HARDWARE ALGORITHMS FOR HIGH-SPEED PACKET PROCESSING

By

Eric Norige

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science — Doctor of Philosophy

2017

ABSTRACT

HARDWARE ALGORITHMS FOR HIGH-SPEED PACKET PROCESSING

By

Eric Norige

The networking industry is facing enormous challenges of scaling devices to support the exponential growth of internet traffic as well as increasing number of features being implemented inside the network. Algorithmic hardware improvements to networking components have largely been neglected due to the ease of leveraging increased clock frequency and compute power and the risks of implementing complex hardware designs. As clock frequency slows its growth, algorithmic solutions become important to fill the gap between current generation capability and next generation requirements. This paper presents algorithmic solutions to networking problems in three domains: Deep Packet Inspection (DPI), firewall ruleset compression and non-cryptographic hashing. The improvements in DPI are two-pronged: first in the area of application-level protocol field extraction, which allows security devices to precisely identify packet fields for targeted validity checks. By using counting automata, we achieve precise parsing of non-regular protocols with small, constant per-flow memory requirements, extracting at rates of up to 30 Gbps on real traffic in software while using only 112 bytes of state per flow. The second DPI improvement is on the long standing regular expression matching problem, where we complete the HFA solution to the DFA state explosion problem with efficient construction algorithms and optimized memory layout for hardware or software implementation. These methods construct automata too complex to be constructed by previous methods in seconds, while being capable of 29 Gbps throughput with an ASIC implementation. Firewall ruleset compression enables more firewall entries to be stored in a fixed capacity pattern matching engine, and can also be used to reorganize a

firewall specification for higher performance software matching. A novel recursive structure called TUF is given to unify the best known solutions to this problem and suggest future avenues of attack. These algorithms, with little tuning, achieve a 13.7% improvement in compression on large, real-life classifiers, and can achieve the same results as existing algorithms while running 20 times faster. Finally, non-cryptographic hash functions can be used for anything from hash tables to track network flows to packet sampling for traffic characterization. We give a novel approach to generating hardware hash functions in between the extremes of expensive cryptographic hash functions and low quality linear hash functions. To evaluate these mid-range hash functions properly, we develop new evaluation methods to better distinguish non-cryptographic hash function quality. The hash functions described in this paper achieve low-latency, wide hashing with good avalanche and universality properties at a much lower cost than existing solutions.

To my advisor, who didn't give up on me despite all the reasons to.

ACKNOWLEDGMENTS

I would like to thank Chad Meiners and Sailesh Kumar for providing strong shoulders to stand upon.

This work is partially supported by the National Science Foundation under Grant Numbers CNS-0916044, CNS-0845513, CNS-1017588, CCF-1347953, CNS-1318563 and CNS-1017598, and the National Natural Science Foundation of China under Grant Numbers 61272546 and 61370226, and by a research gift from Cisco Systems, Inc..

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The packet processing domain offers algorithm authors a unique combination of challenges and opportunities. The phrase "packet processing domain" means the wide range of tasks done in networking components' data planes. This includes topics as varied as buffer management strategies, to manage buffering packets between reception and transmission, and IP lookup, to determine how to forward the packet. It also includes security topics such as deep packet inspection and statistics gathering topics such as packet sampling and counter implementations. Until coming into the networking field, I had no idea how much work has been put into the implementation of simply counting how many packets/bytes were transferred by a router. It turns out that there's a lot of improvement that can be done on even this simple task.

One reason these tasks need improvement is the incredible demand for Internet bandwidth. Internet use is still growing at astronomical rates, and the infrastructure needed to support this growth keeps being pushed to its limits. Electrical and optical engineers keep improving the "pipes" of the internet, giving the ability to send more and more data through wires and fiber optic strands. Historically, the data path in the boxes that connected these "pipes" had little difficulty keeping up with the data transfer rates. Further, as semiconductor technologies scaled up in frequency, the data path logic performance was automatically upgraded, so less attention was paid to its implementation. Semiconductor technology is

not scaling in frequency anywhere near as fast as it has in the past, so this easy source of performance has been lost, making algorithmic improvements valuable to keep pace with demand.

Further, new security and monitoring requirements are being added to networks, increasing the burden on the datapath to do complex processing on packets in flight. Firewall rulesets and IP routing tables are growing in size even as the time available to process each packet decreases with higher line rates. Deep packet inspection is moving up the complexity ladder to being able to deeply understand application protocols with a combination of string matching, regular expression matching and protocol parsing. Monitoring of networks is becoming more involved, with network analytics becoming important to managing even mid-size networks, with predictive systems alerting administrators to problems before failures occur. For high security networks, network situational analysis and network behavior anomaly detection tools can be deployed to monitor for and identify a wide range of security problems from information exfiltration to bot-nets for further investigation.

The opportunity is that the networking industry is able to make use of custom hardware solutions. Normally, custom hardware is well out of the reach of most algorithm designers for three reasons: lifetime, wide scope and experience. An algorithm that will have to be frequently changed, such as Google PageRank, is not a good candidate for hardware implementation, as the hardware will take much time to develop and will be out of date by the time it can be used. In the networking field, the task of a router hasn't fundamentally changed in decades. Even the new requirements being added to network data-paths can be built on top of hardware primitives for orders of magnitude performance improvements. The cost of custom hardware is high, but the large number of installed units allows this cost to be amortized across hundreds of thousands to millions of devices. Further, the cost

of hardware is a small part of the total cost of ownership of a network, allowing expensive, high-performance hardware to be in high demand. Finally, the networking industry has a long history of developing custom hardware as part of the "pipes" portion of networking. This has led to the industry having experience implementing custom hardware solutions, making it possible for a custom-hardware solution to their problems to be integrated in new products.

This paper develops algorithmic solutions to four separate problems. It first tackles deep packet inspection at two levels: protocol parsing and regular expression matching. An efficient protocol parsing engine is developed in Chapter 2 to extract application-layer fields from raw flows. Chapter 3 follows with novel methods of extended automaton construction for regular expression matching. Chapter 4 develops a framework for building firewall-style ruleset compression. It ends with Chapter 5 on ASIC-optimized non-cryptographic hashing, useful in hash tables and measurement sampling.

The results described in Chapter 2 have been published in JSAC Vol 32 No. 10. The results described in Chapter 3 chapter have been published in ICNP 2013. The results described in Chapter 4 have been published in ANCS 2013 and are in preprint for TON 2015. The results described in Chapter 5 have been published in ANCS 2011.

Chapter 2

Protocol Parsing

2.1 Introduction

2.1.1 Motivation

Content inspection is the core of a variety of network security and management devices and services such as Network Intrusion Detection/Prevention Systems (NIDS/NIPS), load balancing, and traffic shaping. Currently, content inspection is typically achieved by Deep Packet Inspection (DPI), where packet payload is simply treated as a string of bytes and is matched against content policies specified as a set of regular expressions. However, such regular expression based content policies, which do not take content semantics into consideration, can no longer meet the increasing complexity of network security and management.

Semantic-aware content policies, which are specified over application protocols fields, *i.e.*, Layer 7 (L7) fields, have been increasingly used in network security and management devices and services. For example, a load balancing device would like to extract method names and parameter fields from flows (carrying SOAP [1] and XML-RPC [2] traffic for example) to determine the best way to route traffic. A network web traffic analysis tool would extract message length and content type fields from HTTP header fields to gain information about current web traffic patterns. Another example application that demonstrates the need for and the power of semantic-aware content policies is vulnerability-based signature checking for

detecting polymorphic worms in NIDS/NIPS. Traditionally, NIDS/NIPS use exploit-based signatures, which are typically specified as regular expressions. The major weakness of an exploit-based signature is that it only recognizes a specific implementation of an exploit. For example, the following exploit-based signature for Code Red,

```
urlcontent:‘‘ida?NNNNNNNNNNN...’’,
```

where the long string of Ns is used to trigger a buffer overflow vulnerability, fails to detect Code Red variant II where each N is replaced by an X. Given the limitations of exploit-based signatures and the rapidly increasing number of polymorphic worms, vulnerability-based signatures have emerged as effective tools for detecting zero-day polymorphic worms [3–6]. As a vulnerability-based signature is independent of any specific implementation, it is hard to evade even for polymorphic worms. Here is an example vulnerability-based signature for Code Red worms as specified in Shield [3]:

```
c = MATCH_STR_LEN(>> P_Get_Request.URI,‘‘id[aq] \ ?(.*)$’’,limit);
IF (c > limit) # Exploit!
```

The key feature of this signature is its extraction of the string beginning with “ida?” or “idq?”. By extracting this string and then measuring its length, this signature is able to detect any variant of the Code Red polymorphic worm.

We call the process of inspecting flow content based on semantic-aware content policies Deep Flow Inspection (DFI). DFI is the foundation and enabler of a wide variety of current and future network security and management services such as vulnerability-based malware filtering, application-aware load balancing, network measurement, and content-aware caching and routing. The core technology of DFI is L7 field extraction, the process of extracting the values of desired application (Layer 7) protocol fields.

2.1.2 Problem Statement

In this paper, we aim to develop a high-speed online L7 field extraction framework, which will serve as the core of next generation semantic aware network security and management devices. Such a framework needs to satisfy the following six requirements. First, it needs to parse at high-speed. Second, it needs to use a small amount of memory per flow so that the framework can run in SRAM; otherwise, it will be running in DRAM, which is hundreds of times slower than SRAM.

Third, such a framework needs to be *automated*; that is, it should have a tool that takes as input an extraction specification and automatically generates an extractor. Hand-coded extractors are not acceptable because each time when application protocols or fields change, they need to be manually written or modified, which is slow and error prone.

Fourth, because of time and space constraints, any such framework must perform *selective parsing*, *i.e.*, parsing only relevant protocol fields that are needed for extracting the specified fields, instead of *full parsing*, *i.e.*, parsing the values of every field. Full protocol parsing is too slow and is unnecessary for many applications such as vulnerability-based signature checking because many protocol fields may not be referenced in given vulnerability signatures [5]. Selective parsing that skips irrelevant data leads to faster parsing with less memory. To avoid full protocol parsing and improve parsing efficiency, we want to dramatically simplify parsing grammars based on extraction specification. *We are not aware of existing compiler theory that addresses this issue.*

Fifth, again because of time and space constraints, any such framework must support *approximate protocol parsing* where the actual parser does not parse the input exactly as specified by the grammar. While *precise parsing*, where a parser parses input exactly as

specified by the grammar, is desirable and possibly feasible for end hosts, it is not practical for high-speed network based security and management devices. First, precise parsing for recursive grammars is time consuming and memory intensive; therefore, it is not suitable for NIDS/NIPS due to performance demand and resource constraints. Second, precise parsers are vulnerable to Denial of Service (DoS) attacks as attackers can easily craft an arbitrarily deep recursive structure in their messages and exhaust the memory used by the parser. However, it is technically challenging to perform approximate protocol parsing. Again, since most existing compiler applications run on end hosts, *we are not aware of existing compiler theory that addresses this issue.*

At last, any such framework must be able to parse application protocols with field length descriptors, which are fields that specify the length of another field. An example field length descriptor is the HTTP Content-Length header field, which gives the length of the HTTP body. Field length descriptors cannot be described with CFG [7,8], which means that CFGs are not expressive enough for such framework.

2.1.3 Limitations of Prior Art

To the best of our knowledge, there is no existing online L7 field extraction framework that is both automated and selective. The only selective protocol parsing solution (*i.e.*, [5]) is hand-coded and all automated protocol parsing solutions (*i.e.*, [7–16]) perform full parsing. Furthermore, *prior work on approximate protocol parsing is too inaccurate.* To reduce memory usage, Moscola *et al.*proposed ignoring the recursive structures in a grammar [13–15]. This crude approximate parsing is not sufficient because recursive structures often must be partially parsed in order to extract the desired information; for example, the second field in a function call.

2.1.4 Proposed Approach

To address the above limitations of prior work on application protocol parsing and extraction, in this paper, we propose FlowSifter, an L7 field extraction framework that is *automated*, *selective*, and *approximate*. The architecture of FlowSifter is illustrated in Figure 2.1. The input to FlowSifter is an extraction specification that specifies the relevant protocol fields that we want to extract values. FlowSifter adopts a new grammar model called *Counting Regular Grammars (CRGs)* for describing both the grammatical structures of messages carried by application protocols and the message fields that we want FlowSifter to extract. To our best knowledge, CRG is the first protocol grammar model that facilitates the automatic optimization of extraction grammars based on their corresponding protocol grammars. The extraction specification can be a partial specification that uses a corresponding complete protocol grammar from FlowSifter's built-in library of protocol grammars to complete its specification. Changing the field extractor only requires changing the extraction specification, which makes FlowSifter highly flexible and configurable. FlowSifter has three modules: *a grammar optimizer*, *an automata generator*, and *a field extractor*. The grammar optimizer module takes the extraction specification and the corresponding protocol grammar as its input and outputs an optimized extraction grammar, by which FlowSifter selectively parses only relevant fields bypassing irrelevant fields. The automata generator module takes the optimized extraction grammar as its input and outputs the corresponding *counting automaton*. The field extractor module applies the counting automaton to extract relevant fields from flows.

FlowSifter achieves low memory consumption by stackless approximate parsing. Processing millions of concurrent flows makes stacks an unaffordable luxury and a vulnerability for

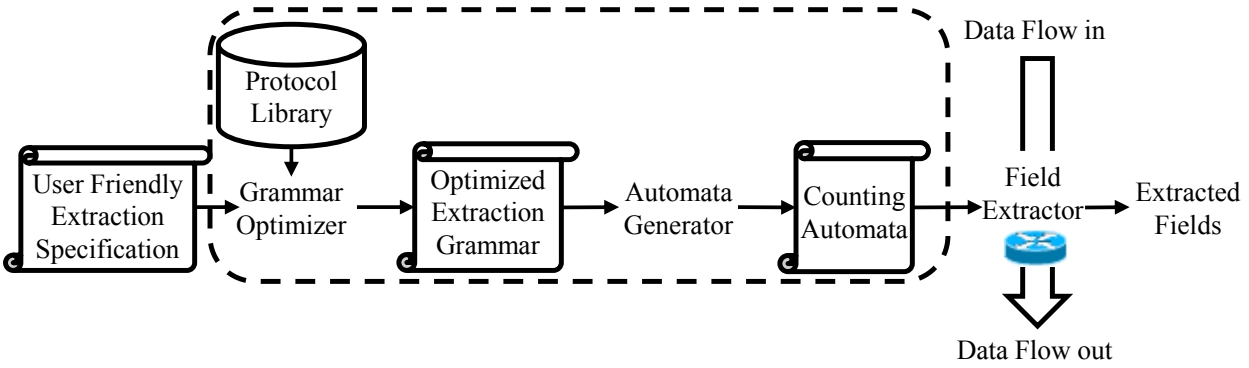


Figure 2.1: FlowSifter architecture

DoS attacks. For some application protocols such as XML-RPC [2], an attacker may craft its flow so that the stack used by the parser go infinitely deep until memory is exhausted. To achieve a controlled tradeoff between memory usage and parsing accuracy, we use a formal automata theory model, counting automata (CA), to support approximate protocol parsing. CAs facilitate stackless parsing by using counters to maintain parsing state information. Controlling memory size allocated to counters gives us a simple way of balancing between memory usage and parsing accuracy. With this solid theoretical underpinning, FlowSifter achieves approximate protocol parsing with well-defined error bounds.

FlowSifter can be implemented in both software and hardware. For hardware implementation, FlowSifter can be implemented in both ASIC and Ternary Content Addressable Memory (TCAM). For ASIC implementation, FlowSifter uses a small, fast parsing engine to traverse the memory image of the CA along with the flow bytes, allowing real-time processing of a vast number of flows with easy modification of the protocol to be parsed. For TCAM implementation, FlowSifter encodes the CA transition tables into a TCAM table, thus allowing FlowSifter to extract relevant information from a flow in linear time over the number of bytes in a flow. FlowSifter uses optimization techniques to minimize the number

of TCAM bits needed to encode the transition tables. Note that TCAM has already been installed on many networking devices such as routers and firewalls.

2.1.5 Key Contributions

In this paper, we make the following key contributions:

1. We propose the first L7 field extraction framework that is automated, selective, and approximate.

2. We propose for the first time to use Counting Context-Free Grammar and Counting Automata to support approximate protocol parsing. By controlling memory size allocated to counters, we can easily tradeoff between memory usage and parsing accuracy.

3. We propose efficient algorithms for optimizing extraction grammars.

4. We propose an algorithm for the automatic generation of stackless parsers from non-regular grammars.

2.2 Related Work

Prior work on application protocol parsing falls into three categories: (1) hand-coded, full, and precise parsing, (2) hand-coded, selective, and precise parsing, and (3) automated, full, and precise parsing,.

Hand-coded, full, and precise parsing: Although application protocol parsers are still predominantly hand coded [7], hand-coded protocol parsing has two major weaknesses in comparison with automated protocol parsing. First, hand-coded protocol parsers are hard to

reuse as they are tightly coupled with specific systems and deeply embedded into their working environment [7]. For example, Wireshark has a large collection of protocol parsers, but none can be easily reused outside of Wireshark. Second, such parsers tend to be error-prone and lack robustness [7]. For example, severe vulnerabilities have been discovered in several hand-coded protocol parsers [17–22]. Writing an efficient and robust parser is a surprisingly difficult and error-prone process because of the many protocol specific issues (such as handling concurrent flows) and the increasing complexity of modern protocols [7]. For example, the NetWare Core Protocol used for remote file access has about 400 request types, each with its own syntax [7].

Hand-coded, selective, and precise parsing: Full protocol parsing is not necessary for many applications. For example, Schear *et al.*observed that full protocol parsing is not necessary for detecting vulnerability-based signatures because many protocol fields are not referenced in vulnerability signatures [5]. Based on such observations, Schear *et al.*proposed selective protocol parsing [5], which is three times faster than binpac. However, the protocol parsers in [5] are hand-coded and henceforth suffer from the weaknesses of hand-coded protocol parsing.

Automated, full, and precise parsing: Recognizing the increasing demand for application protocol parsers, the difficulty in developing efficient and robust protocol parsers, and the many defects of home-brew parsers, three application protocol parser generators have been proposed: binpac [7], GAPA [8], and UltraPAC [16]. Most network protocols are designed to be easily parsed by hand, but this often means their formal definitions turn out complex in terms of standard parsing representations. Pang *et al.*, motivated by the fact that the programming language community has benefited from higher levels of abstraction for many years using parser generation tools such as yacc [23] and ANTLR [24], developed the pro-

tocol parser generator BinPAC. GAPA, developed by Borisov *et al.*, focuses on providing a protocol specification that guarantees the generated parser to be type-safe and free of infinite loops. The similarity between BinPAC and GAPA is that they both use recursive grammars and embedded code to generate context sensitive protocol parsers. The difference between BinPAC and GAPA is that BinPAC favors parsing efficiency and GAPA favors parsing safety. BinPAC uses C++ for users to specify code blocks and compile the entire parser into C++ whereas GAPA uses a restricted and memory-safe interpreted language that can be proven free of infinite loops. UltraPAC improves on BinPAC by replacing BinPAC's tree parser with a stream parser implemented using a state machine to avoid constructing the tree representation of a flow. UltraPAC inherits from BinPAC a low-level protocol field extraction language that allows additional grammar expressiveness using embedded C++ code. This makes optimizing an extraction specification extremely difficult, if not impossible. In contrast, FlowSifter uses high-level CA grammars without any inline code, which facilitates the automated optimization of protocol field extraction specifications. When parsing HTTP, for example, BinPAC and UltraPAC need inline C++ code to detect and extract the Content-Length field's value whereas FlowSifter's grammar can represent this operation directly. In addition, FlowSifter can automatically regularize non-regular grammars to produce a stackless approximate parser whereas an UltraPAC parser for the same extraction specification must be converted manually to a stackless form using C++ to embed the approximation.

2.3 Protocol and Extraction Specifications

FlowSifter produces an L7 field extractor from two inputs: a protocol specification, and an extraction specification. Both specifications are specified using a new grammar model called

Counting Context Free Grammar (CCFG), which augments rules for context-free grammars with counters, guards, and actions. These augmentations increase grammar expressiveness, but still allows the grammars to be automatically simplified and optimized. In this section, we first formally define CCFG, and then explain how to write protocol specification and extraction specification using CCFG.

2.3.1 Counting Context Free Grammar

Formally, a counting context-free grammar is a five-tuple $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ where $\mathbb{N}, \Sigma, \mathbb{C}$, and $\mathbb{R}$ are finite sets of *nonterminals*, *terminals*, *counters*, and *production rules*, respectively, and S is the *start nonterminal*. The terminal symbols are those that can be seen in strings to be parsed. For L7 field extraction, this is usually a single octet. A *counter* is a variable with an integer value, which is initialized to zero. The counters can be used to store parsing information. For example, in parsing an HTTP flow, a counter can be used to store the value of the "Content-Length" field. Counters also provide a mechanism for eliminating parsing stacks.

A production rule is written as $\langle \text{guard} \rangle : \langle \text{nonterminal} \rangle \rightarrow \langle \text{body} \rangle$. The *guard* is a conjunction of unary predicates over the counters in $\mathbb{C}$, *i.e.*, expressions of a single counter that return `true` or `false`. An example guard is $(c_1 > 2; c_2 > 2)$, which checks counters c_1 and c_2, and evaluates to `true` if both are greater than 2. If a counter is not included in a guard, then its value does not affect the evaluation of the guard. Guards are used to guide the parsing of future bytes based on the parsing history of past bytes. For example, in parsing an HTTP flow, the value of the "Content-Length" field determines the number of bytes that will be included in the message body. This value can be stored in a counter. As bytes are processed,

the associated counter is decremented. A guard that checks if the counter is 0 would detect the end of the message body and allow a different action to be taken.

The *nonterminal* following the guard is called the *head* of the rule. Following it, the *body* is an ordered sequence of terminals and nonterminals, any of which can have associated actions. An empty body is written ϵ. An *action* is a set of unary update expressions, each updating the value of one counter, and is associated with a specific terminal or nonterminal in a rule. The action is executed after parsing the associated terminal or nonterminal. An example action in CCFG is $(c_1 := c_1 * 2; c_2 := c_2 + 1)$. If a counter is not included in an action, then the value of that counter is unchanged. An action may be empty, *i.e.*, updates no counter. Actions in CCFG are used to write "history" information into counters, such as the message body size.

To make the parsing process deterministic, CCFGs require leftmost derivations; that is, for any body that has multiple nonterminals, only the leftmost nonterminal that can be expanded is expanded. Thus, at any time, production rules can be applied in only one position. We use leftmost derivations rather than rightmost derivations so that updates are applied to counters in the order that the corresponding data appears in the data flow.

2.3.2 Protocol Specification in CCFG

An application protocol specification precisely specifies the protocol being parsed. We use CCFG to specify application protocols. For example, consider the Varstring language consisting of strings with two fields separated by a space: a length field, B, and a data field, V, where the binary encoded value of B specifies the length of V. This language cannot be specified as a Context Free Grammar (CFG), it can be easily specified in CCFG as shown in Figure 2.2(a). The CCFG specification of another language called Dyck consisting of strings

with balanced parentheses '[' and ']', is shown in Figure 2.2(b). We adopt the convention that the head of the first rule, such as S in the Varstring and Dyck grammars, is the start nonterminal.

1		$\mathrm{S} \rightarrow \mathrm{B}\,\mathrm{V}$
2		$\mathrm{B} \rightarrow$ '0' $(c := c * 2)\,\mathrm{B}$
3		$\mathrm{B} \rightarrow$ '1' $(c := c * 2 + 1)\,\mathrm{B}$
4		$\mathrm{B} \rightarrow$ '␣'
5	$(c > 0)$	$\mathrm{V} \rightarrow \Sigma\,(c := c - 1)\,\mathrm{V}$
6	$(c = 0)$	$\mathrm{V} \rightarrow \epsilon$

examples: "1 a", "10 ba", "101 xyzab"

(a) Varstring Γ_v

1	$\mathrm{S} \rightarrow \epsilon$
2	$\mathrm{S} \rightarrow \mathrm{I}\,\mathrm{S}$
3	$\mathrm{I} \rightarrow$ '[' S ']'

"[[]]", "[][[][]]"

(b) Dyck Γ_d

Figure 2.2: Two protocol specifications in CCFG

We now explain the six production rules in Varstring. The first rule S→BV means that a message has two fields, the length field represented by nonterminal B and a variable-length data field represented by nonterminal V. The second rule B→'0' $(c := c * 2)\,\mathrm{B}$ means that if we encounter character '0' when parsing the length field, we double the value of counter c. Similarly, the third rule B→'1' $(c := c * 2 + 1)\,\mathrm{B}$ means that if we encounter '1' when parsing the length field, we double the value of counter c first and then increase its value by 1. The fourth rule B→'␣' means that the parsing of the length field terminates when we see a space. These three rules fully specify how to parse the length field and store the length in c. For example, after parsing a length field with "10", the value of the counter c will be 2 $(= ((0 * 2) + 1) * 2)$. Note that here "10" is the binary representation of value 2. The fifth rule $(c > 0)$:V→ $\Sigma\,(c := c - 1)\,\mathrm{V}$ means that when parsing the data field, we decrement the counter c by one each time a character is parsed. The guard allows use of this rule as long as $c > 0$. The sixth rule $(c = 0)$:V→ ϵ means that when $c = 0$, the parsing of the variable-length field is terminated.

We now demonstrate how the Varstring grammar can produce the string "10 ba". Each row of the table in Figure 2.3 is a step in the derivation. The c column shows the value of the counter c at each step. The number in parentheses is the rule from Figure 2.2(a) that is applied to get to the next derivation. Starting with the Varstring's start symbol, S, we derive the target string by replacing the leftmost nonterminal with the body of one of its production rules. When applying rule 5, the symbol Σ is shorthand for any character, so it can produce 'a' or 'b' or any other character.

Derivation	c	Rule #	Note
S	0	(1)	Decompose into len and body
B V	0	(3)	Produce '1', $c = 0 * 2 + 1$
1 B V	1	(2)	Produce '0', $c = 1 * 2$
10 B V	2	(4)	Eliminate B, c is the length of body
10␣V	2	(5)	Produce 'b', decrement c
10␣b V	1	(5)	Produce 'a', decrement c
10␣ba V	0	(6)	$c = 0$, so eliminate V
10␣ba	0		No nonterminals left, done

Figure 2.3: Derivation of "10 ba" from the Varstring grammar in Figure 2.2(a)

2.3.3 Extraction Specification in CCFG

An extraction specification is a CCFG $\Gamma_x = (\mathbb{N}_x, \Sigma, \mathbb{C}_x, \mathbb{R}_x, S_x)$, which may not be complete. It can refer to nonterminals specified in the corresponding protocol specification denoted $\Gamma_p = (\mathbb{N}_p, \Sigma, \mathbb{C}_p, \mathbb{R}_p, S_p)$. However, Γ_x cannot modify Γ_p and cannot add new derivations for nonterminals defined in Γ_p. This ensures that we can approximate Γ_p without changing the semantics of Γ_x.

The purpose of FlowSifter is to call application processing functions on extracted field values. Based on the extracted field values, these application processing functions will take application specific actions such as stopping the flow for security purposes or routing the

flow to a particular server for load balancing purposes. Calls to these functions appear in the actions of a rule. Application processing functions can also return a value back into the extractor to affect the parsing of the rest of the flow. Since the application processing functions are part of the layer above FlowSifter, their specification is beyond the scope of this paper. Furthermore, we define a shorthand for calling an application processing function f on a piece of the grammar $f\{\langle body \rangle\}$, where $\langle body \rangle$ is a rule body that makes up the field to be extracted.

We next show two extraction specifications that are partial CCFGs, using the function-calling shorthand. The first, Γ_{xv} in Figure 2.4(a), specifies the extraction of the variable-length field V for the Varstring CCFG in Figure 2.2(a). This field is passed to an application processing function `vstr`. For example, given input stream "`101 Hello`", the field "`Hello`" will be extracted. This example illustrates several features. First, it shows how FlowSifter can handle variable-length field extractions. Second, it shows how the user can leverage the protocol library to simplify writing the extraction specification. The second extraction specification, Γ_{xd} in Figure 2.4(b), is associated with the Dyck CCFG in Figure 2.2(b) and specifies the extraction of the contents of the first pair of square parentheses; this field is passed to an application processing function named `param`. For example, given the input stream `[[[]][[]]]`, the `[[]]` will be extracted. This example illustrates how FlowSifter can extract specific fields within a recursive protocol by referring to the protocol grammar.

