PERG-Rx: An FPGA-based Pattern-Matching Engine with Limited Regular Expression Support for Large Pattern Databases

by

Johnny Tsung Lin Ho

B.A.Sc, University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

The Faculty of Applied Science

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

September, 2009

© Johnny Tsung Lin Ho 2009

Abstract

Pattern-matching is a fundamental technique found in applications like a network intrusion detection system (NIDS) and antivirus. By comparing an input data stream against a set of predefined byte-string patterns, malicious activities can be detected. Due to the increasing pattern database size, hardware engines are frequently being used to accelerate pattern-matching because software-based solution can no longer keep up.

This thesis presents PERG, a FPGA-based pattern-matching engine targeted for the large pattern database found in Clam AntiVirus (ClamAV). Previous hardware pattern-matching engines have targeted the Snort database used in NIDS. PERG is the first hardware pattern-matching engine designed for the ClamAV pattern database, which is several times larger than Snort in both the number of patterns and the range of pattern lengths. Among hash-based pattern-matching hardware, PERG is the first to integrate limited regular-expression support. In ClamAV, this feature is needed for polymorphic virus detection. Finally among hash-based hardware pattern-matching engines, PERG is the first to implement Bloomier filters. With Bloomier filters, the process of exact matching to verify a false positive in PERG is much simpler and faster than other hash-based pattern-matching engines which use traditional Bloom filters.

To reduce hardware resources, PERG uses a pattern compiler to transform the pattern database into an optimized hardware implementation. The compiler packs as many pattern strings as possible into each Bloomier filter unit, a hardware module, by consolidating several different pattern lengths into the same filter unit. To further save on hardware resources, limited regular-expression support for wildcard patterns is provided using a lossy scheme, which trades off a slight increase in false-positive probability to gain significant savings in hardware resources. False-negative probability in PERG remains zero.

Using the ClamAV antivirus database, PERG fits 84,387 patterns containing over 8,645,488 characters into a single modest FPGA chip with a small (4 MB) off-chip memory. It uses just 26 filter units, resulting in roughly 24.7x improved density (characters per memory bit) compared to the next-best NIDS pattern-matching engine which fits only 1/126th the characters. PERG can scan at roughly 170MB/s and match the speed of most network or disk interfaces.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables vi

List of Figures vii

Acknowledgements ix

Chapter 1. Introduction 1

1.1 Motivation 2

1.2 Contributions 4

1.3 Thesis Outline 5

Chapter 2. Background 6

2.1 Pattern Matching Overview 6

2.2 Existing Solutions 7

2.2.1 Finite State Machine (FSM) 8

2.2.2 Hash Checksum 9

2.2.3 Bloom Filter 10

2.2.4 Perfect Hash 12

2.2.5 Cuckoo Hash 13

2.3 Bloomier Filter 14

2.4 Shift-Add-XOR Hash Function 17

Chapter 3. PERG Overview 18

3.1 Clam AntiVirus Signature Database 18

3.1.1 Type of Patterns 18

3.1.2 Database Characteristics 20
3.2 The PERG System 21
3.2.1 Configurable Hardware 22
3.2.2 Pattern Compiler 24
Chapter 4. Pattern Compiler 26
4.1 Initialization 26
4.2 Segmentation 27
4.3 Special Case Removal 28
4.4 Filter Consolidation 28
4.5 Filter Mapping 30
4.6 Or-Expansion 31
4.7 Metadata Generation 31
4.7.1 Unique Segment 32
4.7.2 Common Segment 33
4.7.3 Rule ID Assignment 34
4.8 BFU Generation 35
4.9 RTL Generation 36
Chapter 5. PERG Hardware 37
5.1 Inspection Unit 37
5.1.1 Bloomier Filter Unit 38
5.2 Metadata Unit 40
5.3 Fragment Reassembly Unit 40
5.3.1 Cache 41
5.3.2 Circular Speculative Buffer 41
5.3.3 Exact-Match Request to Host 43
5.4 Wildcard Support 43

5.4.1 Wildcards ... 44
5.4.2 Wildcard Table ... 46
Chapter 6. Experimental Results ... 50
6.1 Experiment Setup ... 50
6.1.1 Cycle-Accurate Simulator ... 50
6.1.2 RTL Implementation ... 51
6.1.3 Pattern Database ... 51
6.1.4 Test Setup ... 51
6.2 Performance Results ... 52
6.2.1 Throughput ... 52
6.2.2 Resource Consumption ... 53
6.2.3 Comparison ... 53
6.3 Impact of Filter Consolidation ... 56
6.4 Scalability and Dynamic Updatability ... 57
6.4.1 Scalability and Dynamic Updatability using ClamAV ... 57
6.4.2 Scalability and Dynamic Updatability using Random Patterns ... 59
Chapter 7. Conclusions ... 64
7.1 Future Work ... 65
7.1.1 Hardware Interface ... 65
7.1.2 Support for Interleaving Filestreams ... 65
7.1.3 Update and Expansion Options ... 65
7.1.4 Alternative Databases ... 66
7.1.5 Integration with Antivirus Software ... 67
7.1.6 Eliminating Special Cases ... 67
References ... 68

List of Tables

Table 3.1: Ratio of different types of ClamAV virus signatures 18

Table 3.2: Regular expression operators in ClamAV.. 19

Table 3.3: Statistical properties of ClamAV pattern database (basic only)................................. 20

Table 3.4: User definitions 25

Table 4.1: Type flag encoding format 32

Table 5.1: Conversion of different ClamAV wildcard operators 46

Table 6.1: PERG system parameters 50

Table 6.2: Simulated performance results 52

Table 6.3: Overall resource consumption 53

Table 6.4: Resource utilization and performance comparison.. 55

Table 6.5: LP, MP, TLP, and TMP comparison 55

Table 6.6: PERG vs ClamAV throughput 56

Table 6.7: Impact of filter consolidation 56

Table 6.8: BFU utilization and insertion 58

Table 6.9: Baseline configurations.. 60

Table 6.10: Impact of database change in Config. 1 61

Table 6.11: Impact of database change in Config. 2 61

Table 6.12: Impact of database change in Config. 3 61

Table 6.13: Impact of database change in Config. 4 61

Table 6.14: Impact of utilization on insertion for Config. 1 62

Table 6.15: Impact of utilization on insertion for Config. 2 62

Table 6.16: Impact of utilization on insertion for Config. 3 62

Table 6.17: Impact of Utilization on Insertion for Config. 4 62

List of Figures

Figure 1.1: An illustration of the pattern matching problem 1

Figure 1.2: Pattern length distribution of Snort database [3].. 3

Figure 1.3: Pattern length distribution of ClamAV database.. 3

Figure 2.1: An AC automaton.. 8

Figure 2.2: A Bloom filter 11

Figure 2.3: Construction of cuckoo hash scheme 13

Figure 2.4: Example Bloomier filter construction in Phase 1 (left), Phase 2 (middle) and Phase 3 (right)... 15

Figure 2.5: Structure with hash pipeline sharing 17

Figure 3.1: Pattern length distribution of ClamAV database... 20

Figure 3.2: High-level architectural dataflow.. 22

Figure 3.3: PERG hardware decision tree 23

Figure 4.1: Flow diagram of the PERG pattern compiler.. 26

Figure 4.2: Offset adjustment 28

Figure 4.3: Simplified C code for filter consolidation step 30

Figure 4.4: Byte-Or expansion... 31

Figure 4.5: Metadata format for unique fragment.. 32

Figure 4.6: Metadata format for unique fragment.. 33

Figure 4.7: Rule ID partition 35

Figure 5.1: Top-level architectural diagram of PERG. 37

Figure 5.2: Inspection Unit. 38

Figure 5.3: Bloomier Filter Unit (BFU) 39

Figure 5.4: Pipelined binary tree.. 39

Figure 5.5: Circular Speculative Buffer (CSB).. 42

Figure 5.6: Attributes of Wildcard Table entry format 46

Figure 5.7: Wildcard state diagram 47

Figure 5.8: Pseudo Code describing behavior of Wildcard Table 49

Acknowledgements

First and foremost, I have to thank my supervisor Dr. Guy Lemieux for everything he has done to help his students. My own adventure as his graduate student began with Guy recruiting me in front of Starbucks while I was chatting with a mutual friend on how I was bored and looking for cool engineering projects to work on. Of all the things I would like to thank him for, my sincerest appreciation is for his trust in my ability to make the right decisions. Guy allowed me to decide on my own research interest and topic as well as detail design decisions of my work. Even when I had to be away for a good portion of my graduate studies for work, Guy has always been supportive on my decisions and I simply cannot thank him enough for it.

The completion of this thesis is not possible without my parents for their unconditional love. I also have to thank all my friends, particularly Johnny Lin, Jack, Orion, and Julia for never stop inviting me to join social events even though I have rejected them 90% of the time due to suffocating personal schedule. Moreover, I have to give a special thank-you to Caleb for rescuing whenever I got stranded at the Seattle airport! Finally, I should probably thank Wynne, who has always been an [edited] to me since we met, but I did had fun I supposed☺.

Other people I had to thank are Dr. John Meech, Mr. Jean Chapoteau, Ms. Julie Lee, my old boss Larry Hung, Sean Chan, and the rest of the Xilinx SEG team, my old landlord Edward and his lovely family, Sam (who probably knows I will never become religious but never stops preaching nonetheless!), and everyone at SoC Lab (especially Darius and Cindy for thanking me!).

Chapter 1. Introduction

Computer virus protection is a ubiquitous part of every modern computer. However, the security of antivirus software comes at a hefty cost in system performance. Tests [1] demonstrate that even on a modern PC powered by multi-core processors, antivirus protection contributes to a significant slowdown (several times increase in runtime) in system performance, particularly for disk I/O intensive applications.

Antivirus software utilizes a plethora of different detection heuristics that vary from vendor to vendor, but matching a file stream against known database of virus patterns is a common fundamental technique that can be easily implemented in hardware. This process, known as *pattern-matching*, compares an input data stream byte by byte against a list of predefined patterns known as *virus signatures*. The simplest pattern is a string of bytes, but more complex patterns are also possible. If the input stream matches with any pattern in the database, virus is said to be found in the input.

Figure 1.1 illustrates an example of pattern-matching input data against a database of n patterns. Input data arrives one character at a time. The pattern matcher scans data in sliding-window fashion. In this simple example, all patterns in the database have the same length, but usually a mixture of lengths is needed.

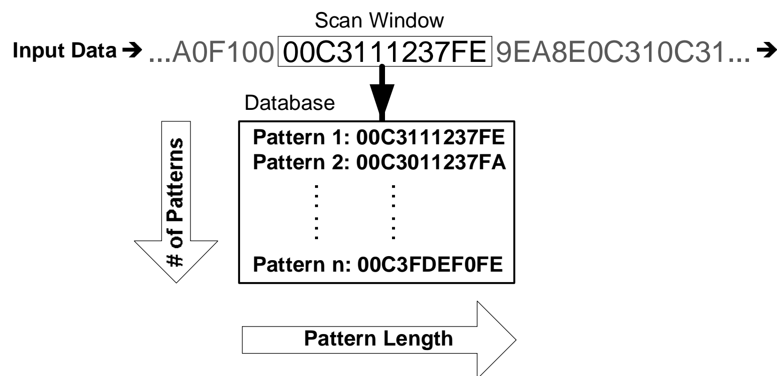

Figure 1.1: An illustration of the pattern matching problem

While scanning input for a single pattern is simple in concept, performing such task with a database of patterns at a consistently high data rate is not. A typical virus signature database contains tens of thousands of virus signatures, and each virus signature may have a different pattern length. The input stream must be inspected against every pattern in the database in order to guarantee its integrity. As a result, even with sophisticated pattern-matching algorithms, pattern matching is the performance bottleneck in many antivirus software programs [2].

1.1 Motivation

Hardware acceleration is one approach to solve the slowdown of software applications. However, in the context of antivirus applications, the use of hardware acceleration is still relatively unexplored. On the other hand, hardware acceleration for pattern-matching in *network intrusion detection system* (NIDS) applications has been studied extensively due to the application's high-throughput requirement [3-10]. A NIDS protects a network from known malicious threats by pattern-matching incoming packets against a database of predefined patterns; this process is referred to as *deep packet inspection* (DPI) in the field of networking. To keep up with network data rate, DPI can be performed on a dedicated hardware pattern-matching engine.

Recent works on pattern-matching engines have been based on field-programmable gate array (FPGA) technology. In comparison with application-specific integrated circuits (ASIC), FPGA devices offer comparable parallelism at significantly lower non-recurring engineering (NRE) cost. Most importantly, FPGA's unique re-programmability is particularly well-suited for the dynamic nature of pattern-matching applications, whose databases are typically subjected to periodic updates. In the case of antivirus applications, the database is updated on a daily basis. FPGA technology allows hardware to keep up with the frequent database changes.

While pattern matching in NIDS and antivirus applications are fundamentally the same, the scale of the antivirus problem is significantly more difficult. The de facto pattern database for NIDS is Snort [11]. Figure 1.2 shows the pattern length distribution of Snort pattern database used by [3], which has implemented the latest version of Snort among the published work in NIDS hardware pattern-matching engines. In comparison, Figure 1.3 shows the same distribution for the pattern database used in this thesis: Clam AntiVirus (ClamAV) [12]. ClamAV is the most popular open-source antivirus software. Clearly, the ClamAV database is much larger in both number of patterns and range of pattern lengths. Since existing NIDS hardware architectures are designed with only a few thousand patterns in mind, fitting a database over ten times larger in a similar FPGA platform is significantly more challenging.

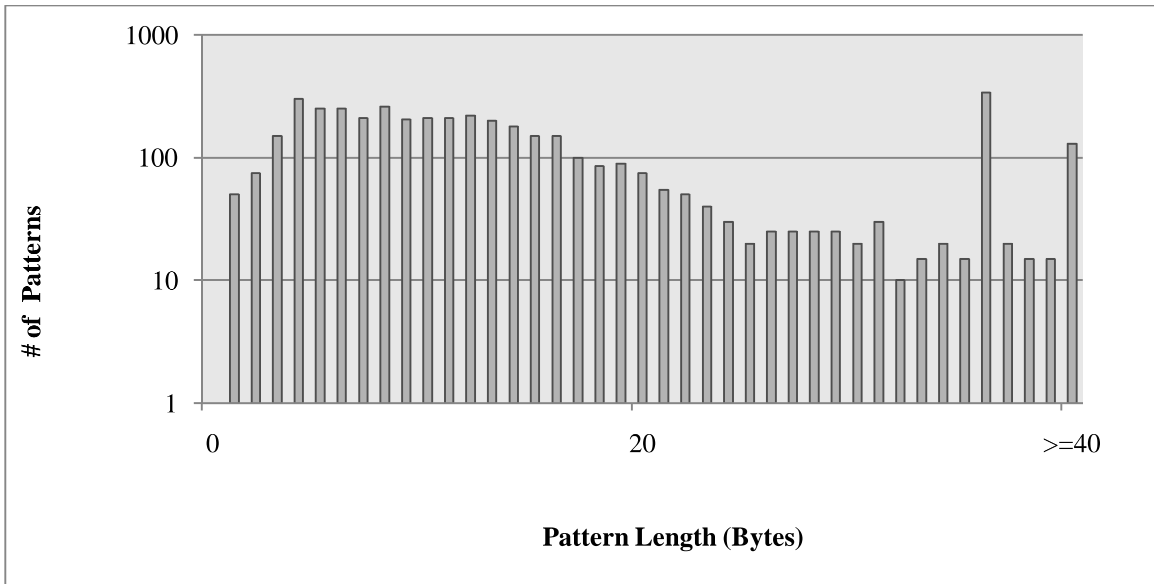

Figure 1.2: Pattern length distribution of Snort database [3]

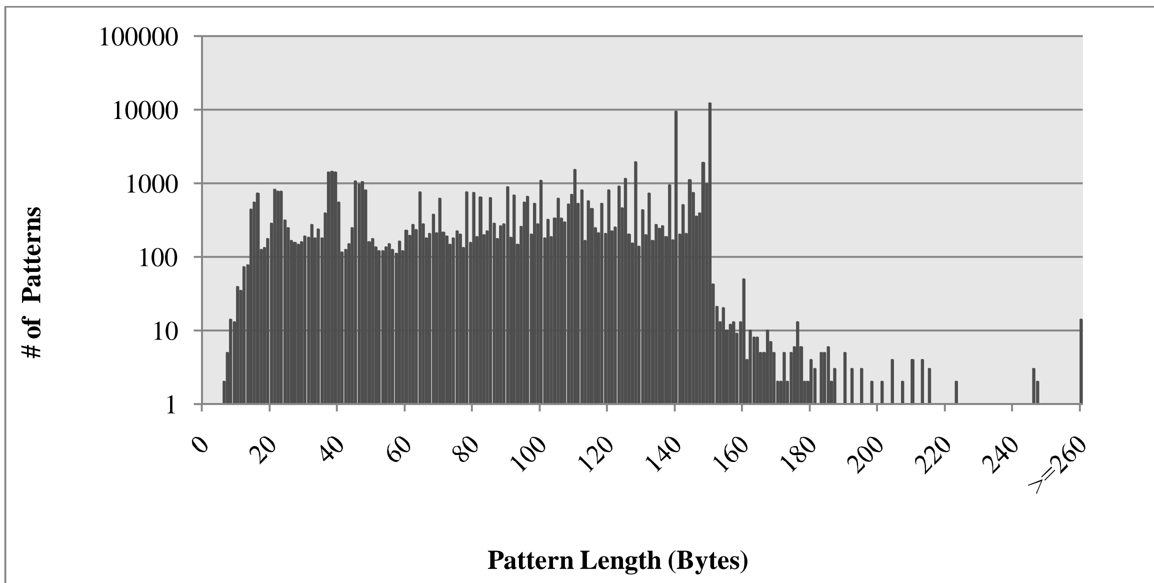

Figure 1.3: Pattern length distribution of ClamAV database

Another challenge with existing pattern-matching solutions is the support for *regular-expression* patterns. Simple patterns are merely continuous character strings. Regular expressions allow patterns to be written in a more flexible manner by including features like wildcards. Although most patterns in both Snort and ClamAV are still simple patterns, the use of regular expression has become increasingly important. A single regular expression can replace a large number of simple patterns, or it can identify a range of viruses which may change rapidly. Existing hardware solutions that support regular expressions tend to have poor area efficiency. On the other hand, hardware solutions offering superior area efficiency

generally lack regular expression support. To handle the ClamAV database, a solution providing the bests of both worlds is needed.

