.260	8.73	2.08	8.00

*1: Derived from BRAM usage only