1 | X → B `vstr`{V}

(a) Varstring Γ_{xv}

1 | X → '[' `param`{S} ']' S

(b) Dyck Γ_{xd}

Figure 2.4: Two extraction CCFGs Γ_{xv} and Γ_{xd}

2.4 Grammar Optimization

In this section, we introduce techniques to optimize the input protocol specification and extraction specification.

2.4.1 Counting Regular Grammar

Just as parsing with non-regular CFGs is expensive, parsing with non-regular CCFGs is also expensive as the whole derivation must be tracked with a stack. To resolve this, FlowSifter converts input CCFGs to Counting Regular Grammars (CRGs), which can be parsed without a stack, just like parsing regular grammars. A CRG is a CCFG where each production rule is regular. A production rule is *regular* if and only if it is in one of the following two forms:

$$\langle \text{guard} \rangle : X \quad \rightarrow \quad \alpha[\langle \text{action} \rangle]Y \tag{2.1}$$

$$\langle \text{guard} \rangle : X \quad \rightarrow \quad \alpha[\langle \text{action} \rangle] \tag{2.2}$$

where X and Y are nonterminals and α is a terminal. CRG rules that fit equation 2.1 are the *nonterminating* rules whereas those that fit equation 2.2 are the *terminating* rules as derivations end when they are applied.

For a CCFG $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ and a nonterminal $X \in \mathbb{N}$, we use $\Gamma(X)$ to denote the CCFG subgrammar $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, X)$ with the nonterminals that are unreachable from X being removed. For a CCFG $\Gamma = (\mathbb{N}, \Sigma, \mathbb{C}, \mathbb{R}, S)$ and a nonterminal $X \in \mathbb{N}$, X is *regular* if and only if $\Gamma(X)$ is equivalent to some CRG.

Given the extraction CCFG Γ_x and L7 grammar Γ_p as inputs, FlowSifter first generates a complete extraction CRG $\Gamma_f = (\mathbb{N}_x \cup \mathbb{N}_p, \Sigma, \mathbb{C}_x \cup \mathbb{C}_p, \mathbb{R}_x \cup \mathbb{R}_p, S_x)$. Second, it prunes any

unreachable nonterminals from S_x and their corresponding production rules. Third, it partitions the nonterminals in $\mathbb{N}_p$ into those we can guarantee to be normal and those we cannot. Fourth, for each nonterminal X that are guaranteed to be normal, FlowSifter regularizes X; for the remaining nonterminals $X \in \mathbb{N}_p$, FlowSifter uses counting approximation to produce a CRG that approximates $\Gamma_f(X)$. If FlowSifter is unable to regularize any nonterminal, it reports that the extraction specification Γ_x needs to be modified and provides appropriate debugging information. Last, FlowSifter eliminates idle rules to optimize the CRG. Next, we explain in detail how to identify nonterminals that are guaranteed to be normal, how to regularize a normal terminal, how to perform counting approximation, and how to eliminate idle rules.

2.4.2 Normal Nonterminal Identification

Determining if a CFG describes a regular language is undecidable. Thus, we cannot precisely identify normal nonterminals. FlowSifter identifies nonterminals in $\mathbb{N}_p$ that are guaranteed to be normal using the following sufficient but not necessary condition. A nonterminal $X \in \mathbb{N}_p$ is guaranteed to be normal if it satisfies one of the following two conditions:

1. $\Gamma_f(X)$ has only regular rules.

2. For the body of any rule with head X, X only appears last in the body and for every nonterminal Y that is reachable from X, Y is normal and X is not reachable from Y.

Although we may misidentify a normal nonterminal as not normal, fortunately, as we will see in Section 2.4.4, the cost of such a mistake is relatively low; it is only one counter in memory and some unnecessary predicate checks.

2.4.3 Normal Nonterminal Regularization

In this step, FlowSifter replaces each identified normal nonterminal's production rule with a collection of equivalent regular rules. Consider an arbitrary non-regular rule

$$\langle guard \rangle : X \to \langle body \rangle.$$

We first express the body as $Y_1 \ \cdots \ Y_n$ where $Y_i, 1 \leq i \leq n$, is either a terminal or a nonterminal (possibly with an action). Because this is a non-regular rule, either Y_1 is a nonterminal or $n > 2$ (or both). We handle the cases as follows.

- If Y_1 is a non-normal nonterminal, Γ_x was incorrectly written and needs to be reformulated.
- If Y_1 is a normal nonterminal, we define CRG $\Gamma' = (\mathbb{N}', \Sigma, \mathbb{C}', \mathbb{R}', \mathbb{Y}_{\not\Vdash}')$ to be $\Gamma_f(Y_1)$ where the nonterminals have been given unique names. We use Γ' to update the rule set as follows. First, we replace the rule $\langle guard \rangle : X \to \langle body \rangle$ with $\langle guard \rangle : X \to \mathbb{Y}_{\not\Vdash}'$. Next, for each terminating rule $r \in \mathbb{R}'$, we create a new rule r' where we append $Y_2 \ \cdots \ Y_n$ to the body of r and add r' to the rule set; for each nonterminating rule $r \in \mathbb{R}'$, we add r to the rule set.
- If Y_1 is a terminal and $n > 2$, the rule is decomposed into two rules: $\langle guard \rangle : X \to Y_1 \ X'$ and $X' \to Y_2 \ \cdots \ Y_n$ where X' is a new nonterminal.

The above regularization process is repeatedly applied until there are no non-regular rules.

For example, consider the Varstring CCFG Γ with non-regular rule S→BV. As both $\Gamma(B)$ and $\Gamma(V)$ are CRGs, so S is a normal non-terminal. Decomposition regularizes $\Gamma(S)$ by replacing S→BV by S→ B′ and B→␣ by B′ →␣ V. We also add copies of all other rules where

we use B' in place of B. Figure 2.5 illustrates the resulting rule set excluding unreachable rules. For example, the nonterminal B is no longer referenced by any rule in the new grammar. For efficiency, we remove unreferenced nonterminals and their rules after each application of regularization.

1		$\mathrm{S} \to \mathrm{B}'$
2		$\mathrm{B}' \to 0\,(c := c * 2)\,\mathrm{B}'$
3		$\mathrm{B}' \to 1\,(c := 1 + c * 2)\,\mathrm{B}'$
4		$\mathrm{B}' \to \sqcup\,\mathrm{V}$
5	$(c = 0)$	$\mathrm{V} \to \epsilon$
6	$(c > 0)$	$\mathrm{V} \to \Sigma\,(c := c - 1)\,\mathrm{V}$

Figure 2.5: Varstring after decomposition of rule S→BV.

Theorem 2.4.1

Given a normal nonterminal X in grammar Γ, applying regularization to any rule $\langle guard \rangle : X \to Y_1 \ \cdots \ Y_n$ in Γ produces an equivalent grammar $\bar{\Gamma}$.

Proof 2.4.1

We define rule $r = \langle guard \rangle : X \to Y_1 \ \cdots \ Y_n$ as the rule in Γ that is replaced by other rules in $\bar{\Gamma}$. We consider two cases: Y_1 is a terminal, and Y_1 is a normal nonterminal.

For the case that Y_1 is a terminal, the only difference between Γ and $\bar{\Gamma}$ is that rule $r = \langle guard \rangle : X \to Y_1 \ \cdots \ Y_n$ is replaced by rules $r_1 = \langle guard \rangle : X \to Y_1 X'$ and $r_2 = X' \to Y_2 \ \cdots \ Y_n$ to get $Y_1 \ \cdots \ Y_n$. Consider any leftmost derivation with Γ that applies the rule r. We get an equivalent leftmost derivation with $\bar{\Gamma}$ that replaces the application of r with the application of rule r_1 immediately followed by the application of rule r_2 to produce the exact same result. Likewise, any leftmost derivation in $\bar{\Gamma}$ that applies rule r_1 must then immediately apply rule r_2 since X' is the leftmost nonterminal in the resulting string. We get an equivalent leftmost derivation with Γ by replacing the application of r_1 and r_2 with r. Finally, r_2 can never be applied except immediately following the application

of r_1 because rule r_1 is the only derivation that can produce nonterminal X'. Therefore, Γ and $\bar{\Gamma}$ are equivalent.

For the case that Y_1 is nonterminal, rule $r = \langle guard \rangle : X \to Y_1 \ \cdots \ Y_n$ in Γ is replaced by rule $r_1 = \langle guard \rangle : X \to Y_1'$ in $\bar{\Gamma}$. Furthermore, we add copies of all rules with head Y_1 that now have head Y_1' where all nonterminals are replaced with new equivalent nonterminals. This also applied to other nonterminals that are in the body of rules in Γ with head Y_1. Finally, for any terminating rule r_t in $\Gamma(Y_1)$, we add a rule r_t' where $Y_2 \ \cdots \ Y_n$ is appended to the body of r_t' and add r_t' to $\bar{\Gamma}$.

Consider any leftmost derivation with Γ that applies the rule r. We get an equivalent leftmost derivation with $\bar{\Gamma}$ as follows. First, we replace the application of r with the application of r_1. Next, until we reach a terminating rule, we replace each application of a rule with head Y_1 or other nonterminal in $\Gamma(Y_1)$ with the equivalent new rule using the new nonterminal names. Finally, we replace the application of terminating rule r_t with terminating rule r_t'. Now consider any leftmost derivation in $\bar{\Gamma}$ that applies rule r_1. This leftmost derivation must eventually apply some terminating rule r_t'. We get an equivalent leftmost derivation in $\bar{\Gamma}$ by replacing r_1 with r, r_t' with r_t, and intermediate rule applications with their original rule copies. We also note that no derivations in $\bar{\Gamma}$ can use any of the new rules without first invoking r_1 since that is the only path to reaching the new nonterminals. Therefore, Γ and $\bar{\Gamma}$ are equivalent. ■

2.4.4 Counting Approximation

For nonterminals in $\mathbb{N}_p$ that are not normal, we use counting approximation to produce a collection of regular rules, which are used to replace these non-normal nonterminals. For a non-normal nonterminal X, our basic idea is to parse only the start and end terminals

for $\Gamma(X)$ ignoring any other parsing information contained within subgrammar $\Gamma(X)$. This approximation is sufficient because our extraction specification does not need to precisely parse any subgrammar starting at a nonterminal in $\mathbb{N}_p$. That is, we only need to identify the start and end of any subgrammar rooted at a nonterminal $X \in \mathbb{N}_p$. By using the counters to track nesting depth, we can approximate the parsing stack for such nonterminals . For nonterminals in the input extraction grammar, we require them to be normal. In practice, this restriction turns out to be minor, and when a violation is detected, our tool will give feedback to aid the user in revising the grammar.

Given a CCFG Γ_f with a nonterminal $X \in \mathbb{N}_p$ that does not identify as normal, FlowSifter computes a counting approximation of $\Gamma_f(X)$ as follows. First, FlowSifter computes the sets of start and end terminals for $\Gamma_f(X)$, which are denoted as *start* and *stop*. These are the terminals that mark the start and end of a string that can be produced by $\Gamma(X)$. The remaining terminals are denoted as *other*. For example, in the Dyck extraction grammar Γ_{xd} in Figure 2.4(b), the set of start and end terminals of $\Gamma_{xd}(S)$ are {'['} and {']'}, respectively, and *other* has no elements. FlowSifter replaces all rules with head X with the four rules in Figure 2.6 that use a new counter *cnt*. The first rule allows exiting X when the

1	$(cnt = 0)\ \mathrm{X} \to \epsilon$
2	$(cnt \geq 0)\ \mathrm{X} \to start\,(cnt := cnt + 1)\,\mathrm{X}$
3	$(cnt > 0)\ \mathrm{X} \to stop\,(cnt := cnt - 1)\,\mathrm{X}$
4	$(cnt > 0)\ \mathrm{X} \to other\,\mathrm{X}$

Figure 2.6: General Approximation Structure

recursion level is zero. The second and third increase and decrease the recursion level when matching start and stop terminals. The final production rule consumes the other terminals, approximating the grammar while $cnt > 0$.

For example, if we apply counting approximation to the nonterminal S from the Dyck extraction grammar Γ_{xd} in Figure 2.4(b), we get the new production rules in Figure 2.7.

1	$(cnt = 0)\ \mathrm{S} \rightarrow \epsilon$
2	$(cnt \geq 0)\ \mathrm{S} \rightarrow$ '[' $(cnt := cnt + 1)\,\mathrm{S}$
3	$(cnt > 0)\ \mathrm{S} \rightarrow$ ']' $(cnt := cnt - 1)\,\mathrm{S}$

Figure 2.7: Approximation of Dyck S

We can apply counting approximation to any subgrammar $\Gamma_f(X)$ with unambiguous starting and stopping terminals. Ignoring all parsing information other than nesting depth of start and end terminals in the flow leads to potentially faster flow processing and fixed memory cost. In particular, the errors introduced by counting approximation do not interfere with extracting fields from correct locations within protocol compliant inputs. However, counting approximations do not guarantee that all extracted fields are the result of only protocol compliant inputs. Therefore, application processing functions should validate any input that its behavior depends upon for proper operation.

2.4.5 Idle Rule Elimination

The CRG generated as above may have production rules without terminals. When implemented as a parser, no input is consumed when executing such rules. We call such rules *idle rules*, and they have the form: $X \rightarrow Y$ without any terminal α. FlowSifter eliminates idle rules by hoisting the contents of Y into X, composing the actions and predicates as well. For a CRG with n variables, to compose a rule

$$(q_1 \wedge \cdots \wedge q_n) : Y \rightarrow \alpha(act)Z(g_1, \cdots, g_n)$$

into the idle rule

$$(p_1 \wedge \cdots \wedge p_n) : X \rightarrow Y(f_1, \cdots, f_n),$$

we create a new rule

$$(p'_1 \wedge \cdots \wedge p'_n) : X \rightarrow \alpha(act)Z(f'_1, \cdots, f'_n)$$

where $p'_i = p_i \wedge q_i$ and $f'_i = f_i \circ g_i$ for $1 \leq i \leq n$. That is, we compose the actions associated with Y in X into Z's actions and merge the predicates.

2.5 Automated Counting Automaton Generation

The *automata generator* module in FlowSifter takes an optimized extraction grammar as its input and generates an equivalent counting automaton (CA) at output. The field extractor module will use this CA as its data structure for performing field extraction.

One of the challenges of field extraction with a CA is resolving conflicting instructions from different CRG rules. For example, consider a CA state A with two rules: $A \rightarrow /ab/B$ and $A \rightarrow /a/[x := x + 1]C$. After processing input character `a`, the state of the automaton is indeterminate. Should it increment counter x and transition to state C, or should it wait for input character `b` so it can transition to state B?

We solve this problem by using a DFA as subroutines in counting automtata. The DFA will inspect flow bytes and return a decision indicating which pattern matched. The DFA will use a priority system to resolve ambiguities and return the highest priorty decision. For the above example, the DFA will have to lookahead in the stream to determine whether /ab/

or /a/ is the correct match; in practice, the lookahead required is small and inexpensive. We describe this novel integrated CA and DFA model in this section.

2.5.1 Counting Automata

A Counting Automata (CA) is a 6-tuple $(Q, C, q_0, c_0, \mathbb{D}, \delta)$ where Q is a set of CA states, C is a set of possible counter configurations, $q_0 \in Q$ is the initial state and $c_0 \in C$ is the initial counter configuration. Normally, the transition function δ of the CA is a function that maps the current configuration (state $q \in Q$ and counter configuration $c \in C$ along with input character $\sigma \in \Sigma$ to a new CA state q' along with some action to update the current counters. We choose a different approach where we use Labeled Priority DFA (LPDFA) as a subroutine to perform this mapping for a given CA. We leave the formal details of LPDFA to Section 2.5.2 but describe now how a CA uses LPDFA to define the CA's transition function δ. The set $\mathbb{D}$ defines the set of LPDFA that the CA may use. The transition function δ then maps each configuration ($q \in Q$, $c \in C$) to an appropriate LPDFA. That is, $\delta : Q \times C \rightarrow \mathbb{D}$. The LPDFA will deterministically process the flow and return a decision belonging to set $D = (Q_C \cup \{DONE, FAIL\}) \times (C \rightarrow C)$ where $DONE$ and $FAIL$ are distinct from all CA states; $DONE$ implies the CA has completed processing of the input stream and $FAIL$ implies that the input flow does not meet the protocol expectations.

FlowSifter generates a CA $(Q, C, q_0, c_0, \mathbb{D}, \delta)$ from a CRG $\Gamma = (N, \Sigma_g, C_g, R, S)$ as follows. We set $Q = N$ and $q_0 = S$. We build C based on C_g but with a bounded maximum size b where typically $b = 2^{sizeof(int)} - 1$; counters are initialized to 0. Formally, $C = \{(c_1, c_2, \ldots, c_{|C_g|}) : 0 \leq c_i < b, 1 \leq i \leq |C_g|\}$ and $c_0 = (0, 0, \ldots, 0)$. If necessary, we can tune the parsing state size by using different numbers of bits for each counter. We set δ as follows. For each configuration (q, c), we identify the set of CRG rules $R(q, c)$ that correspond to

(q, c) and build the corresponding LPDFA from those rules; the set $\mathbb{D}$ is simply the set of LPDFA we build. We describe LPDFA construction in detail in Section 2.5.2.

For example, in Figure 2.8, for the configuration with CA state 3 and counter $c == 0$, rules 5 and 6 are active, so $\delta(3, c == 0)$ is the LPDFA constructed from the right hand sides of those rules: `[` $(c := c+1)\,3$ and $\epsilon\,(p := token(p))\,2$. This LPDFA will return decision $(3, (c := c+1))$ when the flow bytes match [and $(2, p := token(p))$ when the flow bytes match ϵ. On the other hand, $\delta(3, c > 0)$ is constructed from the right hand side of rules 4, 5, and 7.

1		$1 \rightarrow$ `[` $(p := pos())\,4$
2		$2 \rightarrow$ `]` `[` $(c := 1)\,5$
3		$2 \rightarrow$ `]`
4	$(c > 0)$	$3 \rightarrow$ `]` $(c := c-1)\,3$
5		$3 \rightarrow$ `[` $(c := c+1)\,3$
6	$(c = 0)$	$3 \rightarrow \epsilon\,(p := token(p))\,2$
7	$(c > 0)$	$3 \rightarrow$ `/[^[\]]/` 3
8		$4 \rightarrow$ `]` $(c := 1)\,3$
9		$4 \rightarrow \epsilon\,(p := token(p))\,2$
10	$(c > 0)$	$5 \rightarrow$ `]` $(c := c-1)\,5$
11		$5 \rightarrow$ `[` $(c := c+1)\,5$
12	$(c = 0)$	$5 \rightarrow \epsilon$
13	$(c > 0)$	$5 \rightarrow$ `/[^[\]]/` 5

Figure 2.8: CRG for Dyck example from Figures 2.2b and 2.4b

To apply a CA to a flow, we initialize the CA state as (q_0, c_0). Based on the current state (q_i, c_i), we determine $\delta(q_i, c_i) = dfa_i$, and apply dfa_i to the flow bytes. This LPDFA will always return a decision (q_{i+1}, act_i), and if q_{i+1} is either $DONE$ or $FAIL$, parsing ends. After computing $c_{i+1} = act_i(c_i)$, the CA state becomes (q_{i+1}, c_{i+1}), and we repeat the process until we are out of flow bytes.

For example, consider the optimized CRG and CA for our running Dyck grammar example depicted in Figures 2.8 and 2.9, respectively; this CA uses a single counter *cnt*. The CA

state labels have been replaced with integers, and we show the terminal symbols as strings when possible, or as $/regex/$ when a regular expression is needed, such as for rule 7. The initial CA configuration is $(1, cnt == 0)$. The only CRG rule for state 1 is CRG rule 1 which has no conditions associated with it. Thus, the CA invokes the LPDFA that represents the body of rule 1; this LPDFA matches only the string "`[`", and the CA transitions to state 4 after updating p to the result of $pos()$. If the flow bytes do not start with `[`, then the LPDFA returns $FAIL$ and the CA stops processing input because the input does not conform to the CRG. Suppose later the configuration is $(3, cnt > 0)$. In this case, the CA will invoke an LPDFA that matches the input against `[` to increment cnt, `]` to decrement cnt, or any other input character making no change to cnt; this is based on the counting approximation to parse nested brackets. In all cases, the LPDFA sets the next state to be 3. If the configuration is $(3, cnt == 0)$, then the CA will invoke an LPDFA that matches the input against `[` to increment cnt and return to state 3; otherwise no input is consumed and $p := param(p)$ will be evaluated and the CA will leave state 3 for state 2.

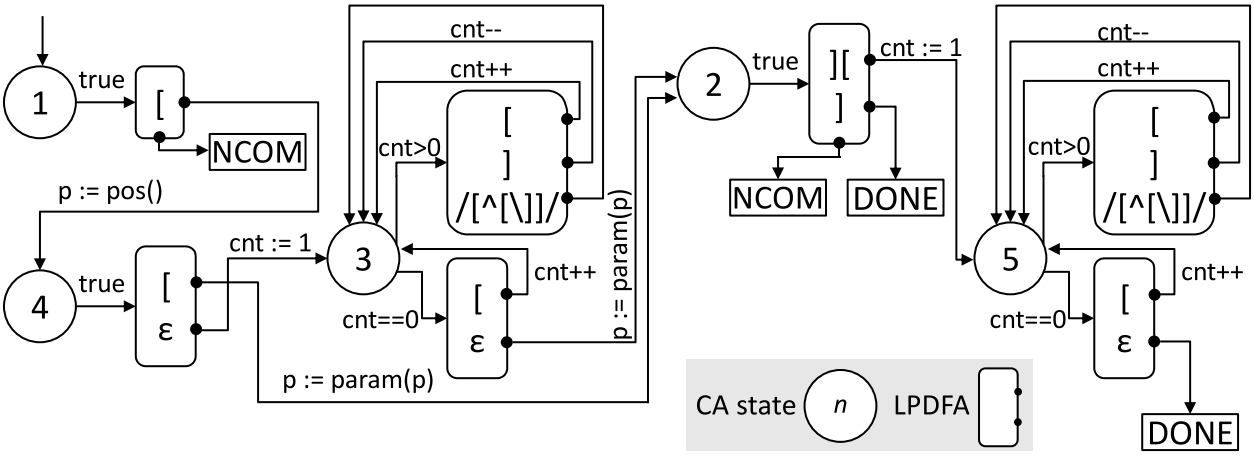


Figure 2.9: Exploded CA for Dyck in Figures 2.2b and 2.4b; each cluster has start CA state on left, and destination CA state on right

The functions *pos* and *param* allow the CA to interact with its runtime state and the outside world. The function *pos*() is built into the FlowSifter environment and returns the current parsing offset in the flow. The CA stores this information in a counter so that it can report the start and end positions of a token when it calls $param(p)$ to report this token to the layer above. The CA waits for a return value from the called application processing function so that it can update the counters before it continues processing the input flow. In many cases, the application processing function never needs to return an actual value to the CA; in such cases, it immediately returns a null value, and the CA immediately resumes processing the input flow.

2.5.2 LPDFA

In this section, we formally define LPDFA focusing on nonstandard LPDFA construction and operation details. A Labeled Priority DFA (LPDFA) is a 7-tuple $(Q, \Sigma, \delta, q_0, D, DF, \pi)$ where Q is a set of states, Σ is an alphabet, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, as normal. The new properties are D, the set of possible decisions, $DF : Q \to D$, a partial function assigning a subset of the states (the accepting states) a decision, and $\pi : Q \to \mathbb{N}$ a total function assigning every state a priority. An LPDFA works as a subroutine for a CA, examining the input for various patterns, consuming the highest priority observed pattern, and returning this pattern's associated decision. We construct and run an LPDFA in a manner similar to a DFA with some modifications to return values instead of just accept/reject and to only consume the correct amount of the input. We focus on the construction of $DF : Q \to D$, a partial function assigning a subset of the states (the accepting states) a decision, and $\pi : Q \to \mathbb{N}$ a total function assigning every state a priority.

As a reminder, $D = (Q_C \cup \{DONE, FAIL\}) \times (C \to C)$, where $DONE$ and $FAIL$ are distinct from all CA states.

For each configuration ($q \in Q$ and $c \in C$) in the δ function of the CA, we construct an LPDFA from the bodies of the rules matching the given configuration. Because we start from a CRG, we can assume the rule bodies are written as (rx_i, act_i, q_i), a terminal regular expression, action and nonterminal. If no rule has a regular expression that matches the empty string ϵ, then we add an ϵ-rule with body $(\epsilon, (), FAIL)$ to guarantee the LPDFA will always return a value. We use the standard process for constructing an automaton from a collection of regular expressions.

We now describe how we set DF. For any accepting state q, we identify all CRG rules' whose regular expressions were matched. If more than one match, we choose the rule r with body (rx, act, q) that has the highest priority (the ϵ-rule has lowest priority). If r is the ϵ-rule, the CA state value is set to $FAIL$. If q is empty, the CA state value is set to $DONE$. Otherwise, the CA state value is set to q. The appropriate action is set to act.

We now describe how the LPDFA operates. We give all states in the LPDFA a priority which is the highest priority decision that is reachable from that state. As the LPDFA processes the input flow, it remembers the highest priority decision state encountered. To prevent the LPDFA from continuing to consume input due to a potential low priority match such as a low priority `/.*/` rule, the LPDFA stops processing the input once it reaches a state with an equal or lower priority. The LPDFA then returns the appropriate decision and consumes only the appropriate input even if more has been seen.

We illustrate the importance of prioritizing CRG rules using the following two rules from the protocol specification for HTTP headers.

```
HEADER 50 -> /(?i:Content-Length):\s*/
```

```
        [bodylength := getnum()];
    HEADER 20 -> TOKEN /:/ VALUE;
```

The second rule is given a lower priority (20) than the first rule's priority (50) to ensure that the first rule is used when the flow prefix is "Content-Length:". In such a case, the rule stores the size of the HTTP body in a counter bodylength for later use. If the priorities were inverted, the first rule would never be used. We must ensure the relative priorities of rules are maintained through all optimizations that we apply. Maintaining these priorities is straightforward but tedious, so we omit these details.

2.5.3 CA Specific Optimizations

We have implemented two key optimizations to speed up our CA implementation. We first avoid processing many bytes of the flow by having actions modify the flow offset in the parsing state. Specifically, if our CA is processing an HTTP flow and does not need to parse within the body, an action can call $skip(n)$ to skip over n bytes of payload. This allows FlowSifter to avoid stepping through the payload byte-by-byte to get to the headers of the next request in the same flow.

We also eliminate some LPDFA to CA transitions. Suppose the optimized CRG has a nonterminal X with a single rule with no actions such as $X \rightarrow /rx/Y$. We can eliminate the switch from LPDFA to CA at the end of $/rx/$ and the switch back to LPDFA at the beginning of Y by inlining Y into X. This is similar to idle rule elimination, but because our terminals are regular expressions, we can concatenate two regular expressions into a single terminal, keeping the grammar in normal form. We also perform this optimization when Y has a single rule and all of X's rules that end in Y have no actions. This increases the number of states in the LPDFA for each non-terminal but improves parsing speed by decreasing the

number of context switches between LPDFA and CA. This optimization has already been performed on the CA in Figure 2.9, specifically in state 2. The pattern "][" is not part of any of the input regular expressions, but is composed of the closing ']' of the Dyck extraction grammar and the opening '[' of the S nonterminal following it.

2.6 Counting Automaton Implementation

We now describe how we implement CA. We first describe incremental packet processing, which is needed because flow data arrives in packets and should be processed as it is received. The alternative solution of buffering large portions of flow is problematic for two reasons. First, it may require large amounts of dynamically allocated memory. Second, it will increase latency in scenarios where undesirable traffic must be blocked.

We then describe two implementations of CA: simulated CA and compiled CA. In a simulated CA, an automaton-independent process parses a flow by referencing a memory image of the simulated CA. In a compiled CA, the structure of the CA is encoded in the process that parses the flow. Typically, compiled CA are more efficient but simulated CA are easier to deploy and modify.

2.6.1 Incremental Packet Processing

Packet processing is difficult because the flow data arrives packet by packet rather than all at once. There are two main ways to process packets: *incrementally* which means processing each packet as it arrives or *buffering* which means buffering a number of packets until a certain amount of flow data is gathered. A DFA supports incremental processing by processing each byte of the current packet and then saving the active state of the DFA in a parsing

state to be used when the next packet of the flow arrives. BinPAC and UltraPAC do not support incremental packet processing. Instead, they buffer packets until they can guarantee sufficient flow data is available to match an entire token. This has the two drawbacks of (i) requiring large amounts of dynamically allocated memory and (ii) increasing latency in scenarios where undesirable traffic must be blocked.