This thesis proposes PERG, an FPGA-based pattern-matching system designed to accelerate pattern-matching against the large database found in ClamAV. The proposed solution aims for the following design targets:

- Area efficiency to support large number of patterns on a single FPGA
- Support for regular expression patterns found in ClamAV
- Dynamic updatability to keep up with daily database update
- Sufficient throughput to keep up with most disk and network interfaces

1.2 Contributions

Through PERG, this thesis demonstrates the feasibility of antivirus acceleration in hardware and opens up a new highly-lucrative potential market for FPGA devices. An initial implementation of PERG in Verilog shows that roughly 80,000 patterns can be fit onto a single low-cost Xilinx Virtex II-Pro VP100 FPGA with 4 MB of external SRAM. In comparison, previous works [3-10] only deal up to a few thousand patterns on similarly-sized FPGAs.

The primary contributions of this thesis are:

- A novel hardware architecture to handle pattern matching in a multi-staged manner without resorting to high-bandwidth off-chip memory requirement
- A novel *filter consolidation* algorithm that significantly reduces the hardware resources required by packing Bloomier filter units into high capacity, thus reducing the number of filter units needed.
- Circular State Buffer, an efficient hardware mechanism capable of supporting multiple traces of multi-segmented patterns with zero false negative probability
- Detection of polymorphic virus signatures through limited regular expression, particularly the support for wildcard characters

PERG is also the first hardware implementation to apply the Bloomier filter algorithm [13] in pattern-matching application. The only known prior hardware to apply Bloomier filters was for longest-prefix IP lookup [14].

The work in this thesis has appeared in three publications. The first [29] described the overall PERG architecture and pattern compiler. The second [30] reported more details about the pattern compiler and mapping results of the Verilog code. The third [31] added support for regular expressions in ClamAV and changed the name to PERG-Rx to signify the importance of this change. In this, we describe the final PERG-Rx implementation, but we use the shorter name PERG for simplicity.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Background information including overview of the pattern-matching problem, existing hardware pattern-matching approaches, and the Bloomier filter algorithm are introduced in Chapter 2. Chapter 3 discusses the targeted ClamAV database and presents a high-level description of the overall PERG system. The PERG pattern compiler and the PERG hardware architecture are discussed in Chapter 5 and 6, respectively. Experimental results are summarized in Chapter 7. Finally, conclusions and future work are given in Chapter 8.

Chapter 2. Background

This chapter is divided to three sections. The first section provides an overview of the pattern matching problem. The second section introduces existing hardware pattern-matching solutions. The last section introduces Bloomier filter, the pattern-matching algorithm PERG built upon.

2.1 Pattern Matching Overview

Pattern matching, in the context of this thesis, refers to the act of matching an input string of characters against one or more predefined strings of characters. In NIDS and antivirus applications, each *character* is a *byte*; the two terms are used interchangeably throughout this thesis. Each predefined string is called a *pattern* or *rule*. A set of patterns is called a *database*. Each pattern in a database is assumed to be unique.

While patterns in traditional applications such as a spell-checking dictionary are simply continuous strings of characters, patterns in newer applications like NIDS and antivirus can also contain regular expressions. Regular-expression support allows patterns to be described more generally and therefore reduces the number of patterns required in the database. Moreover, for antivirus, the flexibility enabled by regular expression is essential for the detection of polymorphic viruses; polymorphic virus is a type of computer virus that can alter its signature appearance through techniques such as encryption to avoid antivirus detection.

Pattern-matching algorithms can be categorized in two types: single-pattern search and multi-pattern search. Single-pattern search matches an input against a single pattern, while multi-pattern search is optimized for comparison with multiple patterns. Both types generally require the patterns to be preprocessed, so advance information about the patterns can be gathered to accelerate the actual pattern-matching process. An example of such information is whether or not any pattern in the database shares a common prefix with others. In some algorithms, such as Aho-Corasick [15], the patterns are preprocessed into completely different data structures.

Single-pattern search algorithms, such as Boyer-Moore [16] or Knuth-Morris-Pratt [17], use heuristics to reduce the number of character comparisons required to determine whether a match exists. By iteratively pattern-matching each pattern in a database, single-pattern search algorithm can also accelerate multi-pattern match. However, such approach does not scale for larger databases as a single-pattern search treats each pattern independently; the algorithm neither shares insights gained through each search nor takes advantage of characteristics of the database.

Multi-pattern search algorithms are designed for pattern matching against a database of patterns.

Software implementations of multi-pattern search typically consume significantly more system resources than a single-pattern search. When presented with a large number of patterns, the sequential nature of the underlying processor-centric software architecture can be slow. As a result, many efforts have been made to implement multi-pattern algorithm on hardware to speed up the process. The following sections cover major types of multi-pattern search algorithms implemented on hardware.

2.2 Existing Solutions

This section discusses existing approaches for pattern-matching hardware. Traditional pattern-matching hardware solutions are based on content-addressable memory (CAM) or application-specific instruction processor (ASIP) platforms. Newer approaches use FPGAs.

In CAM, data is used for inquiry and an address is returned if data is found. Comparison between input and all CAM entries is done in parallel by brute force. As a result, a large CAM can be fast (single-cycled) and very power-hungry. Ternary content addressable memory (TCAM) is an extension of CAM where each storage unit can represent logic 0, 1, and a special third state known as "*don't care*". With this third state, TCAM enables support for various pattern lengths and wildcard. While CAM-based pattern-matching solutions offer simple and deterministic performance, CAM suffers from poor scalability. In terms of transistors required per memory bit, CAM requires 10 transistors while TCAM requires 17; in comparison, SRAM requires only 6 transistors. As a result, CAM is significantly more expensive in cost and power consumption. Finally, the parallel comparison in CAM can limit its operating frequency due to high capacitance.

An application-specific instruction-set processor (ASIP) operates much like a general-purpose processor but also comes with custom-instruction extensions to accelerate the specifically targeted application. Examples of ASIP for pattern-matching include Cavium Networks' OCTEON processor family (based on MIPS64) [18] and Lionic ePavis (based on Altera NIOS-II) [19]. Cavium Networks claims a throughput of 10 Gbps on its OCTEON processor, while Lionic quotes 1.6 Gbps on its fastest ePavis. Since these processors can also perform functions such as encryption and packet-header classification, the exact meaning of the reported throughputs is unknown. Neither Cavium Networks nor Lionic provides any detail about their products or benchmark setups. However, since these solutions are processor-centric, they are unlikely to achieve the level of parallelism offered by FPGAs.

FPGA technology has become the de facto platform for pattern-matching acceleration. An FPGA's parallelism is comparable to ASIC while its low non-recurring cost is well suited for the current niche market of pattern-matching accelerator. Most importantly, the re-programmability is ideal for the dynamic nature of pattern databases in most pattern-matching applications. With FPGAs, hardware can be

optimized for the targeted database which is subject to frequent future updates. Popular FPGA-based approaches include *finite-state machine* (FSM), *Bloom filter,* and *perfect hashing*. The recently proposed *cuckoo hashing* [3] is another noteworthy approach; we believe cuckoo hashing is the best existing pattern-matching approach prior to this work.

2.2.1 Finite State Machine (FSM)

The finite state machine-based approach, as the name suggests, uses a state machine to pattern-match the input. This approach is particularly well-suited for FPGAs due to the abundance of flip flops available for the state machine, which can grow quite large. In general, the FSM approach is based on the Aho-Corasick (AC) algorithm [15] or its variations [4, 24].

The AC algorithm works as follows. The algorithm begins by transforming the pattern database into a *finite automaton*. In its simplest form, each edge in an AC automaton is a character (byte) of a pattern in the database. A simple AC automaton is shown in Figure 2.1.

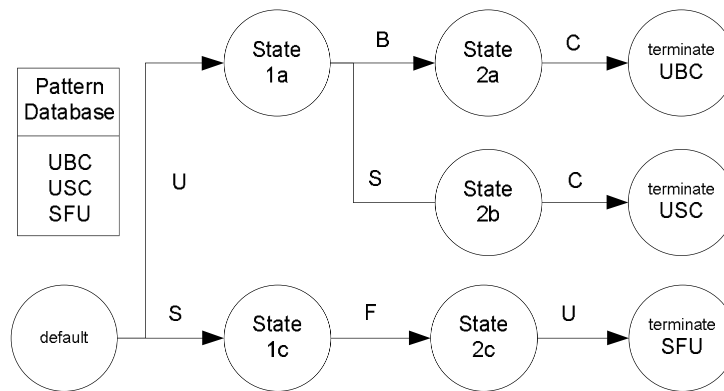

Figure 2.1: An AC automaton

Scanning begins at the *default state* (root) of the automaton. As characters from the input are fed to the automaton one at a time, each character will cause a state transition if it matches the character assigned to that edge. If a pattern from the database does exist in the input string, the state transition will eventually lead to a *terminating state* that indicates a complete match. Otherwise, the automaton transits to the state whose trie-path represents the longest prefix of the current potential match. If the incoming character does not satisfy any prefix path, the automaton transits back to the default state.

Using the example automaton in Figure 2.1, if the input sequence is “US”, the automaton would be at State 2b on the “USC” trace. If the next incoming character is B, then the automaton transits back to the default state since neither “B”, “SB”, nor “USB” is a prefix of any pattern in the database. If however the next incoming character is F, the state will transit to State 2c since “SF” is the prefix of pattern “SFU”.

There are two ways to implement an automaton: deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA). In DFA, there is only one active state. In NFA, there can be multiple active states; i.e., a state may have multiple edges of the same character. This characteristic theoretically reduces the number of states of the resulting automaton and hence the memory storage required. On the other hand, having only one active state makes DFA much simpler to implement (especially on software) than NFA.

The AC algorithm has two advantages. First, its time complexity is linear, dependent only on the length of the pattern and independent of the size of the database. While on average the AC algorithm performs slower than heuristic-based algorithms, its worst case scenario remains constant.

The second advantage of the AC algorithm is that most regular expressions map naturally to an AC automaton. As described earlier, this characteristic is particularly useful for antivirus application. However, a wildcard regular expression leads to exponential growth in FSM edges (NFA) or states (DFA).

In terms of drawbacks, the AC algorithm has two main disadvantages. First, it typically uses much more memory than hash-based approaches like Bloom filter, particularly when the pattern database has a long average pattern length. Second, inserting new pattern updates requires reconstruction of the automaton. This particular drawback is significant for hardware implementations when the state machine is constructed with flip-flops. The need to construct a new state machine with each database update demands the need for reprogrammable hardware. While FPGAs can be used, generating and verifying a new bitstream for each update is a rather costly process in both time and money. The daily antivirus updates lead to another problem: the database may change again before the bitstream is released for the earlier database

Many variants of the AC algorithm have been proposed to improve the algorithm's performance and memory density. Examples include state partitioning [5] and bitmap compression [6]. Another interesting proposal is the B-FSM [4], which features small collision-bounded transition tables for higher performance and memory efficiency. Even with these improvements, however, AC-based hardware solutions in general can still fit no more than a few thousand Snort patterns on a modern FPGA as the memory consumption is proportional to the number of characters in the database.

2.2.2 Hash Checksum

Hash checksum is a hash-based scheme that identifies a pattern by comparing the hash key (checksum) of the input against those of the patterns. The scheme starts by choosing an arbitrary (ideally evenly-distributed) hash function to pre-compute hash keys of all the patterns in the database. The resulted hash

keys are then stored. Using the same hash function, pattern in an input data can then be identified by simply hashing the input and comparing the resulted hash key against those pre-computed ones. A hash key match indicates the input likely is the pattern that corresponds to the matched hash key.

The key advantage of hash-based pattern-matching approach is the simplicity of the algorithm, making this approach extremely easy to implement on both software and hardware. In addition, because only hash keys are stored, storage wise hash-based solutions are generally much more efficient than others; hash keys are not dependent on the length of the pattern and are structurally simpler than complex data structures used like a state machine in the FSM-based approach.

There are two main downsides to hash checksum. First, unlike FSM-based approaches, which perform *exact-match*, hash checksum (as well as other hash-based schemes) performs *approximate match* with a small chance of *false positive*; once in a while an input which does not contain any pattern is falsely reported as one that does. These false positives are results of hash aliasing and inevitable for not just hash checksum but all hash-based solutions. Consequently, another stage of exact-matching with algorithms like AC is still required upon a reported match in a hash-based scheme. Second, a hash checksum such as the common Message-Digest algorithm 5 (MD5) used in ClamAV is intended for single-string matching and performed per file basis (as opposed to searching a specific byte string in the input stream). A pattern (virus) can then easily slip through the scanner if the file integrity is changed even slightly.

2.2.3 Bloom Filter

A Bloom filter [20] is a hash-based scheme that uses a Boolean hash table to pattern-match an input against a set of patterns. Like the hash checksum approach discussed in previous section, Bloom filter is structurally simpler than FSM. Unlike hash checksum however, Bloom filter is designed for multi-pattern search using multiple hash functions as oppose to one.

Another unique characteristic of Bloom filter is that it only provides *membership* information. If the membership is *false*, the input is guaranteed to be free of any pattern in the database; i.e., a Bloom filter has zero *false-negative* probability. If the membership is *true*, the input *should* match with one of the patterns in the database. However, exactly which pattern matched cannot be determined from membership information alone. Upon a hit in a Bloom filter, an exact-match process is needed to perform a complex multi-pattern search and determine which, if any, pattern was detected.

Bloom filter and other hash-based approximate-matching approaches excel in applications where the input stream is expected to be free of patterns in general. Under such an environment, their average throughputs should be on par if not better than FSM-based approaches. With a good set of hash functions

[21], Bloom filter and other hash-based approaches can yield much simpler hardware, and hence higher operating frequency and lower cost.

Construction of a Bloom filter for a database of n patterns is as follows. First, a Boolean hash table containing m entries is created. Each entry in the table is initialized to be *0* (*false*). Second, k different hash functions are chosen. Although not required, typically k is greater than 1 to give a better balance between storage and false-positive probability, which can be approximated by Equation 2.1. Moreover, while the choices of hash functions are arbitrary, their outputs must fall within the range of 0 to m-1. The last step involves hashing each pattern from the database into the hash table. The entry at the resulting hash key is set to *1 (true)*. Because k hash functions are used, each iteration sets up to k entries in the hash table. Note only one hash table is used among the k hash functions.

$$Prob(False\ Positive) \approx (1 - e^{-kn/m})^k \qquad (2.1)\ [8]$$

Pattern-matching input with a Bloom filter is done in two steps. The first step is to hash the input with the same k hash functions used during construction. The second step is to *logic-and* the Boolean values stored at the resulting k hash table entries. If the outcome of the *logic-and* is false, the input is *definitely* not a pattern in the database. If the outcome returns true, the input is *likely* a pattern in the database. The operation of a Bloom filter is illustrated in Figure 2.2.

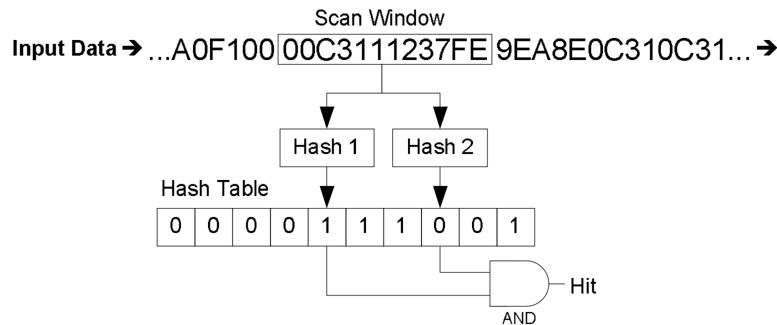

Figure 2.2: A Bloom filter

A Bloom filter has three main advantages over FSM-based approaches. First, computationally it is much simpler as a query in a Bloom filter involves merely hashing the input, accessing k memory locations, and performing a logic-AND operation. Second, a Bloom filter is extremely storage-efficient. Because a Bloom filter only stores one-bit membership information in a Boolean hash table, its memory usage is independent of the pattern length. In addition, at a fixed false-positive probability setting, the size of hash table needed is slightly sub-linearly proportional to the number of patterns. Third, by storing hash tables on embedded memory, dynamic update of the pattern database is easy without the need to generate a new bitstream.

Despite these advantages, a Bloom filter has its downsides. First, because each Bloom filter hash operates on a specific pattern length, each pattern length requires its own Bloom filter. This characteristic can lead to a large number of poorly-utilized filter units in ClamAV, whose database has a wide range of pattern lengths. Second, a complex exact-matching computation is still required upon a Bloom-filter match because a Bloom filter only provides membership information and comes with false positives. Finally, with approximate-matching, supporting regular expressions in a Bloom-filter approach is much more difficult than the AC-based approach, which blends naturally with regular expression thanks to its automaton structure.

Some drawbacks of the Bloom filter can be addressed by combining Bloom filters with the AC algorithm. In [7], patterns are split into s-byte long segments, which are then mapped to s Bloom filters of length s down to one byte. Doing so reduces the range of pattern lengths and therefore the number of filters needed. To link segments back to a complete pattern, AC automaton is constructed from these segments (instead of individual characters). The Bloom filter tables are stored in on-chip memory, but state transitions and the actual string segments are stored in 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 [7], 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.

2.2.4 Perfect Hash

The perfect hash [9, 10] aims to maintain the advantages of a Bloom filter while addressing Bloom filter's drawbacks of false positive probability and membership-only information. Unlike a Bloom filter which suffers from hash table collisions, in a perfect-hash scheme each hash table entry is uniquely associated with one single pattern. Consequently, a pattern may be stored at each hash location and exact matching can be performed immediately through simple direct comparison.

In theory, a perfect hash is superior to the Bloom filter, offering all the advantages of Bloom filter yet enabling a very simple exact-matching step to eliminate false positives. However, in reality, a perfect hash is difficult to achieve even with a large hash table. Patterns are not completely random and hash functions are not evenly distributed. If any pattern hashed results in a collision, the perfect hash construction fails. In this case, a larger table and/or a different hash function must be attempted until construction succeeds without collisions.

The difficulty to construct a perfect hash table leads to several drawbacks of this scheme. First, for the same number of patterns, a perfect hash scheme generally needs a larger hash table than a Bloom filter.

Second, daily updates to a large set of patterns make it increasingly difficult to generate a perfect hash function every time. Third, like a Bloom filter, a dedicated perfect-hash unit is needed for each pattern length. Finally, the ability to perform exact-matching immediately after a hash query assumes that it is possible to store the entire pattern string into each hash entry; this is generally not the case if the hash table needs to reside in on-chip memory. While a checksum of the pattern can be stored instead, perfect hash would become prone to false positives just like Bloom filter. In such a case, the false positive probability can be approximated by Equation 2.2, where n is number of patterns, m is the size of the hash table, and c is the bit length of the checksum.

$$Prob(False\ Positive) \approx (1 - e^{-\frac{n}{m}})/(2^c) \tag{2.2}$$

Another alternative is to keep the patterns off-chip and store memory pointers at on-chip hash table entries. This approach however suffers from degraded performance and may not scale well as pointer length is proportional to the size of the off-chip memory.

2.2.5 Cuckoo Hash

Cuckoo hashing [3, 22] is a unique hash-based approach that we believe provides the best tradeoff among existing hardware pattern-matching solutions. Cuckoo hashing relies on k hash functions like the Bloom filter, but it uses k distinct hash tables as opposed to one. When preparing the tables and inserting a new pattern R, aliasing with a previously mapped pattern S in *Table 0* with hash function H_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. The operation is illustrated in Figure 2.3.

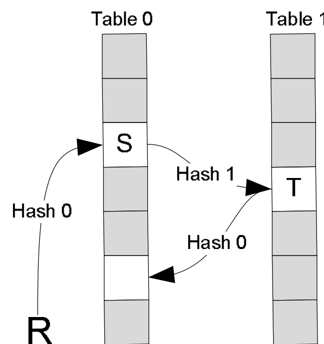

Figure 2.3: Construction of cuckoo hash scheme

It is possible to get caught in an infinite loop while setting up a cuckoo hash scheme; when that happens, the setup process fails and a different set of hash functions need to be used. The probability of setup failure can be reduced by increasing the number of hash functions or the size of each hash table. However, the fact that cuckoo hashing tolerates some hash collision however does theoretically give it a much higher setup success rate than perfect hashing.

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 [3], the hash functions and table lookups proceed in parallel using BlockRAM. Like a perfect hash, the exact-matching stage in cuckoo hashing is simple. Pattern strings can be stored in the hash table allowing direct comparison against the input string to be made immediately after hash query. However, it should be emphasized that cuckoo hash is *not* a perfect hash scheme. If a checksum comparison is used, the cuckoo hash is unable to uniquely identify the matched pattern because the k hash entries may return common checksum values in parallel.

2.3 Bloomier Filter

This section covers the Bloomier algorithm used in PERG. The algorithm was first proposed by [13] and implemented by [14] for IP lookup. PERG is the first to use Bloomier Filter in a pattern-matching hardware application.

The Bloomier filter is an extension of the Bloom filter. Structurally, the two schemes are very similar. However instead of providing a Boolean membership result, the Bloomier filter uses a perfect hash to indicate exactly *which pattern* matched. The one-to-one association allows PERG to perform a checksum comparison to rule out most false positives without further verification. The final exact-matching stage is also simple because only a single pattern needs to be compared.

Bloomier filters operate as follows. When an input string appears, k independent hash functions are computed on the input to point to multiple table entries. In a traditional Bloom filter, all table entries must be 1 to indicate a match. In a Bloomier filter, there is the additional requirement that a match must indicate which pattern is matched. To determine which pattern, exactly one hash function is selected and, through the one-to-one mapping produced at filter construction time, a direction comparison with the pattern stored at that hash entry is performed. The goal during construction of a Bloomier filter is to determine which of the k hash functions points to the correct table location of the matched pattern. This is done by XORing all k table entries. When the table entries are constructed properly, this XOR result will produce a hash select $s(x)$ which chooses the right hash function.

Construction of a Bloomier filter begins by creating a Bloom filter. Each pattern x is mapped to k table entries by the hash keys $Y_1(x)$, $Y_2(x)$,.. $Y_k(x)$; this set of k locations is the *neighborhood* of the pattern. As patterns are mapped, their neighborhoods are recorded. Neighborhoods may or may not overlap with each other. A pattern containing a hash entry not found in another pattern's neighborhood is called a *singleton*.

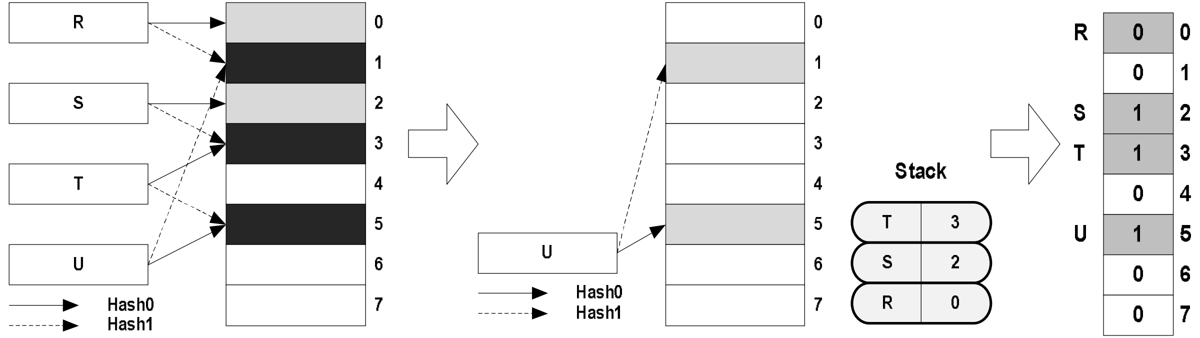

Figure 2.4: Example Bloomier filter construction in Phase 1 (left), Phase 2 (middle) and Phase 3 (right).

The ultimate goal is to construct a one-to-one mapping of patterns to table entries. Consider a simple example of mapping four patterns R, S, T, and U on a table with 8 entries using hash functions H_0 and H_1, as shown in Figure 2.4 (left). Being singletons (highlighted by light grey), it should be apparent that R and S can be uniquely mapped to entry 0 and entry 2, respectively.

Once the Bloom filter is constructed, neighborhoods of each pattern are removed from the hash table one by one. The process starts with singletons (i.e., R and S) because they are readily mapped to unique locations; which singleton is removed first does not matter. Each time a pattern's neighborhood is removed from the table, the pattern and its mapped location is pushed onto a stack. After removing a singleton, more singletons may appear. For example, with S removed, T becomes a singleton mapped to entry 3. In the unlikely scenario where no new singleton is available, the setup fails and a new hash table must be created with a new set of hash functions. The probability of setup failure is bounded by Equation 2.3, where k is the number of hash functions used, m is the number of hash table entries, and n is the number of patterns to be mapped. To construct a Bloomier filter, $m >= k*n$.

$$Prob(Setup\ Fails) \leq \sum_{i=0}^{n} \left(\frac{e^{\frac{k}{2}+1}}{2^{\frac{k}{2}*i}} \right)^{i} \left(i * \frac{k}{m} \right)^{i*k/2} \qquad (2.3)\ [14]$$

Figure 2.4 (middle) shows a snapshot of the stack and hash table after three iterations. At this point, entry 1 and 5 can both be uniquely associated with *U*; which one is used does not matter. Assume *U* is associated with entry 5 and pushed onto the stack, making the hash table completely empty with all entries equal to zero. The patterns can now be popped from the stack and mapped back to the hash table. Each pattern *x* is associated with a unique table location. $V(x)$, used to compute hash select $s(x)$ during inquiry, is stored in the table. The formula for calculating $V(x)$ is given in Equation 2.4, where $V_i(x)$ to $V_k(x)$ are the table entries of the neighborhood excluding the unique entry for this pattern, and $s(x)$ is the value that indicates which hash function points to this entry.

$$V(x) = s(x)\,\mathbf{XOR}\,V_i(x)\,\mathbf{XOR}\,V_{i+1}(x)\ldots V_{k-1}(x)\,\mathbf{XOR}\,V_k(x) \tag{2.4}$$

Initially, all entries in the entire table store $V(x) = 0$. In our example, *U* is popped first and stored in entry 5. Since entry 5 is given by H_1, $s(x)=1$. To determine $V(x)$ for *U*, $s(x)$ is XORed with all remaining $V(x)$ in *U*'s neighborhood (entry 1). Because entry 1 is still initialized to zero, $V(x)$ for *U* equals 1. The process iterates until new $V(x)$ values are calculated for all patterns as shown in Figure 2.4 (right). At this point, the Bloomier filter construction is complete.

To look up a rule, the input is hashed with H_0 and H_1, giving hash key $H_0(x)$ and $H_1(x)$. To determine whether the matching table entry should be selected from $H_0(x)$ or $H_1(x)$, hash select $s(x)$ is calculated by XORing all $V(x)$ entries in the neighborhood as shown below:

$$s(x) = V_1(x)\ \mathbf{XOR}\ V_2(x)\ \ldots V_{k-1}(x)\ \mathbf{XOR}\ V_k(x) \tag{2.5}$$

For *R*, $s(x)=0$ XOR $0=0$. Therefore, if the input equals to *R*, it will be compared with rule associated with hash key 0 result $H_0(R)=0$. Similarly for *S*, since $s(x)=1\text{XOR}1=0$, if the input equals to S it will be identified with the rule associated with $H_0(S)=2$.

The features offered by a Bloomier filter are similar to cuckoo hashing. Both enable simpler exact-matching than Bloom filter and offer higher setup success rate than the perfect-hash scheme. However, one striking difference is that Bloomier filter is a true perfect-hashing scheme, making it possible to filter out false positives with a checksum yet still be able to uniquely identify the potential match. In the case of cuckoo hash, it is not possible to guarantee all checksums returned from the *k* entries are unique. As a result, the one-to-one association would be lost in a cuckoo hash if a checksum is used.

2.4 Shift-Add-XOR Hash Function

This section discusses the Shift-Add-XOR (SAX) hash function used in PERG to construct its Bloomier filters. With the discussion of hash-based pattern matching approaches like cuckoo hash and Bloomier filter, clearly the choice of hash functions has a direct impact on the overall system. The ideal hash function would be universal (uniformly distributed) to yield higher setup success rate with the same number of hash entries and therefore higher utilization of memory. At the same time, the hash function should remain logically simple in order to save logic gates and increase the achievable operating frequency.

PERG uses SAX hash proposed by [21] and implemented by [3] on its cuckoo hash-based pattern matching engine. SAX is a class of hash function that hashes string inputs as shown in Equation 2.6.

$$H_i = H_{i-1} \textbf{ XOR } ((H_{i-1} << S_{Li}) + (H_{i-1} >> S_{Ri}) + C_i) \qquad (2.6)$$

H_i is the partially-formed hash value after seeing i bytes. C_i is the ith byte of the input string, S_{Li} is the amount to shift left, and S_{Ri} is the amount to shift right. By choosing the initial hash input H_0 and shift values (S_{Li} and S_{Ri}) at each byte position randomly, the resulted hash function is relatively universal [21].

SAX is well-suited for hardware implementation. With mere shift, add, and XOR operations, a hardware implementation of SAX involves relatively simple logic gates without need for any expensive multiplier or multi-cycle operations. In addition, Equation 2.6 can be fully unrolled in a hardware pipeline. By selecting an overall length equal to the longest pattern length, different Bloomier filters (as well as any other hash scheme) can then share the same hashing pipeline by picking off the appropriate intermediate hash. This structure is shown in Figure 2.5.

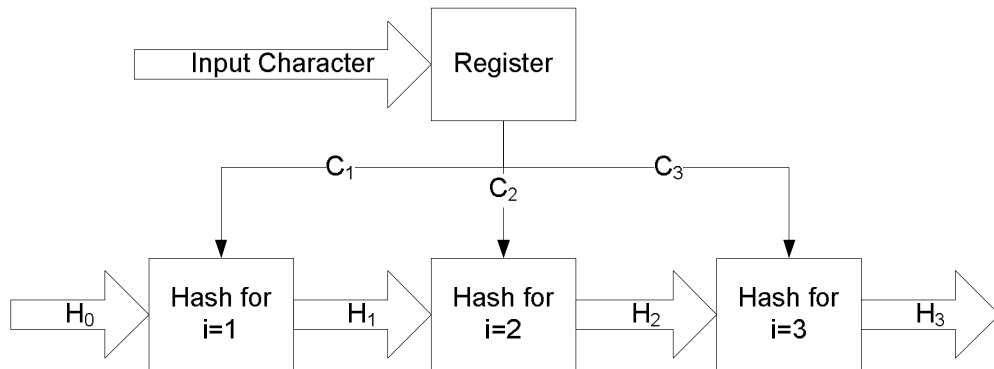

Figure 2.5: Structure with hash pipeline sharing

Chapter 3. PERG Overview

This chapter is divided to two sections. The first section analyzes the ClamAV pattern database used in this thesis. The second section provides a high-level overview of how PERG works as a system to match these patterns on hardware.

3.1 Clam AntiVirus Signature Database

The project source code for Clam AntiVirus (ClamAV) [12] is available on SourceForge.net and has been ported across various operating system platforms including Linux and Windows. All work presented in this thesis is based on the ClamAV version 0.93.1 released on June 9, 2008.

To detect computer virus, ClamAV relies on comparing file inputs against patterns in its virus signature database. The ClamAV database is released in two packages: the *main* database, which comes with each stable software release, and the *daily* database, which is updated regularly between official version releases. The *main* package provides a more consistent and accurate view of the ClamAV database characteristics than the *daily* package, which varies widely between each release and contains significantly fewer samples.

A ClamAV virus signature consists of the name of the virus and its unique pattern. Each signature may also contain header information like the specific file extension the pattern should only be matched against. Virus header information is not supported in the PERG system presented; such support requires higher-level software support, which is left as future work in this thesis.

3.1.1 Type of Patterns

Each virus signature pattern in ClamAV can be categorized as one of three types: *Message-Digest algorithm 5(MD5) checksum*, *basic pattern*, and *regular-expression pattern*. Table 3.1 shows the distribution of each type in the ClamAV 0.93.1 main database, which contains a total of 231,800 patterns.

Table 3.1: Ratio of different types of ClamAV virus signatures

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

MD5 checksum is a simple 128-bit hash value that can be used to identify malicious files by computing the MD5 checksum of the entire input file and comparing it against MD5 checksums of known malicious files. The advantage of using MD5 checksum is that any user can generate and submit as MD5

checksum of malicious files easily, providing an immediate solution to react against new threats. On the downside however, any slight variation in the identified malicious file generally requires a new MD5 checksum signature, because the new MD5 checksum most likely is completely different.

A basic pattern is a continuous byte string that presumably only exists in an infected file. A virus can therefore be detected by scanning through the input file for the basic pattern. In comparison with MD5 checksum, a basic pattern provides a more flexible way to describe the virus without being restricted to a fixed 128-bit length. Although much less sensitive to file integrity than MD5 checksum, a basic pattern is still prone to changes in the known malicious file.

A regular-expression pattern is an extension of the basic pattern which includes regular expression operators. Regular expression support allows a virus pattern to be described more generically, hence reducing the total number of patterns needed in the database. Most importantly, a regular-expression pattern is necessary for detection of polymorphic viruses, where a portion of the virus undergoes mutation such as encryption or compression to avoid being matched by an MD5 checksum or basic pattern. A pattern of this type contains one or more regular expression operators shown in Table 3.2. Note in comparison with regular expressions found in the Perl-Compatible Regular Expression (PCRE) de facto standard [26], the regular expressions in ClamAV form a more limited set. Operators like “match one or more times” or “match the end of line” are not supported. On the other hand, a single-nibble (half-byte) wildcard is found only in ClamAV. The main reason for the differences is that PCRE operators are intended for ASCII text processing while ClamAV operates on raw bit stream.

Table 3.2: Regular expression operators in ClamAV

Symbol	**Definition**
(X|X)	Byte-Or
?	Single-Nibble Wildcard
??	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

The pattern-matching engine presented in this thesis deals only with basic and regular-expression patterns. MD5 checksum, whose computation differs greatly from the pattern-matching problem described in Chapter 2, is completely left out from this thesis. While MD5 checksum pattern represents

the largest portion of the ClamAV database, due to the algorithm's simplicity, MD5 checksum computation only accounts for 0.64% of entire software runtime [23]. As a result, there is little motivation to accelerate MD5 checksum with dedicated hardware.

3.1.2 Database Characteristics

Figure 3.1 shows the length distribution of basic patterns in ClamAV. Statistical information is summarized in Table 3.3. Regular–expression patterns are not included here because most of them contain wildcard or range operators and therefore do not have fixed lengths.

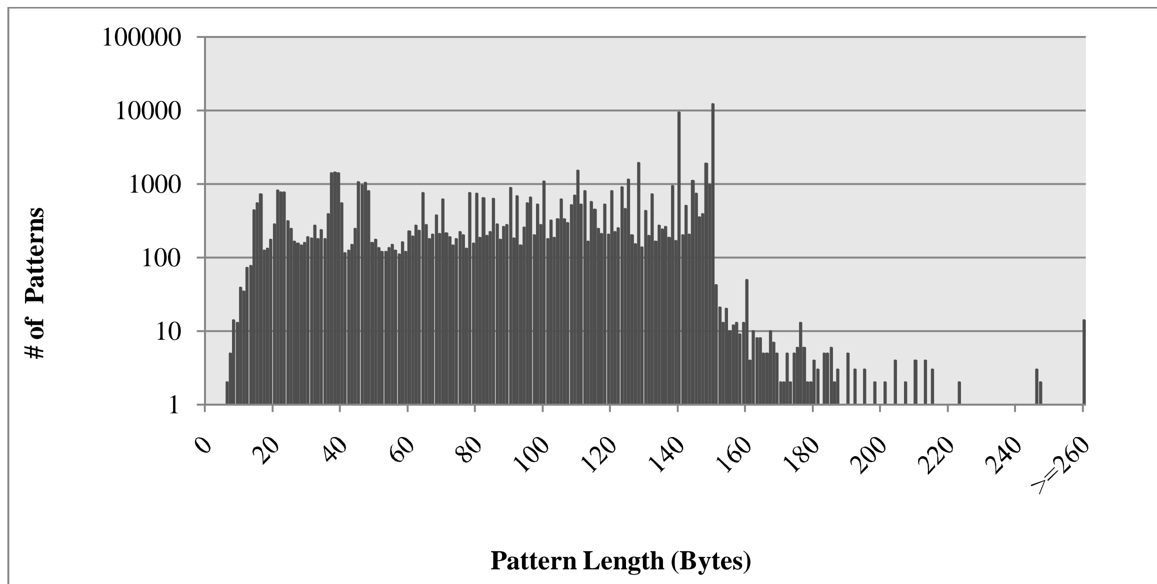

Figure 3.1: Pattern length distribution of ClamAV database

Table 3.3: Statistical properties of ClamAV pattern database (basic only)

Measurement	Value (Byte)
Length (Mode)	150
Length (Mean)	102
Shortest Length	4
Maximum Length	347
# of Unique Lengths	220

Even without accounting for regular-expression patterns, basic patterns alone already contain over 8,236,243 characters (bytes) in total. Using the density in [24], whose density of 8.7 bit/char is the best known among existing FSM-based solutions, 8,236,243 characters would require more than 8.57 MB of on-chip memory. On the other hand, if a hash-based approach (Bloom filter, perfect hash, or cuckoo hash) is used, 220 unique lengths demand 220 individual hash units to operate in parallel. Assuming the

availability of arbitrarily-sized memory blocks, for a Bloom filter with a moderate theoretical 0.00001% false positive probability (twice that of PERG), the best configuration using two hash functions (a BlockRAM on a Xilinx/Altera FPGA has two read ports) would result in a memory requirement of 6.00 MB for the hash table alone. If cuckoo hashing is used, assuming an extremely high 50% utilization for each cuckoo hash filter and again the availability of arbitrarily-sized memory blocks, the resulted memory requirement would still require an enormous 17.14 MB. This leaves perfect hash with checksum as the only remaining option. However, as discussed in Chapter 2, with the challenge of low setup success rate, perfect hash generally results in a poorly utilized hash table. In addition, the optimal number of bits needed for each checksum is still in question. In conclusion, even with the conservative estimations, the resulting memory requirement from existing approaches exceeds the available resources on the FPGA available today, which tops at about 4 MB of on-chip memory. A new approach is needed clearly.