As FlowSifter is built on automata-theoretic constructs, we present an incremental packet processing approach similar to DFA, though we do require some buffering of flow data since an LPDFA may look at more flow data than it consumes. Unlike other buffering solutions, FlowSifter only buffers input data that the LPDFA needs to determine the highest priority match; this is typically a small amount of data, much smaller than the amount of data needed to guarantee a token can be matched. There are three parts of the per-flow state that we must record: the state of the input flow, the state of the CA and the state of the LPDFA.

We record two byte offsets for each flow: an offset within the current packet of the next byte to be processed, and a flow offset of the current packet, indicating the position in the flow of the first byte of that packet. We keep both of these in our parsing state to be able to check for errors in flow reassembly and to aid in supporting the *skip*() builtin. After processing a packet, we subtract its size from the packet offset, which normally resets the packet offset. The *skip*() action can be implemented by increasing the packet offset by the number of bytes to be skipped. If the resulting offset is within the current packet, processing will resume there. If the offset is within the next packet, the subtraction rule will correctly compute the offset within that next packet to resume processing. If the offset is beyond the next packet's data, that packet can be skipped entirely and the subtraction rule will account for that packet's bytes being skipped.

As both the CA and LPDFA are deterministic, they each have only one active state. Furthermore, when an LPDFA has an active state, the CA state is not needed, and when a CA state is active, no LPDFA state is active; thus we only need to store one or the other. In addition to the active state, the CA also must keep the values of each counter. In addition to the active state, the LPDFA must track the state id and flow offset of the current highest priority match, if any. A small buffer may be needed when the LPDFA lookahead occurs at a packet boundary, to buffer the flow data since the last match was found. This allows the LPDFA to consume only the bytes needed to reach its decision while allowing the next consumer of input bytes to resume from the correct location, even if that is in the previous packet.

Finally, we must address incremental parsing for actions that alter the parsing state. For example, *skip*() can change the packet offset. We might create other actions such as *getByte*() that can read a current input character into a counter as an unsigned integer; *getByte*() is useful in the DNS protocol where the length of a name field is indicated by the byte preceding it. Instead of using *getByte*(), we could use an LPDFA that transitions to a different state for each possible value of that byte and have an action for each value that would set the counter correctly. Using a function like *getByte*() makes this much easier to implement and faster to execute. However, this does introduce a corner case where the byte we need may be in the next packet. In general, the CA must be able to resume processing from the middle of an action. Our simulated CA and compiled CA take different approaches to solving this problem, as described in their sections below.

2.6.2 Simulated CA

In an ASIC implementation of CA, we create a fixed *simulator* that cannot be changed after construction. The simulator uses a memory image of the CA to be simulated when processing an input flow. To assess the challenges of a simulated CA implementation and develop a proof of concept, we implemented a software CA simulator. The most challenging issues we faced were (i) efficiently implementing the function δ to find the appropriate LPFDA based on the current CA counter values and (ii) incremental packet processing.

The biggest challenge to implementing δ efficiently is the potentially huge size of the counter state space. With two 16-bit counters and just 10 CA states, a direct lookup table would have over 40 billion entries, ruling out this solution for the entirety of δ. Instead, we break the lookup into two steps. In the first step, we use the CA state number as a direct lookup.

For a CA with C counters, a given CA state may correspond to a CRG nonterminal with n rules. We must find the correct LPFDA that implements the combination of these rules that can be applied,based upon the current counter values. We present three different solutions to solving this LPDFA selection problem. We evaluate each solution based upon two criteria: (i) space and (ii) time. The space complexity of a solution corresponds to the number of LPDFA that need to be precomputed and stored. The time complexity of a solution corresponds to the number of predicate evaluations that must be performed. Each solution differs in how to index the LPDFA and how to perform the predicate evaluations.

Our first solution indexes LPDFA by rule. That is, we construct 2^n LPDFA, one for each combination of rules and store pointers to these LPDFA in an array of size 2^n. Note that not all combinations of rules are possible. For example, in the Dyck CA, state 3 has 4 rules,

so the LPDFA table has 16 entries. However, only two LPDFAs are possible for state 3: one encoding rules 5 and 6 when $cnt == 0$, and one encoding rules 4, 5 and 7 when $cnt > 0$. We perform an analysis of rule predicates and save space by leaving as many of the LPDFA array entries empty as possible. To find the current predicate, we evaluate the predicates for each rule and pack these true/false results as 1/0 bits in an unsigned integer used to index the array of LPDFA. In the worst case, this requires nC counter predicate evaluations. Of course, some counters may not need to be evaluated for some rules, and the results from some counter evaluations may eliminate the need to perform other counter evaluations.

Our second solution indexes LPDFA by predicate, taking advantage of redundancy in predicates among different rules. For example, using state 3 from the Dyck CA, we previously observed there are only two relevant LPDFA: one for when $cnt == 0$ and one for when $cnt > 0$. We can store this list of relevant predicates and index the LPDFA by their truth values. For this example, we would have just these two predicates and two LPFDA. For this example, we would evaluate both predicates and use these two bits to choose the LPDFA to execute.

A third solution is to build an optimal decision tree. Each node of the decision tree would correspond to a counter and the children of that node would either be the LPDFA to execute or further nodes corresponding to other counters. Each edge would be labeled with the values of the counter it can be traversed on. In this example, the decision tree would have one node, with an edge to the rule {5,6} LPDFA labeled 0 and an edge to the rule {4,5,7} LPDFA on all other values. This solution can reduce the number of counter evaluations required by optimally exploiting redundancies and optimally using the results of previous predicate evaluations. We plan on investigating this potential solution further in future work. In our

software simulator, we observed that indexing by rule appeared to be more efficient than indexing by predicate in our experiments.

Our simulator uses closures to handle saving and resuming state. Whenever it runs out of input, it returns a new function that takes the next packet's payload as input and continues processing where it left off. This is possible because OCaml allows us to encapsulate a portion of our environment of variables inside a newly created function, allowing this new function to make use of any local state available at the point of its creation. The details of the function differ depending on where we run out of input. In general, this resume function does some bookkeeping on the input string to put it into the main shared state record, and then it calls a function to resume processing at either a CA state (to processes any delayed actions) or at an LPDFA state, to continue consuming input.

2.6.3 Compiled CA

In this section, we give a compilation from CA to C++. This non-ASIC compiled CA implementation can achieve very high performance because modern CPUs are highly optimized simulators for their machine language. The major difficulty is minimizing the overhead in the compilation process; the Turing completeness of CPUs guarantees that this is always possible for any reasonable automata construction, but the efficiency of the result is strongly dependent on the semantics of the automaton and the specifics of the translation process.

The basic structure of the compiled CA is similar to that of the simulated CA. We briefly describe all of the details but focus on the key differences. We encapsulate the parsing state of a flow using a C++ struct that contains unsigned integers for each counter, a few fields to hold a pointer to the flow data and the offsets relative to the pointer and to the start of the flow. We also include fields for the LPDFA to store the offset, state, and priority of

the highest priority matching state seen so far. Finally, we store the current LPDFA state, if any, to support incremental packet processing.

We represent each CA state, each LPDFA, and each action update function with a procedure. Each procedure exhibits tail call behavior where it ends by calling some other procedure. For example, after the CA state procedure determines which LPDFA to run, it ends its operation by calling the appropriate LPDFA procedure. Likewise, an LPDFA procedure will typically end by calling a series of update actions and then the next CA state procedure. To ensure the stack does not grow to unbounded size, our CA must implement Tail Call Optimization (TCO). Ideally, the C/C++ compiler will implement TCO. If not, we manually implement TCO by having a main dispatch loop. Within this loop, we maintain a function pointer that represents the next procedure to be called. Each procedure then ends by returning a pointer to the next function to be called rather than calling the next function.

The procedure for each CA state has access to all the flow state, and its job is simply to determine which LPDFA to run. As with simulated CA, the compiled CA can index the LPFDA in many different ways such as by rule, by predicate, or by decision tree. Our compiler produces a CA that indexes by rule where the resulting Boolean values are concatenated into a single integer. The CA state procedure uses a switch statement on this integer to determine which LPDFA is initialized and started. If all the rules for a CA state have no predicates, then there is only one LPDFA that is always called. We can eliminate this CA state procedure and simply call the corresponding LPDFA procedure instead.

We run each LPDFA using a its own LPDFA simulator procedure. The transitions and priorities are stored in integer arrays. To make a transition, the current state and input character are used to index the transition array which stores the next state that this automaton will be in. The simulator only proceeds to the next state q if q's priority exceeds the priority

of the highest priority match found so far. When we have determined the highest priority match, the simulator runs a switch statement to implement the actions and the transition to the next CA state dictated by this match. We choose to implement a separate simulator for each LPDFA to support finely tuned branch prediction for the CPU and to facilitate pausing and resumption caused by incremental packet processing. The cost of replicating the very small common simulation code is negligible.

It is possible that multiple CA states may call identical LPDFA. With each new LPDFA that we create, we compare it to previously created ones and only keep the new one if it is indeed different than all previous ones.

Finally, we discuss our procedures which implement the actions of a CA. We create a procedure for each action so that we can pause and resume them at packet boundaries. To support incremental packet processing, we ensure that each input-consuming function in an action is in its own action function. We improve performance by leaving inline actions that do not consume any input as the CA will never need to resume from such actions.

2.7 Extraction Generator

FlowSifter is unique in its use of protocol and extraction grammars to specify the desired fields to be extracted. In this section, we describe our simple Extraction Generator GUI tool that allows a user to easily create an extraction grammar from a given protocol grammar. It simplifies the process of writing the extraction grammar to exploring expansions of the protocol grammar and ticking off boxes next to the fields to extract.

A screenshot of this interface is shown in Figure 2.10. At the top is a drop-down dialog box that lists all the available protocol grammars; in this case, we have chosen the `dyck.pro`

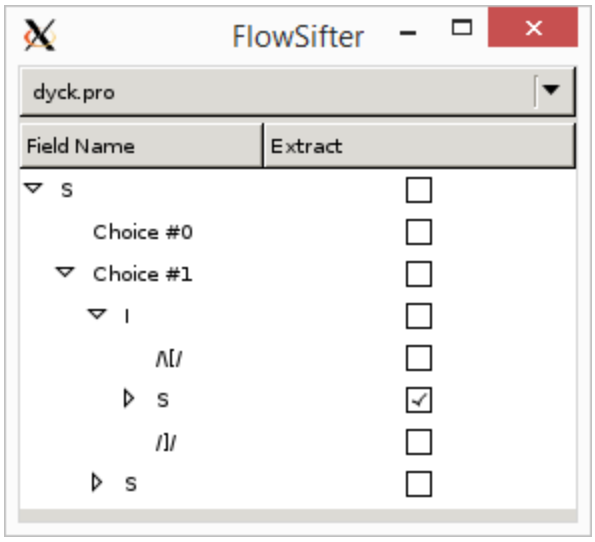


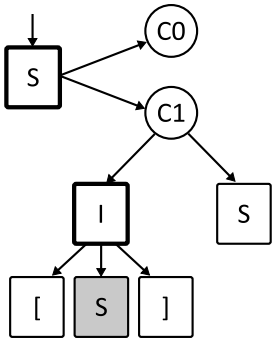
Figure 2.10: Extraction Generator Application Screenshot

grammar for the Dyck protocol. Below that is a TreeView that shows a tree view of the grammar, which we call the MetaTree. Figure 2.11a shows the MetaTree corresponding to the tree view in the screenshot. The root of the MetaTree is the start symbol of the grammar. "Choice" nodes in the tree allow us to show the rules for each non-terminal. The body of that non-terminal makes up the children of its "Choice" children. Terminal symbols are the leaves in the MetaTree.

The MetaTree for a grammar contains all possible parse trees as subtrees. For grammars whose language is infinite, this tree is infinite in size, so we construct it on demand. To preserve the efficiency of generating the MetaTree, it does not include the predicates or actions of the grammar and thus does not maintain counter values. This simplification means that a given MetaTree may include fields that are not producible by the original grammar. That is, the counter values may prevent an application of a derivation rule, but the MetaTree will allow it since it does not track the counter values.

The Extraction Generator works by allowing the user to explore the MetaTree and choose subtrees to extract as fields by selecting nonterminals and rules as follows. Initially, the

extraction generator shows the start nonterminal for the protocol grammar. In the example screenshot, this start nonterminal is called S. A nonterminal is selected by clicking the ▷ adjacent to the nonterminal. Selecting a nonterminal displays the rules that have that nonterminal as head. As rules typically have no name, we identify these rules using the notation "Choice #0" to "Choice #k" if there are $k+1$ rules. When there is only one rule for a nonterminal, the "Choice" nodes are omitted. For example, the nonterminal I node in Figure 2.10 has no Choice node children. A nonterminals and choice nodes are expanded to show their children by clicking the ▷ adjacent to that . Selecting a rule displays the sequence of terminals and nonterminals in the body of that rule. Finally, any nonterminal or rule node can be selected for extraction by clicking the box next to it.



(a) MetaTree for Dyck example; field to extract in grey

1	$\mathrm{XS} \rightarrow \epsilon$
2	$\mathrm{XS} \rightarrow \mathrm{XI\,S}$
3	$\mathrm{XI} \rightarrow$ /\[/ token(S) /]/

(b) Extraction Grammar

Figure 2.11: Intuitive MetaTree and Extraction Grammar for Dyck example

After a MetaTree has been constructed, it can be easily converted into an extraction grammar. First, the nonterminals with descendant nodes to be extracted are renamed with unique names. The other nonterminals keep their names and reference the corresponding rules in the protocol grammar. For any renamed nonterminals, we generate new rules for

them replacing any nonterminals with renamed nonterminals as dictated by the MetaTree. A nonterminal T marked for extraction is encapsulated in a $token(T)$ in the corresponding rewritten rule. If a choice node is marked for extraction, the corresponding rule is encapsulated by $token()$.

In the example depicted in Figures 2.10 and 2.11, a single S field is chosen to be extracted from within the first brackets of the Dyck grammar. The internal nonterminals, the root S and the I in the root S's "Choice #1" child, are marked by a thick box in Figure 2.11 and are renamed to XS and XI in the extraction grammar. We produce two rewritten rules for XS which are rule numbers 1 and 2 in Figure 2.11b. Rule 1 corresponds to "Choice #0" and has an empty body. Rule 2 corresponds to "Choice #1" and has a renamed XI in its rule body. Rule 3 is produced by XI directly as it has only one choice, and its rule body has no renamed nonterminals as the child S is a leaf. That S nonterminal is placed inside $token()$ because it will be extracted as a field.

This extraction generator is able to handle a wide range of extraction scenarios, but it cannot create arbitrary extraction grammars. One limitation is that it cannot generate extraction grammars with cycles in their nonterminal symbols because the MetaTree representation prevents cycles from being created. If such an extraction grammar is required, it can be written by hand in some cases. The user must ensure that the resulting grammar is regular.

2.8 Experimental Results

We evaluate field extractor performance in both speed and memory. Speed is important to keep up with incoming packets. Because memory bandwidth is limited and saving and

loading extractor state to DRAM is necessary when parsing a large number of simultaneous flows, memory use is also a critical aspect of field extraction.

2.8.1 Methods

2.8.1.1 Traces

Tests are performed using two types of traces, HTTP and SOAP. We use HTTP traffic in our comparative tests because the majority of non-P2P traffic is HTTP and because HTTP field extraction is critical for L7 load balancing. We use a SOAP-like protocol to demonstrate FlowSifter's ability to perform field extraction on flows with recursive structure. SOAP is a very common protocol for RPC in business applications, and SOAP is the successor of XML-RPC. Parsing SOAP at the firewall is important for detecting parameter overflows.

Our trace data format has interleaved packets from multiple flows. In contrast, previous work has used traces that consist of pre-assembled complete flows. We use the interleaved packet format because it is impractical for a network device to pre-assemble each flow before passing it to the parser. Specifically, the memory costs of this pre-assembly would be very large and the resulting delays in flow transmission would be unacceptably long.

Our HTTP packet data comes from two main sources: the MIT Lincoln Lab's (LL) DARPA intrusion detection data sets [25] and captures done in our research lab. This LL data set has 12 total weeks of data from 1998 and 1999. We obtained the HTTP packet data by pre-filtering for traffic on port 80 with elimination of TCP retransmissions and delaying out-of-order packets. Each day's traffic became one test case. We eliminated the unusually small traces ($<$ 25MB) from our test data sets to improve timing accuracy. The

final collection is 45 LL test traces, with between 0.16 and 2.5 Gbits of data and between 27K and 566K packets per trace, totaling 17GB of trace data.

The real life packet traces come from two sources: 300 MB of HTTP spidering traces (HM) and 50GB (roughly 2/3 of which is HTTP) of Internet use (IU) over a period of one month. We capture the HTTP spidering traces using the HarvestMan [26] web spider on web directories like `dmoz.org` and `dir.yahoo.com`. This method produces mostly smaller requests for `.html` files instead of downloading large graphic, sound or video files. HarvestMan also uses HTTP 1.1's keep-alive pervasively, resulting in many HTTP requests/responses being included in a single TCP flow. Thus, it is imperative to support the Content-Length header to identify the end of one request's data. The IU trace was recorded in 100MB sections. We use each section as a separate datapoint for testing.

We construct SOAP-like traces by constructing 10,000 SOAP-like flows and then merging them together into a trace. We create one SOAP-like flow by encapsulating a constructed SOAP body in a fixed HTTP and SOAP header and footer. We use a parameter n ranging from 0 to 16 to vary the depth of our flows as follows. The SOAP body is composed of nested tag; on level l, a child node is inserted with probability $(0.8^{\max(0,l-n)})$. After inserting a child node, the generator inserts a sibling node with probability $(.6 * .8^{\max(0,l-n)})$. The average depth of a flow with parameter n is about $n + 5$.

For each value of n, we generate 10 traces for a total of 170 traces. Each trace consists of 10,000 flows generated using the same parameter n. These 10,000 flows are multiplexed into a stream of packets as follows. Initially we set no flow as active. During each unit of virtual time, one new flow is added to the set of active flows. Each active flow generates data that will be sent in the current unit of virtual time as follows: with equal probability, it sends `0`, `rand(50)`, `rand(200)`, `rand(1000)` and `1000+rand(500)` bytes. If the transmission

amount for a flow exceeds its remaining content, that flow sends all remaining data and is then removed from the set of active flows. All the data sent in one unit of virtual time is merged and accumulated into a virtual packet flow. The typical trace length is roughly 28K packets for $n = 0$ and 106K packets for $n = 16$. The total length of all 170 traces is 668MB.

2.8.1.2 Field Extractors

We have two implementations of FlowSifter: a compiled FlowSifter (csift) and a simulated FlowSifter (sift). We have written one FlowSifter package that generates both implementations. This package is written in 1900 lines of Objective Caml (excluding LPDFA generation, which uses normal DFA construction code) and runs on a desktop PC running Linux 2.6.35 on an AMD Phenom X4 945 with 4GB RAM. The package includes additional optimizations not documented here for space reasons. We emphasize that in our experiments, the compiled FlowSifter outperforms the simulated FlowSifter because we are only doing software simulation. If we had an ASIC simulator, the simulator would run much more quickly and likely would outperform our compiled implementation.

We constructed HTTP field extractors using FlowSifter, BinPAC from version 1.5.1 of Bro, and UltraPAC from NetShield's SVN r1928. The basic method for field extractor construction with all three systems is identical. First, a base parser is constructed from an HTTP protocol grammar. Next, a field extractor is constructed by compiling an extraction specification with the base parser. Each system provides its own method for melding a base parser with an extraction specification to construct a field extractor. We used UltraPAC's default HTTP field extractor which extracts the following HTTP fields: `method`, `URI`, `header name`, and `header value`. We modified BinPAC's default HTTP field extractor to extract these same fields by adding extraction actions. FlowSifter's base HTTP parser was written

from the HTTP protocol spec. We then wrote an extraction specification to extract these same HTTP fields.

For SOAP traffic, we can only test the two implementations of FlowSifter. We again wrote a base SOAP parser using a simplified SOAP protocol spec. We then made an extraction specification to extract some specific SOAP fields and formed the SOAP field extractor by compiling the extraction specification with the base SOAP parser. We attempted to develop field extractors for BinPAC and UltraPAC, but they seem incapable of easily parsing XML-style recursive structures. BinPAC assumes it can buffer enough flow data to be able to generate a parse node at once. UltraPAC's Parsing State Machine can't represent the recursive structure of the stack, so it would require generating the counting approximation by hand.

2.8.1.3 Metrics

For any trace, there are two key metrics for measuring a field extractor's performance: *parsing speed* and *memory used*. We use the term *speedup* to indicate the ratio of FlowSifter's parsing speed on a trace divided by another field extractor's parsing speed on the same trace. We use the term *memory compression* to indicate the ratio of another parser's memory used on a trace divided by FlowSifter's memory used on the same trace. The *average speedup* or *average memory compression* of FlowSifter for a set of traces is the average of the speedups or memory compressions for each trace. Parser Complexity is measured by comparing the definitions of the base HTTP protocol parsers. We only compare the HTTP protocol parsers since we failed to construct SOAP field extractors for either BinPAC or UltraPAC.

We measure parsing speed as the number of bits parsed divided by the time spent parsing. We use Linux process counters to measure the user plus system time needed to parse a trace.

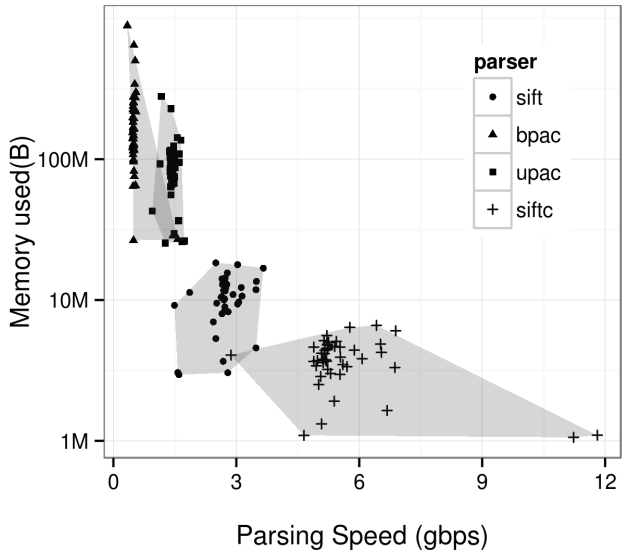
We record the memory used by a field extractor on a trace by taking the difference between the memory used by the extractor at the end of processing the trace and the memory used by the extractor just before processing the trace. This is not the peak memory across the trace. Instead, it is a somewhat random sample of memory usages for each trace. In particular, traces end at arbitrary points typically with many active flows. BinPAC, UltraPAC and compiled FlowSifter use manual memory management, so we get our memory used values via tcmalloc's [27] `generic.current_allocated_bytes` parameter. This allows us to precisely identify the exact amount of memory allocated to the extractor and not yet freed. Although this does not measure stack usage, none of the implementations makes extensive use of the stack. Simulated FlowSifter runs in a garbage collected environment that provides an equivalent measure of live heap data.

2.8.2 Experimental Results

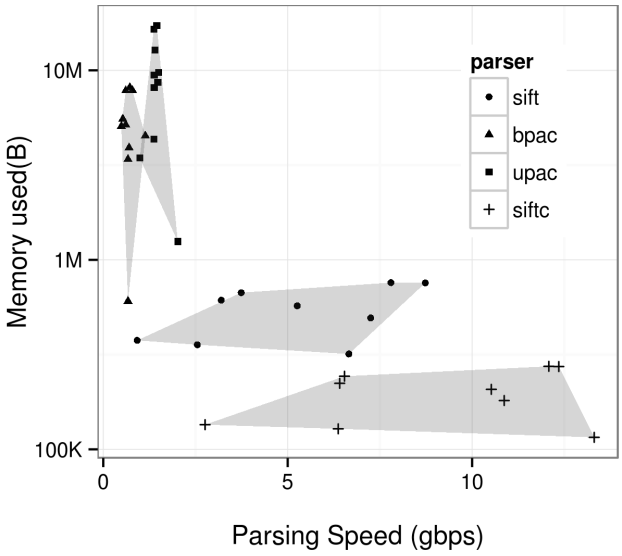
We show empirical CDFs for all three field extractors' memory usage and extraction speed on each dataset. These show FlowSifter's memory use dominates both BinPAC and UltraPAC. There is slight overlap in parsing speed, but FlowSifter clearly has better best case, worst case and average speed than both BinPAC and UltraPAC. The efficiency curve nearly matches the speed curve, with FlowSifter having infrequent worst and best efficiency, and still showing much improvement over BinPAC and UltraPAC.

2.8.2.1 Parsing Speed

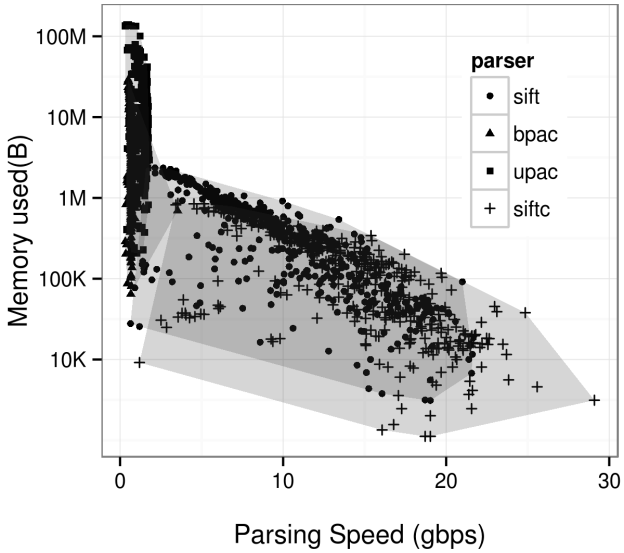
As shown in Figure 2.12, compiled FlowSifter (siftc) is significantly faster than simulated FlowSifter (sift) which is faster than both BinPAC (bpac) and UltraPAC (upac). Each cluster of points has its convex hull shaded to make it easier to separate the clusters while still



(a) LL traces



(b) HM traces



(c) IU traces

Figure 2.12: Comparison of parsers on different traces

showing the individual data points. Across the range of traces, compiled FlowSifter's average speedup over BinPAC ranged from 11 (LL) to 21 (IU). UltraPAC was able to parse faster than BinPAC, but compiled FlowSifter still kept an average speedup of between 4 (LL) to 12 (IU). This means that on a real-life dataset, FlowSifter was able to achieve an average performance of over 12 times that of the previous best field extraction tool. Further speed improvement is possible with ASIC implementation, which could make one LPDFA transition per cycle at 1GHz, resulting in up to 80Gbps performance on the IU traces with a single engine.

Simulated FlowSifter's LPDFA speed is 1.8Gbps; compiled FlowSifter's LPDFA speed is 2.6Gbps. These speeds were measured by running a simple LPDFA on a simple input flow. As shown in Figure 2.12, the FlowSifter implementations can run both faster and slower than their LPDFA speed. FlowSifter can traverse flows faster by using the CA to perform selective parsing. For example, for an HTTP flow, the CA can process the `ContentLength` field into a number and skip the entire body by ignoring that number of bytes from the input. BinPAC and UltraPAC improve their performance similarly through their `&restofflow` flag.

However, the CA introduces two factors that can lead to slower parsing: evaluating expressions and context switching from an LPDFA to a CA and then back to an LPDFA. Evaluating predicates and performing actions is more costly in the simulated FlowSifter implementation than in the compiled implementation. Context switches are also more costly for the simulated implementation than the compiled implementation. Further, the compiled implementation can eliminate some context switches by compiler optimizations. That is, it can inline the code for a CA state into the decision for the LPDFA so that one LPDFA can directly start the next one.

To test FlowSifter's approximation performance, we made a SOAP field extractor that extracts a single SOAP data field two levels deep and then ran it on our 10 traces for each value of n ranging from 0 to 16. The resulting average parsing speeds with 95% confidence intervals for each value of n are shown in Figure 2.13a. For both compiled and simulated implementations, the parsing speed on this grammar is much less than on plain HTTP. The major reason for this is that with HTTP, there are no fields to extract from the body, so the body can be skipped. With SOAP traces, we cannot exploit any skip.

Also, as the recursion level increases, the number of CA transitions per DFA transition increases. This causes FlowSifter to check and modify counters more often, slowing execution.