3.2 The PERG System

PERG is a pattern-matching system designed to accelerate ClamAV by performing the pattern-matching process on FPGA hardware. The pattern-matching system is designed to operate as an offloading engine for antivirus application on a personal computer or a network file server. The FPGA would communicate with CPU through PCI-Express or Front-Side Bus technology like Intel QuickAssist [25]. The underlying operating system can then direct all file traffic, whether from hard disk or network interface, through PERG for virus scan. Antivirus software protection is still needed for exact matching, but only called upon (through hardware interrupt) when a highly-potential match is reported by PERG. As a result, most data traffic and scan are performed without interfering the user applications other than possible increase in latency.

The PERG system is actually made up of two main parts: a configurable hardware and a pattern compiler. The configurable hardware, designed for an FPGA platform, is the heart of the pattern-matching engine. Given a database, the pattern compiler preprocesses the patterns to generate parameters for the configurable hardware to target different databases.

Data input to the PERG hardware is assumed to be raw bytes of a sequential file stream. A virus is detected if the input data stream contains any pattern in the ClamAV database.

To fit the enormous ClamAV database on a single FPGA, PERG uses a resource-efficient approximate-matching technique to scan the input stream for defined patterns. If a match is detected, this is to be followed by exact matching. The exact matching stage is not part of the PERG hardware and assumed to be performed in software at a slower rate. However, the overhead due to exact matching

however is kept minimal to a single-pattern comparison. In a real environment, most input data traffic is presumably free of any viruses. By keeping false positive probability sufficiently low, the proposed architecture is able to keep up with the input line speed of a modern disk or network interface (approximately 200 MB/s).

The remainder of this chapter provides a high-level overview of the configurable hardware and pattern compiler. The two components are closely intertwined and difficult to comprehend individually without a basic knowledge of the counterpart. Full implementation details of the pattern compiler and the configurable hardware are discussed in Chapter 4 and Chapter 5, respectively.

3.2.1 Configurable Hardware

The configurable hardware pattern-matches input data for predefined patterns through multiple filtering stages. Throughout the remainder of this thesis, the terms *approximate matching* and *filtering* are used interchangeably. When multiple filtering techniques are performed in series, no new false positive is introduced in subsequent stages, since only the intersection of previous stages is passed down. While false positives may still slip through, false positive probability decreases with each additional filtering stage.

A high-level architectural diagram of the pattern-matching engine and its decision tree are shown in Figure 3.2 and Figure 3.3, respectively. Unlike a traditional single-stage architecture, in PERG's multi-stage paradigm, each stage can operate at a lower rate than its previous stage because the hit rate decreases with each consecutive stage (as explained in previous section, the majority of file inputs is presumably clean and hit traffic is mostly contributed by false positives). FIFOs are connected between stages to buffer input data in the unlikely scenario of a burst of match hits in the input stream.

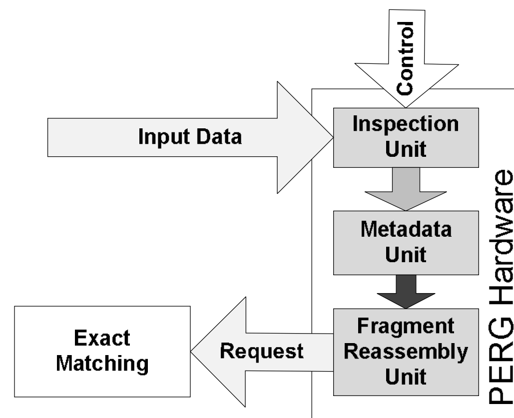

Figure 3.2: High-level architectural dataflow

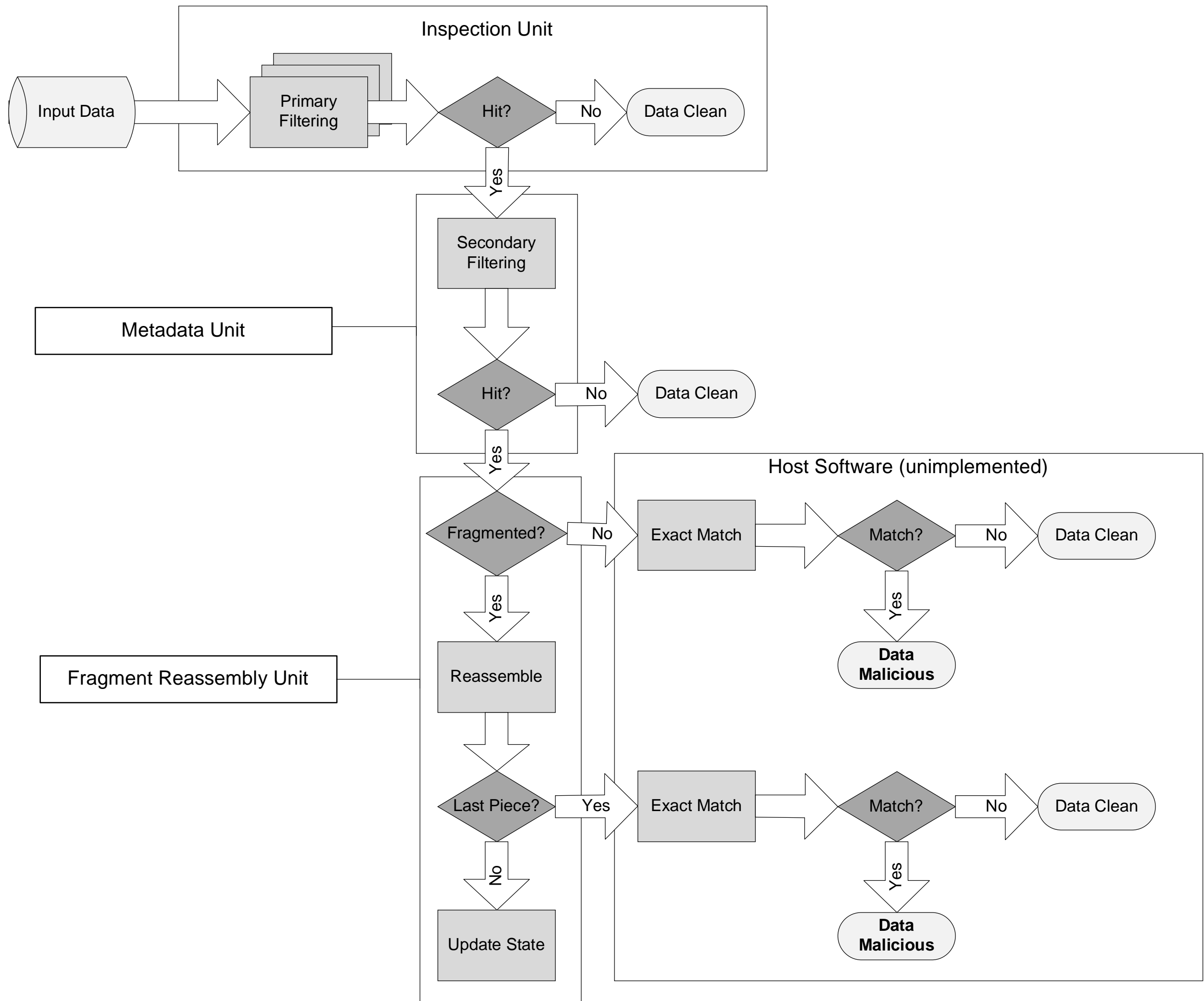

Figure 3.3: PERG hardware decision tree

The configurable hardware is partitioned into three units: *Inspection*, *Metadata*, and *Fragment Reassembly Units*. Each unit is accessed in a pipelined manner and in the order as shown in Figure 3.3. The control input informs the hardware the arrival of a new file stream and resets the internal states of PERG hardware at the start of each file stream. Data flow begins at the Inspection Unit which scans the input data one byte per clock cycle with a set of parallel Bloomier filters discussed in Section 2.3. A checksum of the pattern is stored at each hash entry to enable immediate comparison between the input and suspected pattern. This checksum is referred to as the *primary checksum*. Together, the filters and primary checksums provide the first level of defense; majority of activities in PERG hardware ends here as most data are clean and filtered out safely.

In the rare event where a pattern hit is reported by any of the filters, additional *metadata* information about the suspected pattern is retrieved from the Metadata Unit, which resides on external memory. Metadata includes three pieces of information. The first is the identification (ID) of the virus signature. The second piece provides linkage information: whether this pattern is a complete pattern or a fragment of a pattern, and if it is the latter, where should this fragment reside in sequence in the original pattern. The last piece of information is a cyclic-redundancy checksum (CRC), used to match against that of the reported input string to reduce false positives even further. This checksum is also referred to as the *secondary checksum*.

The secondary checksum comparison is performed in the Fragment Reassembly Unit. If the result returns true, two scenarios can happen. If the reported string is one complete pattern, an exact-matching request is sent out immediately. If the reported string is a fragment of a pattern, then the Fragment Reassembly Unit keeps track of the state information of the corresponding pattern. Only when all fragments of this pattern arrive in the exact order specified by the virus signature is an exact-matching request sent.

3.2.2 Pattern Compiler

The pattern compiler preprocesses the pattern database to generate database-dependent parameters and Metadata information used by the configurable hardware. The design flow, shown in Figure 3.4, begins by inputting the targeted pattern database and user definitions to the pattern compiler.

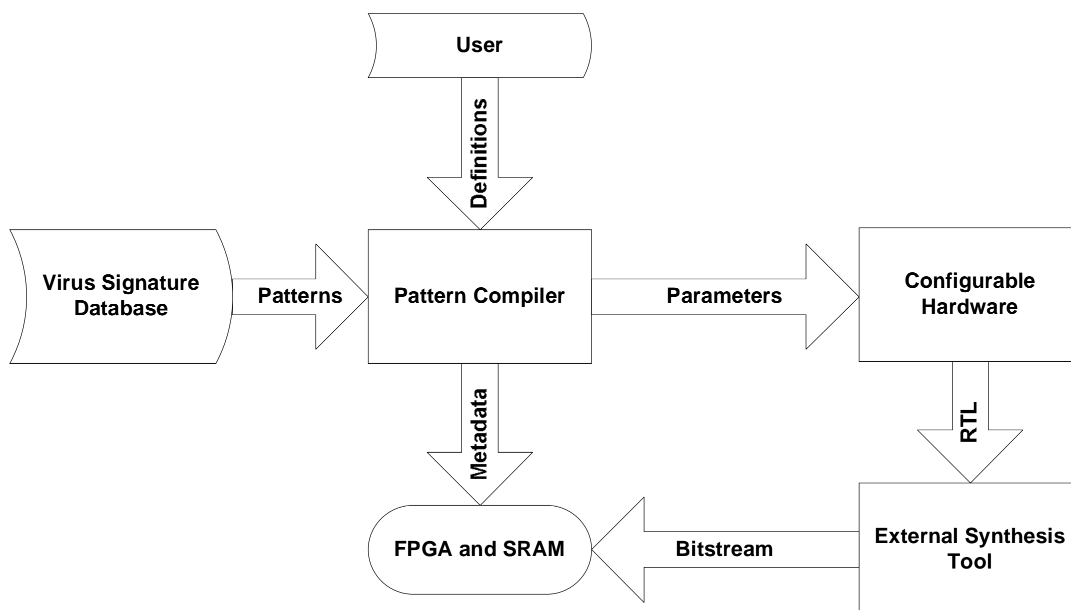

Figure 3.4: PERG design flow

User definitions define the boundaries of the pattern database and underlying hardware. In concept, these definitions work much like the user constraints in FPGA synthesis; similarly, patterns can be thought of as hardware description files. These parameters have a direct effect on the performance and resource consumption of the resulting hardware. As an example, increasing the number of checksum bits reduces the false positive probability but requires more memory storage and wider comparators in the inspection unit.

The complete list of user defined parameters is summarized in Table 3.4. Their values used in this thesis are shown as well as a reference of their appropriate ranges. The significance of each parameter will become clear when the pattern compiler and hardware architecture are explained in detail in Chapter 4 and 5.

Table 3.4: User definitions

Parameter	**Value**
Maximum # of bytes per fragment	256 bytes
Minimum # of bytes per fragment	4 bytes
Maximum # of fragments per pattern	16
Maximum # of OR operator per pattern	6 operators
Utilization threshold of each hash table	90%
Maximum byte displacement between two consecutive segments	512 bytes
# of bits used for primary checksum	8 bits
# of bits used for secondary checksum	32 bits

After a series of data-parsing and optimizing processes, the compiler outputs two types of information. The first set of information is the hardware parameters used by the configurable hardware. Such information is recorded as header files to be used along with the register-transfer-level (RTL) source code of the baseline hardware in the synthesis tool. Although hardware parameters are used to optimize the hardware against the targeted database, it is not necessary ot change these parameters unless there is a significant change to the targeted database distribution (i.e., significant change in number of patterns or length distribution).

The second set of information from the compiler is the memory initialization data, which can be further divided to two types. The first type is the filter hash table data stored in the block memory on FPGA. The second type is the Metadata information and stored in off-chip memory due to its larger storage requirement.

Unlike the hardware parameters, memory initialization data is database-specific; an addition or removal of any pattern in the database would require the memory initialization data to be updated accordingly.

Chapter 4. Pattern Compiler

The pattern compiler takes in the pattern database along with the user constraint definitions to outputs hardware parameters and memory initialization data. The transformation is performed through a series of processes as shown in Figure 4.1. This chapter walks through each stage one by one in the order they are executed in the compiler.

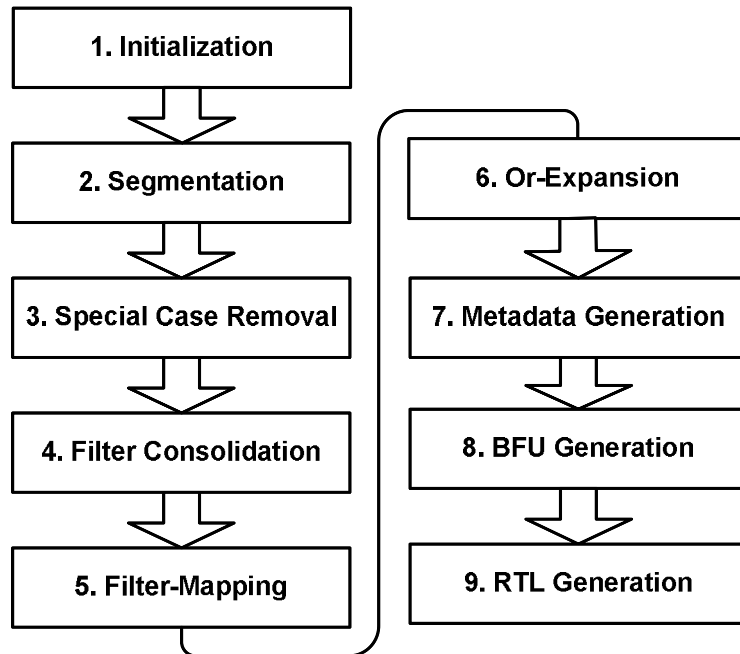

Figure 4.1: Flow diagram of the PERG pattern compiler

4.1 Initialization

Initialization fetches patterns from the appropriate ClamAV database files (main.db and main.ndb) and eliminates any duplicate pattern encountered. While most signatures in ClamAV apply to *all* file extensions, sometimes certain extensions (such as *.*exe* and *.*dll* files) share identical pattern strings. As mentioned in previous chapter, the PERG hardware itself does not distinguish 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.

4.2 Segmentation

In segmentation, regular-expression patterns containing displacement and/or wildcard operators (see Table 3.2) are divided into segments. The point of separation is at where the displacement/wildcard operator appears in the pattern. For example, pattern "ABCD**{10}**DEF", containing the displacement operator "**{10}**", which means a gap of 10 arbitrary bytes, and would be segmented into segment "ABC" and "DEF". For the remainder of this thesis, the terms *segment* is used to refer to a string of a bytes. In addition to byte string, each segment in PERG also comes with an overhead (*metadata*) including information like whether this segment is followed by a wildcard. A pattern consisting of one single segment is called *single-segmented pattern*. A pattern with more than one segment is called a *multi-segmented pattern*.

As a result of segmentation, metadata is uniquely associated to a segment rather than a pattern. The order in which each segment appears in a pattern is preserved by assigning a pair of *link numbers* to each segment. A segment identifies itself with the *current link number* and uses the *expected link number* to identify which segment shall follow. Segments of the same rule can then be linked together by comparing the expected link number of the most recently discovered segment with the current link number of a newly arriving segment. Current link numbers start at zero for the initial (prefix) segment and increment by one with each consecutive segment until the last (suffix) segment, which has a special terminal of zero for its expected link number.

In addition to a link number, it is also necessary to keep track how far consecutive segments are apart from each other. This information can be directly extracted from the displacement operator. Because a segment of n bytes can only appear and become detectable (hashed) after n bytes have passed, the actual displacement value in the metadata must go through *offset adjustment*. In PERG, a displacement amount is the distance in bytes between the last characters of adjacent segments. However in ClamAV, displacement in its original format is only the gap length between two segments. As a result, offset adjustment is needed in PERG to sum the original displacement value with the byte length of the segment that immediately follows the displacement operator. Using the example in Figure 4.2, the original pattern has a displacement of 10 bytes between two segments. Since the second segment is 3 byte long, displacement value is recorded as 10+3=13 bytes in PERG's metadata information.

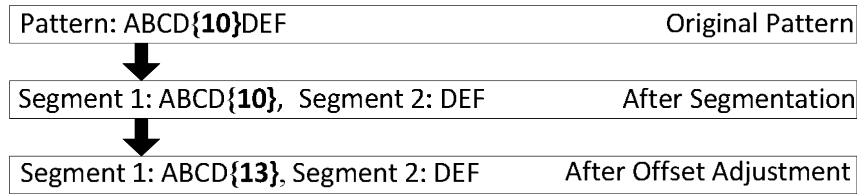

Figure 4.2: Offset adjustment

For segments separated by a wildcard, in addition to range information, a wildcard flag is used to record the type of wildcards in metadata. Due to the complexity and close interaction with hardware components, detail of how wildcard support is done in PERG is saved for Chapter 5.

4.3 Special Case Removal

This stage removes a relatively small number of patterns from the database for different reasons. Some removed patterns can be handled more efficiently by alternative pattern-matching approaches. Others simply occur so rarely that they do not justify dedicated hardware resource or architectural change.

In PERG, a pattern is removed if it contains one or more segments that meet any of the following criteria:

1) Equal or shorter than four bytes long: regardless of the false-positive probability, such short segments would have high hit rates and saturate the fragment reassembly process which is sequential in nature. Since these types of rules are relatively scarce, they are bettered handled by alternative methods like CAM or FSM-based approach. Alternatively, a separate mini version of PERG's hardware engine using on-chip memory for its Metadata Unit can be used.
2) Contain seven or more Byte-Or operators: See Section 4.5.
3) Contain half-byte wildcard operator(s) (?) or odd number of half-bytes: Like other pattern-matching engines, PERG operates only operate on byte boundaries.

1128 patterns are removed based on criteria above. A total of 84,387 patterns remain.

4.4 Filter Consolidation

The next stage determines the number of Bloomier filter units (BFUs) needed in the configurable hardware and the hash 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, the compiler selects a *small number* of distinct filter lengths and splits any segments of the wrong length into two *overlapping fragments* of the correct length. This process consolidates several BFUs of differing lengths and differing utilization levels into a single highly-utilized BFU. The actual process of splitting segments is performed in the next stage (Section 4.5).

The C code to select filter lengths is simplified and shown in Figure 4.3. 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 the *segmentation* stage. The algorithm starts from the longest segment length down to the shortest length supported; both parameters are user defined.

Xilinx's 18kb BRAM can be configured to 9-bit mode with 2048 entries, so PERG uses 8 bits for a primary checksum and 1 bit for the Bloomier hash selection data. Moreover, the BRAM has two read ports, allowing two hash functions to index the table in parallel. A Bloomier filter table requires $n*k$ entries for n patterns and k hash functions. Consequently, BFU_TABLE is set to 1024 as that is the theoretical maximum number of patterns that can be mapped on a table of 2048 entries with two hash functions.

The algorithm tries to accumulate as many segments as possible (set by UTILIZATION in term of percentage) without over-filling (<1024) a Bloomier filter table of 2048 entries. If the *accum_fragment* exceeds BFU_TABLE, a larger table will be created by consolidation multiple BRAMs to support more than 2048 entries. In the actual implementation (not shown in Figure 6), this multiple is forced to be a power of 2 to simplify address decoding for the consolidated filter table. The maximum number of BRAMs allowed to be consolidated into one filter is user-defined. A new filter is created whenever the current utilization exceeds the UTILIZATION constant while remaining within 100%.