2.8.2.2 Memory Use

While the graphs in Figure 2.12 show FlowSifter using less memory than BinPAC and UltraPAC, the different number of active flows in each capture make direct comparison harder. Each point in Figure 2.13b shows the total memory used divided by the number of flows in progress when the capture was made. This shows FlowSifter uses much less memory per flow (and thus per trace) than either BinPAC or UltraPAC. On average over our 45 LL traces, FlowSifter uses 16 times less memory per flow (or trace) than BinPAC and 8 times less memory per flow (or trace) than UltraPAC.

FlowSifter's memory usage is consistently 344 bytes per flow for simulated automaton and 112 bytes for compiled automaton. This is due to FlowSifter's use of a fixed-size array of counters to store almost all of the parsing state. BinPAC and UltraPAC use much more memory averaging 5.5KB and 2.7KB per flow, respectively. This is mainly due to their buffering requirements; they must parse an entire record at once. For HTTP traffic, this means an entire line must be buffered before they parse it. When matching a regular expression against

flow content, if there is not enough flow to finish, they buffer additional content before trying to match again.

2.8.2.3 Parser Definition Complexity

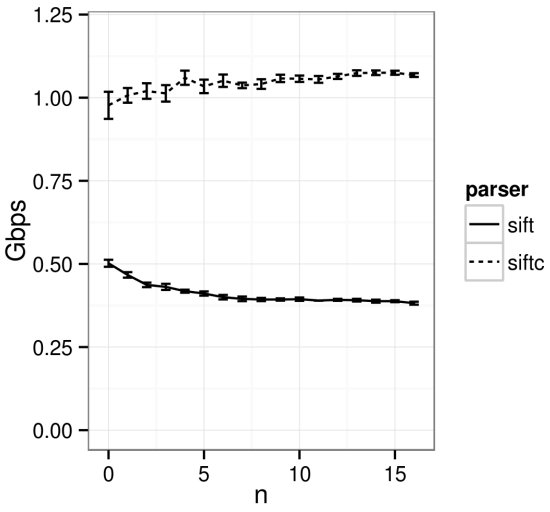
The final point of comparison is possibly less scientific than the others, but is relevant for practical use of parser generators. The complexity of writing a base protocol parser for each of these systems can be approximated by the size of the parser file. We exclude comments and blank lines for this comparison, but even doing this, the results should be taken as a very rough estimate of complexity. Figure 2.13c shows a DNS and HTTP parser size for BinPAC and FlowSifter and HTTP parser size for UltraPAC. UltraPAC has not released a DNS parser. The FlowSifter parsers are the smallest of all three, with FlowSifter's DNS parser being especially small. This indicates that CCFG grammars are a good match for application protocol parsing.

2.9 Conclusions

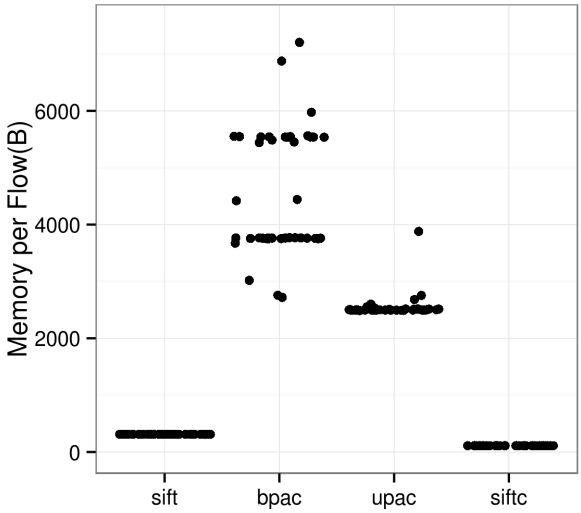
In this work, we performed a rigorous study of the online L7 field extraction problem. We propose FlowSifter, the first systematic framework that generates optimized L7 field extractors. Besides the importance of the subject itself and its potential transformative impact on networking and security services, the significance of this work lies in the theoretical foundation that we lay for future work on this subject, which is based on well-established automata theory.

With this solid theoretical underpinning, FlowSifter generates high-speed and stackless L7 field extractors. These field extractors run faster than comparable state of the art parsers,

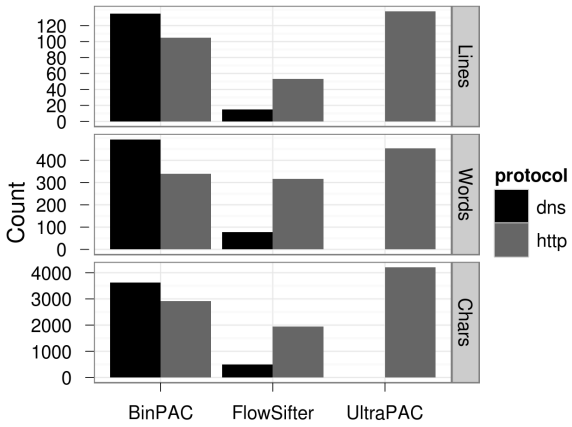
use much less memory, and allow more complex protocols to be represented. The parsing specifications are even by some measures simpler than previous works. There are further improvements to be made to make our field extractor even more selective and efficient by further relaxing the original grammar.



(a) Average parsing speed for SOAP-like flows versus recursion depth n



(b) Memory per flow on LL traces



(c) Complexity of base protocol parsers

Figure 2.13: Various experimental results

Chapter 3

Regex Matching

3.1 Introduction

3.1.1 Motivation

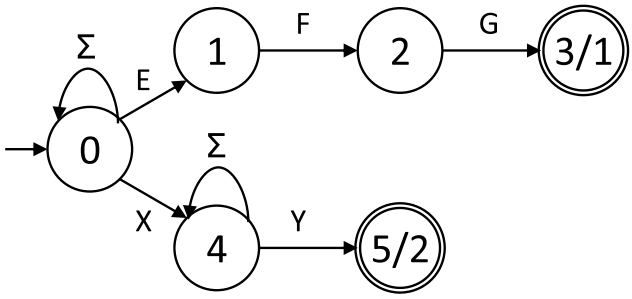
Deep Packet Inspection (DPI) is the core operation in a wide range of networking and security services (such as malware detection, data leak prevention, application protocol identification, load balancing, quality of service, differential billing, copyright enforcement, and usage monitoring) on most networking middleboxes and security devices (such as routers, firewalls, and Network Intrusion Detection/Prevention Systems (NIDS/NIPS)). In the past, string matching was used in DPI; nowadays, Regular Expression (RegEx) matching has become the industry standard because RegExes are more expressive than strings. Given a flow as a stream of bytes, a RegEx matching algorithm needs to determine which RegExes in a predefined set are matched in that flow. As each packet of each flow needs to go through RegEx matching, the memory and time efficiency of RegEx matching is critical to achieve low latency and high throughput. However, it is difficult to achieve both memory and time efficiency for RegEx matching. The Non-deterministic Finite Automata (NFA) model gives us the best memory efficiency as memory grows linear with the size of RegExes, but has the worst time efficiency because we need to maintain multiple active states and perform one memory lookup per active state for each input character. The Deterministic Finite Au-

tomata (DFA) model, on the other hand, gives us the best time efficiency of one lookup per character, but has the worst memory efficiency because of the well known state explosion problem - the number of DFA states can be exponential in the size of RegExes. Memory efficiency is critical for RegEx matching because it needs to run in on-chip SRAM to achieve high speed as it needs low latency random-access memory reads.

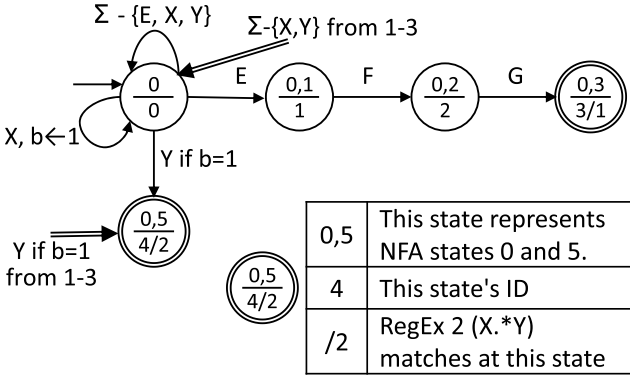
To achieve high speed, RegEx matching algorithms need to be Application Specific Integrated Circuit (ASIC) friendly, *i.e.*, they can be implemented in a small and fast ASIC block. The size of an ASIC block depends on the number of logic gates needed to implement the algorithm, and its speed (*i.e.*, clock frequency) depends on circuit complexity. Not only chip fabrication cost is proportional to its die area, but also for networking and security devices such as IPSes, area efficiency of the circuit board is a critical issue.

3.1.2 Limitations of Prior Art

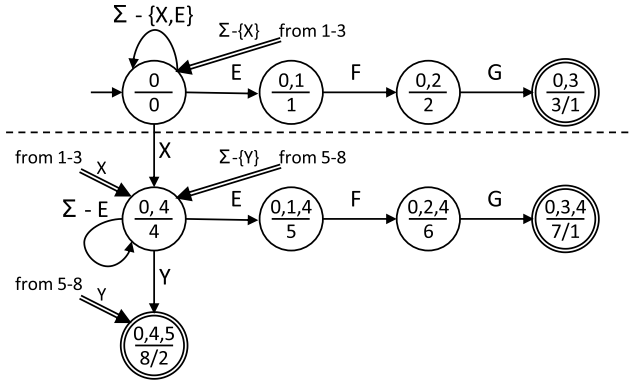
The software RegEx matching solution that represents the state of the art is XFA [28, 29]. An XFA is a DFA where each state is augmented with a program. The transition from a source state to a destination state triggers the execution of a program associated with the destination state. Despite its theoretical elegance, XFA has two fundamental limitations. (1) *XFA does not have a fully automated construction algorithm* (with a given RegEx set as input and a memory image for runtime packet processing as output) fundamentally because the XFA construction process requires human experts to annotate the given RegEx set [30]. Furthermore, both the non-automated manual processing and the automated computation required in XFA construction may take a long time. For manual processing, XFA authors reported that the manual processing may take one hour even for one RegEx, and it is "prohibitively time-consuming" for some RegExes from Snort [28]. For automated computation,



(a) NFA



(b) HFA



(c) DFA

Figure 3.1: NFA, HFA, and DFA generated from RegEx set: {`EFG`, `X.*Y`}

the required time to make the transitions deterministic varies for different RegEx sets, with 13% of the test set in [28] taking between 16 minutes and 2 hours. Slow construction renders XFA unsuitable for applications where RegEx updating is frequent. (2) *XFA is not ASIC friendly* because the ASIC implementation of XFA would require much of the complexity of a general purpose CPU to implement the programs associated with states. Such an ASIC chip will not only be overly expensive but also too complex to develop and verify due to the complex interactions among different components. Furthermore, because each XFA state's program may take a very different amount of time to process, in the ASIC implementation of XFA, the processing elements at different stages would be very difficult to pipeline. These two reasons partially explain why XFA has not been implemented in ASIC since its invention. Nevertheless, XFA is no doubt an important RegEx matching solution as it shows the power of DFAs with annotated instructions.

3.1.3 Proposed Approach

In this paper, we propose HASIC, a History-based Finite Automaton (HFA) based RegEx matching scheme that is both fully automated and ASIC friendly. An HFA is a DFA with an auxiliary vector of bits (called *history bits*) where each transition is augmented with a boolean *condition* specified in history bits and some *actions* of setting/clearing history bits.The current HFA state, input character, and history bits jointly determine a single outgoing transition, and following this transition the actions to the history bits are applied.

We now explain why history bits can exponentially reduce the state explosion in DFAs. As a large number of RegExes can be partially matched in parallel, the DFA must track the progress of every combination of partial matches with a separate state. An exponentially large number of partial matches may occur leading to an exponential number of states. In

real-life RegEx sets, many partially matched patterns are persistent; that is, once they have been found, they stay active for a long period of time, waiting to continue matching the rest of their pattern. Such partial matches are often the primary culprit behind the state explosion in DFAs. The bit-vector in HFAs can compactly represent these partial matches and therefore alleviate state explosion. For example, in the HFA in Figure 3.1b built from RegEx set {`EFG`, `X.*Y`}, we use one bit `b` to remember the partial match of character `X` for RegEx `X.*Y`, which eliminates the duplication of the NFA states corresponding to RegEx `EFG`.

Although both HFA and XFA can exponentially reduce state explosion, they are fundamentally different. First, HFA places conditions and actions on transitions, using the history value to direct the operation of the automaton. XFA instead places actions on states and conditions on accepting states. This means that the sequence of states that XFA enters is not affected by its memory values, only whether or not it reports a match when it reaches an accept state. Second, they differ on operations. In HFA, the operations only include three simple bit operations: `test`, `set`, and `clear`, which are very easy to implement in ASIC. In XFA, the operations can be an arbitrary program, which requires CPU complexity to implement in ASIC. When limiting primitive XFA operations to a set of so called "efficiently implementable operations" [28], it is unclear what operations this set should include and whether some set of efficiently implementable operations can guarantee that an XFA can be constructed for any given RegEx.

Compared with XFA, HASIC advances the state of the art from two perspectives. First, HASIC construction can be fully automated (without human annotation). In this paper, we present efficient algorithms to construct an HFA from a RegEx set and then generate an optimized memory image for runtime packet processing. Second, HASIC is ASIC friendly.

HASIC only involves three simple bit operations: `test`, `set`, and `clear`, and only adds a small fixed-size bit array to the state of each flow, which makes storing and restoring the active state of each flow efficient. Furthermore, these operations are combined without the need for a software layer to tie them together, allowing us to represent the processing logic with a simple fixed combination of gates.

3.1.4 Challenges and Proposed Solutions

There are two key challenges in HASIC: (1) automated and fast HFA construction (converting a RegEx set to an HFA), (2) memory image construction (converting an HFA to a memory image) for fast packet processing.

Automated and Fast HFA Construction: Automatic HFA construction is challenging because of two reasons. First, the number of choices for history bits is exponential in the number of NFA states; yet, choosing the right set of history bits is most critical for eliminating state explosion. Second, the number of history bits in a high speed implementation is limited; we must construct an automaton that uses no more than this many history bits. We solve this by identifying NFA states that stay active over many transitions, and ranking them so that we can choose the best states to use. Fast HFA construction is challenging because the intermediate DFA is often exponential (in the number of NFA states) to construct in both time and space; and sometimes the DFA is too big to be practically built. In this paper, we propose a fast and automated HFA construction algorithm whose time and space complexity is linear in the size of the final HFA, which is the best possible complexity for HFA construction. Our key idea is to eliminate the bit states from the NFA before doing subset construction. We can then modify the generated states to compensate, allowing us to avoid exploring all the potential DFA states.

Memory Image Construction for Fast Packet Processing: The number of memory reads per input character for finding the next state(s) is critical for RegEx algorithms to achieve high throughput. It is challenging to minimize the number of memory reads per character in an HFA because there can be multiple HFA transitions for a state and character pair. In this paper, we minimize the number of transitions to be searched by merging compatible transitions into a single transition with the same net effect. To further reduce the size of each transition in memory, we compress the actions by using an auxiliary action mask table to store atomic actions so that the actions for any transition can be represented as the union of these atomic actions; thus, for each transition, instead of storing its actions, we only need to store the indexes of the atomic actions in the action mask table.

3.1.5 Key Novelty and Contributions

The key novelty is in proposing automatic HFA construction algorithms and memory image construction algorithms, where we identify many optimization opportunities. The main contribution is in proposing an automated and ASIC friendly RegEx matching scheme. Specifically, we make the following key contributions. First, we propose an automated and optimized HFA construction algorithm, and then propose a fast HFA construction algorithm with linear complexity in the final HFA size. Second, we propose an optimized memory image construction algorithm for fast runtime packet processing. Finally, we conducted experiments using real-world RegEx sets and various traffic traces. As we cannot construct XFA from our RegEx sets, we estimate the packet processing speed on our hardware based on the results reported in [28]. Our results show that HFA runs an average of 3.34 times faster than XFA. In comparison with DFA, for automata construction speed, HFA is orders of magnitude faster than

DFA; for memory image size, HFA is an average of 20.4 and 16.5 times smaller than DFA for memory bus width of 16 and 32 bytes, respectively.

3.2 Related Work

Existing RegEx matching schemes fall into three categories based on their underlying implementation technology: software based, FPGA based, and TCAM based.

Software-based schemes are generally based on deterministic automata to achieve high throughput. The difference between them is in their approach to solving the DFA state explosion problem. We divide software-based schemes based on whether they introduce auxiliary memory to the automaton or not. Schemes that do not introduce auxiliary memory include D^{2}FA [31], mDFA [32], and Hybrid-FA [33]. D^{2}FA achieves significant compression of the transition table, but does not solve the exponential explosion in the number of states. mDFA and Hybrid-FA avoid building too large a DFA by either building multiple DFA or by producing a partially-deterministic automaton. In both cases, there will be multiple simultaneous active states, causing a large reduction in throughput. Schemes that augment auxiliary memory include XFA [28,29], extended-FA [34], and HFA [35]. XFA and extended-FA both propose hardware designs that are, in broad strokes, a plain DFA that processes all traffic plus a much more complex logic that handles the parts of the RegExes that are too complex to add to the DFA. The complexity of this second layer makes them unsuitable for ASIC implementation. Additionally, the task of coupling these two layers together to achieve guaranteed high performance is even more difficult, as the second layer's processing cost per input byte is highly variable. In [35], Kumar *et al.*briefly proposed the theoretical model of HFA and a manual HFA construction method that requires human experts to design history

bits. However, they proposed neither the methods for the automatic construction of HFAs nor the methods for generating memory images for runtime packet processing. Furthermore, the manual HFA construction method in [35] requires first constructing a DFA from the given RegEx set and then constructing the HFA from the DFA. Although the final HFA is memory efficient, the intermediate DFA is often exponential (in the number of NFA states) to construct in both time and space; and sometimes the DFA is too big to be practically built. Note that the solution for handling large counters in [35] can be applied in our scheme as well.

The field of regular expression matching using FPGA includes a huge breadth of work [36–43]. These techniques all develop a circuit on FPGA that takes in packet data and reports match information. Important to their methods is using the reprogrammability of the FPGA to have the flexibility to handle many pattern sets. These techniques are effective for a fixed pattern set or for environments where the pattern matching tool can be taken offline without penalty. Because the re-synthesis procedure to update the patterns is complex and requires taking the FPGA offline, FPGA solutions have issues with practical deployment in many scenarios. As well, the matching state of many FPGA solutions is large, making it expensive to save and restore this state when matching a large number of interleaved network flows. As an example, in Bando *et al.* [36], the internal state includes 1.5 Kbits of status flags for string matching modules. This makes the handling of interleaved flows much more complex, as saving and loading that state is very expensive.

A newer line of research is the use of TCAM technology to encode pattern matching automata [44, 45]. TCAM are content addressable memory with wild-cards, meaning that a binary string is input, and the set of ternary patterns stored in the memory are checked to find the first pattern that matches the query. An SRAM associated with the TCAM

allows a value to be associated with each pattern in the TCAM. The transition table of an automaton can be implemented by creating query strings that indicate the current state and input character and storing the destination state of the transition in the SRAM. The downside of TCAM is its high power usage, as every query is matched against every pattern.

3.3 Automatic HFA Construction

3.3.1 Basic Construction Method

The original algorithm to construct an HFA from a RegEx set first constructs an NFA and uses subset construction to create the corresponding DFA. The label of each DFA state is the subset of NFA states that the DFA state is constructed from. Second, the DFA is "folded" into an HFA by repeatedly turning a single NFA state into a history bit, removing this label from all DFA states, and merging those DFA states that now have identical NFA state labels. We call the removed NFA states "*bit states*". To convert an NFA state s into a bit state, we partition the DFA into two groups: P, which consists of the states that have the NFA state s in their label, and N, which consists of those that do not. For example, to convert NFA state 4 in Figure 3.1a into a bit state, we partition the DFA in Figure 3.1c into two groups: $P = \{4, 5, 6, 7, 8\}$ and $N = \{0, 1, 2, 3\}$, those states that have the NFA state 4 and those that do not. We use an example state label $\frac{0,3,4}{7/1}$ to explain the way that we label DFA (or HFA) state in this paper: $0, 3, 4$ is the set of NFA states that corresponds to this DFA state, 7 is the DFA state ID, and /1 denotes that this is an accepting state and the RegEx ID 1 (namely `EFG`) is matched upon reaching this state. Removing label s from each state in P will allow us to merge each state and their transitions in group P with the corresponding state and transitions in N. For example, removing NFA state label 4 from each DFA state

in $P = \{4, 5, 6, 7, 8\}$ will allow us to pairwise merge DFA states 4 and 0, 5 and 1, 6 and 2, and 7 and 3.

Transitions leaving a state in P can now be taken when the corresponding history bit for N is set and transitions leaving a state in N can only be taken when that bit is clear. For example, in Figure 3.1b, the transitions from HFA states 1, 2, and 3 to state 4 on character `Y` can only be taken when b is set. The transitions that were going from P to N must clear the history bit, and from N to P must set it. For example, in Figure 3.1b, the transition from HFA state 0 to 0 on character `X` corresponds to the transition from DFA state 0 to 4. Because this transition goes from N to P, it sets bit b. Transitions that stay within a group do not need an action to modify the history bits. For example, in Figure 3.1b, in the transitions among HFA states 0, 1, 2, and 3, there is no action to modify bit b. Repeating the above process constructs an HFA with multiple history bits.

3.3.2 Bit State Selection

Choosing bit states is critical for HFA. The best case for compressing the DFA by converting an NFA state s into a history bit is when the DFA states that include s in their labels exactly mirror the DFA states that do not, which allows us to halve the number of automaton states. For example, in Figure 3.1a, by choosing NFA state 4 to be a bit state, we almost halve the number of HFA states.

Before presenting our method for choosing the right bit states, we first introduce some new concepts. The *self-looping degree* of an NFA state is defined as the percentage of the number of input characters that the state transitions to itself on. An NFA state is complete-self-looping if its self-looping degree is 100%. For example, in Figure 3.1a, both states 0 and 4 are complete-self-looping. RegExes with `.*` cause most complete-self-looping states. *Once*

a complete-self-looping becomes active, it remains active. An NFA state s_1 *shadows* another state s_2 if and only if every time when state s_2 is active, s_1 is also active. For example, in Figure 3.1a, state 0 shadows every other state, and state 4 shadows state 5.

The shadowing relationship reduces state explosion by eliminating the combinations of states for which we need to generate new DFA states. For example, in Figure 3.1a, we do not need to generate a new DFA state for the combination of states $\{0, 5\}$ because whenever state 5 is active, state 4 is active. Two NFA states s_1 and s_2 are *exclusive* if and only if they cannot be simultaneously active. For example, in Figure 3.1a, states 1, 2, and 3 are mutually exclusive. *Exclusive relationship also reduces state explosion* by eliminating the combinations of states that we need to generate new DFA states for. For example, in Figure 3.1a, we do not need to generate a new DFA state for the combination of states $\{1, 2\}$ because they cannot be simultaneously active. NFA states s_1 and s_2 are *independent* if and only if two conditions are satisfied: (1) there is no shadowing relationship between them, and (2) they are not exclusive.

Independent states cause state explosion in DFAs. Given an NFA with n independent persistent states and m other states, using $d(m)$ to denote the size of the DFA constructed only from the m states, the size of the final DFA is in the order of $2^n * d(m)$. This comes from the $d(m)$ states being copied for all 2^n ways for the independent states to be active.

For an NFA state that is independent from most of other states, if we choose it to be a bit state, then we almost halve the DFA size. As NFA states with a high self-looping degree tend to remain active for a long time, we use the NFA states with a high self-looping degree that are shadowed by complete-self-looping states as bit states, as these are likely to be independent with a large number of other states.

3.3.3 HFA Construction without DFA

In constructing an HFA from a RegEx set, the intermediate DFA may be too big to be practically generated due to state explosion, even if the final HFA is small. Next, we present our HFA construction algorithm that can directly build the HFA from an NFA without generating and storing the intermediate DFA. Before we present our algorithm, we first introduce a new concept of *equivalent state classes*: given an NFA with a set of bit states B, for two DFA states that correspond to two sets of NFA states S_1 and S_2, if the two NFA state sets only differ on bit states (*i.e.*, $S_1 - B = S_2 - B$), then the two DFA states are *equivalent with respect to bit state set* B. This relationship partitions the DFA states into equivalence classes. Considering the example in Figure 4.3, when choosing NFA state 4 as a bit state, the DFA states 1 and 5 are in the same equivalence class as they correspond to NFA state sets {0, 1} and {0, 1, 4}, respectively.

The this HFA construction algorithm is similar to the standard subset construction algorithm as in DFA construction, but it only generates one HFA state per equivalence class. Let B be the set of bit states. Each time we generate a DFA state that corresponds to a set of NFA states S, we append its transitions to the HFA state $S - B$. For each transition from DFA state S to DFA state D, we add an HFA transition, with a condition and an action, from $S - B$ to $D - B$ on the same character. The condition corresponds to $S \cap B$, meaning that this transition can only be taken when the history bits corresponding to $S \cap B$ are set and the remaining history bits are clear. The action consists of two parts: setting the history bits corresponding to $(D - S) \cap B$ and clearing the history bits corresponding to $(S - D) \cap B$.

An HFA with a vector H of k history bits, has 5-tuple transitions $(S, c, \mathbb{C}, \mathbb{A}, D)$ which are the source state, input character, condition, action, and destination state. The *source*

state and *destination state* of an HFA are HFA states, which are written as a set of NFA states. The *condition* of an HFA transition is represented as a vector $\mathbb{C}$ of k ternary bits, denoting the condition $\wedge_{i=0}^{k-1}(\mathbb{C}[i] = H[i])$. A ternary bit has three possible values: 0, 1, or *. Note that for each $0 \leq i \leq k-1$, when $\mathbb{C}[i]$ is *, $\mathbb{C}[i] = H[i]$ is true regardless of the value of $H[i]$. The *action* of an HFA transition is represented as a vector $\mathbb{A}$ of k bit-wise operations. Each bit-wise operation is either `set` (denoted as `s`), `clear` (denoted as `c`), or `do-nothing` (denoted as `n`). For each $0 \leq i \leq k-1$, $\mathbb{A}[i] =$ `s` means the action assigns 1 (sets) to $H[i]$, $\mathbb{A}[i] =$ `c` means the action assigns 0 (clears) to $H[i]$, and $\mathbb{A}[i] =$ `n` means the action does nothing to $H[i]$. Table 3.1 shows example transitions with 3 history bits for a HFA state and character pair. The pseudocode of this HFA construction algorithm, called HASIC$_D$ is shown in the appendix.

Condition	Action	Destination State
*00	nnn	{1}
*10	ncn	{1}
*01	nnn	{1, 5}
*11	ncn	{1, 5}

Table 3.1: Example transitions before optimization

3.3.4 Transition Table Optimization

The above HFA construction algorithm avoids DFA state explosion by merging many variants of the same NFA states into one HFA state; however, all it adds each DFA transition to the HFA. Next, we introduce our HFA transition table optimization algorithm that can efficiently store transitions while allowing fast transition lookup. We first introduce a new concept called *mergeable bit actions*. Given an action a on a ternary bit t, we use $a(t)$ to denote the resulting value after applying action a on t. Two action and ternary bit pairs (a_1, t_1) and (a_2, t_2) are

mergeable if and only if there exists an action a_3 so that $a_1(t_1) = a_3(t_1)$ and $a_2(t_2) = a_3(t_2)$. We call a_3 a *merged action* of (a_1, t_1) and (a_2, t_2). For example, $(\texttt{n}, 0)$ and $(\texttt{s}, 1)$ are mergeable and the merged action is `n`. One merging may have two merged actions, either `n` and `c` or `n` and `s`. For example, action `n` on bit 0 and action `c` on bit 0 have two possible merged actions: `n` and `c`. In such cases, we choose `n` to be the merged action because it has less bit operations than `c` or `s`. Note that the choice of merged actions can never be between `c` and `s` because their results are always different. Table 3.2 shows the merged actions of all possible pairs of bit-action pair, where - denotes that the two bit-action pairs are not mergeable.

	n0	n1	n*	c0	c1	c*	s0	s1	s*
n0	n	n	n	n	c	c	-	n	-
n1	n	n	n	n	-	-	s	n	s
n*	n	n	n	n	-	-	-	n	-
c0	n	n	n	n	c	c	-	n	-
c1	c	-	-	c	c	c	-	-	-
c*	c	-	-	c	c	c	-	-	-
s0	-	s	-	-	-	-	s	s	s
s1	n	n	n	n	-	-	s	n	s
s*	-	s	-	-	-	-	s	s	s

Table 3.2: HFA transition mergeability table

Now we introduce another concept called *mergeable transitions* and discuss how to minimize HFA transitions by identifying and merging such transitions. In an HFA with k history bits, a transition $(S, c, \mathbb{C}_1, \mathbb{A}_1, D)$ and a transition $(S, c, \mathbb{C}_2, \mathbb{A}_2, D)$ are *mergeable* if and only if both of the following conditions are satisfied: (1) $\mathbb{C}_1$ and $\mathbb{C}_2$ differ in only one bit and (2) for $0 \leq i \leq k-1$, $(\mathbb{A}_1[i], \mathbb{C}_1[i])$ and $(\mathbb{A}_2[i], \mathbb{C}_2[i])$ are mergeable. For these two mergeable transitions, assuming $\mathbb{C}_1$ and $\mathbb{C}_2$ differ in bit i, we merge them into one rule by both replacing $\mathbb{C}_1[i]$ by $*$ and replacing $\mathbb{A}_1[i]$ by the merged action from Table 3.2. For example, in Table 3.1, the first two transitions and the last two transitions are mergeable. Table 3.3

shows the two merged actions. This optimization allows many HFA transitions to be stored in a small memory while allowing fast packet processing.

Condition	Action	Destination State
**0	ncn	{ 1 }
**1	ncn	{ 1, 5 }

Table 3.3: Table 3.1 transitions after optimization

3.4 Fast HFA Construction

In this section, we propose the first algorithm to generate deterministic automata without exploring all the possible NFA state interactions. Compared with the HFA construction algorithm in Section 3.3, this algorithm runs significantly faster at the price of producing a possibly larger HFA.

3.4.1 Observation and Basic Ideas

In observing the HFAs that our previous algorithm constructed from real-life RegEx sets, we observe that many HFA states have exactly 2^n outgoing transitions for a particular input character where n is the number of bit states. This occurs when n bit states have departing transitions to distinct NFA states on the same input character. When we merge all the transitions from all the DFA states in an equivalence class, our previous algorithm has to create a conditional transition for each reachable combination of history bits. This constitutes a significant portion of the HFA construction time. To speed up HFA construction, our key idea is to simply assume each bit state is independent from all other states; thus, we can precompute an incoming transition table and an outgoing transition table, which we call the

mixin incoming table and *mixin outgoing table*, respectively. These two tables consist of all the transitions introduced by all combinations of bit states. Then, we can mix this table with the transition table of each HFA state. This may introduce some transitions that can never be taken, but they do not affect the correct execution of the automaton. We choose the term "mixin" because of its resemblance of the use of mixin classes in object-oriented programming, where a mixin is a class that provides a certain functionality to be inherited or reused by derived classes.

To construct an HFA from an NFA using this method, we first identify bit states as described earlier. Second, we generate a *mixin incoming table* that is constructed from all incoming transitions to bit states and a *mixin outgoing table* that is constructed from all outgoing transitions from bit states. Third, we remove all bit states and their incoming and outgoing transitions from the NFA to produce a *pruned NFA* that has only non-bit states. Finally, we generate HFA states using subset construction on the pruned NFA, "mixing in" the transition information from the two mixin tables.

To illustrate this process, we will show step by step conversion of the example NFA in Figure 3.2 into the HFA in Figure 3.5.

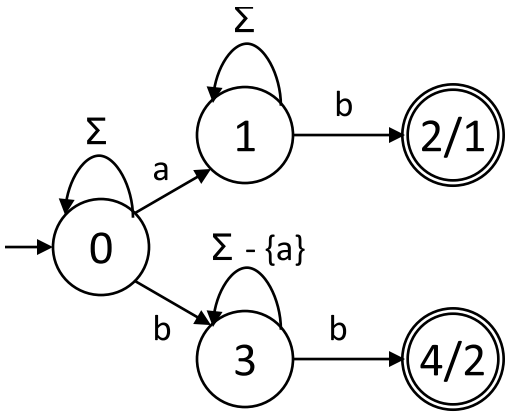


Figure 3.2: Input NFA

3.4.2 Bit State Pruning

Given an NFA and its bit states, we produce a pruned NFA by removing the bit states from the NFA as well as all their incoming and outgoing transitions. The information about these bit states, which is missing in the pruned NFA, will be kept in two mixin tables. When a history bit is set, this impacts the action of a transition and/or its destination. The mixin tables capture both of these effects and allow us to apply them to the full HFA. In the NFA in Figure 3.2, which is constructed from the regular expressions `a.*b` and `b[^a]*b`, we choose states 1 and 3 as the bit states, which means that we have two history bits that correspond to NFA states 1 and 3. The pruned HFA is shown in Figure 3.3.

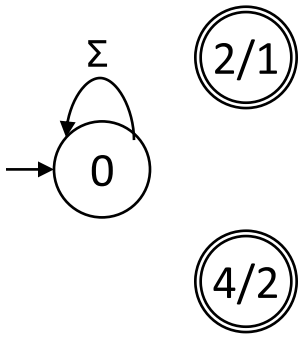


Figure 3.3: Pruned NFA

3.4.3 Mixin Table Generation

We use two tables to store the information about bit states: the mixin incoming table and the mixin outgoing table. For the mixin incoming table, we generate it directly from the NFA. For the mixin outgoing table, we first generate an outgoing table for each bit state and then merge them into one table. Note that these two tables are only used for HFA construction and they are not part of the final HFA memory image.

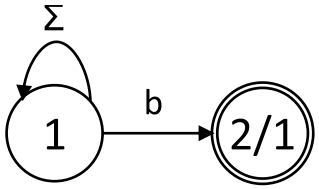
To generate the mixin incoming table, we simply record all the transitions from non-bit states to bit states. Figure 3.4 shows the mixin incoming table for the NFA in Figure 3.2, whose entries are three tuples, $(q, c, \mathbb{A})$. Note that the source field of this table is a single NFA state. The first entry shown comes from the incoming transition to NFA state 1 on input character a. The source field of the transition is 0, indicating that this is only available from HFA states constructed from NFA state 0. The action "`sn`" means that we should set the first history bit and do nothing to the second history bit. Similarly, on b, the action "`ns`" sets the second history bit.

Src.	input	Act.
0	a	`sn`
	b	`ns`

Table 3.4: Mixin Incoming Table

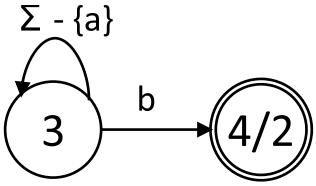
The outgoing table for a bit state has entries that are 4-tuples, $(c, b, \mathbb{A}, D)$, representing an input character, single history bit, action, and destination state. The input character and history bit value uniquely determine an action and a destination state. For input characters that are not shown, the default entry is an action that does nothing and an empty destination state, marked "else". To generate the mixin outgoing table entries for a bit state, we first examine its outgoing transitions that are not looping back to itself. Figure 3.5 shows the mixin outgoing table of bit state 1, which has a transition on b to state 2. The corresponding entry in this table means that in the final HFA, whenever bit state 1 is active, on character b, we take no additional action and we transition to a combination of NFA states containing state 2. For transitions that leave a bit state to go to another bit state, the outgoing table entry sets the history bit for the destination state instead of putting that state in the destination set. When processing an input character for which some bit state has a self-looping transition, the

corresponding history bit will not change. But we must clear the history bit upon processing a character lacking a self-loop transition. Figure 3.6 shows an example of this: bit state 3 does not have a transition to itself on character a, thus we need the first entry in its outgoing table, which clears the history bit for state 3 when it is set.



	input	Cond.	Act.	Dest.
	b	1	nn	{2}
		0	nn	∅
else:	*	*	nn	∅

Table 3.5: Bit State 1 Outgoing Table



	input	Cond.	Act.	Dest.
	a	1	nc	∅
		0	nn	∅
	b	1	nn	{4}
		0	nn	∅
else:	*	**	nn	∅

Table 3.6: Bit State 3 Outgoing Table

After we generate a mixin outgoing table for each bit state, we merge these individual outgoing tables into a single mixin outgoing table, which will be used during the subset construction process. The entries in this table are 4-tuples, $(c, \mathbb{C}, \mathbb{A}, D)$, similar to the outgoing table for a single state, but with k-bit conditions. Given multiple individual mixin outgoing tables, we construct all combinations of rules in the input tables that have the same input character, σ. The resulting destination of two rules is the union of their destination state sets. The rule for merging actions is based on NFA semantics, where a state becoming active takes precedence over that state becoming inactive from the lack of a transition. Thus, the result of merging two actions is `s` if either input is `s`, `c` if some action is `c`, otherwise `n`. For

example, the first entry in Figure 3.4 is the result of merging the entry for a in Figure 3.6 with the default entry from Figure 3.5. For actions, we have `nn`+`nc` = `nc` as the merging result. For destinations, we have $\emptyset \cup \emptyset = \emptyset$ as the merging result. Since we have two entries on b in each table, the mixin outgoing table will have 4 entries, as shown in Figure 3.4.

	input	Cond.	Act.	Dest.
	a	*1	`nc`	$\emptyset$
		*0	`nn`	$\emptyset$
	b	00	`nn`	$\emptyset$
		10	`nn`	{2}
		01	`nn`	{4}
		11	`nn`	{2,4}
else:	*	**	`nn`	$\emptyset$

Figure 3.4: Mixin Outgoing Table for 1&3

3.4.4 HFA Transition Table Generation

We now construct HFA from the pruned NFA, not the original NFA. We still use subset construction considering all possible combination of states in the pruned NFA. But note that the number of all combinations of pruned NFA states is orders of magnitude smaller than that of the original NFA. This explains why this HFA construction method is much faster than the one in Section 3.3. The start state of the generated HFA corresponds to the start state of the NFA without any bit states active. The starting value of the history bits has all bits cleared except those corresponding to history states in the start state of the NFA.

In generating transitions, we mix in the transitions in the mixin transition table as follows. The transitions generated from the pruned NFA, which has no bit states, can be represented as 3-tuples: *source NFA state set*, *input character*, and *destination NFA state set*. Let (S, c, D) denote a transition generated from the pruned NFA. The transitions in the mixin incoming table can be represented as 3-tuples: *source NFA state*, *input character*, and *action*. Let

$(q_i, c_i, \mathbb{A}_i)$ $(1 \leq i \leq m)$ denote the m entries of the mixin incoming table with $c_i = c$ and $q_i \in S$. We will merge all these actions into the result, so we can write M as the result of merging all the $\mathbb{A}_i$. Recall that the result of merging two actions is `s` if either input is `s`, `c` if some action is `c`, otherwise `n`. Let $(c_j, \mathbb{C}_j, \mathbb{A}_j, D_j)$ $(1 \leq i \leq n)$ denote the n entries of the mixin outgoing table with $c_j = c$. In generating the transitions for each NFA state, we merge the destination set constructed from the pruned NFA with the two mixin tables in a manner similar to how we merge individual mixin outgoing tables. For each entry $(c_j, \mathbb{C}_j, \mathbb{A}_j, D_j)$ in the mixin outgoing table, we create an HFA transition $(S, c, C_j, M + \mathbb{A}_j, D \cup D_j)$.

In our example, the subset construction process generates an HFA state for NFA state $\{0\}$ and constructs the transition $(\{0\}, \mathrm{a}, \{0\})$. This is augmented by the mixin tables to become $(\{0\}, \mathrm{a}, *1, \mathtt{sc}, \{0\})$ and $(\{0\}, \mathrm{a}, *0, \mathtt{sn}, \{0\})$, which can be compressed into the transition from HFA state 0 to itself on input character a, setting the history bit for bit state 1 and clearing the history bit for bit state 3. The subset construction process also constructs a similar transition for `b`: $(\{0\}, \mathrm{b}, \{0\})$. This transition is expanded to four transitions, each with different destinations: $(\{0\}, \mathrm{b}, 00, \mathtt{nn}, \{0\})$, $(\{0\}, \mathrm{b}, 10, \mathtt{nn}, \{0, 2\})$, $(\{0\}, \mathrm{b}, 01, \mathtt{nn}, \{0, 4\})$, $(\{0\}, \mathrm{b}, 11, \mathtt{nn}, \{0, 2, 4\})$. The newly reachable states from $\{0\}$ have their transition tables generated in the same manner until all states are constructed. Applying this algorithm to the pruned NFA in Figure 3.3, the Mixin Incoming Table in Figure 3.4 and the Mixin Outgoing Table in Figure 3.4 produces the HFA in Figure 3.5.

The work necessary to produce HFA by this process is not proportional to the size of the DFA. When we prune the bit states that cause exponential increase in DFA size and construct HFA using mixin tables, we avoid enumerating exponentially many combinations of NFA states. Our experimental results show that this fast HFA construction algorithm

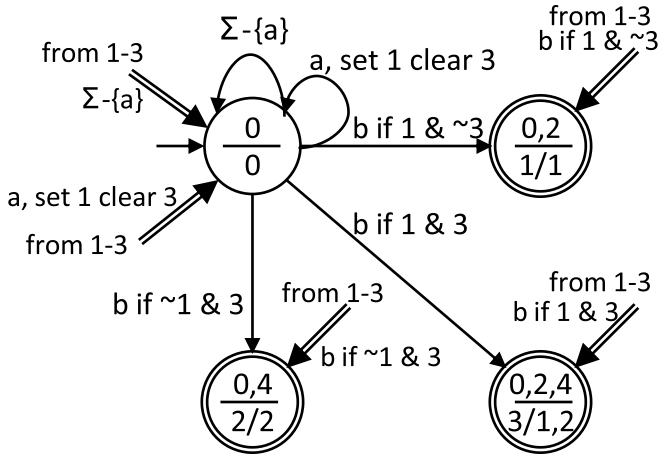


Figure 3.5: Output HFA

produces HFAs of similar size as our previous HFA construction algorithm at orders of magnitude higher speed.

3.4.5 Correctness of Fast HFA Construction

Next, we prove the correctness of our fast HFA construction algorithm. We first introduce some new concepts and notations. The *configuration* of an automaton means the entirety of its internal state. For a DFA, this is the single active state, for an NFA, it is the set of active states, and for HFA it is both the active state and the value of the history bits. The configuration of an HFA after processing some input can be written (S, H), where S is the active HFA state, identified by its equivalent NFA state set, and H is the current history bit vector, also identified with a set of bit states. We can partition the configuration of an NFA into $\{S, H\}$, where S is the set of active non-bit states and H is the set of active bit states. For an automaton (HFA or NFA) A, we can use $A : S \xrightarrow{c} D$ to indicate that automaton A transitions from configuration S to configuration D on input character c. Further, $A \overset{s}{\rightsquigarrow} D$ indicates that D is the final configuration after processing string s with automaton A, starting from the automaton's initial state. Theorem 3.4.1 states the

correctness of our fast HFA construction algorithm. For an action A and history bit vector H, we use $A(H)$ to denote the resulting history bit vector after applying action A on H. Because a history bit vector uniquely defines a set of bit states, we also use H and $A(H)$ to denote the bit state sets defined by them.

Theorem 3.4.1

For any NFA N with state set Q and alphabet Σ, string s over Σ, character $c \in \Sigma$, state sets $S_1, H_1, S_2, H_2 \subseteq Q$, and HFA H constructed by our fast HFA construction algorithm from N, if $N \overset{s}{\rightsquigarrow} \{S_1, H_1\}$, $N : \{S_1, H_1\} \overset{c}{\rightarrow} \{S_2, H_2\}$, and $H \overset{s}{\rightsquigarrow} (S_1, H_1)$, then we have $H : (S_1, H_1) \overset{c}{\rightarrow} (S_2, H_2)$.

Proof 3.4.1

We must show that the HFA transition for (S_1, H_1) on character c has destination state S_2 and action A such that A applied to H_1 results in H_2. We will do this by decomposing the NFA transition based on the bit states and showing how each part is mirrored in the HFA construction. Using S to indicate non-bit states and H to indicate bit states, the HFA must account for the effects $S \rightarrow S$, $S \rightarrow H$, $H \rightarrow S$ and $H \rightarrow H$. We divide the NFA transition $N : \{S_1, H_1\} \overset{c}{\rightarrow} \{S_2, H_2\}$ into two partial transitions: $N : S_1 \overset{c}{\rightarrow} \{S_a, H_a\}$ consisting of transitions from active non-bit states in S_1 and $N : H_1 \overset{c}{\rightarrow} \{S_b, H_b\}$ consisting of transitions from active bit states in H_1. Note that $S_a \cup S_b = S_2$ and $H_a \cup H_b = H_2$.

$S \rightarrow S$: Because S_1 is reachable, the subset construction process will generate exactly S_a as the destination on c from S_1.

$H \rightarrow S$: The mixin outgoing table will generate exactly S_b as it includes each non-bit state destination of a transition on c from all the active bit states.

$S \to H$: The mixin incoming table contributes M, which is the combined actions of all NFA transitions on c from S_1 to bit states, so the result of these actions will ensure that $M(\emptyset) = H_a$.

$H \to H$: The mixin outgoing table contributes an action $\mathbb{A}$ that simulates transitions on c from bit states to bit states (themselves and other bit states), corresponding to $H_1 \to H_b$ in the NFA.

The destination state of HFA transitions consists of the NFA states calculated by the subset construction procedure and the NFA states obtained from the mixin destination table. Thus, the HFA destination state is $S_a \cup S_b = S_2$. The action for HFA transitions is constructed from the mixin incoming table and the mixin outgoing table. By the semantics of action merging, their combination $M + \mathbb{A}$ has the property $(M + \mathbb{A})(H_1) = M(\emptyset) \cup \mathbb{A}(H_1) = H_a \cup H_b = H_2$. ■

Theorem 3.4.2

The HFA generated by mixin construction simulates the NFA it was generated from correctly.

Proof 3.4.2

We will proceed by induction on the input string using Lemma 3.4.1. Given an input string s, NFA N and HFA H constructed from N by fast HFA construction algorithm, we will show that if $N \overset{s}{\rightsquigarrow} \{S_1, H_1\}$ then $H \overset{s}{\rightsquigarrow} (S_1, H_1)$. Base case: s is the empty string. In this case, the initial state of the HFA corresponds to the initial state of the NFA by construction. Recursive case: s can be decomposed into $s' + c$, where c is the final character of s. Let $N \overset{s'}{\rightsquigarrow} \{S_2, H_2\}$ define the state that N reaches after processing s'. From the recursive hypothesis, $H \overset{s'}{\rightsquigarrow} (S_2, H_2)$. Using Theorem 3.4.1 to make the final transition on character c, we conclude that $H \overset{s}{\rightsquigarrow} (S_1, H_1)$. ■

3.5 Fast Packet Processing

Having constructed an HFA, we need to create a memory image to represent this HFA so that it can be used for runtime fast packet processing. In this section, we first examine design considerations for an HFA memory layout to achieve high throughput with low required memory bandwidth. Then, we introduce a compact format for an individual HFA transition, and present a data structure for storing HFA transitions. Finally, we explore optimizations on transition ordering. The pseudocode of our packet processing algorithm is shown in the appendix. Note that if RegExes are specified in Unicode, we can convert them into RegExes over bytes.

3.5.1 Design Considerations

The key optimization metric in designing the memory layout for HFA is to minimize the average number of memory accesses per input character because the key limiting factor for packet processing performance in routers/IPSes is the memory bandwidth of random accesses. Furthermore, we want to bound the computational complexity required to calculate the next state per input character to be small because simpler chip circuitry leads to higher clock rate, smaller die area, and reduced power consumption.

3.5.2 Transition Format

To best suit ASIC implementation, we propose two formats to encode state transitions so that the required processing includes only ASIC friendly operations such as bitwise logical operations and shifting.For a 128-bit memory bus and 32 history bits, we propose the transition format with a total of 128 bits as shown in Figure 3.6. We choose 128 as the number

of bits for encoding each transition because memory buses in routers/IPSes are often 128-bit and we want to minimize the number of memory accesses for each input character. Here, the first 32 bits constitute the condition mask, the second 32 bits constitute the condition value, and together they represent a ternary condition. Thus, testing whether the current history bit-vector satisfies such a ternary condition requires only simple bitwise logical operations. The third 32 bits encode the ID of the next state and some flags (where an example flag is for indicating whether the next state is an accepting state). The last 32 bits encode some actions (for setting/clearing history bits) that can be executed in parallel upon taking this transition. The format of each action is shown as "Action Format" below the transition format in Figure 3.6. The first bit in an action indicates whether that action is a valid one that needs to be taken. The second bit in an action indicates whether the action is for setting bits or clearing bits. The remaining six bits form the index of the entry that specifies the target bits that need to be set or cleared in a table that we call the *action mask table*. The action mask table consists of $64(= 2^6)$ entries where each entry has 32 bits and is called an *action mask*. The i-th bit in an action mask being one indicates that the corresponding i-th bit in the history is a target for this action.

For other memory bus width and number of history bits, we can design transition formats similarly. For example, the bottom of Figure 3.6 has a transition format for a 64-bit memory bus and 16 history bits. Compared with the 128-bit format, this format has only two actions, no extra flag space and a smaller next state id field, but it is able to represent very complex automata.

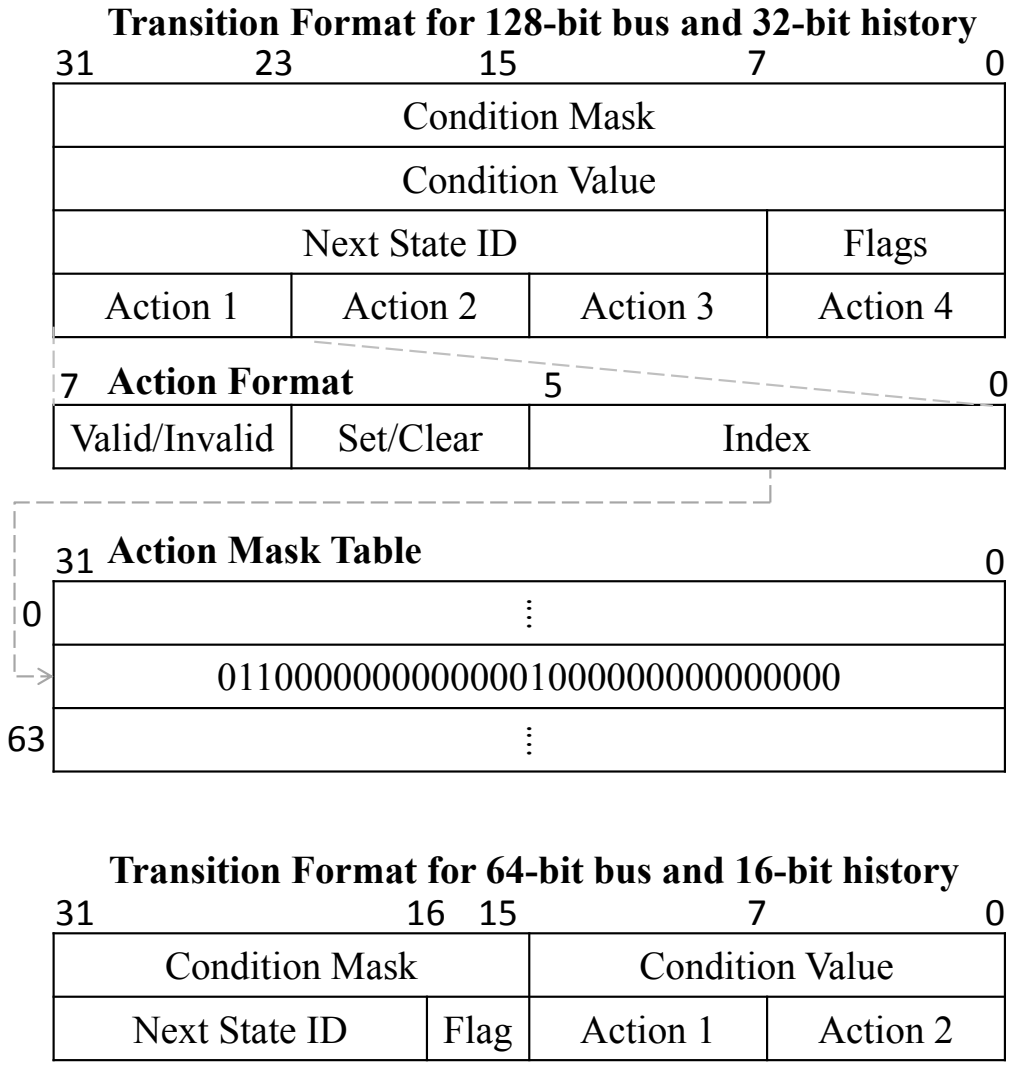


Figure 3.6: Transition Formats

3.5.3 Action Compression Algorithm

Each HFA has one action mask table whose width is the number of history bits. If there is no limit on the number of entries that this table can have, then we can store each distinct action mask used by the HFA. However, we have to limit this number because the index field in each action has a fixed number of bits. Generating such a fixed size table for an HFA of arbitrary size becomes technically challenging. Next, we present our action compression

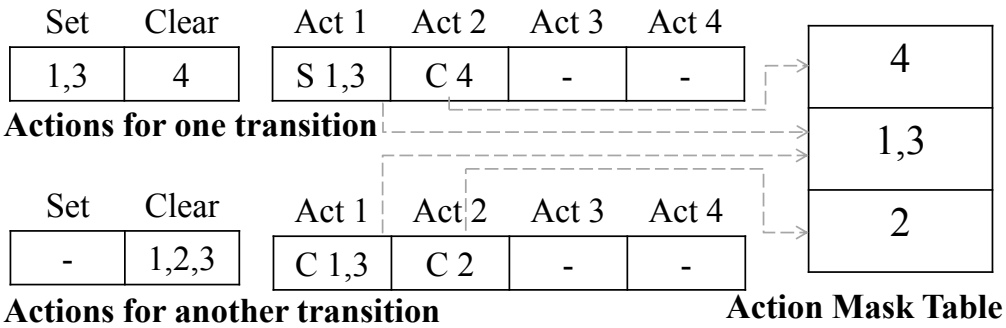


Figure 3.7: Action Mask Example

algorithm for generating the action mask table for an HFA using the transition format in Figure 3.6.

Let us first have a deeper understanding of the action mask table generation problem. The actions taken upon an HFA transition are setting and/or clearing some history bits. We want to generate the action mask table for an HFA so that the following two conditions hold. First, performing the actions for each HFA transition can be accomplished by performing no more than four sub-actions where each sub-action is either setting or clearing the history bits indicated by an action mask table entry. Second, the number of sub-actions for each HFA transition is maximum four because there are a total of four actions in the transition format. This problem is a special case of the known NP-complete problem called *set basis* [46]: given two sets of finite sets, S and B, we say B is a *set basis* for S if and only if each set in S can be represented as the union of some elements of B; the set basis problem asks for a set S and positive integer k, whether there exists a set basis of size no greater than k. The major differences lie in that we have a limit of four and there are two types of actions (of set and clear).

Our action compression algorithm for generating the action mask table of an HFA runs in a greedy fashion. The input is a set of *set action set* and *clear action set* pairs where each pair corresponds to an HFA transition, each set action set contains all history bits to

be set, and each clear action set contains all history bits to be cleared. First, we search for the smallest action set among all action sets and create an action mask corresponding to that action set. Second, we subtract this smallest action set from all remaining action sets that are the supersets of this smallest action set and remove all duplicate action sets. We repeat the above two steps until all action sets become either empty or the action sets of each transition cannot be further decomposed. For example, suppose one HFA transition has the actions of setting bits $\{1, 3, 5, 7\}$ and clearing bits $\{2, 4, 6\}$. If we have created one action mask entry for $\{1\}$ and another entry for $\{3\}$, then we must create two action mask entries, one for $\{5, 7\}$ and one for $\{2, 4, 6\}$ because any further decomposition of the two sets $\{5, 7\}$ and $\{2, 4, 6\}$ will result in more than four action mask entries for this transition.

Due to the limitation of 64 entries imposed by our transition table format, our algorithm does not guarantee the successful generation of the action mask table for any HFA, although we have not encountered such failures in our experiments with even complex RegEx sets. There are two simple solutions to this problem. One is to increase the number of bits allocated for encoding each transition. For example, if we allocate 256 bits to each transition instead of 128 bits, with 64-bit history and 4 actions, we can have 12 bits for the index for each action, which means that the action mask table can have 2^{12} entries instead of 2^6 entries. Of course, this will increase the number of memory accesses per input character. This solution can be adopted in the ASIC design phase in trading off efficiency and capability. Another solution is to split the input RegEx set into two RegEx sets and build two HFAs and the corresponding two memory images. For each input character, we will give it to both HFAs. This solution can be adopted after ASIC has been fabricated.