A new filter is also created if the last-created filter uses a length that is twice the current length – this ensures no information is lost through segment splitting (see next stage in Section 4.5). Another exception that will result in a new BFU is if the *basic_patterns[i]* is exceedingly large relatively to BFU_TABLE; this is done to avoid the overhead from splitting each of them into two segments. Finally, the algorithm has a lower UTILIZATION value as *i* approaches close to the minimal length of 4 bytes. In the ClamAV database, the most common fragments are relatively short in length. Creating more filter units at those lengths reduces the need to break up these short strings and therefore reduces *cache size* and average *burst length* (see Section 4.7).

```
for( i=longest_segment_or_fragment_length; i>=min_length; i--) {
    new_segments = basic_patterns[i] + segments[i];
   if( new_segments > 0 ) {
      accum_fragments += new_segments;
      if ((accum_fragments > UTILIZATION*BFU_TABLE && BFU_TABLE > accum_fragments )
        || (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 4.3: Simplified C code for filter consolidation step

4.5 Filter Mapping

The next stage maps all the patterns to the BFU lengths and sizes selected during the filter consolidation stage. This stage performs the actual splitting of segments into two overlapping fragments to match the chosen BFU lengths. If a segment is split in this stage, its link numbers are adjusted accordingly to reflect the new arrangement.

Previous approaches [2, 4] have also suggested splitting long patterns into smaller segments. However, in their case, splitting is done to create two disjoint (non-overlapping) segments. In comparison, the novelty of splitting to two overlapping segments as presented in this thesis has three main advantages. First, this greatly simplifies the filter consolidation's cost function; a split always results in two segments of equal length and therefore doubles the cost for the next filter length. Second, overlap also reduces the number of unique filter lengths needed. For example, if the pattern "ABCDEFG" is split to "ABCDEF" and "G" instead of two overlapping "ABCDEF" and "BCDEFG", two filter lengths (7 and 1 byte long) are needed as oppose to one (7 byte long). Third, two long overlapping segments have lower hit rate than one long segment with a short segment. Using the same example, "G" would have a very high hit rate than "BCDEFG" due to the probability of its natural appearance in a string. A high hit rate in a filter can slow down the overall system performances by increasing the number of unnecessary memory and exact-matching requests.

4.6 Or-Expansion

In this stage, each segment containing one or more Byte-Or operators is expanded to all possible string combinations as demonstrated in Figure 4.4. The newly created segments will have their own metadata as different string combinations should result in different checksums. Their link numbers however will not change since their order within a pattern is not affected by Or-expansion. As a result, each of the expanded segments will have the same link numbers.

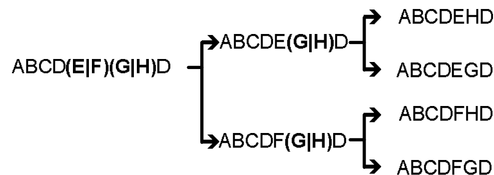

Figure 4.4: Byte-Or expansion

Expanding a pattern with n Byte-Or operators results in 2^n patterns in total. The pattern compiler enforces a user-defined limit of six Byte-Or operators per pattern. Patterns containing more than six Byte-Or operators are removed in the special-case removal stage as discussed in Section 4.3. With a cap of six operators, the Or-expansion of the ClamAV database generates 492 additional segments.

Due to exponential growth, a cleverer scheme may be needed for other database. One technique is breaking down any segment with larger number of Or-operators into smaller segments such that individually, each of them have only a portion of the Or-operators and overall they contribute less expansion overhead in total.

4.7 Metadata Generation

This stage is responsible for the creation of the metadata information for each segment. Metadata contains the following attributes:

- Rule ID
- Current and expected link numbers
- Displacement
- Wildcard prior
- Wildcard type
- Secondary checksum

The meanings of link numbers, displacement, and secondary checksum have already been defined in previous sections. Wildcard prior indicates whether the segment is immediately preceded by a wildcard operator in its original pattern form. Similarly, a wildcard type indicates whether the segment is followed by a within wildcard, at-least wildcard, or neither. How these are actually used to support wildcard patterns is detailed in Chapter 5.

Rule ID is simply a substitution of the original virus name. Instead of keeping the virus's name in string form, an ID number is assigned and used to identify each virus pattern for computation and storage reasons. Naturally, a rule ID is common for metadata within the same pattern and unique among patterns. Segments expanded from the same segment during Or-expansion share the same metadata as well.

4.7.1 Unique Segment

PERG implements the Metadata Unit in an off-chip memory that is 64-bits wide. Under such a configuration, the metadata for each segment has the encoding shown in Figure 4.5. This encoding is for a segment that uniquely appears in just one *multi-segmented rule*.

Type Flag (3 bits)	Link # (4 bits)	Byte Distance (9 bits)	Rule ID (16 bits)	Secondary Checksum (32 bits)

Figure 4.5: Metadata format for unique fragment

Type flag encodes the type of metadata and wildcard information using the scheme in Figure 4.6. The *Cache Pointer* will be explained in the next section. *Regular Fragment* refers to fragment of a multi-segmented pattern that is separated by a *fixed* byte distance (due to either displacement operator or overlap-splitting).

Table 4.1: Type flag encoding format

Binary Encoding (3 bits)	**Preceded by**	**Followed by**
011	Wildcard	Wildcard - Within
001	Wildcard	Regular Fragment
010	Wildcard	Wildcard – At-least
000	Cache Address Pointer	
101	Regular Fragment	Wildcard – Within
100	Regular Fragment	Wildcard – At-least
110	Regular Fragment	Regular Fragment
111	Single-segmented Pattern	

To improve storage efficiency, *Link #* stores only the current link number; expected link number is always one plus the current number. Suffix segment is indicated by a *Byte Displacement* of zero bytes. Displacement is always non-zero for prefix and substring segments, and zero for a suffix segment since nothing follows it.

A different encoding format is used for *single-segmented patterns*. Single-segmented patterns account for 40% (33,674) of the final database. Clearly, a 16-bit rule ID does not give enough range (0 to 65,535) for both multi-segmented and single-segmented patterns (84,387 in total). Since this type of pattern does not need overhead information like *Byte Distance* (displacement), some of those bits can be reallocated to Rule ID. The format for single-segmented patterns is shown in Figure 4.6. Type flag and Link # bits are always high in this case.

Type Flag (3 bits)	Link # (4 bits)	Reserved (1 bit)	Rule ID (24 bits)	Secondary Checksum (32 bits)

Figure 4.6: Metadata format for unique fragment

4.7.2 Common Segment

In some situations (1303 cases), a segment string is *shared* by multiple patterns. For example, if pattern “ABCD” and “ABCF” are to be mapped to a BFU whose hash length is three characters long, after filter-mapping stage, both patterns would contain the fragment “ABC”. Each instance however needs its own Rule ID, Link #, and etc.

For each common segment, its secondary checksum, common to all patterns sharing this segment, is stored off-chip in the Metadata Unit. The unique per-pattern data is stored on-chip in a small yet fast storage called *cache*. Cache is read once for each pattern sharing the segment. Consequently, if a common segment shared by two patterns is detected, the cache will be read twice to retrieve the data for each pattern. The number of cache accesses, which is equivalent to the number of patterns sharing this segment, is referred to as *cache burst length*. Like Metadata Unit, cache is read-only and written only during database compilation.

Since most information for a common segment is stored on cache, its metadata on the Metadata Unit has a different set of attributes than those of unique segments as shown below:

- Address pointer to cache memory location
- Cache burst length
- Secondary checksum

The encoding format is shown in Figure 4.6. Type flag and Link # bits are always low. The format for information stored in the cache is exactly the same as Figure 4.5 but without the secondary checksum.

Type flag (3 bits)	Link # (4 bits)	Reserved (1 bit)	Cache Burst Length (8 bits)	Cache Address Pointer (16 bits)	Secondary Checksum (32 bit)

Figure 4.6: Metadata format for common segments

4.7.3 Rule ID Assignment

Rule ID is assigned according to the type of segment into different ID partition ranges. The process first divides single-segmented and multi-segmented patterns into two partitions. This division ensures that each multi-segmented pattern, which has only a 16-bit rule ID, is unique within this bit range.

As a result of filter-mapping, it is possible for a single-segmented pattern to be also a common fragment of another multi-segmented pattern. For the purpose of rule ID assignment, these special patterns are categorized as multi-segmented patterns as their information are stored in cache. They can be still recognized as single-segmented patterns by their values of Link # and Byte Distance, which are both zero (i.e. it is both a prefix and suffix).

The multi-segmented pattern partition can be further partitioned into two sub partitions: one for any pattern with one or more wildcards and one for those without. This division is done so Rule ID can directly address the *Wildcard Table*; details about this mechanism are discussed in Section 5.4.2. The resulting Rule ID partition map in PERG using the ClamAV database is shown in Figure 4.7.

<table>
<tr><td rowspan="2">Multi-segmented Pattern</td><td>With Wildcard</td></tr>
<tr><td>Without Wildcard</td></tr>
<tr><td colspan="2">Single-segmented Pattern</td></tr>
</table>

Figure 4.7: Rule ID partition

While the 64-bit encoding presented in this section works fine for the database used in this thesis, the current format has a few limitations. For example, each pattern cannot contain more than 16 segments since Link # is only 4 bit wide. As a result, a larger database may demand a wider metadata format. However, the same scheme can be used; the only adjustment needed is widening the bit widths of Link #, Rule ID, and Cache Address Pointer.

4.8 BFU Generation

This stage generates the Bloomier Filter Units (BFUs), starting from the shortest length to the longest. For each BFU, the construction creates two hash functions based on the SAX hash discussed in Chapter 2 and shown again in Equation 4.1. In this equation, H_i is the partially-formed hash value after seeing i bytes and H_0 is an initial constant seed. C_i is the ith 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.

$$H_i = H_{i-1} \textbf{ XOR } ((H_{i-1} << S_{Li}) + (H_{i-1} >> S_{Ri}) + C_i) \qquad (4.1)$$

As mentioned in Section 2.3, 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, if failure does occur, 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 a good H_0 selection. However, if the failure occurs later after many filters have been constructed, a complete reconstruction of all filters is tedious. Instead, construction algorithm modifies

the shift values for that filter length without disturbing the previous generations. This incremental hashing significantly reduces the time required for filter generation. The process is completely automated and takes only 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. Checksum entries in the table that do not map to a pattern are initialized to all 1's. Our experimental observations show that it is common to receive a chain of 0's in a file stream but not 1's. Initializing checksum entries to 1's therefore reduce the false positive probability.

4.9 RTL Generation

The last stage generates the Verilog instantiation of BFUs: the hash functions used, its byte length, the size of its hash table, etc. In addition, the bus interface for BFUs to the Metadata Unit is also generated at this stage since it is dependent on the number of BFUs.

Another hardware component generated at this stage is the memory decoder for the Metadata Unit. The memory decoder is dependent on both Rule ID partition and the number of BFUs. In PERG, each metadata entry is stored in a unique location in the Metadata Unit. By assigning a unique ID number to each BFU generated, the memory address for the metadata entry can be derived by concatenating the hash value of each stored segment with its BFU ID. The resulting number is guaranteed to be unique, because the hash value is uniquely associated with each segment within each BFU and the BFU ID is unique among the BFUs.

These hardware configurations are recorded as Verilog header files and are synthesized along with the rest of the configurable hardware architecture.

Chapter 5. PERG Hardware

This chapter covers in detail the PERG hardware architecture. The PERG hardware architecture, shown in Figure 5.1, is divided into *Inspection*, *Metadata*, and *Fragment Reassembly Units*. The Inspection Unit and Fragment Reassembly Unit (FRU) reside in the FPGA while the Metadata Unit is stored in an off-chip memory. A high-level description of what these units are and how they interact with each other is already covered in Chapter 3. This remainder of this chapter walks through the implementation detail of each unit, in the order of the data flow. Due to the complexity of the design, regular-expression support is covered in a section of its own in Section 5.4.

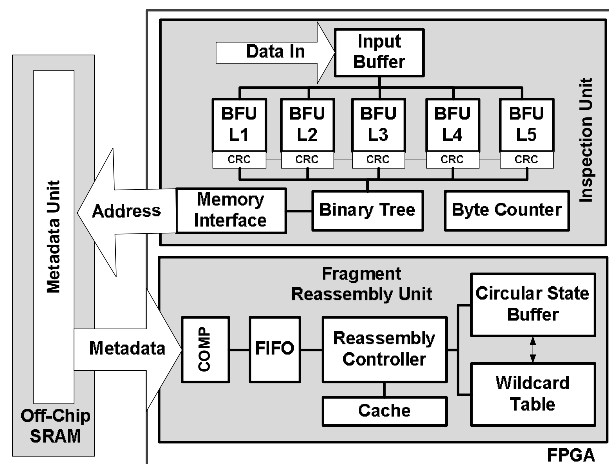

Figure 5.1: Top-level architectural diagram of PERG.

5.1 Inspection Unit

The block diagram of the Inspection Unit is shown in Figure 5.2. At each cycle, a new data byte from the input file stream arrives at the Inspection Unit. A 32-bit Byte Counter counts the number of bytes in the current file stream.

Hash values as well as primary and secondary checksums are calculated with each arriving input byte. The SAX hashing pipeline is fully unrolled and shared by all parallel BFUs in the same way as [3], resulting in a sliding-window hash.

The primary and secondary checksums use the same pipelined structure. The primary check is based on cyclic-redundancy check 8 (CRC) while the secondary checksum is based on 2's complement sum.

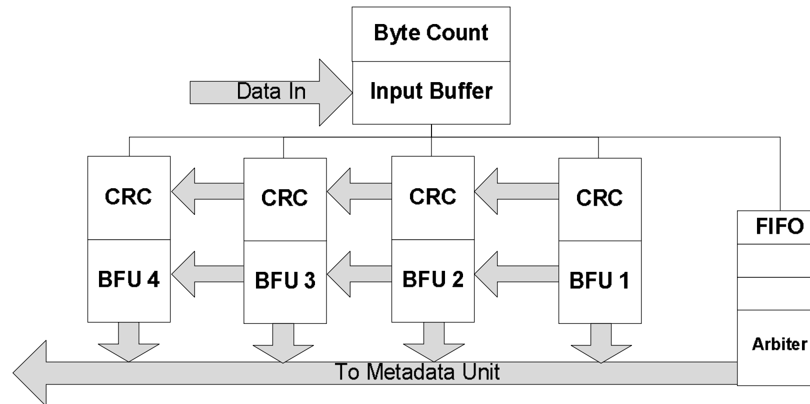

Figure 5.2: Inspection Unit.

5.1.1 Bloomier Filter Unit

The internal structure of a BFU is shown Figure 5.3. The hash table is stored in Xilinx 18kb BRAMs. Each 18kb BRAM can be configured to have 2,048 entries. Under this configuration, each entry is 9-bits wide. Eight bits are used for the primary checksum. While the 9th bit is originally intended as a parity bit, it serves nicely as storage for the Bloomier filter select bit $V(x)$.

Each BRAM also comes with two true access ports. Both ports are used for read only. One of the ports is multiplexed to allow writing to the hash table to facilitate quick initialization and online database updates.

Each port is addressed by the output of the hash pipelines. Each data output contains a $V(x)$ and primary checksum. Using $V(x)$, the hash select can be used to select the following:

- Primary checksum to compare against the input data checksum
- Hash value to pass to memory decoder to derive metadata address

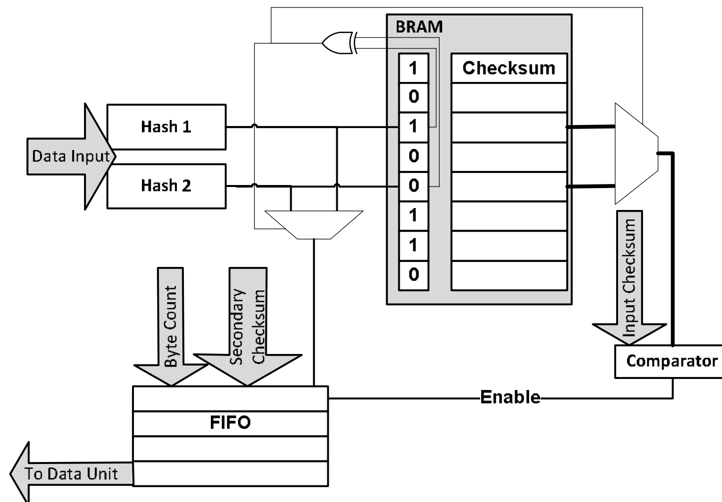

Figure 5.3: Bloomier Filter Unit (BFU)

Multiple BFUs can report hits in the same cycle, so each BFU has a small BFU FIFO (16 stages to map with Xilinx SRL16). Burst of near-simultaneous hits need to be serialized by a bus interface and processed in byte count order. Because PERG contains 26 BFUs, traditional arbiter-based bus architecture would scale poorly with such a large number of bus masters. Instead, a pipelined binary tree is used as shown in Figure 5.4.

Requests flow from leaf to root node; the leaf nodes takes input from the output of the BFU FIFOs, and root node is connected to the memory interface of the Metadata Unit. Each node (circle in Figure 5.4) is a mini state machine consisting of mainly a register and a comparator. Comparison of byte count values are always between two nodes only. The pipelined structure also forms a natural FIFO to buffer additional requests to the off-chip Metadata Unit. Additional FIFO can be added at the output of the tree if needed.

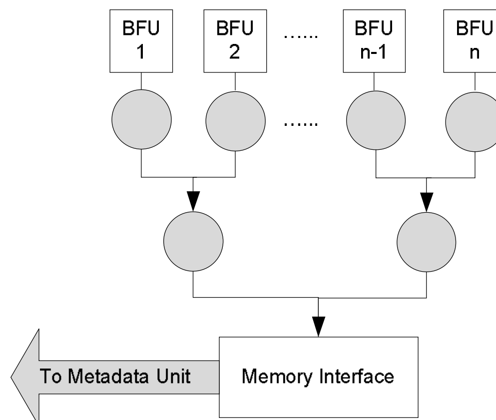

Figure 5.4: Pipelined binary tree

5.2 Metadata Unit

Metadata Unit is a small external memory used to store metadata. The unit receives metadata addresses from the Inspection Unit and outputs the requested data to the FRU. Writing to Metadata Unit is only performed during initialization or database updates.

Because the request (hit) rate is relatively low from the Inspection Unit, the underlying memory can operate at a lower clock rate than the Inspection Unit. However, faster memory can help reduce the size of FIFOs needed inside the Inspection Unit.

Metadata Unit memory is always less than $1/N$ utilized, where N is the number of hash functions used in the Bloomier filters, because of the way the metadata address is derived as described in Chapter 4. While it is possible to form compact metadata addresses through an on-chip pointer table as in [14] to improve utilization, this option increases latency, adds complexity, and uses expensive on-chip FPGA memory to save relatively cheap external memory.

5.3 Fragment Reassembly Unit

The Fragment Reassembly Unit (FRU) reconstructs full patterns from fragments or segments. Inputs to the FRU include metadata, byte count of the request, and secondary checksum of the input string. Both byte count and secondary checksum of the input are received directly from Inspection Unit, while metadata comes from the Metadata Unit.

The first stage compares secondary checksums from Metadata and Inspection Units. A mismatch allows the request to be dropped. A match with a single-segmented pattern generates an interrupt to the host for exact matching. A match of a fragment of a multi-segmented pattern results in the fragment's metadata sent to the Reassembly Controller. Matching a suffix fragment of a pattern is insufficient; all prior fragments to the pattern must also have matched in the correct order at their expected positions, forming a trace of the pattern. The metadata indicates how fragments link together in a trace.

When the Reassembly Controller receives metadata, it passes the same metadata to both Circular Speculative Buffer (CSB) and Wildcard Table (WT) in parallel. CSB is used to reconstruct segments separated by fixed byte gaps and is discussed in Subsection 5.3.2. WT is accessed whenever the metadata indicates it is preceded or followed by a wildcard operator. Details about WT and wildcard support are presented in Section 5.4.

For both CSB and WT, the metadata process can be divided to two conceptual 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 CSB may indicate it is followed by a wildcard. In such a scenario, CSB is used to verify that the incoming fragment is expected by an ongoing trace. Upon the verification, CSB will send a control signal to the WT so it will record the expectation for the next segment.

5.3.1 Cache

Cache, described in Chapter 4, is also an important part of the FRU. Unlike typical caches, cache in PERG is simply a fast read-only memory for storing metadata of segments shared in common by multiple patterns. For each pattern sharing the segment, one cache access is needed to retrieve its metadata, and one update to the pattern's overall progress is made in either the Circular Speculative Buffer or Wildcard Table, or both. Because metadata is always processed in time (byte count) order, metadata output from the cache are processed as a continuous burst until all metadata of patterns sharing the common segment are served. In comparison, when serving only metadata from the Metadata Unit, not all components in the FRU pipeline are active because the FRU, like Inspection Unit, is expected to operate many times faster than the Metadata Unit.

While the Reassembly Controller is directing metadata from cache to CSB and WT, new arriving metadata from Metadata Unit must be buffered until the cache burst finishes. However, since cache operates at a much higher speed, the cache burst is generally transparent. In PERG, the average cache burst length is two patterns, and most cache burst lengths are less than four patterns.

5.3.2 Circular Speculative Buffer

The *Circular Speculative Buffer* (CSB) as shown in Figure 5.5 tracks the progress of multiple traces for various multi-segmented patterns. 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 used buckets used is called the *Occupancy*. Unlike a time wheel, however, CSB schedules events up to only N-1 bytes into the future, where N is the number of entries (rows) in the circular buffer (512 here). This limitation is done to conserve hardware resource.

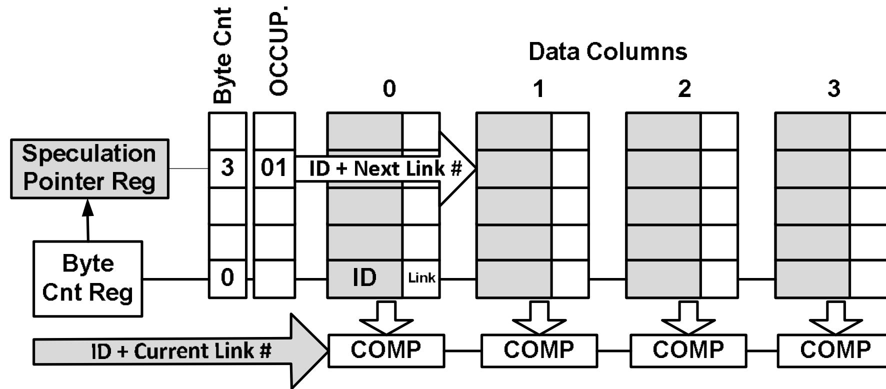

Figure 5.5: Circular Speculative Buffer (CSB)

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 is three bytes, meaning the last byte of the next fragment for this rule will appear 3 bytes into the future. This three 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 # of the next fragment and the current Rule ID. As additional input is scanned, additional fragments may be matched and the current Byte Counter 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 # 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 the CSB implementation presented in this thesis, the number of Data Columns is fixed at four columns. It is possible, though improbable, that more than four traces are expected to have a fragment match at exactly the same Byte Count. To handle this, the Occupancy is incremented once more (to five) to signify this condition. On a fragment match at this particular Byte Count, CSB ignores the Data Column contents and always assumes 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 potential 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.

5.3.3 Exact-Match Request to Host

Upon a match in the FRU, request for exact matching is sent to the host in the form of interrupt. PERG also provides two additional information: the Rule ID and Byte Count of where the virus pattern is detected. Since the value Rule ID is bounded, it can be used to derive memory pointers to the full virus pattern in the host memory. The Byte Count indicates the byte position in the file stream where the last byte of the suspected malicious pattern resides; the exact matcher of the host system can then pattern-match in reverse byte order for the reported pattern.

5.4 Wildcard Support

This section describes how wildcards of different types are supported, in particular with the Wildcard Table (WT) in the FRU. Before getting to wildcards, below is a summary of regular-expression operators in Table 3.2 that are already supported by the mechanisms presented up to this point (Note: *single-nibble wildcard* is removed by the pattern compiler as described in Section 4.3).

- *Byte-Or*: Support for the byte-or operator is done through the Or-expansion phase in the pattern compiler as discussed in Section 4.6.
- *Displacement*: Displacement operator is supported through the CSB as explained in Section 5.3.1. Because the number of entries in the CSB is limited by the available hardware resources, a user-defined limit of 512 byte displacement is enforced by the pattern compiler. Displacement operators greater than 512 bytes are converted by the compiler to an a*t-least wildcard* to be discussed in Subsection 5.4.1.
- *Single-byte wildcard*: A single-byte wildcard is simply a displacement of one byte. Single-byte wildcard is then treated exactly the same as the displacement operator in the segmentation stage of the pattern compiler.

With the above list, the remaining operators are *at-least*, *any-number-of-byte*, *within*, and *range wildcards*. The treatments of these operators are discussed next.

5.4.1 Wildcards

The remaining wildcards can all be converted to at-least or within wildcards. This conversion is possible because PERG is allowed to generate false positive and rely upon an exact matching done by the host. As a result, by supporting at-least and within wildcards, all the remaining wildcard types are also supported by PERG.

As described in Chapter 4, a pattern with wildcards is partitioned into segments at the position where a wildcard operator is found. Consequently, the resulting segments' metadata goes through the same offset adjustment described in Section 4.4. Regardless of the type of wildcard, if a segment is n bytes long, at least n bytes must have passed since the prior segment before it is possible to detect this segment.

A *within wildcard* of N bytes between two segments means in order for the pattern to match, the appearance of the segment that immediately follows the wildcard operator must appear within N bytes or less after the appearance of the segment immediately before the wildcard operator. If the second segment has not arrived after N bytes of the first segment, this particular pattern trace shall be terminated as a mismatch even if the second segment eventually arrives later in the same input stream.

Due to offset adjustment in PERG as described in Section 4.2, within wildcards actually are converted to range wildcards. The newly-introduced lower bound equals to the byte length of the segment prior the within wildcard. The upper bound is increased by the length of the segment following the within wildcard. This conversion exists only in intermediate stage of the pattern compiler however as the lower bound shall be dropped. The reason for this conversion is discussed in the later part of this section.

An *at-least wildcard* of n bytes between two segments means n bytes must have passed after the last byte of the first segment before the pattern matcher needs to match for the second segment in the input. In other words, any appearance of the second segment in the input before n bytes has passed has absolutely no effect on the progress of the trace. However, once n bytes have passed, the pattern matcher shall always look out for the arrival of the second segment. A consequence of this behavior is that once the first segment is detected, there is no need to create a new trace if another instance of the first segment ever appears again in the same input stream; this will be reset when a new file is presented to PERG.

An *any-number-of-bytes wildcard* between two segments means the second segment can arrive any time after the appearance of the first segment. Once the first segment is detected in the input, the pattern matcher shall always look out for the second segment. As a result of offset adjustment, any-number-of-byte wildcard is simply an at-least wildcard with *n* bytes, where *n* is the byte length of the segment that follows immediately after the wildcard operator. This transformation is perfect; it does not introduce any new false positives.

A *range wildcard* of *n*-to-*N* bytes between two segments has two conditions that must be satisfied in order for the pattern match to progress forward. First, *n* bytes must have passed between the appearances of the two segments in the input. Second, the following segment must appear within *N* cycles after the appearance of the prior segment.

The fact that each range wildcard contains two values (lower and upper bounds) as opposed to one raises two implementation problems. The first is hardware simplicity: it is difficult to justify the tradeoff of additional complex logic just for range wildcards, which have relatively few occurrences in the actual ClamAV database. Second, an additional value means extra storage requirements, higher external memory bandwidth demand, and possibly a wider memory datapath; again with the sparse usage of range wildcards in the database, these costs are not justifiable.

Instead supporting range wildcards directly, this type of wildcard is converted to within wildcard of *N* bytes by discarding the lower bound information (*n*). For example, the pattern "ABC**{10-20}**DEF" indicates a lower bound of *n*=10 bytes and an upper bound of *N*=20 bytes. This now becomes "ABC**{-20}**DEF". This conversion is performed at the segmentation stage in the pattern compiler.

The above approach is lossy as it introduces false positives that would otherwise be filtered out by the lower bound condition. However, the system remains with zero false negative probability as whatever satisfies the range condition of *n-N* bytes must also satisfy the up-to *N* byte condition. Using the previous example, if the input satisfies the pattern "ABC**{-20}**DEF", it must also return true when matched against "ABC**{10-20}**DEF".

The upper bound is kept instead of the lower bound because an upper bound implies a life time to the pattern trace and therefore has a lower overall increase in false positive probability. In comparison, if a range wildcard is converted to an at-least wildcard, the system will remain prone to false positives after the detection of the first segment until the very end of the file stream.

In summary, the remaining wildcards can be converted to either at-least or within wildcards as shown in Table 5.1. The next section describes how at-least and within wildcards are handled in with the Wildcard Table hardware.

Table 5.1: Conversion of different ClamAV wildcard operators

Symbol	Original	After Conversion
??	Single-Byte Wildcard	Displacement
*	(Any-Number-of-Byte) Wildcard	At-Least Wildcard
{n}	n Byte Displacement	Displacement
{n-}	At-Least (n-Byte) Wildcard	At-Least Wildcard
{-N}	Within (n-Byte) Wildcard	Within Wildcard
{n-N}	Range wildcard	Within Wildcard

5.4.2 Wildcard Table

The Wildcard Table (WT) is essentially a one-dimensional array used to record the state of each pattern containing one or more within or at-least wildcards. Each every entry is mapped to one unique pattern with one or more wildcards and indexed directly by the rule ID of the incoming metadata. For this reason, the wildcard patterns are assigned the first block of Rule IDs starting at 0. The contents of each WT entry are shown Figure 5.6.

Valid (1 bit)	Wildcard Type (1 bit)	Byte Range (9 bits)	Link # (4 bits)

Figure 5.6: Attributes of Wildcard Table entry format

The *Valid bit* is used to indicate the relevance of the data in the WT entry. A WT entry shall only be interpreted if its valid bit is set; otherwise, a WT entry behaves as if all bits are zeroes. The Valid bit, stored as a flip flop, is cleared at the start of each input stream and set whenever the WT writes to it. The remaining attributes of the WT entry are stored in a Xilinx 18kb BRAM; while these attributes are never reset, they can be safely overwrittenb when the entry's Valid bit is cleared.

Wildcard type is a flag that indicates whether the segment is followed by an at-least or within wildcard; When a trace is started and the Valid bit is set, the Wildcard type bit is determined from the Type flag attribute in the metadata. The type of wildcard decides whether lower (at-least) or upper (within) bound condition needs to be satisfied for the incoming segment in order for the trace to progress.

Byte Range is the displacement value, *n* or *N*. This value is also determined from the *byte distance* attribute in the metadata; whether the value is interpreted as the lower (*n*) or upper (*N*) bound depends on the metadata's type flag.

Link # is used to track the arrival sequence order among multiple segments in a pattern. This is exactly the same as in CSB.

Because each pattern may contain multiple wildcards, multiple traces of the same pattern may arise while scanning an input stream. Since there is only one WT entry per pattern, the WT can only track one trace per pattern. In other words, within the same input stream, different traces of the same pattern are mapped or *compressed* in a lossy fashion into one storage location to eliminate the need to keep and maintain separate state information for a potentially exponential number of traces of the same pattern.

To compress, a trace of a pattern only progresses forward (i.e., advances in Link #) and never backtracks. Each entry remains static only until the next expected segment arrives. The repeated arrival of current or previous segment of the same pattern is ignored. The effective state diagram is shown in Figure 5.7.

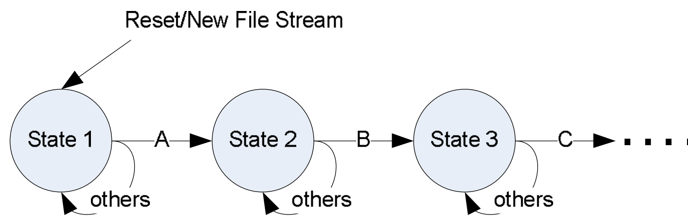

Figure 5.7: Wildcard state diagram

Moving forward not only saves storage but also ensures false negative probability remains at zero. In comparison, if a traditional AC automaton is used here instead, a false negative is possible since the automaton may be misled by false positives (emitted by the BFUs) to a *false state* and overlook the segment it should have been looking out for.

For at-least wildcards, the proposed WT mechanism works without any increase in false positive probability. Take the example pattern "ABC**{3-}**DEF**{3-}**EFG" and assume its WT entry has already detected up to the second fragment "DEF". Regardless of when and how many instances of "ABC" and

"DEF" appear later, what matters is the appearance of "EFG", which can appear any time after three bytes have passed after the first "DEF" is detected.

In the case where a fragment is preceded by an at-least wildcard and followed by a regular fragment (as the "OMG" in the pattern "LOL**{3-}**OMG**{3}**UBC"), 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 continue to watch for the arrival of "OMG" while the CSB waits for the next fragment "UBC". In this case, the simultaneous use of the CSB and WT allows two traces of the same pattern to be followed.

For within wildcards, everything works the same as at-least wildcards 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 segment. 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. However this approach would increase false positives so it is not used.

Instead the WT keeps the upper bound and refreshes the Byte Range condition every time the current segment is matched again. With this approach, the WT is monitoring two traces: both the current and expected link numbers of the incoming metadata. Whenever the current segment (whose expected link number is the same as the link number in the WT entry) reappears in the input stream, the WT entry's byte range, which represents the upper bound for the next segment, is updated accordingly. The new value should be the sum of the metadata's Byte Count (when the hit occurred in the BFU) and its byte distance attribute; calculation of this value can be computed ahead of time as metadata flows through the FRU FIFO.

The operation of wildcards is summarized in the pseudo code in Figure 5.8.

In any true trace of a within wildcard (i.e., the real pattern does actually appear in the input stream), the current segment must always arrive before the appearance of the next fragment at a byte position satisfying the bound condition. In the proposed 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 Byte Range is refreshed by a current segment that appears but does not satisfy all of the proper preceding segments in the overall rule.