3.5.4 Transition Table Image Construction

Next, we present our data structure for storing all the transitions of an HFA. The complexity is due to the fact that at any HFA state, for any possible input character, there may be multiple transitions, one of which will be taken based on the current value of history bits. Different HFA state and input character pairs may have different number of transitions. For an HFA with $|Q|$ states over the alphabet Σ, the natural solution for storing all the transitions is to use an array T of size $|Q|*|\Sigma|$, where for state Q_i and character σ, treating σ as an integer $T[i*|\Sigma|+\sigma]$ stores a pointer that points to a linked list of all the corresponding transitions. However, linked lists are inefficient for high speed networking and security devices for two main reasons. First, pointers consume memory. As our transition format does not leave room for storing such pointers, we have to use extra memory to store pointers, which may cause more memory access per transition. Second, traversing a linked list leads to random accesses of noncontiguous memory blocks, which is less efficient than sequential memory reads. Based on the fact that for any HFA state and any possible input character, there is always a transition that the current value of history bits matches, we can store all transitions of an HFA in one array S where the transitions for each state and input character pair are stored as one block in the array. Thus, for any state Q_i and character σ, $T[i*|\Sigma|+\sigma]$ simply stores the index of their first transition in S. When processing character σ at state Q_i, we first fetch transition $S[T[i*|\Sigma|+\sigma]]$ and check whether the current history matches the condition of the transition. If it matches, then we consume the character and move to the next state; otherwise, we fetch transition the next transition $S[T[i*|\Sigma|+\sigma]+1]$ and repeat the above process.

By the above design of the ragged array T, for any state Q_i and character σ, there is always one extra memory access for fetching $T[i * |\Sigma| + \sigma]$ in addition to sequentially fetching their transitions for S. To reduce memory accesses, we move the first $k \geq 1$ transitions for each state Q_i and character σ to $T'[i * |\Sigma| + \sigma]$, where T' is a new two dimensional array T' with a height of $|Q| * |\Sigma|$ (the same as T) and a width of k. If state Q_i and character σ have less than k transitions, then we leave unused transitions empty in T'. Thus, when processing character σ at state Q_i, we first sequentially fetch the k transitions from $T'[i * |\Sigma| + \sigma]$ until we hit a match; if the k transitions result in no matches, we follow the above process of fetching transitions starting from $S[T[i * |\Sigma| + \sigma]]$ until we hit a match. There is a design tradeoff for the value of k. The larger the k is, the more empty transitions T' has. The smaller the k is, the higher chance that the extra memory access for reading the index from T occurs. Figure 3.8 illustrates this tradeoff. In our experiments, $k = 1$ provides the best tradeoff between memory size and throughput for real life HFAs.

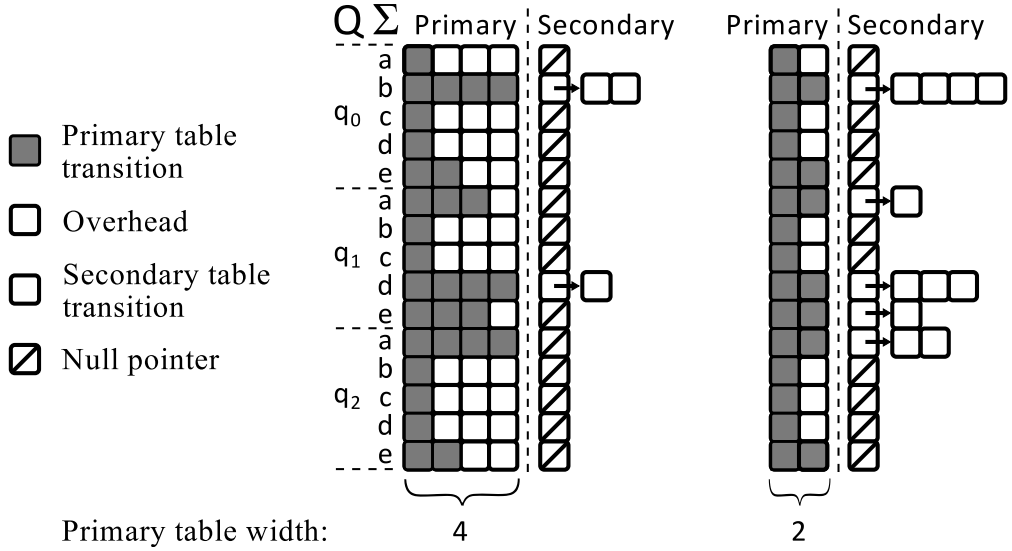


Figure 3.8: Effects of Table Width

As we sequentially read the transitions for a state-character pair until a match is found and different transitions have different probabilities of matching the history, we can reduce

memory accesses by ordering the transitions in the decreasing order of hit probabilities. Note that for any state-character pair, we can freely order its transitions because they are all non-overlapping. The matching probability of each transition can be estimated using past traffic. The transitions in both T' and S can be dynamically reordered at runtime to adapt to traffic conditions.

3.6 Hardware Design

The HFA constructed from our improved algorithm, the corresponding memory layout, and the DPI engine have been designed to allow efficient hardware implementation in ASIC or Field Programmable Gate Arrays (FPGA). This is ensured with a structured memory layout that systematically organizes the state transitions, conditions and actions of the HFA, and minimum logical complexity involved in the inspection engine that evaluates the transition conditions and performs appropriate actions. In this section, we present a reference hardware design of a HFA-based DPI engine that can be coupled with a variety of memory technologies or on-chip cache hierarchies to store the DFA. This design accommodates wide variations in the memory bandwidth and latencies, providing the flexibility to be deployed in a variety of platforms and achieve optimal DPI throughput.

The DPI engine receives the input character stream and performs memory operations for state transitions. It consists of two interfaces: a host interface to receive DPI requests and return the match responses, and a standard memory interface to load state transition information. The host interface receives DPI requests that contain the payload data stream to be inspected and the initial state of the automaton from where the matching operation should start. Notice that the initial state information is important to enable stateful inspection

across multiple packets of a network flow. The engine responds with a list of matches to indicate which RegExes are matched, the corresponding offsets to indicate the position in the payload where the match occurred, and the final DFA state where the inspection ended, which is needed to resume inspection of the next packet of the traffic flow. To perform the RegEx matching operation, the engine uses a standard memory read interface to access DFA transitions and an auxiliary read-write access to a small internal memory to record the internal states of the engine. The internal memory has two parts. The first part is used to store the input data and track the current byte location in input data and the current automaton state while the matching or automaton traversal is in progress. The second part accumulates the matches found during the inspection, forming the list of match results for the response. Upon receiving a new request, the internal auxiliary memory is initialized with the state information in that request. Using the current active automaton state and the next input character, the finite automaton memory is read to load the the next active state and a match flag. If the match flag is set, the match id and current byte position are appended to the response buffer. If there are additional input characters to process, the address of the next memory load to fetch the next transition data is computed using the next state ID from the loaded transition and the next character value. If there are no additional input characters left to process, the match responses collected in the response buffer are returned through the host interface.

The core engine logic can be pipelined over n clock cycles to achieve higher clock frequency. In this case, the data received from automaton memory and auxiliary memory will be presented at the input of the pipeline and the results will be available after n cycles, which will used to process the next input character and perform next set of memory operations. Additionally, there may be multiple clock cycles of latency between issuing a request

to memory and receiving the load data. To compensate for the latency of the engine pipeline and the memory accesses, multiple inspection contexts should be maintained. Each context can process a separate request and different contexts can be issued one after other every cycle in the pipeline to keep the pipeline and memory busy. To illustrate, considering a simple example in which the core engine is not pipelined (*i.e.*, it performs computations in a single cycle), it takes 6 cycles to access the memory that keeps the automaton image. Without using multiple contexts, the peak throughput would be one transition per 7 cycles, as we have to wait for the memory load to fetch the current transition data before we can process the next character. Using multiple processing contexts, we can process multiple requests for different traffic flows simultaneously. The contexts are like lightweight threads, each thread has its own internal state to track their its matching progress and multiple threads are issued independently whenever they are ready to make progress. With 6 threads, each can issue one load every cycle, keeping the memory at full utilization. If the core engine logic is also pipelined at the circuit level to achieve higher clock frequency, the effective memory latency will further increase the number of threads needed to cover the latency. This level of circuit pipelining will increase the throughput at which loads will be issued to memory, and will be useful only when the memory subsystem can support such throughput levels. Notice that when multiple contexts are used, the internal auxiliary memory must be sized accordingly so that it can store the internal information of all threads.

The reference DFA engine design is easily expanded to support HFA operations. Figure 3.9 shows the schematic diagram of the HASIC engine design to support HFA. The key distinctions between HFA and DFA, along with the associated modifications in the hardware design, are described below.

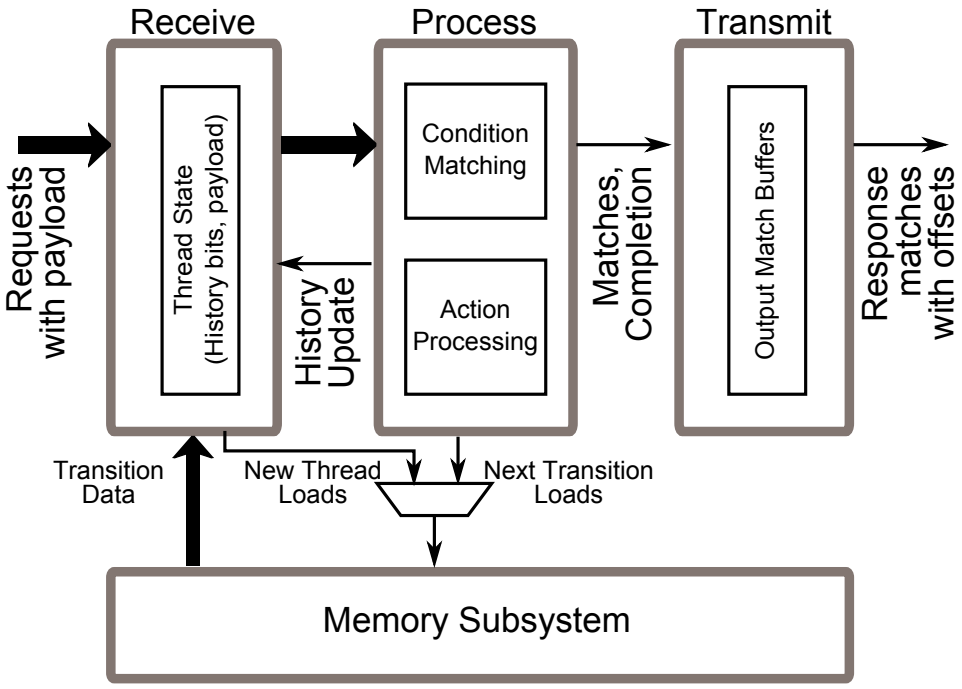


Figure 3.9: Hardware design for HFA module

First, HFAs may have a variable number of outgoing transitions for a single state and input character. The HFA construction algorithm and the memory layout are geared towards one memory access per input character on average, although the hardware circuit is designed to accommodate the need to fetch additional transitions for the same character. The thread issue logic is revised accordingly to allow these operations. The offset of the current input character being inspected is incremented only when the transition condition is matched, unlike the DFA design in which it is incremented at every load. Second, HFA transitions are more advanced than DFA transitions as they contain a set of conditions and actions on the history bits. Our algorithm represents the condition as two fixed length bit vectors, which simplifies the logic design to test conditions. If the condition matches, the action needs to be decoded and history bits are updated accordingly before proceeding to the next input character and loading its associated transitions. If the condition does not match, we load the next transition on this state and character. This process repeats until hitting the match Finally, in addition to tracking a single automaton state in a DFA based system, HFA based systems must also track the state of the history bits associated with a traffic flow. The initial history bits are provided as part of the incoming requests and are stored in the internal auxiliary memory to be read and updated as transitions are processed. Finally, the history bits are returned with the standard return data of the DFA design when the request completes.

3.7 Experimental Results

We demonstrate the capability of HASIC by comparing it with DFA for speed because DFA is the fastest (although the biggest), with NFA for memory size because NFA is the

smallest (although the slowest), and with XFA because XFA represents the state-of-the-art. We compare the construction time of the direct HFA construction algorithm in Section 3.3.3, denoted HASIC$_D$, and the mixin-based HFA construction algorithm in Section 3.4, denoted HASIC$_M$, with that of DFA and NFA. We cannot compare with XFA construction time as it cannot be automated and XFA construction code is not available.

3.7.1 Data Set

The RegEx sets that we use come from a variety of sources. Sets C7, C8, and C10 are proprietary and come from a large networking vendor. Sets S24, S31, S34, and B217 are public sets from Snort [47] and Bro [48] that has been used in prior literature [49]. Within C7, S31, and B217, there are a number of RegExes with `.*`s that have been commented off. We further created RegEx sets C7p, S31p, and B217p by restoring the RegExes containing `.*`s from these three sets respectively. As C7⊂C7p, S31⊂S31p, B217⊂B217p, we focus the collection of RegEx sets C7p, C8, C10, S24, S31p, S34, and B217p, denoted as BCS.

The 217 RegExes in Bro217 are almost entirely string matching, although the additional RegExes that we restored have one to three wildcard closures (`.*`) each. The Snort sets have a higher density of wildcard and near-wildcard (*e.g.*, `[^\r\n]*`) closures, and the C7, C8, C10 sets have a very high density of wildcard and wildcard closures, with as many or more closures than the number of RegExes. The density of these closures makes their corresponding DFA much larger. To determine the scaling properties of HASIC, we created an additional set Scale by combining all the distinct RegExes from S24, S31p, and S34. Table 3.7 summarizes the properties of these RegEx sets with their corresponding numbers of NFA states, DFA states, HFA states (by HASIC$_M$), and history bits. Note that the HFAs

generated by HASIC$_M$ and HASIC$_D$ for the same RegEx set have almost the same number of states.

Set	RegExes	NFA St.	DFA St.	HASIC$_M$ St.	Hist. Bits
B217p	224	2553	-	16527	10
C7p	11	295	244366	616	12
C8	8	99	3786	117	7
C10	10	123	19508	238	9
S24	24	702	10257	925	7
S31p	40	1436	39977	2323	18
S34	34	1003	12486	1362	8
Scale	77	2631	593810	7401	30

Table 3.7: RegEx set Properties

We evaluated our solution using both synthetic and real-life traffic traces. The synthetic trace comes from Becchi *et al.*'s flow generator [50], which is a useful tool for generating synthetic streams of various degrees of maliciousness based on given RegExes. The degree of maliciousness depends on parameter p_M where a higher value indicates more malicious traffic. More specifically, the trace is generated such that with probability p_M, each input character transitions the automaton away from the start state, which will cause a large amount of the DFA states to be visited, forcing a small cache hit rate and thus many accesses to main memory. Our synthetic traces are generated with the default p_M values of 0.35, 0.55, 0.75, 0.95 as specified in the tool. Furthermore, we generate a trace called *rand* consisting of purely random data.

To test the overhead of handling interleaved packets from many simultaneous flows, we also use real traffic traces. This allows us to test not only raw processing speed but also the ability to save and load the matching state for each flow. We use three sources for realistic trace data: (1) the DARPA intrusion detection data set, generated by the MIT Lincoln Laboratory (LL) [25], (2) traces captured during the 2009 Inter-Service Academy Cyber

Defense Competition (CDX) [51], and (3) a locally gathered traffic trace of a student lab PC (RL). For LL, we process the ten traces from week 5 for a total of 5.8GB. For CDX, we process the Border Data Capture traces 3 through 8, for a total of 550M. For RL, we capture 10 traces, the size of each is 0.1GB.

3.7.2 Metrics & Experimental Setup

To compare these algorithms, we measure automaton construction time, memory image size, and packet processing throughput. Memory image size is measured by the amount of contiguous memory needed to store an automaton. Throughput is measured by the RegEx matching time per byte in processing packets. Error bars in graphs represent standard deviation. The experiments are carried out on a server with a Sandy Bridge Core(I7-2600K@3.4GHz) and 8GB RAM. The image construction code is 1.5K lines of OCaml, and the pattern matching engine is 300 lines of C++. For all experiments, we use a single thread running on a single core. To remove disk overhead from our measurements and make results more consistent, the entire trace file on disk is read into memory and payloads are pre-extracted from pcap files, although flows are not pre-assembled.

3.7.3 Automaton Construction: Time & Size

Figure 3.10 shows the automaton construction time for each of the BCS sets divided by the number of NFA states. This normalization reduces the variation in construction time due to the underlying complexity of the RegEx set, and allows for easier comparison of the construction methods. Comparing HASIC$_M$ and HASIC$_D$, we observe that *HASIC$_M$ is much faster in construction* (peaking at 4700 times faster than HASIC$_D$ for C7p), and

has a more consistent construction time per NFA state than HASIC$_D$. Comparing DFA and HASIC$_D$, we observe that they take almost the same construction time for each RegEx set. For B217p, which is too complex to be generated by both HASIC$_D$ and DFA, it can be generated by HASIC$_M$ in 25.2 seconds.

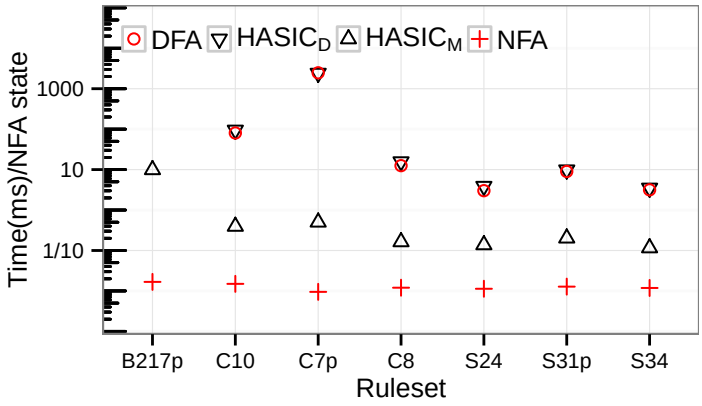


Figure 3.10: Construction Time BCS Sets

To evaluate how well HASIC$_M$ construction scales with the complexity of the input, we use the Scale RegEx set. We generated DFA and HFA for the first rule, then the first two rules, then the first three, etc. We stopped generating DFA when the DFA generation time exceeded 2 minutes, while each of the 77 HFAs were generated in under 2.2 seconds. The results in Figure 3.11 show that *HASIC$_M$ scales linearly with the number of RegExes* while the DFA has exponential construction cost.

Table 3.12 shows memory image sizes for the various rulesets. The DFA memory images use 4-byte transitions, and the HFA memory images uses 16-byte transitions. Although HFA has much larger memory size per transition than DFA, for complex RegEx sets, *the memory image size of HFA is orders of magnitude smaller than DFA*, because HFA has orders of magnitude fewer states. The memory image size of the HFAs constructed by HASIC$_M$ is on average 33% larger than that of HASIC$_D$.

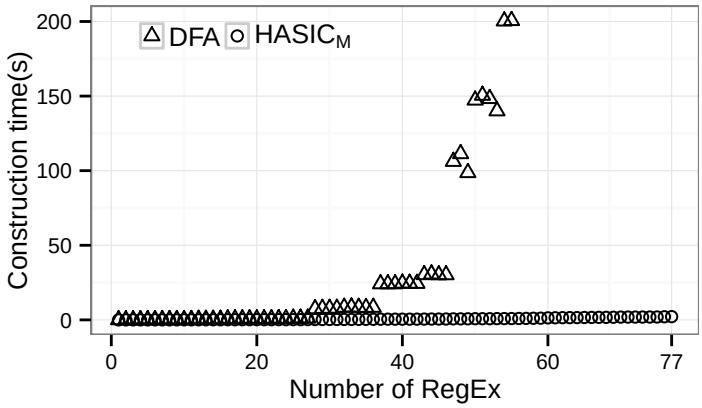


Figure 3.11: Construction Time Scale Sequence

Ruleset	NFA	DFA	HASIC$_D$	HASIC$_M$
B217p	0.5	-	-	108
C7p	0.1	250	4	4
C8	0.1	4	0.7	0.8
C10	0.1	20	2	2
S24	0.2	10	5	6
S31p	0.4	41	9	16
S34	0.3	13	6	9
Scale	0.8	608	32	54

Figure 3.12: Memory Image Sizes

3.7.4 Packet Processing Throughput

Figure 3.13 shows 6 categories synthetic traces of increasing degree of maliciousness and their impact on the processing throughput of different automata. Note that malicious traces cause RegEx matching performance to drop significantly because they cause the automaton to access a wide variety of its states, which causes the cache hit rate to drop, requiring more accesses to slower main memory. We observe that HFA throughput is 2.9 to 3.6 times, with an average of 3.3x, faster than XFA. XFA performance is estimated from the measurements in [28], which shows that XFA is 75.6/11.6 = 6.5 times slower than DFA. Applying this proportion to our DFA results gives estimated XFA throughput.

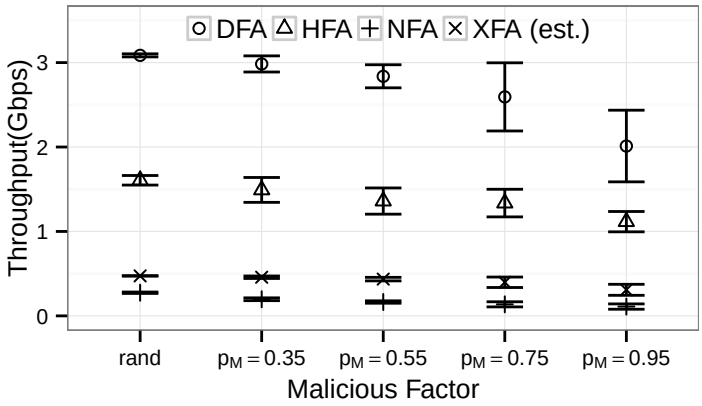


Figure 3.13: Throughput Synthetic Traces

For real network traces, the results are similar to random traces, as shown in Figure 3.14. Each marker indicates the throughput of a RegEx engine on a single trace. For all automaton types, RegEx sets, and traces, the results were nearly identical to random traces because these traces have a low number of matches on our RegEx sets. The real traces had additional overhead of switching between flow states as packets of different flows arrived and were analyzed, but this did not reduce their performance significantly as compared to the random traces.

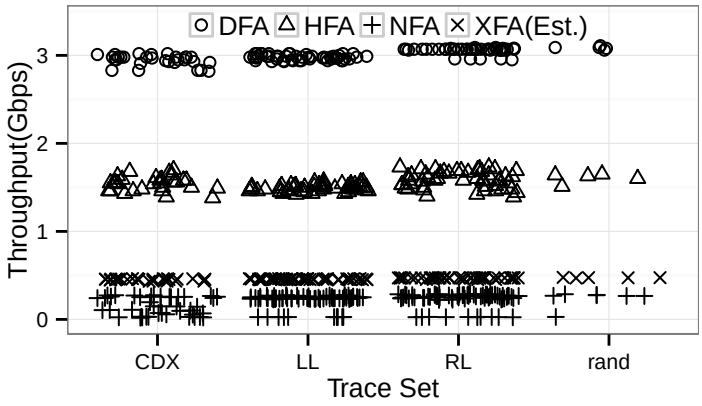


Figure 3.14: Throughput Real Traces

Finally, we evaluate the impact of the HFA optimization technique of ordering transitions based on hit probability of each transition. Figure 3.15 shows the average number of HFA transitions examined for each input character without and with this optimization for synthetic traces generated with p_M being 0.35 and 0.75, respectively. The results show that *this optimization technique greatly reduces the average number of HFA transitions examined for each input character and improves packet processing throughput.* The optimization is performed using the first 10% of the trace, and then the number of transitions examined on the rest of the trace is shown. In practice, the average number of HFA transitions examined for each input character with this optimization on a real packet trace will vary depending the similarity between past traffic and current traffic.

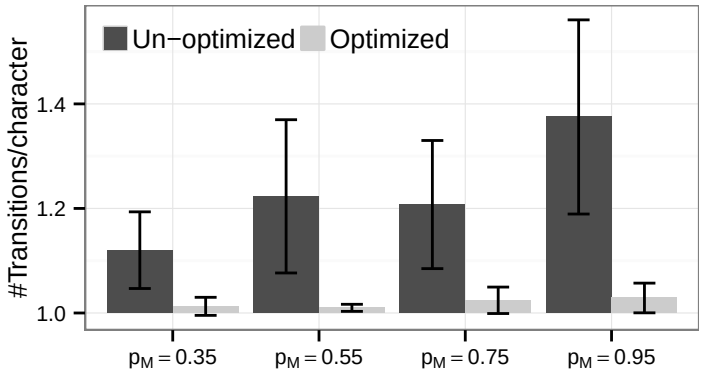


Figure 3.15: Transition Order Optimization

The hardware implementation of HASIC has throughput dependent on two factors. The more important factor, and the one that we have control over, is the memory subsystem it uses. The random-access throughput of the memory subsystem determines the number of transitions per second it can process. The second factor is the number of transitions we need to examine for an input character. Figure 3.15 shows that we can get the average number of transitions per character down to less than 1.1 after optimization. Table 3.16 shows the throughput based on an estimate of 1.1 transitions per character and IBM 32nm eDRAM technology.

Memory speed	#Read Ports	Read Latency	# HASIC Engines	Throughput (Gbps)
eDRAM @ 1GHz	1	2	1	7.3
eDRAM @ 1GHz	2	2	2	14.5
eDRAM @ 1GHz	4	2	4	29.1

Figure 3.16: HASIC hardware implementation throughput

3.8 Conclusions

As the core problem of many security and networking applications, RegEx matching has received much work; however, it has remained an unsolved problem as it is inherently difficult to achieve high speed with low memory. This work significantly pushes forward the direction of ASIC friendly RegEx matching with high speed and low memory. Using only a few history bits, our algorithms are able to achieve DFA-like matching speed with NFA-like memory. Our algorithms are not only fast in its software implementation but also easy to implement in ASIC due to the simplicity of the RegEx matching process and memory image.

Chapter 4

Firewall Compression

4.1 Introduction

4.1.1 Background and Motivation

Packet classification is the core mechanism that enables many networking devices, such as routers and firewalls, to perform services such as packet filtering, virtual private networks (VPNs), network address translation (NAT), quality of service (QoS), load balancing, traffic accounting and monitoring, differentiated services (Diffserv), etc. A packet classifier is a function that maps packets to a set of decisions, allowing packets to be classified according to some criteria. These classifiers are normally written as a sequence of rules where each rule consists of a *predicate* that specifies what packets match the rule and a *decision* for packets that match the rule. For convenience in specifying rules, more than one predicate is allowed to match a given packet. In such cases, the decision used comes from the first rule in the sequence whose predicate matches the packet. Table 4.1 shows a simplified example classifier with three rules. This packet classifier's predicates examine 5 fields of the packet, and has decision set {accept, discard}, as might be used on a firewall. Note that 1.2.0.0/16 denotes the IP prefix 1.2.*.*, which represents the set of IP addresses from 1.2.0.0 to 1.2.255.255. The final rule, r_3, is the default rule; it matches all packets, guaranteeing a decision for each packet.

Rule	Source IP	Dest. IP	Source Port	Dest. Port	Protocol	Action
r_1	1.2.0.0/16	192.168.0.1	[1,65534]	[1,65534]	TCP	accept
r_2	*	*	*	6881	TCP	discard
r_3	*	*	*	*	*	accept

Table 4.1: An example packet classifier

Packet classification is often the performance bottleneck for Internet routers as they need to classify every packet. Current generation fiber optic links can operate at over 40 Gb/s, or 12.5 ns per packet for processing. With the explosive growth of Internet-based applications, efficient packet classification becomes more and more critical. The de facto industry standard for high speed packet classification uses Ternary Content Addressable Memory (TCAM). Since 2003, most packet classification devices shipped were TCAM-based [52]. Although a large body of work has been done on software-based packet classification [53], due to its parallel search capability, TCAM remains the fastest and most scalable solution for packet classification because it is constant time regardless of the number of rules.

The high speed that TCAM offers for packet classification does not come for free. First, a TCAM chip consumes a large amount of power and generates lots of heat. This is because every occupied TCAM entry is tested on every query. The power consumption of a TCAM chip is about 1.85 Watts per Mb [54], which is roughly 30 times larger than an SRAM chip of the same size [55]. Second, TCAM chips have *large die area* on line cards - 6 times (or more) board space than an equivalent capacity SRAM chip [56]. Area efficiency is a critical issue for networking devices. Third, TCAMs are *expensive* - often costing more than network processors [57]. This high price is mainly due to the large die area, not their market size [56]. Finally, TCAM chips have small capacities. Currently, the largest TCAM chip has 72 megabits (Mb). TCAM chip size has been slow to grow due to their extremely high circuit density. The TCAM industry has not been able to follow Moore's law in the past, and it

is unlikely to do so in the future. In practice, smaller TCAM chips are commonly used due to lower power consumption, heat generation, board space, and cost. For example, TCAM chips are often restricted to at most 10% of an entire board's power budget; thus, even a 36 Mb TCAM may not be deployable on many routers due to power consumption reasons.

Furthermore, the well known range expansion problem exacerbates the problem of limited capacity TCAMs. That is, converting range-based packet classification rules to ternary format typically results in a much larger number of TCAM entries. In a typical packet classification rule, the three fields of source and destination IP addresses and protocol type are specified as prefixes which are easily written as ternary strings. However, the source and destination port fields are specified in ranges (*i.e.*, integer intervals), which may need to be expanded to one or more prefixes before being stored in a TCAM. This can lead to a significant increase in the number of TCAM entries needed to encode a rule. For example, 30 prefixes are needed to represent the single range $[1, 65534]$, so $30 \times 30 = 900$ TCAM entries are required to represent the single rule r_1 in Table 4.1.

4.1.2 Problem Statement

Formally, a classifier C is a function from binary strings of length w to some decision set D; that is, $C : \{0,1\}^w \rightarrow D$. While the classifier function may be specified as a combination of range and ternary predicates on a number of fields, the underlying function takes a binary string and returns a classification decision. In firewalls, the classifier commonly takes 104 bits of packet header and returns either *Accept* or *Reject*. A TCAM classifier T is an ordered list of rules $r_1, r_2, \ldots, r_n$. Each rule r_i has a ternary predicate $p \in \{0,1,*\}^w$ and a decision $d \in D$. A ternary predicate $p = p_0 \ldots p_w$ matches a binary string b if all non-star entries in

p match the corresponding entries in b, that is,

$$\bigwedge_{j=1}^{w} (p_j = b_j \vee p_j = *).$$

The decision of a TCAM classifier T for an input $p \in \{0,1\}^w$ $T(p)$ is the decision of the first matching rule in T; that is, TCAMs use first-match semantics. A TCAM classifier T implements a classifier C if $T(p) = C(p)$ for all $p \in \{0,1\}^w$, that is, if all packets are classified the same by both.

This paper focuses on the fundamental *TCAM Classifier Compression* problem: given a classifier C, construct a minimum size TCAM classifier T that implements C. TCAM classifier compression helps to address all the aforementioned TCAM limitations - small sizes, high power consumption, high heat generation, and large die area. Note that even for the same TCAM chip, storing fewer rules will consume less power and generate less heat because the unoccupied entries can be disabled in blocks.

4.1.3 Limitations of Prior Art

Prior TCAM Classifier Compression algorithms fall into two categories: list based algorithms and tree based algorithms. List based algorithms (*e.g.*, Bit Weaving [55], Redundancy Removal [58], Dong's scheme [59]) take as input a list of TCAM rules and search for optimization opportunities between rules. These algorithms are sensitive to the representation of their input ruleset which means they can produce very different results for equivalent inputs. Tree based algorithms (*e.g.*, TCAM Razor [60] and Ternary Razor [55]) convert the input rules into a decision tree and search for optimization opportunities in the tree. Tree based algorithms typically produce better results because they can find optimization opportunities

based on the underlying classifier without being misled by a specific instantiation of that classifier. A key limitation of prior tree based algorithms is that at each tree level, they only try to optimize the current dimension and therefore miss some optimization opportunities.

4.1.4 Proposed Approach

In this paper, we propose the Ternary Unification Framework (TUF) for TCAM classifier compression, which consists of three basic steps. First, TUF converts the given classifier to a BDD, a binary tree representation that puts all decisions at the leaves. Second, TUF collapses the BDD, converting leaves into sets of equivalent ternary data structures and combining these at internal nodes to produce a set of ternary data structures that represent the classifier. Finally, TUF converts the ternary data structures to TCAM rules and chooses the smallest as the final result. Broadly, the two decisions that define a specific TUF algorithm are (1) the ternary data structure to represent the intermediate classifiers and (2) the procedure to combine intermediate classifiers.

TUF advances the state of the art on TCAM classifier compression from two perspectives. First, it is a general framework, encompassing prior tree based classifier compression algorithms as special cases. Because of the structure that TUF imposes on tree based classifier compression algorithms, it allows us to understand them better and to easily identify optimization opportunities missed by those algorithms. Second, this framework provides a fresh look at the TCAM classifier compression problem and allows us to design new algorithms along this direction.

4.1.5 Key Contributions

We make five key contributions in this paper. First, we give a general framework for optimizing ternary classifiers. The framework allows us to find more optimization opportunities and design new TCAM classifier compression algorithms. More specifically, the choices of which ternary data structures to use and how to combine them give new flexibility in designing such algorithms. Second, by designing novel ternary data structures and heuristic combining procedures, we develop three concrete compression algorithms for three types of classifier compression. Third, we implemented our algorithms and conducted side-by-side comparison with the prior algorithms on both real-world and synthetic classifiers. The experimental results show that our new algorithms outperform the best prior algorithms by increasing amounts as classifier size and complexity increases. In particular, on our largest real life classifiers, the TUF algorithms improve compression performance over prior algorithms by an average of 13.7%. Fourth, we give a series of problem variants that we have needed to solve and give their solutions in terms of TUF. Fifth, we developed tools to allow practical redundancy removal on ternary classifiers.

4.2 Related Work

Several papers have addressed the problem of optimizing TCAM packet classifiers. Some major categories of this research include techniques for redundancy removal, compressing one and two-dimensional packet classifiers and techniques for compressing higher-dimensional packet classifiers.

Redundancy removal techniques identify rules within a classifier whose removal does not change the semantics. Since firewall policy is specified indirectly, redundant rules are com-

monly introduced into real life classifiers. Discovering these redundant rules and removing them reduces the storage requirements of the classifier in TCAM, as well as potentially simplifying maintenance of the classifier. This technique has been investigated by Liu [61, 62], using FDD variants to efficiently identify a maximal (not necessarily optimal) set of redundant rules. More recently, Acharya and Gouda [63] have shown a correspondence between redundancy testing and firewall verification and used this to build a novel redundancy removal algorithm not based on trees that achieves the same compression results as FDD-based redundancy removal. Redundancy removal is an important component of our algorithms, but alone it misses many opportunities to combine or re-represent policy using new rules.

While the problem of efficiently producing minimum prefix classifiers for a one-dimensional classifier has been solved [64, 65] optimally, there are still ongoing efforts to produce optimal prefix classifiers in two or more dimensions. Suri *et al.* [65] solved the case of optimizing a two-dimensional classifier where any two rules are either disjoint or nested, and Applegate *et al.* [66] solved the special case for strip rules where all rules must span one dimension as well as providing approximation guarantees for the general two-dimensional case. These solutions do not generalize to higher dimensions, so they provide little assistance with typical five-dimensional classifiers.

Dong *et al.* [59] proposed the first heuristic five-dimensional prefix classifier minimization algorithm. Meiners *et al.* [60] improved upon this in their heuristic TCAM Razor algorithm. Meiners *et al.* [55] then developed two heuristic ternary classifier minimization algorithms Bit Weaving and Ternary Razor. Ternary Razor adds the bit merging algorithm from Bit Weaving into TCAM Razor. McGeer *et al.* [67] demonstrated an optimal algorithm for finding the minimum representation of a given classifier, but this algorithm is impractical for all but the smallest classifiers due to its exponential runtime. Kogan *et al.* [68] also

target optimal compression by reducing problem to MinDNF for non-overlapping rulesets or Min-AC^0_d for overlapping rulesets, but no efficient solutions to these NP-complete problems are likely. Rottenstreich *et al.* [69] attack the optimal encoding problem from a different perspective, producing optimal encodings of individual rules and then combining these using a novel TCAM architecture to perform pattern classification.

There are many papers that attack the larger problem of packet classification by using non-standard TCAM architectures [70–72] or by reencoding packet fields [73–79]. While this type of work may have theoretical elegance, the cost of engineering new TCAM architectures or re-encoding makes these works less practical. Algorithms like TUF that work within the constraints of the standard TCAM architecture have the advantage that they can be deployed immediately on existing hardware.

4.3 TUF Framework

In this section, we outline our Ternary Unification Framework (TUF), which gives a general structure for ternary classifier compression algorithms. Section 4.3.1 gives the basic structural recursion to compress a TCAM classifier. Section 4.3.2 specifies how TUF facilitates efficient merging of partial solutions in the structural recursion step.

4.3.1 TUF Outline

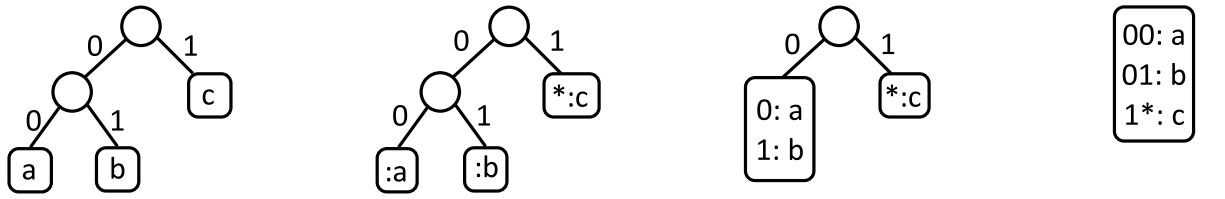
We first present the basic steps of TUF. TUF takes a TCAM classifier as input and returns an optimized TCAM classifier as output. Because TCAM classifiers are written as rule lists, determining a simple property such as whether a TCAM classifier has a decision for every input is NP-complete. The first step of TUF is to represent the classifier as a binary tree

where every internal node has two children and the decisions are in the leaves, called a BDD. An example BDD with three leaves is shown in Figure 4.1a. Converting to BDD resolves the priorities of overlapping classifier rules and gives exact decisions for any input value or range. Note that the construction is dependent on the order of the bits in the BDD, and the resulting classifier can be different with different orderings. By considering multiple bit orderings, typically organized by packet header fields, better compression can be achieved.

The second step of TUF converts the leaves of the BDD into instances of a ternary data structure. As each leaf represents some collection of input values assigned a single decision, we can convert it into an equivalent ternary data structure, such as a trie or a TCAM classifier. This is demonstrated in Figure 4.1b, where the BDD leaves are replaced by TCAM classifiers. The predicate for each classifier depends on the height of the leaf; in this case, the bottom-most leaves are zero-bit classifiers, while the upper c leaf is a one-bit classifier as its height is 1.

The third step, the core of TUF, merges these ternary data structures to form ternary data structures that encode larger sections of the input space. Figure 4.1c shows the result of merging the left subtree of Figure 4.1b. It is in the merging process that compression is possible; similarities in the two halves can be identified and a smaller merge result constructed. After all BDD nodes have been merged, a ternary data structure that represents the entire classifier is created. If the ternary data structure used is not a TCAM classifier, then it is converted to a TCAM classifier as the final step. The TCAM classifier for this example is shown in Figure 4.1d.

TUF can use a number of different ternary data structures such as tries, nested tries (tries of tries), and TCAM classifiers. To support a particular ternary data structure, TUF requires that the data structure support two operations: `Singleton` and `LRMerge`. `Singleton` converts



(a) Input (b) Leaves (c) Merge (d) Output

Figure 4.1: Structural Recursion Example, converting a BDD to a TCAM Classifier

a BDD leaf to a ternary data structure and `LRMerge` joins two ternary data structures A and B into one, $A + B$. Pseudocode for the TUF Core recursive merging is given in Algorithm 1.

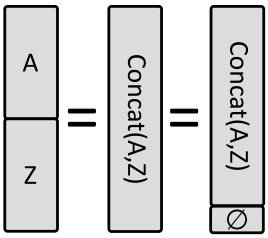
Algorithm 1: TUFCore(c)

```
Input: A 2-tree c representing a classifier
Output: An equivalent ternary data structure
switch c do
    case Leaf dec do
        return Singleton (dec);
    case Node(left,right) do
        LeftSol := TUFCore (left);
        RightSol := TUFCore (right);
        return LRMerge (LeftSol,RightSol);
```

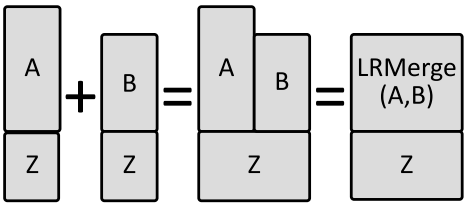
To support classifier compression, the `LRMerge` operation should find commonalities between the two halves and use ternary rules to represent any found commonalities only once in the ternary data structure. This may be a complex operation, spending significant effort to both find and then efficiently represent such commonalities. We next describe how we can simplify the task required by `LRMerge` by tracking backgrounds.

4.3.2 Efficient Solution Merging

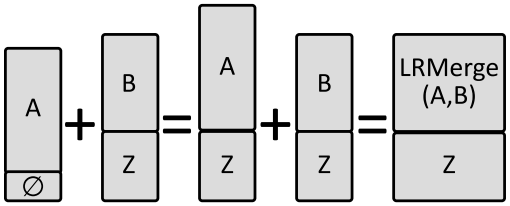
The goal of `LRMerge` is to combine two ternary data structures into one ternary data structure representing both. Using just a single ternary data structure at each step can cause overspecialization. The minimum-size solution for a small part of the input space is often



(a) Concat



(b) LRMerge



(c) LRMerge with EB

Figure 4.2: TUF operations w/ backgrounds

not the best representation for it in the context of the complete classifier. By keeping multiple representations at each step, the right representation of a subtree can be used to create the final solution. Spending more time to keep extra representations of each subtree allows smaller ternary classifiers to be constructed. Taking this idea to the logical extreme, an algorithm could keep all combinations of sub-solutions every time two solution sets are merged. This could cause an exponential amount of work to be spent creating and combining them, leading to an impractical algorithm. The rest of this section explores the use of backgrounds as a way to limit the number of combinations that are created, allow pruning of useless solutions from the solution set and improve the speed of merging solutions.

As ternary classifiers can have multiple matching decisions for an input, they have a precedence structure. For TCAM classifiers, earlier rules in the list have higher precedence than rules later in the list. Using this relationship, a ternary classifier can be divided into two classifiers: a foreground of higher precedence and a background of lower precedence. The operation $\texttt{Concat}(A, Z)$, denoted $\langle A, Z \rangle$, joins a foreground and background ternary classifier into a single ternary classifier, as shown in Figure 4.2a. Intuitively, the joined

classifier searches A first, and if no decision is found, the result from searching Z is used. If the background is non-empty, the foreground classifier should be an incomplete classifier so that some inputs do not have a decision. We denote the classifier that has no decision for any input as $\emptyset$ and note that it is the identity element for the `Concat` operation, *e.g.* $\langle x, \emptyset \rangle = x = \langle \emptyset, x \rangle$.

We write $\frac{F}{B}$ to represent a classifier split into separate foreground, F, and background, B, ternary data structures. For each BDD leaf, we will create a set of solutions that encodes that leaf in different ways. This solution set has two split classifiers, one that encodes the decision in the foreground, and one that encodes it in the background. The solution set for a BDD leaf with decision d is

$$\left\{ \frac{\texttt{Singleton}(d)}{\emptyset}, \ \frac{\emptyset}{\texttt{Singleton}(d)} \right\} \tag{4.1}$$

One of these two solutions will be used in the final classifier to represent this part of the input having decision d. The first solution will be used if the decision d is sufficiently rare in this subtree that it is best to encode this part of the classifier function as its own rule. The second solution will be used if d is sufficiently common in this will be common in this subtree, and that this decision will be encoded as part of a rule with a more general predicate.

TUF maintains the invariant that every solution set will have a solution with an empty background. Because the empty background solution will be handled differently from the other pairs, we give it the name EB, for Empty Background. In (4.1), the first solution is the EB as its background is empty. The EB has the best complete encoding of the classifier represented by a BDD subtree, while the other solutions are the best encoding that assumes some background.

TUF uses multiple solutions and backgrounds to efficiently merge its two input sets of solutions into a new set of solutions. TUF will create a new solution for every distinct background in either input set. For a background found in both input sets, TUF merges the two associated foregrounds together to make the solution for that background. For ternary data structures A, B, and Z, to merge $\frac{A}{Z}$ with $\frac{B}{Z}$, the result is $\frac{A+B}{Z}$, as shown in Figure 4.2b. When one child has a solution with a background that the other lacks, TUF substitutes the EB for the missing foreground. This will produce correct results because the EB is a complete classifier, so can be placed over any background without changing the semantics, as shown in Figure 4.2c. An example merge of two solution sets can be written as

$$\left\{\frac{A}{X}, \frac{B}{Y}, \frac{C}{\emptyset}\right\} + \left\{\frac{D}{X}, \frac{E}{\emptyset}\right\} = \left\{\frac{A+D}{X}, \frac{B+E}{Y}, \frac{C+E}{\emptyset}\right\}.$$

This is implemented as `SetMerge`(l, r) in Algorithm 2.

Backgrounds simplify `LRMerge`'s search for commonalities by allowing `LRMerge` to focus on merging sibling ternary data structures that have the same background. The use of backgrounds also simplifies the merging process by producing the most useful solutions; instead of trying to merge all pairs of solutions (for merge with m and n solutions on left and right, $O(mn)$ merges), we instead merge solutions with the same background ($O(m+n)$ merges). Finally, the use of backgrounds allow a set of solutions to be simplified.

To simplify a set of solutions, TUF incorporates a cost function `Cost`(C) which returns the cost of any ternary classifier. Let A be the foreground of the EB in a solution set. For any solution $\frac{X}{Y}$, if `Cost`$(A) \leq$ `Cost`(X), then TUF removes $\frac{X}{Y}$ from the set. It is a useful simplification because substituting A for X in future merges will supply `LRMerge` with lower cost inputs, which should produce a lower cost result while maintaining correctness. TUF

can also replace the EB by $\frac{\langle X,Y\rangle}{\emptyset}$ if there is a solution $\frac{X}{Y}$ for which $\texttt{Cost}(A) > \texttt{Cost}(\langle X, Y\rangle)$. The result of these simplifications is that the EB will have the highest cost foreground of any solution, and the difference in cost between the EB and any other solution must be less than that solution's background cost.

So far, we have treated the classifier as an unstructured string of bits. In actual usage, the bits being tested are made up of distinct fields, and there is structure to the classifier rules related to these fields. For example, ACL rulesets often have 5 fields: Protocol, Source IP, Source Port, Destination IP, and Destination Port. Once we have developed a good ternary classifier for a section of one field, it is often beneficial to simply store that classifier without modification as we extend it to consider bits from other fields. To support this, we use a function `Encap(d)` that creates a field break by encapsulating the ternary data structure as a decision of another 1-dimensional classifier. The `LRMerge` operation is not required to respect this field break, although doing so will reduce the complexity of merging, as it will have fewer bits to consider. While encapsulating, we also promote the EB to be a background to make it easy to find as a commonality.

Pseudocode for this enhanced TUF Core is given in Algorithm 3. In it, the pair (X, Y) is used to represent $\frac{X}{Y}$.

4.4 Prefix Minimization using Tries

The TUF framework can be used to create a multi-dimensional prefix classifier compression algorithm by using tries. Prefix classifiers have prefix predicates where all stars follow all 0's and 1's. TUF will represent multi-dimensional prefix rules with tries of tries.

Algorithm 2: SetMerge (l,r)

Input: solution sets l and r
Output: merged solution set

```
1 Out = empty solution set;
2 NullLeft = foreground of ∅ in l;
3 NullRight = foreground of ∅ in r;
4 foreach bg in l.backgrounds ∪ r.backgrounds do
5 |   ForeLeft = foreground of bg in l or NullLeft;
6 |   ForeRight = foreground of bg in r or NullRight;
7 |   Out.add(LRMerge (ForeLeft,ForeRight),bg);
8 return Out
```

In this paper, tries are binary trees with nodes optionally labeled with decisions. As with BDDs, the binary search key determines a path from the root, taking the left (right) child if the next bit is 0 (1). The decision of a trie for a search key is the last label found on the path for that key. Each labeled node corresponds to a prefix rule; the path to it from the root matches a prefix of the search key, and all other bits are ignored. Tries are commonly used to represent Internet routing tables where the longest matching prefix determines the route taken. To handle multi-dimensional prefix classifiers, the solution is to use tries where the decision is another trie.

We now describe 1-dimensional and multi-dimensional prefix classifier minimization in TUF.

4.4.1 1-Dimensional Prefix Minimization

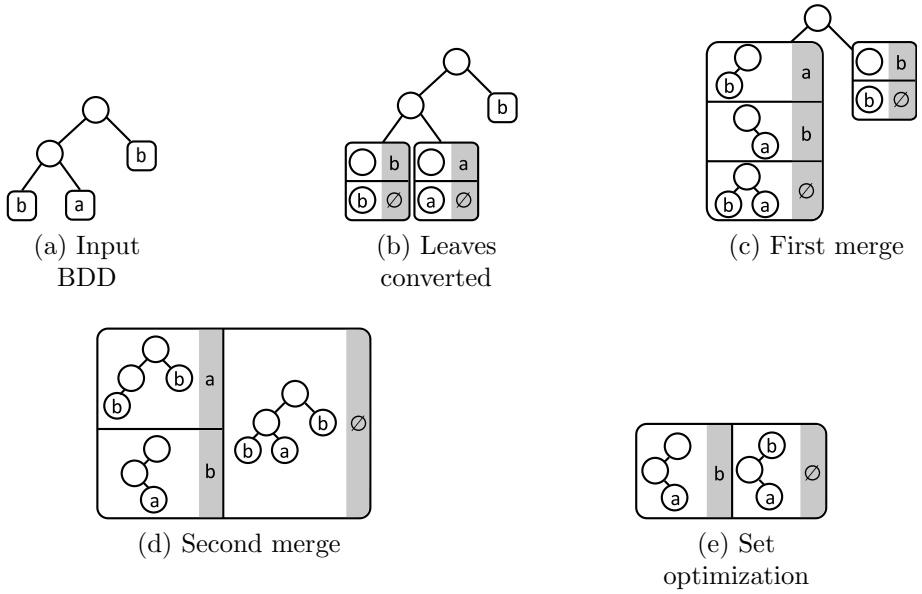
Adapting tries into the TUF framework is very natural. The empty classifier, $\emptyset$, is a trie node with no decision. To implement `Singleton`(d) and produce a classifier where all inputs have decision d, simply create a trie node with decision d. The `Cost(t)` of a trie t is the number of nodes with a decision, which corresponds to the number of TCAM rules needed

Algorithm 3: TUFCore (b) with backgrounds

```
   Input: A BDD b
   Output: A solution set of (fg,bg) pairs
 1 switch b do
 2     case Leaf d do                                  /* BDD Leaf w/ decision d */
 3         return {(Singleton (d),∅),
 4                 (∅,Singleton (d))};
 5     case Node(left,right) do                              /* Internal Node */
 6         LeftPairs := TUFCore (left);
 7         RightPairs := TUFCore (right);
 8         MergedPairs := SetMerge (LeftPairs, RightPairs);
 9         Solutions := Concat() to all of MergedPairs;
10         BestSol := lowest Cost() solution in Solutions;
11         MergedPairs.removeIf(Cost (x) ≥ Cost (BestSol));
12         MergedPairs.add(BestSol, ∅);
13         if at field boundary then
14             Encap (MergedPairs);
15             MergedPairs.add(∅, Encap (BestSol));
16         return MergedPairs;
```

for that trie. To `LRMerge(l,r)` two tries, we create a new trie node with no decision and assign the left child as l and the right child as r. The `Concat(f,b)` operation only has to handle the case where the foreground has a root node without decision and the background is a singleton trie node with a decision. This is because backgrounds are always singleton tries and because `Concat` is applied immediately after `LRMerge` which produces a foreground trie where the root has no decision. In this situation, `Concat` just moves the decision of the background to the root node of the foreground.

Figure 4.3 illustrates the compression of an example classifier into a trie using these operations. The input classifier assigns decision `a` to binary input value 01 and `b` to binary input values 00, 10 and 11. The BDD representation of this classifier is Figure 4.3a, which has a leaf labeled `a` in position 01 (left, right), and leaves with decision `b` elsewhere. At each BDD leaf, two solutions are created, one with the decision in the foreground and one



(a) Input BDD (b) Leaves converted (c) First merge

(d) Second merge (e) Set optimization

Figure 4.3: TUF Trie compression of simple classifier

with the decision in the background, shown in Figure 4.3b. As backgrounds will always be a singleton trie, we will show them as the decision of that trie over a shaded background. The merging step combines solutions that have the same background. In the case where the other solution set is missing a solution with the same background, a solution is combined with the EB of the other solution set. We will apply this merging step twice to our example BDD, as shown in Figures 4.3c and 4.3d. The first merge produces three solutions: the EB, one solution with `a` as background, and one solution with `b` as background. To produce the EB, the foregrounds of the existing EBs are merged. To produce the solution with background `a` or `b`, the corresponding foreground is merged with the EB of the opposite solution set. Note that `LRMerge` is not symmetric, producing differently shaped tries. The second merge is done similarly, with the two `b` solutions merged, the two EBs merged, and the `a` solution merged with the EB.

After we finish merging two solution sets, we optimize the result. Optimization has no effect after the first merge in our example, but it does improve the solution set after the

second merge. Recall that if any solution has total (foreground + background) cost less than the EB, then the EB can be replaced by a `Concat` of that solution. In the example, the total cost of the solution in Figure 4.3d with background `b` is 2; the foreground cost is 1 and the background cost is 1. The EB has a higher cost of 3, which is greater than the total for `b`, so we replace the EB. The result of `Concat` is to put the background decision into the root node of the trie, as shown in the final solution set, where the EB has decision `b` in its root node. Next, recall that if any solution has foreground cost no smaller than the EB, it is replaceable by the EB and can be removed. In this case, we removed the solution with background `a` because its foreground cost of 2 is the same cost as that of the new EB. As we have finished reducing our BDD to a single solution set, the result of compressing this classifier is that newly created EB. The final result is shown in Figure 4.3e.

4.4.2 Multi-dimensional prefix minimization

To represent a multi-dimensional prefix classifier, tries of tries are the natural solution. In a trie of tries, each decision of an n-dimensional trie is an $(n-1)$-dimensional trie. The lowest level 1-dimensional tries have the final decisions of the classifier in them. Tries of tries are turned into decisions for the next level trie at field boundaries using an `Encap` function which is run on both the foreground and background classifiers. In this case, `Encap` simply takes the existing classifier and sets it as the decision of a singleton trie, producing a $(n+1)$-dimensional trie from an n-dimensional trie. For the case of an empty trie such as the background of the EB, `Encap` returns an empty trie. This is analagous to how leaf solution sets are created for tries.

The result of encapsulating the solution set in Figure 4.3e is shown in Figure 4.4. The two existing solutions have been encapsulated as decisions of 2-dimensional tries, which is shown

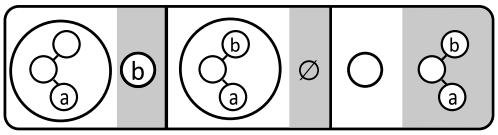


Figure 4.4: Encapsulation at a field boundary

with the existing trie inside a new, large trie node. For the second solution, the background is just an empty background as encapsulation of the empty background is a null operation. One new solution is added at this step (the rightmost solution in Figure 4.4) where the EB is set as a background decision to an empty foreground.

The number of solutions at any stage is bounded by the number of different backgrounds that can be possible. In the 1-dimensional case, the bound is simply $|D|$, the number of classification decisions plus one for the EB. Once the input is 2-dimensional (or higher), the encapsulation operation to cross a field boundary creates backgrounds out of the solution tries that result from compressing the child (already compressed) dimensions at that point. Worst case, this might be $O(|D|2^{w_{tail}})$, where w_{tail} is the number of bits classified by those dimensions. This bound is unlikely, as the filtering and promotion of solutions to BestSol places bounds on the costs of these solutions relative to the best solution. When the BestSol has cost c, solutions with foreground cost at least c will be eliminated. Intuitively, if the backgrounds have high cost, more complicated solutions (foreground + background cost) will be tolerated by the algorithm as they may combine with other solutions with same background to produce a better global solution.