```
//Is the incoming metadata a segment preceded by a wildcard operator?
If ( metadata.preced == WILDCARD){   //Yes
        //Is the incoming metadata expected by an ongoing trace?
        If (WildcardTable[metadata.ruleID].link == metadata.link){   //Yes
                //What type of wildcard operator is it?
                If (WildcardTable[metadata.ruleID].type == WITHIN ) { //Within
                        //Did the metadata arrive within the ByteRange condition?
                        If (WildcardTable[metadata.ruleID].byte_cnt > metadata.byte_cnt ){
                                //What type of metadata to expect next?
                                If ( metadata.follow == WILDCARD) {
                                        Update_WT(metadata);
                                }
                                else {
                                        Update_CSB(metadata);
                                }
                        }
                }
                else {  //At-least
                        //Did the metadata satisfy the ByteRange condition?
                        If (WildcardTable[metadata.ruleID].byte_cnt < metadata.byte_cnt ){
                                //What type of metadata to expect next?
                                If ( metadata.follow == WILDCARD) {
                                        Update_WT(metadata);
                                }
                                else {
                                        Update_CSB(metadata);
                                }
                        }
                }
        }
        //Special Case for Within Wildcard:
        //If the incoming metdata is a Within type, check if it is the same as the previous metadata
        else if (WildcardTable[metadata.ruleID].link == metadata.next_link ) {
                if (WildcardTable[metadata.ruleID].type == WITHIN ) {
                        //Update ByteRange if extend the trace's lifetime
                        WildcardTable[metadata.ruleID].byte_cnt= metadata.range+metadata.byte_cnt;
                }
        }
}
else { //No, it is an ordginary displacement segment
        //Verify with CSB
        if (Verify_CSB(metadata) == EXPECTED){
                if ( metadata.follow == WILDCARD) {
                        Update_WT(metadata);
                }
                else {
                        Update_CSB(metadata);
                }
        }
}
```

Figure 5.8: Pseudo Code describing behavior of Wildcard Table

One issue with the WT is that forward-only lossy progress may lead to a large number of verifications. For example, for the pattern "ABC**{-3}**DEF**{-3}**EFG", consider that DEF may have been detected by a false positive. Once progress is made to the last fragment EFG, verification must be done whenever another EFG segment is matched in the data stream. The only time this history state is cleared is when a new file begins. However, our experiments show that with the Bloomier filters and two levels of checksum, it is very unlikely a trace would progress that far in the first place. Alternatively, host software can roll back the WT for a rule when verification by the host fails. This is left as future work.

Chapter 6. Experimental Results

This chapter is divided in three main sections. The first section discusses the experiment setup. The second section evaluates PERG's performance and resource density and compares them with existing pattern-matching solutions. The third section analyzes the scalability and updatability of the PERG architecture.

6.1 Experiment Setup

PERG is evaluated in two ways using: 1) a cycle-accurate simulator and 2) a synthesizable Verilog model. The cycle-accurate simulator is used to measure performance with a real file stream, to monitor false positives, FIFO utilization, and the back-pressure flow control. The Verilog model is used to evaluate FPGA resource consumption and operating frequency of the hardware. Both the simulator and Verilog model have the same system parameter configurations as shown in Table 6.1. Note there is a minor difference in operation between the Verilog and simulator in the Metadata arbitration; a simple FIFO is modeled in the simulator as opposed to a binary tree in the Verilog since there is no fan-out issue involved in simulation.

Table 6.1: PERG system parameters

Compiler Parameters	**User Defined**	**Value**
Minimum Segment Length (bytes)	Yes	4
Maximum Segment Length (bytes)	Yes	256
# of Hash Functions	Yes	2
# of Data Columns in CSB	Yes	4
Maximum Byte Displacement	Yes	512
Hardware System Parameters		
Longest BFU Length (bytes)	Generated	4
Longest BFU Length (bytes)	Generated	150
Number of Bloomier Filter Units	Generated	26
Maximum Hash Table Size (entries)	Generated	32,768
BFU FIFO Depth	Yes	16
Maximum # of Cache Entries	Generated	512
State FIFO Depth	Yes	64
Arbiter FIFO Depth	Yes	64
Secondary Checksum (bits)	Yes	32

6.1.1 Cycle-Accurate Simulator

The cycle-accurate simulator, written in C, captures the expected behavior of the underlying hardware it is mimicking. The simulator allows collection of internal statistical information from a large amount of

input data which cannot be done easily on hardware. The statistics collected include such information as false positive probability and the number of off-chip memory requests. In addition, the simulator also provides an early functional model for development purposes, allowing early evaluation of the feasibility of the proposed algorithm and architecture. Finally, the simulator is also used to generate test vectors which are then fed to a Verilog testbench for the verification of the Verilog implementation.

6.1.2 RTL Implementation

PERG is implemented in synthesizable Verilog targeted for a Xilinx Virtex-II Pro VP100 FPGA on the Amirix VP1000 board. The circuit has been verified through Modelsim simulation and implemented with Xilinx ISE 9.1 with a core operating frequency of 180MHz. For the Metadata Unit, the model assumes a 4 MB CY7C1380 SRAM with a 64-bit data width running at a quarter of PERG's operating frequency at 45 MHz with a 2-cycle read latency. This choice of off-chip memory is also based on the Amirix VP1000 board and simulated with a Verilog model from Cypress Semiconductor.

The Verilog model here is intended as a proof of concept without much optimization effort. For example, the current bottleneck (worst timing slack failure) is at the 32-bit comparator used to compare the input Byte Count with the Byte Range of a Wildcard Table entry.

6.1.3 Pattern Database

The pattern database used is from main.db and main.ndb in ClamAV 0.93.1, containing 85,625 basic and regular expression rules in total. In total, 1,128 patterns are removed as special-cases in the pattern compiler, resulting in a total of 84,387 rules containing a total of 164,864 fragments and 12,049,565 bytes (characters). A total of 1,230 segments/fragments are identified as shared by two or more rules.

6.1.4 Test Setup

To evaluate the average performance as well as false positive probability of a realistic file stream, the Ubuntu 7.10-Desktop i386 ISO image is used as the data source. The file is scanned with ClamAV software to ensure no virus matches in this data. Hence, all matches found by PERG are false positives.

The use of an Ubuntu ISO enables two types of tests: Single File test and Extracted Files test. In the Single File test, the entire ISO image file (696 MB) is scanned. 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).

6.2 Performance Results

This section evaluates the throughput and resource density of the PERG hardware using the setup described in Section 6.1.

6.2.1 Throughput

The performance parameters for PERG is summarized Table 6.2. In both tests, four false positives have been reported. Three of out four are actually true pattern matches, but ClamAV considers them safe due to mismatched file extensions. It is worthwhile to note there are no false positives resulting from the Wildcard Table.

Table 6.2: Simulated performance 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 Probability for Each Byte Scanned	0.0000005%	0.0000005%
# of Off-chip Memory Requests	82,499,591	80,500,329
Probability of Off-chip Memory Request for Each Byte Scanned	11.31%	11.07%
Off-chip Memory Throughput	19.4 MB/s	19.0 MB/s
# of Secondary Check Hits	5,931,478	4,005,953
Probability of Secondary Check Hits for Each Byte Scanned	0.81 %	0.55 %
Average Throughput	166 MB/s (0.922 B/cycle)	168 MB/s (0.933 B/cycle)
Modeled Frequency	180 MHz	180 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, PERG sometimes (~7%) stops accepting a new byte and applies backpressure flow control. Backpressure can be reduced with larger FIFOs and faster off-chip memory.

6.2.2 Resource Consumption

PERG's resource consumption on Virtex II Pro VP100 is shown in Table 6.3. While PERG's LUTs consumption can fit well under most modern FPGAs, its block memory usage requires higher-end Spartan 6 or Cyclone III.

Table 6.3: Overall resource consumption

Resource	Value
LUTs	32,870
Flip Flops	13,624
BRAMs	168

6.2.3 Comparison

This section compares PERG's performance with existing pattern-matching solutions. Candidates chosen are those that published quantitative information regarding the database used and resource consumed; many previous publications on pattern-matching engines actually lack such information.

One immediate challenge to compare results is the lack of the same database input. PERG is the only known FPGA-based pattern-matching engine to use the ClamAV database. Among the previous works targeted for Snort, each of them has a dramatically different Snort database. For instance, [27], one of the most recently published works in 2009, still uses Snort 2.4 released over 3 years ago. Others like [28] use a proprietary pattern generator to evaluate the effectiveness of their architectures. Attempts to reproduce databases used by previous work were unsuccessful.

The latest and only publicly-available version of Snort 2.8 available from SourceFire contains myriad of text-specific operators from the PCRE standard like case-insensitivity that are currently not supported in PERG. While a lossy support for such operators may be possible through modification of the pattern compiler, the resulting impact on false-positive probability cannot be evaluated without a network testbench. As a result, mapping of Snort on PERG for comparison with previous work is left as future work.

For the reasons describe above, the observations on results shown Table 6.4 should emphasize on the throughput and density, while the total number of characters and patterns are purely for reference. Even among the Snort-targeted designs, the actual pattern sets used appear to vary greatly.

To address this issue for lack of a standard database, two papers [9, 10] use a Performance Efficiency Metric (PEM). This is now used by several other works in this area. PEM is defined as the ratio of throughput (Gps) to logics cells per character; i.e., *PEM = Throughput*/(*LC*/*Chars*) [9]. The assumption with PEM is that this ratio is relatively independent of the targeted database. A higher PEM means the system offers better performance and character density ratio.

Table 6.4 summarizes the performance and resource consumption of PERG and prior solutions. The results are meant to be a quantitative comparison; some of these candidates offer different features that others do not, such as regular expression support. As a result, the chosen list of candidates includes those that stand out in terms of quantitative results and provide sufficient quantitative information about their test setups. The best value at each metric is highlighted in bold.

Another worthy note is that other than PERG and [24], which is an FSM-based approach, none of the hash-based solutions shown [3, 9, 10] support regular expressions. It is unclear whether regular expression patterns are included in the number of patterns claimed in [3, 9, 10]. The density shown in Table 6.4 is derived from the assumption that only patterns actually supported are reported when it is unclear.

Note the throughput presented here for PERG has already factored in the average throughput shown in Table 6.2. Moreover, because of PERG's pattern compiler design, the actual number of characters and string mapped to PERG is greater than the number presented in Table 6.4. Finally, memory consumption reported only accounts for BRAMs, since on-chip memory is the limiting factor while off-chip memory is abundant in size and inexpensive in cost.

Although PERG lags behind in throughput performance, most of this is due to the lack of hardware optimization effort on the Verilog model. Nevertheless, PERG's current throughput is sufficient to keep up with most data interfaces, assumed to be in the range of 1 Gbps (125 MB/s). Most importantly, PERG outperforms the nearest competitor by 8.7x better in logic cell density and by 24.7x better in memory density.

Table 6.4: Resource utilization and performance comparison

System	Device (Xilinx)	Freq. (MHz)	# of Chars	# of LCs	Mem (kb)	LCs per Char	Mem. per Char (bits/char)	Throughput (Gbps)	PEM
PERG (raw)	XC2 VP100	180	**8,645,488**	42,809	3,024	**0.00495**	**0.354**	1.3	**262.62**
Cuckoo Hashing [3]	XC4 VLX25	285	68,266	2,982	1,116	0.043	16.7	2.28	53.02
HashMem [9]	XC2 V1000	**338**	18,636	**2,570**	630	0.140	34.6	**2.70**	19.60
PH-Mem [10]	XC2 V1000	263	20,911	6,272	288	0.300	14.1	2.11	7.03
ROM+Coproc [24]	XC4 VLX15	250	32,384	8,480	**276**	0.260	8.73	2.08	8.00

In terms of PEM, PERG also leads the nearest competitor by nearly 5 times. However, all these achievement may be due to the use of the much larger ClamAV database. One major problem with using LC/char, bits/char, and PEM is that they all are proportional to the number of characters in the database used. As a result, this approach strongly favors hash-based solutions whose consumption depends little on the number of characters in databases used. Instead, this thesis proposes two new metrics: **L**ogic cells per **P**attern (LP) and **M**emory bits per **P**attern (MP). With LP and MP, two additional metrics can be derived similar to PEM: **T**hroughput to **L**ogic cells per **P**attern (TLP) and **T**hroughput to **M**emory bits per **P**attern (TMP). Higher TLP or TMP means the system offers better performance and pattern density ratio.

Comparison using the new metrics is shown in Table 6.5. Note for patterns mapped in PERG, the number here refers strictly to the actual number of ClamAV signatures mapped to the system as opposed to the total number of segments generated by the pattern compilers and mapped to BFUs.

Table 6.5: LP, MP, TLP, and TMP comparison

System	Patterns mapped	LC per Pattern	Memory per Pattern (kb/pattern)	TLP	TMP
PERG	**84,387**	**0.5073**	**0.0358**	2.56	**36.28**
Cuckoo Hashing [3]	5,026	0.5933	0.2220	**3.84**	10.27
HashMem [9]	1,474	1.7436	0.4410	1.55	6.12
PH-Mem [10]	2,200	2.8509	0.1309	0.74	16.12
ROM+Coproc [24]	2,031	4.1753	0.1359	0.50	15.31

While PERG continues to lead in TMP, in terms of TLP, Cuckoo Hashing actually outperforms PERG by nearly 50%. The superiority of [3] over PERG is interesting considering that the two solutions share the same underlying SAX hash algorithm. The result is likely to be due to the extra logic required by the CRC-8 and 2's complement checksum pipelines in PERG. Fragment Reassembly Unit is another area that likely contributes to PERG's logic use. Finally, Cuckoo Hashing logic is simpler and runs at a higher throughput than PERG.

In addition to FPGA-based pattern matching engines, PERG's throughput is also compared against the ClamAV 0.93.1 software running on a modern PC powered by an Intel Core Duo E4500 2.2 GHz processor with 3GB of RAM. In this experiment, ClamAV performs recursive scan on the extracted files. The ClamAV scan uses the same database as PERG and therefore excludes the MD5 checksums. The throughputs of both solutions are shown in Table 6.6. The throughput of ClamAV is determined by dividing the size of the scanned data with its reported run time. Note unlike PERG, the ClamAV software can distinguish file extensions and therefore match against only the necessary patterns for each file. On the other hand, ClamAV does perform exact-matching, whereas PERG ignores the timing overhead required by the exact-matching of the four false positives detected.

Table 6.6: PERG vs ClamAV throughput

	Average Throughput	**Speedup**
PERG	166 MB/s	15.76x
ClamAV 0.93.1	10.5 MB/s	1x

6.3 Impact of Filter Consolidation

This section evaluates the impact of the filter consolidation technique introduced by this thesis. Comparison results are summarized in Table 6.7. The emphasis here is on the resource saving. While filter-mapping does increase the number of segments by 51,724 segments (57.8%), the use of filter consolidation results in 194 less BFUs, which translates to 88 fewer BRAMs.

Table 6.7: Impact of filter consolidation

	Without Filter Consolidation	**With Filter Consolidation**
Total # of Segments Mapped to BFUs	89,423	141,147
Total # of BFUs	220	26
Total # of BRAMs used by BFUs	256	168
# of Cache Entries	132	3823

In terms of performance impact, additional BFUs has little impact; as indicated later in Table 6.8, the hit rate of BFU has no obvious if any relation with the utilization of each BFU. Due to the pipeline structure, operating frequency is unlikely to change as well; additional resource usage however will have an indirect impact through placement and routing contention.

6.4 Scalability and Dynamic Updatability

Scalability refers whether the architecture can achieve high density/utilization of patterns in each BFU without reporting a large number of false positives. Dynamic update refers to the ability to add new rules to previously generated hardware without the need to recompile the RTL. This is important because recompiling and re-verifying the RTL can take hours or even days; this allows immediate virus protection during that time period.

This section evaluates the scalability and dynamic updatability of the PERG architecture. Given the rapid growth and frequent updates to virus databases, it is important to verify PERG scalability and updatability. The first part of this section evaluates PERG scalability realistically using a daily release of the ClamAV database. The second part explores the theoretical scalability and dynamic updatability; this part uses random patterns to provide avoid any dependency on the specific database used.

6.4.1 Scalability and Dynamic Updatability using ClamAV

This part evaluates scalability and dynamic updatability of the PERG architecture by generating a baseline system with ClamAV 0.93 and updating this baseline with new additional patterns from the ClamAV *daily* release updates. Note that this subsection uses a slightly modified pattern compiler compared to the rest of this thesis. While experimenting with the filter consolidation step, we decided to force a new BFU to be created for all segments less than 8 bytes in length. This result in slightly more BFUs (27 vs 26), but avoids breaking up very short strings into even shorter fragments; these short fragments run the risk of increased BFU hit rate due to the higher probability of true appearances in the data stream.

After compiling the main database, the utilization and configuration of each BFU is shown in Table 6.8. Recall from Chapter 4 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 (for example, the 32/16 split) or if the current length has an exceeding number of fragments that is much greater than the BFU_TABLE size. Since the maximum number of patterns that can be mapped onto a Bloomier hash table of m entries with two hash functions is $m/2$, theoretically, given there are 1.7 fragments per rule on

average, with 86.7% utilization in filter table entries, the current system is still capable of accepting 12,935 more rules with the exact same hardware resource usage. This estimation does not account for the fact that these new rules may contain common fragments, which utilize cache and do not need to be hashed separately into a BFU, nor does it account for the fact cache may already be full.

Table 6.8: BFU utilization and insertion

BFU Frag Len	Max # of Frags	Main Database			Update test 1		Update test 2	
		# of Frag	Util.	Hit Rate	Frag Succ.	Frag Fail	Frag Succ.	Frag Fail
150	16384	12994	79.3%	3%	50	0	50	0
149	1024	975	95.2%	3%	3	0	3	0
148	2048	1915	93.5%	3%	4	0	4	0
144	4096	4041	98.7%	3%	12	0	12	0
140	16384	11305	69.0%	3%	13	0	13	0
134	4096	3939	96.2%	3%	11	0	11	0
131	2048	1999	97.6%	3%	0	0	0	0
129	1024	101	99.5%	3%	3	0	3	0
128	2048	1939	94.7%	3%	6	0	6	0
125	2048	1845	90.1%	3%	2	0	2	0
116	8192	7511	91.7%	3%	18	0	18	0
101	16384	15103	92.2%	3%	56	0	56	0
97	4096	4066	99.3%	3%	128	0	128	0
95	2048	1926	94.0%	3%	10	0	10	0
76	16384	14868	90.7%	3%	70	0	70	0
69	4096	3991	97.4%	3%	31	0	31	0
57	8192	7710	94.1%	**5%**	81	0	81	0
52	2048	1961	95.8%	3%	50	0	50	0
39	16384	15109	92.2%	3%	268	0	268	0
35	8192	7536	92.0%	3%	79	0	79	0
32	2048	1828	89.3%	3%	56	0	56	0
16	16384	14024	85.6%	**6%**	575	0	575	0
11	4096	3620	88.4%	**6%**	229	0	229	0
7	2048	1202	58.7%	3%	202	**1**	202	**1**
6	1024	145	14.2%	3%	26	0	26	0
5	1024	156	15.2%	3%	33	0	33	0
4	1024	147	14.4%	3%	24	0	24	0

One possible counter argument to the scalability claim is that as a deviation of the Bloom filter, the false positive probability in each Bloomier filter will increase with higher level of utilization. In reality however, as shown in Table 6.8, the hit rate for each BFU, which is represented in percentage of total BFU hits, is relatively evenly distributed at 3% except for bursts at length 11, 16, and 57. While these bursts suggest one cannot conclude the hits in BFUs are evenly distributed, they also suggest that there is no obvious correlation between the hit rate and utilization. The reason is PERG relies mostly on the CRC8 checksum to immediately reduce for false positive hits.

Another possible argument against the scalability claim is that the probability of perfect-hashing a new pattern into the BFUs would be low. To determine this behavior, the dynamic updatability of PERG is tested experimentally using the daily signature update (Sept. 29 2008) from ClamAV.net as a test subject. After preprocessing through the compiler flow, 1033 rules were identified and broken down into 2,040 unique fragments.

The probability of successful insertion to the BFUs is tested follows. First, the BFUs were pre-generated with the original database from main.ndb and main.db. Then, two tests were run. In Test 1, each fragment is inserted one at a time, and removed before insertion of the next fragment so they do not interfere each other. For 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.

To insert the last few un-mapped fragments, the hash functions in PERG needs to be reprogrammable without recompiling the RTL. This can be done by making the initial hash constant H_0 a user-configurable value. Also, the shift amounts S_{Li} and S_{Ri} of the SAX hash can be made reprogrammable at each stage as well. However, making them fully programmable would require two full barrel shifters in each stage, which is probably too much area overhead. Instead, each shifter can select between two constant shift amounts, giving each stage four different configurations. Adding re-programmability to the hash functions is left as future work.

6.4.2 Scalability and Dynamic Updatability using Random Patterns

This part evaluates the theoretical scalability and dynamic updatability of the PERG architecture using randomly generated databases. In comparison with the previous part, this part focuses on evaluating the PERG architecture's sensitivity to database pattern changes (scalability) and ability to accept new (dynamic updatability). These aspects are explored in two separate tests.

In both scalability and dynamic updatability tests, four baseline BFU configurations are used and outlined in Table 6.9. Like PERG, the hash function used is also based on SAX. These baseline configurations are reconstructed at the start of each test trial to ensure independent results; each trial will test a parameter value at a configuration for 50 trials. The BFU in all baseline configurations have 10,000 hash table entries and have two hash functions; i.e., each BFU has a maximum theoretical capacity of 10,000 patterns. This number is chosen arbitrarily based on observations of BFUs size made while using the ClamAV database.

Table 6.9: Baseline configurations

Baseline Name	# of Bloomier filters	Length of each Bloomier filter (characters)
Config. 1	2	8, 16
Config. 2	4	8, 16, 24,32
Config. 3	8	8, 16, 24, 32, 40, 48, 56, 64
Config. 4	16	8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128

Both tests share the same random pattern generator, which can generate an arbitrary number of random byte strings. All generated patterns are unique and map perfectly with one of the predefined BFU lengths of the chosen baseline. There are two reasons not to include patterns of random lengths into each test. First, it is reasonable to assume that the general distribution of whatever targeted application database will not change dramatically on a daily basis (which uses the dynamic updatability). Second, the overlapping technique described in Section 4.3 would effectively eliminate the different lengths; introducing an unfitted pattern is no different from adding two patterns that fit perfectly with existing BFU lengths. Note this reasoning is based on three assumptions: 1) existing BFU lengths are no more than two times longer or shorter than its neighbors', 2) whatever new length introduced must be greater than the current shortest BFU length, and 3) common segments are stored in cache.

To limit the scope of exploration, two limits are set. First, once a baseline is constructed, the initial hash constant H_0 is the only variable to create new hash keys during dynamic update. The limitation also stresses the dependency among the BFUs in each configuration. Second, a hard limit of up to 50 rehashes is allowed for each pattern insertion or database reconstruction. Without these two limitations, results can be tuned quite favorably by exhausting each test iteration; it is unreasonable in a real application scenario to expect an exhaustive test for each update.

The first test determines how sensitive each baseline platform is to changes in the database patterns. Utilization of each BFU is set to be the same as PERG's default minimum threshold value at 90%, translating to 9,000 randomly generated patterns in each BFU. After construction, each BFU has 9,000 patterns mapped to it. Each test iteration overwrites a portion of the 9,000 patterns of all the BFUs in the configuration with randomly generated patterns. The same portion is overwritten in each trial so patterns outside of the portion are unchanged.

The results are shown in Table 6.10 to 6.13. As the number of portion changed has minor influence on the success of the BFU reconstruction. This finding suggests that other than storage limitation, having a highly utilized BFU will have negligible impact on the scalability of the system. It is worth pointing out that Config. 3 in Table 6.12 indeed do not have any setup failure in the trials conducted.

As the number of BFUs changes in the configuration, the number of rehashes needed for reconstruction as well as the number of failed reconstructions increased rapidly. The trend suggests the need have more flexibility in the SAX hash parameters, as suggested in Subsection 6.3.1.

Table 6.10: Impact of database change in Config. 1

Number of BFUs	2	2	2	2	2
Total number of patterns	180000	180000	180000	180000	180000
Utilization	90%	90%	90%	90%	90%
Change %	10%	25%	50%	75%	100%
Average number of rehashes	3.08	4.66	4.78	4.34	5.04
Number of setup failures (out of 50)	0	0	0	0	0

Table 6.11: Impact of database change in Config. 2

Number of BFUs	4	4	4	4	4
Total number of patterns	360000	360000	360000	360000	360000
Utilization	90%	90%	90%	90%	90%
Change %	10%	25%	50%	75%	100%
Average number of rehashes	20.62	18.44	18.56	20.08	21.5
Number of setup failures (out of 50)	7	9	6	8	6

Table 6.12: Impact of database change in Config. 3

Number of BFUs	8	8	8	8	8
Total number of patterns	720000	720000	720000	720000	720000
Utilization	90%	90%	90%	90%	90%
Change %	10%	25%	50%	75%	100%
Average number of rehashes	20.98	21	20.16	22.48	23.38
Number of setup failures (out of 50)	0	0	0	0	0

Table 6.13: Impact of database change in Config. 4

Number of BFUs	16	16	16	16	16
Total number of patterns	1440000	1440000	1440000	1440000	1440000
Utilization	90%	90%	90%	90%	90%
Change %	10%	25%	50%	75%	100%
Average number of rehashes	13.98	11.36	15	16.56	15.72
Number of setup failures (out of 50)	32	36	31	30	29

The second test evaluates how close each configuration can reach to its theoretical max capacity by inserting 100 new patterns (1% of the theoretical max of each BFU capacity) to a BFU at each iteration. A new BFU is chosen per iteration in a round-robin fashion. The number of new patterns inserted that the system can absorb until one of its BFU fails to construct is measured.

The results for the second test are shown in Table 6.14 to 6.17. When existing BFUs are already highly utilized, they actually have better probability of reaching the theoretical maximum capacity of the BFU. What is more interesting is that this trend continues at configuration with larger number of BFUs. A simple explanation of this behavior is that higher-utilized BFUs need fewer insertions to reach to its theoretical max. However, the important takeaway form this result is that a *good* (closeness to even distribution) set of hash parameters (shift values) at the start is crucial. Poorly utilized BFUs are more likely to have a bad set of hash parameters since the initial construction did not need to be stressed enough at all.

Table 6.14: Impact of utilization on insertion for Config. 1

Number of BFUs	2	2	2	2	2
Total number of patterns	10000	12000	14000	16000	18000
Utilization	50%	60%	70%	80%	90%
Average number of patterns inserted	9086	7510	5938	3600	1946
Average number of insertions until failure	181.72	150.2	118.76	72	38.92
% of theoretical max reached	95.43	97.55	99.69	98.00	99.73

Table 6.15: Impact of utilization on insertion for Config. 2

Number of BFUs	4	4	4	4	4
Total number of patterns	20000	24000	28000	32000	36000
Utilization	50%	60%	70%	80%	90%
Average number of patterns inserted	16464	13292	9508	6412	2716
Average number of insertions until failure	329.28	265.84	190.16	128.24	54.32
% of theoretical max reached	91.16	93.23	93.77	96.03	96.79

Table 6.16: Impact of utilization on insertion for Config. 3

Number of BFUs	8	8	8	8	8
Total number of patterns	40000	48000	56000	64000	72000
Utilization	50%	60%	70%	80%	90%
Average number of patterns inserted	23110	22620	13478	8340	2470
Average number of insertions until failure	462.2	452.4	269.56	166.8	49.4
% of theoretical max reached	78.8875	88.275	86.8475	90.425	93.0875

Table 6.17: Impact of Utilization on Insertion for Config. 4

Number of BFUs	16	16	16	16	16
Total number of patterns	80000	96000	112000	128000	144000
Utilization	50%	60%	70%	80%	90%
Average number of patterns inserted	37466	27774	17892	3892	48
Average number of insertions until failure	749.32	555.48	357.84	77.84	0.96
% of theoretical max reached	73.41625	77.35875	81.1825	82.4325	90.03

The results from both the first and second tests in this part all reemphasize the importance and effectiveness of the filter consolidation algorithm, which not only reduces the number of BFUs but also increases their utilization. In conclusion, the filter consolidation stage is not only critical in compressing the resource usage of the initial baseline but also has a positive impact on future scalability and dynamic updatability of the system.

Chapter 7. Conclusions

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.

Contributions of this thesis are summarized below:

- The novel use of Bloomier filters in a pattern-matching application to simplify exact matching
- Filter consolidation to reduce the number of Bloomier filters and improve density
- 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
- The support of wildcard operators using a lossy but storage-efficient and zero-false positive mechanism

In terms of memory density, PERG shows over 24x improvements over the best NIDS pattern-matching engine. While PERG lags somewhat in throughput, it still provides nearly 16x improvements over software-based virus scanning. With PERG, this thesis demonstrates the feasibility of hardware-based virus scanning. Most importantly, the amount of on-chip memory required can be found in current low-cost Spartan 6 and Cyclone III FPGAs.

PERG is the first hash-based pattern matching engine 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 comparably low to the original PERG design.

In terms of scalability and dynamic updatability, this thesis has shown that PERG can reach very close to the theoretical maximum capacity of m/k patterns. The filter consolidation algorithm introduces a controlled amount of slack for future updates. The Bloomier perfect hash functions generated by the pattern compiler work very well, allowing a large number of updates even when the filter unit is nearly full.

7.1 Future Work

While this thesis has described a pattern-matching engine hardware and its pattern compiler, several aspects are left as future work.

7.1.1 Hardware Interface

As a pattern-matching engine designed to scan for computer viruses, the exact hardware interface for PERG and how it would interact with the host system is never specified. Currently, PERG is assumed to operate as a co-processor over a PCI bus interface, while antivirus software would monitor and intercept I/O operations and direct the file stream to PERG. While bandwidth of the bus is unlikely to become the bottleneck, latency and communication overhead may turn out to be a significant factor to the usability of PERG.

Unlike previous pattern-matching engines designed for NIDS, PERG's application in computer virus protection goes beyond network equipment to every personal computer. It is natural to explore the possibility of integrating PERG onto the south bridge of a chipset much like what nVidia has done by integrating firewall onto their chipset. Most importantly, in such a scenario, PERG would need to be implemented as an ASIC to be integrated with a south bridge. Additional resources must be reserved to support future updates and hash reconfiguration.

7.1.2 Support for Interleaving Filestreams

In our test, the filestream is assumed to arrive one by one in order. In practice, file streams usually are interleaved and governed by the underlying operating system. To support multiple ongoing streams, additional registers would be required to maintain state information. Multiple Wildcard Tables and Circular State Buffers are required to allow fast switching between streams. When the number of ongoing streams exceeds the number of WT and CSB pairs available, WT and CSB contents must be backed up on to external memory temporarily and restored when appropriate.

In NIDS, designers assume TCP packets are reassembled by a TCP reassembly engine to detect malicious activities that would span over multiple packets. A similar mechanism may also be applied for file streams.

7.1.3 Update and Expansion Options

Virus vendors have witnessed rapid growth of viruses – it jumped 137% in 2007 alone. Generally, most of the new viruses are identified by file-integrity hash functions like MD5 checksum. In addition, older

virus signatures (e.g. DOS-based viruses) may be removed or replaced with more generic descriptions. As a result, PERG's high rule/resource density PERG architecture should be able to keep up with such growth.

With advancement in process technology, newer generations of PERG may be released according to the general product support life-cycle of antivirus software, which is about 2 years on average. However, depending on how PERG is integrated to the system, it may be impractical to expect users to upgrade their hardware (e.g. mounted on the motherboard). At the same time, partial protection would be just as inadequate as no protection at all.

One possible way for users to retain protection would be adding new PERG engines through the PCI-Express bus that could work in conjunction or independently with the existing PERG much like the PC graphic cards today. Alternatively, full protection can be retained with the same PERG at the cost of latency and performance. Since the BFU tables in PERG are memory-programmable, it is possible to swap the table content dynamically; in such a scenario, the number of virus signatures supported is no longer bounded by PERG's hardware. Swapping mechanism can be implemented at the cost of latency (buffering the input stream) , the cost of performance (multiple swaps for each input byte), or a combination of both. Since scanning with PERG is expected to be several magnitudes faster than pure software solution, this method should be sufficient within the life time of the host system.

7.1.4 Alternative Databases

In this thesis we were only able to map ClamAV database onto PERG. A database from a different antivirus vendor may exhibit different results. Unfortunately, other than ClamAV, antivirus databases generally are proprietary and unavailable to public in their raw forms.

Another worthy note is the Snort NIDS database. Our analysis of previous work on both CAM and FPGA-based pattern matching engines reveals that there is no existing solution which provides support for the entire Snort database.

To map Snort, however, the pattern compiler would require significant changes due to the vast difference in syntax. Compiler-specific parameters such as the Metadata encoding format would also likely need to be re-optimized.

7.1.5 Integration with Antivirus Software

PERG is not aimed to replace antivirus software but complement it as an accelerator. Since ClamAV is an open-source project, a port to support PERG can be implemented. Drivers and other low-level interfaces should be kept in separate layers to ensure future expandability to other antivirus software.

7.1.6 Eliminating Special Cases

In the thesis we mentioned that patterns with short fragments (<4 bytes) are to be handled by another pattern matching engine. This pattern-matching engine would be significantly smaller in resource usage than the main PERG due to the insignificant number of special cases. Such a pattern matching engine can be an FSM-based engine or a mini version of PERG whose Metadata Unit resides completely on-chip. Similarly, special cases of patterns with excess number of Or operators can also be solved by a small FSM-based engine working along with PERG. Or-expansion is expensive due to the number of output patterns resulting from each Or operator found in the input pattern. Finally, by dropping fragments intelligently and replacing them with wildcard operators, many special cases can be eliminated at the cost of higher false positive probability.

References

[1] AV-Comparative, "Anti-Virus Comparative Performance Test," http://av-comparatives.org/images/stories/test/performance/performance08a.pdf, Nov. 2008.

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

[3] N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," International Conference on Field-Programmable Technolog, 2007, pp. 121- 128.

[4] J. van Lunteren, "High-performance pattern-matching for intrusion detection," IEEE International Conference on Comp. Comm., 2006, pp. 1-13.

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

[6] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," IEEE Conference on Computer Communications, vol.4, 2004, pp. 2628-2639.

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

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

[9] G. Papadopoulos and D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," International Conference on Field Programmable Logic and Applications, 2005, pp. 39-44.

[10] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," International Conference on Field Programmable Logic and Applications, 2005, pp. 644-647.

[11] Snort, http://www.snort.org, accessed on August 2009.

[12] Clam AntiVirus, http://www.clamav.net, accessed on August 2009.

[13] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," Symposium on Discrete Algorithms, 2004, pp. 30-39.

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

[15] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," Communicatoins of the ACM, vol.18, 1975, pp. 333-340.

[16] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," Communications of the ACM, vol.20, 1977, pp. 762–772.

[17] Donald Knuth, James H. Morris, and Vaughan Pratt, "Fast Pattern Matching in Strings," SIAM Journal on Computing, vol.6, 1977, pp. 323–350.

[18] Cavium Networks, OCTEON Multi-Core Processor Family. http://www.caviumnetworks.com/OCTEON_MIPS64.html, accessed on August 2009.

[19] Lionic Corp., ePavis. http://www.lionic.com, accessed on August 2009.

[20] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, vol.13, 1970, pp. 422-426.

[21] M. V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," International Conference on Database Systems for Advanced Applications, World Scientific Press, 1997, pp. 215-224.

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

[23] X. Zhou, B. Xu, Y. Qi, and J. Li, "MRSI: A fast pattern matching algorithm for anti-virus applications," International Conference on Networking, 2008, pp. 256-261.

[24] 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.

[25] Intel, Corp. Intel QuickAssit, http://developer.intel.com/technology/platforms/quickassist/index.htm, accessed on August 2009.

[26] PCRE - Perl Compatible Regular Expressions, http://www.pcre.org, accessed on August 2009.

[27] N. Yamagaki, R. Sidhu, S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," International Conference on Field Programmable Logic and Applications, 2008, pp. 131-136.

[28] Y. Yang, W. Jiang, V. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2008, pp. 30-39.

[29] J. Ho, G. Lemieux, "PERG: A Scalable Pattern-matching Accelerator," CMC Microsystems and Nanoelectronics Research Conference, Ottawa, pp. 29-32, October 2008.

[30] J. Ho, G. Lemieux, "PERG: A Scalable FPGA-based Pattern-matching Engine with Consolidated Bloomier Filters," IEEE International Conference on Field-Programmable Technology, Taipei, Taiwan, December 2008, pp. 73-80.

[31] J. Ho, G. Lemieux, "PERG-Rx: A Hardware Pattern-matching Engine Supporting Limited Regular Expressions," ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, pp. 257-260, February 2009.