4.5 Ternary Minimization using Terns

Additional compression can be achieved by exploiting the full ternary nature of TCAM-based classifiers rather than limiting ourselves to prefix classifiers. To facilitate this goal, we

develop a novel data structure called a ternary trie or tern that overcomes this restriction and allows us to represent any TCAM rule in tree form. With the additional capability of terns, we extend the `LRMerge` operation to find additional commonalities between its inputs and compress them further in its output.

TCAM rulesets allow ternary rules, where each bit of the rule can be a 0, 1 or *. Tries are capable of representing prefix rules, with leading 0 and 1 bits represented by the path to a node with the decision and trailing stars implied by the height of the node. With terns, we change the structure to explicitly allow * bits along the path to a decision. Instead of having only a left and right child, corresponding to 0 and 1 bits, the nodes of a tern have three children: 0, 1 and *. This allows us to represent any TCAM rule, so we can attain higher levels of compression. Details of converting a tern into a TCAM ruleset are covered later in this section.

With terns, we update our `LRMerge` algorithm to better identify and merge commonalities. Our updated `LRMerge`'s input is two terns representing the left and right children of the new tern we will create. The output is a single tern with the commonalities from the inputs moved into the * child. Logically, given two terns l and r (for "left" and "right"), we want to produce a new "star" tern $s = l \cap r$.

This leads to a simple `LRMerge` algorithm for tries which simply traverses the two inputs in parallel, looking for the same decision in the same position. When one is found, that decision is moved from the two side terns to the star tern. The most interesting thing about this algorithm is how the two results are folded back together. After merging the left children, we get a single tern node, but the children of that node will not end up as siblings in the final result, but as cousins. That is, they will become the left child of the left, right and

star children of their grandparent node, in positions 00, *0 and 10 relative to the root we're compressing. Pseudocode for this merging step is shown as Algorithms 4 and 5.

Algorithm 4: ReChild

Input: Three ternary tries, l, s and r, all non-null
Output: One ternary trie with children striped from the input

```
1 left := Tern(l.l, s.l, r.l);
2 star := Tern(l.s, s.s, r.s);
3 right := Tern(l.r, s.r, r.r);
4 return Tern(left, star, right);
```

Algorithm 5: LRMerge for Terns

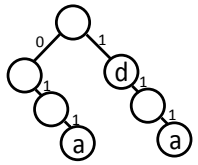
Input: Two ternary tries, l for left and r for right
Output: A ternary trie with children l', s, r', where $s = l \cap r$, $l' = l - s$, $r' = r - s$

```
1  if isEmpty (l) or isEmpty (r) then
2      return Tern(l,empty,r);
3  else
4      lefts := LRMerge (l.l, r.l);
5      stars := LRMerge (l.s, r.s);
6      rights := LRMerge (l.r, r.r);
7      out := ReChild (lefts, stars, rights);
8      if l.dec == r.dec then
9          out.s.dec = l.dec;
10     else
11         out.l.dec = l.dec;
12         out.r.dec = r.dec;
13     return out;
```

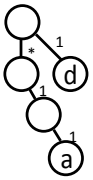
To illustrate the power of using terns as compared to tries, consider the example classifier consisting of three rules shown in Figure 4.5a. The equivalent trie for this classifier, shown in Figure 4.5b, has three labeled nodes which means that no merging of rules was found. Thus, the output classifier will still have the same three rules. On the other hand, using TUF with terns, we produce the tern shown in Figure 4.5c. It has merged the two rules with decision `a` into a single node that corresponds to the rule `*11:a`. The resulting classifier will have two

Pred.	Decision
011	a
111	a
1**	d

(a) Rules



(b) Trie Classifier



(c) Tern Classifier

Figure 4.5: Example Tern compression

rules. Such compression is impossible in tries because it tries can only produce prefix rules, and this tern is encoding a non-prefix rule.

To use a compressed tern in a TCAM, we must convert it to a TCAM classifier. As with tries, each labeled node becomes a rule with the path to the node indicating the predicate and the label of the node becoming the decision of the rule. It remains to order the rules to achieve the correct priority. For a trie, doing a post-order traversal will emit rules in the correct order, as the only consideration is that a parent be emitted after its children. For a tern, there are ordering requirements between rules in different subtrees, so we must use a different strategy. The * branch of the tern must have lower priority then the 0 and 1 branches (otherwise it will override decisions there), but nodes deeper in the tree should have priority over shallower nodes. In our example, a post-order traversal would emit the rule `1**:d` first, because at the root, the `1` subtree is generated before the `*` subtree. This results in the input `111` having decision `d`, which is not the semantics of our input.

To convert a tern into a TCAM classifier, we instead use a bottom-to-top, level-order traversal of the tree. This results in the combination of two decisions with height h still shadowing decisions higher in the tree. Correctness follows because for terns generated in TUF, all rules within the same level are non-overlapping. This is because the rules are produced from non-overlapping BDD prefixes; furthermore, our tern merging operation will

not produce overlapping prefixes. Thus, the order of rules within a level does not change the semantics of the resulting ruleset.

4.6 Ternary Minimization using ACLs

In this section, we use TCAM classifiers as the ternary data structure in the TUF algorithm. Using tree structures makes the `LRMerge` operation very natural to represent, but limits the variety of TCAM classifiers that can be produced. Using TCAM classifiers as the ternary data structure makes the `Concat` operation trivial, but has additional complexity in implementing a compressing `LRMerge` operation.

Merging TCAM classifiers without factoring out any commonalities can be done by prefixing all rules in the left input by 0 and those from the right input by 1 and concatenating them: $A + B = \langle 0A, 1B \rangle$. As there is no overlap between the two groups of rules, the order of concatenation doesn't matter, and the two prefixed inputs can be interleaved arbitrarily. Factoring out commonalities is superficially easy; find a rule that appears in both inputs and put a copy of that rule prefixed by * into the result instead of the two rules prefixed by 0 and 1. This simple method does not take into account the ordering of rules in the inputs, so it produces incorrect results. A rule that appears in both inputs may be needed to shadow other rules. We must take this into account when merging.

To preserve correctness in all cases, we must ensure that rules combined in this way stay in the same order relative to other rules in both inputs. Graphically, this is illustrated in Figure 4.6. This leads to a recursive procedure to merge two TCAM classifiers. After identifying a common rule, split both ACLs into the part before the common rule and the part after the common rule, called the tops and bottoms, respectively. Next, merge the two

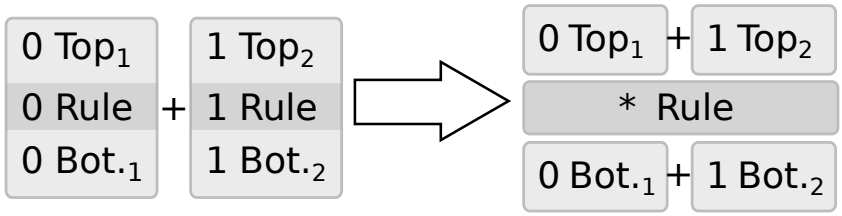


Figure 4.6: Recursive merging left and right ACLs

(a) All pairs (b) Invalid pairing (c) Maximal valid pairing (d) Maximum valid pairing

Figure 4.7: ACL pairing example

tops and the two bottoms, recursively and join the pieces back together. We can write this algebraically as

$$\langle T_1, x, B_1\rangle + \langle T_2, x, B_2\rangle = \langle (T_1 + T_2), *x, (B_1 + B_2)\rangle.$$

Given two rulesets, we can maximize the number of rules merged by examining the pattern of which rules could be merged. Figure 4.7a shows an abstracted pair of ACLs, with letters representing rules. Each pair of rules that can be merged is connected by a line. Two pairs of rules conflict if after merging one pair, the other pair cannot be merged. Graphically, two pairs of rules conflict if their corresponding lines intersect. We define a pairing to be a subset of the pairs of rules that can be merged without conflict. Figure 4.7b shows an invalid pairing, as splitting the rules for one pair prevents the other pair from merging. A maximal pairing is a valid pairing in which no pairs can be added without it becoming invalid. Figure 4.7c shows a maximal pairing; when we split the rulesets into tops and bottoms, we can see there are no further pairings. A maximum pairing is a pairing with the property that no other

pairing has more pairs. Figure 4.7d shows a maximum pairing; there is no larger set of pairs that has no conflict.

The problem of finding a maximum pairing can be reduced to the maximum common subsequence problem [80]. This problem is NP-complete for an arbitrary number of input sequences, but has polynomial-time solutions for the case of two input sequences. In our experiments on the real-life rulesets used in Section 4.9, we observe that there is little difference in the number of pairings identified between the optimal solution and our greedy solution.

4.7 Revisiting prior schemes in TUF

TUF provides a new perspective for classifier compression that leads to new opportunities for compression and more efficient implementations. We illustrate this feature by studying previously developed one-dimensional and multi-dimensional prefix classifier minimization algorithms from the perspective of TUF. Specifically, we examine an existing algorithm for one-dimensional prefix classifier minimization [65] and a non-optimal but effective algorithm for multi-dimensional prefix classifier minimization [60]. Both Suri *et al.*'s one-dimensional algorithm and Meiners *et al.*'s multi-dimensional algorithms can be viewed as instantiations of the TUF framework. Furthermore, when viewed within TUF, we immediately find improvements to both algorithms.

Suri *et al.* first build a BDD out of the input trie, then apply a union/intersection operation to label interior nodes with sets of decisions, and finally traverse the tree once more from the root to give each tree node its final decision. The TUF Trie algorithm presented in Section 4.4 follows a very similar structure. The set of solutions generated at each step follows the same pattern of union/intersection. Because of the simple background structure,

all foregrounds are always equal cost, except for the EB, which has a cost that is greater by one. When the children of an internal node have no matching backgrounds, all the merge results will have the same cost and will be preserved for the next step; this corresponds to the union case. When there are matching backgrounds with the same decision, the resulting solution will replace the EB and eliminate all non-paired solutions; this corresponds to the intersection case.

There is one important difference between the algorithms which is how they manage the tree. The existing algorithms are in-place algorithms that modify an existing tree in multiple passes. This requires the whole BDD to be generated and kept in memory. The TUF Trie algorithm does a reduction over the structure of the BDD, but does not require it all to be generated, or in memory at once. The BDD can be lazily generated, and only a few solution sets need be in memory at once. In this way, TUF Trie can be more efficient than existing algorithms for very large classifiers.

We next consider the multi-dimensional TCAM Razor algorithm developed by Meiners *et al.*. TCAM Razor builds a multi-dimensional algorithm from Suri *et al.*'s one-dimensional algorithm. It first builds an FDD from the input classifier and compresses the 1D classifiers at the leaves using a weighted version of 1D compression. It then treats the results of this compression as a single decision with weight equal to the number of rules in the result, effectively reducing the dimension by one. By repeating this process until all the dimensions of the FDD have been compressed, Razor produces a single classifier as output.

Looking at TCAM Razor from the perspective of TUF, we identify ways to improve compression speed. Razor's weighted one-dimensional compression keeps a full solution for each decision. As the last dimension is compressed, its decisions are compressed tries for the other dimensions. There may be tens or hundreds of these, and it is wasteful to do

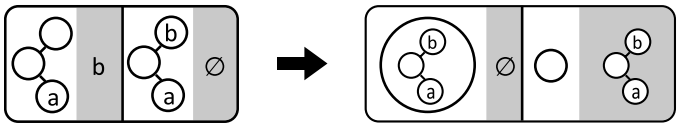


Figure 4.8: Razor hoisting the null solution as a decision

merges for all these solutions, when few of them will be used. A TUF-based solution can remove solutions that are strictly inferior and start with a small set of solutions at the leaves to greatly decrease the amount of work done. These changes give an average of 20x speed improvement on the classifiers that take more than 1/4 second to compress.

The compression level achievable by TCAM Razor can also be improved. Looking at both algorithms from the perspective of solution sets, we see that when Razor finishes compressing a field, it keeps only a best solution (the EB). When the next level is processed, only this one solution is used, and it is treated as an opaque value. The resulting transformation is illustrated in Figure 4.8, which can be compared with Figure 4.4. The classifier with background b is discarded, and the EB is shown encapsulated on the far right and promoted as a new background in the middle.

The Multi-dimensional TUF Trie improves compression by encapsulating more solutions at field boundaries which can lead to the discovery of more potential similarities. Figures 4.9a and 4.9b give example outputs of TCAM Razor and the simple multi-dimensional prefix TUF algorithm described in section 4.4 for a 2-dimensional prefix compression problem. TCAM Razor treats the rulesets from already compressed fields as opaque values, so once the last field is processed, the results have no flexibility. Because of TUF's ability to keep multiple possible solutions for already compressed fields, it is able to apply the default rule across the field boundary, resulting in better compression.

F_1	F_2	Dec.
0*	10*	: a
0*	***	: d
1*	***	: d

(a) Result of TCAM Razor

F_1	F_2	Dec.
0*	10*	: a
**	***	: d

(b) Result of TUF Trie

F_1	F_2	Dec.
0*	X (cost 2)	
1*	Y (cost 1)	

(c) Razor Midpoint

F_1	F_2 + Dec.
0*	10*:a, Bg: d
1*	Bg: d

(d) TUF Midpoint

Figure 4.9: TUF and Razor comparison on same input

The critical step in Razor compression is shown in Figure 4.9c, where sections of field 2 have been compressed into rulesets X and Y (with cost shown), and Razor is compressing field 1. At this point, Razor only knows that X and Y are different, so it must represent them separately in the compression of the first dimension. Only when X and Y are expanded to reconstruct the final TCAM classifier is the similarity revealed. Most of the time that this happens, the final redundancy-removal pass eliminates these redundancies, but in this case, there are no redundant rules, only rules that should have been merged. On the other hand, Figure 4.9d shows part of TUF's view of the problem at this point. The partial solutions for X and Y have a common background, so they can be merged into a simpler solution

4.8 Ternary Redundancy Removal

The final step in TUF is to remove any redundant rules from the resulting TCAM classifier. Existing algorithms for redundancy removal [61–63] are designed for range rules, taking advantage of the low dimensionality of most classifiers to efficiently identify redundant rules. These algorithms can be applied to fully ternary classifiers, but the runtime explodes as

mapping a ternary classifier into the lower dimension space of range rules produces an exponential number of non-adjacent regions for a single rule. As a running example, we consider the problem of compressing the following abstract classifier in Table 4.2 The rule r_1 expands to over 2 billion range rules because the first field matches every even number between 0 and 2^{32}. Here, we explore two ways to perform redundancy removal on ternary classifiers that allow practical removal of redundant rules.

Rule	Field 1	Field 2	Action
r_1	********************************0	0*******	accept
r_2	11111111111111*11111111111111110	1*******	discard
r_3	*********************************	********	accept

Table 4.2: A classifier equivalent to over 2B range rules

The first way to mitigate this expansion problem is to map the rules to a higher dimensional range format. A 32-bit field can be replaced by four 8-bit fields, where the concatenation of the smaller fields equals the larger field. Table 4.3 shows an equivalent classifier with 5 fields that expands to only 131 range rules. In this example, we can see that replacing a wide field by multiple smaller fields can greatly reduce the number of ranges that this rule expands to. This is because only the field containing the 0-bit needs to be expanded into multiple ranges. Care must be taken not to split fields too finely, as the running time of redundancy removal algorithms is usually exponential in the dimension of the classifier.

Rule	Field 1.1	Field 1.2	Field 1.3	Field 1.4	Field 2	Action
r_1	********	********	********	*******0	0*******	accept
r_2	11111111	111111*1	11111111	11111110	1*******	discard
r_3	********	********	********	********	********	accept

Table 4.3: An equivalent classifier equivalent to 131 range rules

Another way to mitigate the expansion from ternary to range rules is by reordering the bits within a field. Permuting the order of the input bits does not change which rules are

redundant, but can simplify conversion to range rules. Table 4.4 shows our original classifier with bits permuted to minimize the number of ranges. Rearranging bits to maximize the number of '*' bits on the right edge of the field gives a classifier that converts to only 3 range rules. To maintain correctness, we must permute the predicate bits of all rules identically, and this prevents us from always converting classifiers to prefix format in this way. The permutation that converts one rule to prefix form may greatly increase the expansion cost for another rule. A simple heuristic ordering is to sort the ternary bit positions based on how many rules have a '*' in that position. Putting '*'s on the right edge of a field tends to reduce the number of ranges represented by a rule.

Rule	Field 1	Field 2	Action
r_1	0******************************	0*******	accept
r_2	01111111111111111111111111111*	1*******	discard
r_3	******************************	********	accept

Table 4.4: An equivalent classifier equivalent to 3 range rules

In Section 4.9, we present experimental results using these transformations to reduce the complexity of redundancy removal.

4.9 Experimental Results

4.9.1 Evaluation Metrics

The critical metric of a classifier compression algorithm is the number of rules in the output TCAM classifier. As the input classifiers are in range form, they may contain rules that must be rewritten to be stored in TCAM. When computing compression ratios, we compare against the result of direct range expansion. This means we replace each non-ternary rule with a collection of prefix rules that compose the same predicate. We denote the result of

this process Direct(C) for a classifier C. We denote the size of the result of running algorithm A on classifier C as $|A(C)|$. For example, |Razor(C)| is the number of rules after running TCAM Razor on a classifier C. Then, the compression ratio for an algorithm A on a classifier C is

$$CR(A, C) = \frac{|A(C)|}{|\text{Direct}(C)|}.$$

A smaller compression ratio indicates that the algorithm produced a smaller output and thus needs less TCAM space. To measure an algorithm's performance on a set of classifiers, we use Average Compression Ratio (ACR). For a set of classifiers S, the ACR of algorithm A is the mean compression ratio across those classifiers;

$$ACR(A, S) = \frac{1}{|S|} \sum_{C \in S} CR(A, C).$$

We evaluate how much TUF advances TCAM classifier compression using the following improvement metric. First, for any classifier C, we define $CR_{prior}(C)$ to be the best possible compression ratio for C using any algorithm excluding TUF algorithms. We then define $CR_{new}(C)$ to be the best possible compression ratio for C using any algorithm including TUF algorithms. In both cases, we use the best possible field permutation order for each algorithm for the given classifier. We define the percent improvement of TUF as $1 - \frac{CR_{new}}{CR_{prior}}$. In this case, a higher Improvement percentage means that TUF performs better and saves more TCAM space.

4.9.2 Results on real-world classifiers

We test these algorithms on a collection of real-life classifiers in three categories. The categories are based on the number of non-redundant rules and difficulty converting the rules to ternary format. Table 4.5 gives a breakdown and statistics on these categories.

Cat.	Count	Avg # Non-Red.	Avg. # Prefix Exp.
Small	13	9	1578
Large	8	3221	7525
Med.	17	209	641

Table 4.5: Classifier categories

Classifiers with an expansion ratio over 20 are categorized as Small RL. These 13 classifiers have an average of 9 non-redundant rules each, yet their prefix expansions have 1600 ternary rules. The remaining classifiers with more than 800 non-redundant rules are categorized as Large RL. The remaining 17 classifiers have neither extreme expansion ratios nor are extremely large, so we categorize them as Medium RL.

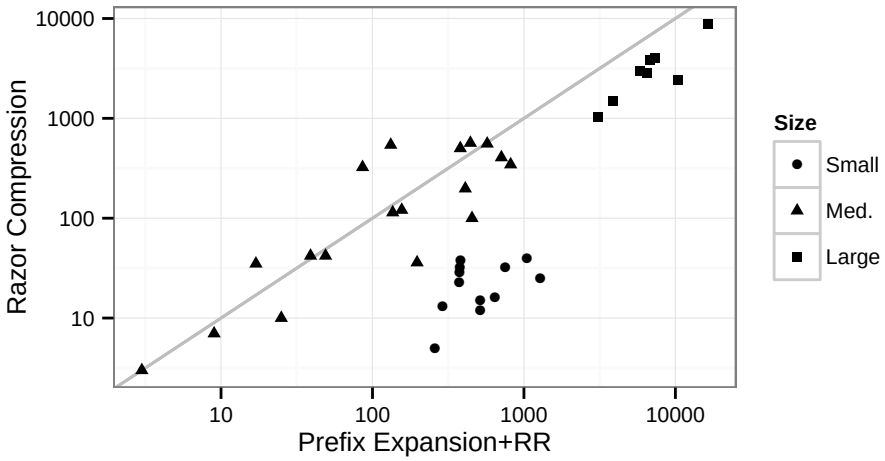
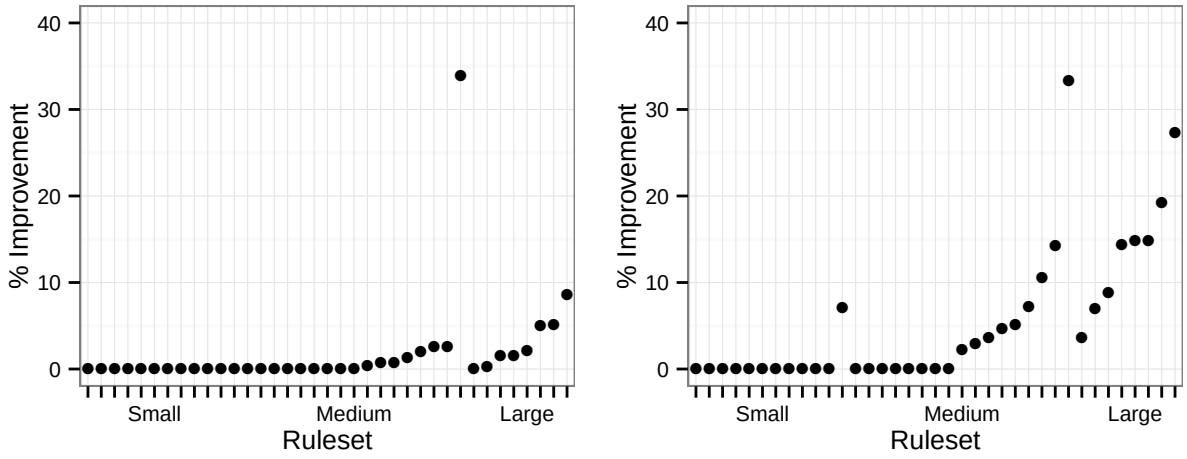
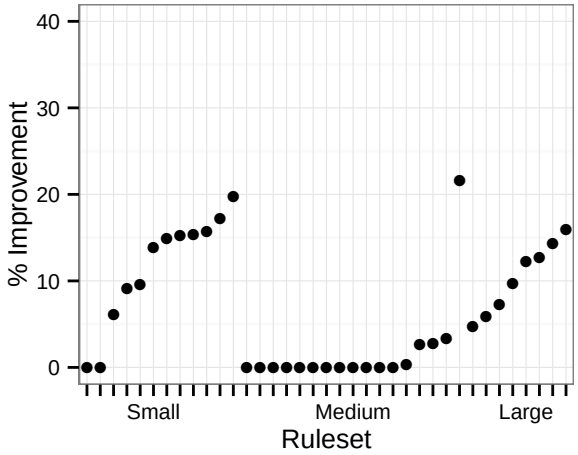


Figure 4.10: Razor vs. Redundancy Removal, for classifier grouping purposes



(a) Prefix Compression

(b) Ternary Compression



(c) Ternary Compression, Unique decisions

Figure 4.11: Improvement of TUF over the state of the art for real life classifiers

These groupings are visually distinguishable by plotting $|Direct(RR(C))|$, the prefix expansion of non-redundant rules, against the compressed size using TCAM Razor, as shown in Figure 4.10.

To test the sensitivity of algorithms to the number of decisions, we also compressed a variant of each of these classifiers. These variants are modified to have a unique decision for each rule. One practical reason for considering unique decisions is that we may need to know which rule matched; for example, we may be tracking hit counts for each rule.

	Algorithm	All		Large	Med.	Small
Orig.	TUF Trie	26.2	%	30.8	41.8	0.8
	TUF ACL	22.8		22.8	38.4	0.7
Uniq.	TUF Trie	45.3		56.9	70.2	2.4
	TUF ACL	43.3		50.9	68.8	2.1

Table 4.6: ACR on real world classifiers

Table 4.6 shows the results of compressing the real-life classifiers and their unique variants with TUF Trie and TUF ACL. Both variants are very effective at compressing classifiers, but TUF ACL does outperform TUF Trie by roughly 13% on all classifiers and 26% on the Large classifiers where there are more compression opportunities to be exploited by a full ternary compression algorithm.

4.9.2.1 Sensitivity to number of unique decisions

When the input classifier has a unique decision for each rule, less compression is possible because there is less potential to apply a background that applies to multiple rules. As a result, TUF ACL and TUF Trie both have reduced compression performance, needing two to three times as many TCAM rules to represent the classifiers. TUF ACL still outperforms TUF Trie but by a smaller amount, roughly 4.5% on all classifiers and 10.5% on Large classifiers.

4.9.2.2 Comparison with state-of-the-art results

We present a direct comparison between TUF algorithms and the previous best algorithms in Figure 4.11. For prefix compression, $C_{prior}(C)$ uses only TCAM Razor [60] and $C_{new}(C)$ uses the best of TCAM Razor and TUF Trie. For ternary compression, $C_{prior}(C)$ uses the best of Ternary Razor and BitWeaving [55] and $C_{new}(C)$ uses the best of Ternary Razor, BitWeaving, and TUF ACL. Each graph shows the percent improvement of the TUF

algorithm over the comparable state of the art for each of our real-life classifiers. The x-axis of each graph is broken into three parts corresponding to the Small, Medium and Large classifiers. Within each group, classifiers are sorted in order of increasing improvement from left to right.

We first consider prefix compression. In Figure 4.11a, we can see that adding TUF Trie improves performance by an average of 1.9 % on all classifiers. The improvement is small but does increase as we move from Small to Medium to Large classifiers from 0% to 2.6% to 3.0%. Furthermore, while the improvement is generally small, the percentage of classifiers where adding TUF Trie improves performance increases as we move to larger classifiers. Adding TUF Trie improves performance on 0 of the 13 Small classifiers (0%), 8 of the 17 Medium classifiers (47%), and 7 of the 8 Large Classifiers (87.5%). There is one notable outlier where TUF trie outperforms TCAM Razor by 34%.

We next consider ternary compression. In Figure 4.11b, we see how much adding TUF ACL improves compression over using only Ternary Razor and BitWeaving on our set of real life classifiers. We see that the improvement is greater than for prefix compression. Specifically, adding TUF ACL improves performance by an average of 5.4 % on all classifiers. As with prefix compression, the improvement does increase as we move from Small to Medium to Large classifiers from 0.6% to 4.9% to 13.7%. As with prefix compression, the number of classifiers where adding TUF ACL improves performance increases as we move to larger classifiers. Specifically, adding TUF ACL improves performance on 1 of the 13 Small classifiers (7.7%), 11 of the 17 Medium classifiers (64.7%), and 8 of the 8 Large Classifiers (100%).

For prefix compression with unique decisions, TUF Trie offers almost no improvement over the state of the art, giving a maximum of 1.7% improvement and only improving 3 of the classifiers. We omit the plot for this uninteresting result.

For ternary compression with unique decisions, from Figure 4.11c, we see that TUF ACL improves performance but the pattern of improvement is quite different. Adding TUF ACL improves performance by an average of 6.2% on all classifiers with unique decisions, but now the best performance is on the Small Classifiers followed by the Large classifiers, with almost no improvement for the Medium classifiers. As we move from Small to Medium to Large classifiers, the average improvement goes from 11.4% to 1.8% to 10.3%. For the Medium classifiers, many of them are already in prefix form, so with unique decisions, the only optimization possible is removing redundant rules. The remainder are nearly in prefix form, so there is little opportunity for compression not already found by prior algorithms, although we still find some. For the Small classifiers, TUF ACL achieves improved performance by better ordering backgrounds to minimize the effects of prefix expansion. For the Large classifiers, TUF ACL is still better able to find global commonalities than previous algorithms.

TUF Tern	10	9	14	13	13	13	12	8	12	5	7	21
Ternary Razor	10	9	14	13	14	13	12	8	12	5	7	21

Table 4.7: Small Classifiers Compressed # of Rules

TUF Tern	92	331	89	34	9	250	135	100	269
Ternary Razor	138	331	91	34	9	271	138	95	270
TUF Tern	110	518	20	140	12	6	33	108	
Ternary Razor	113	520	20	140	14	6	33	108	

Table 4.8: Medium Classifiers Compressed # of Rules

TUF Tern	723	1733	1189	1079	2253	1833	4227	2624
Ternary Razor	724	1784	1312	1049	2184	1868	4307	2770

Table 4.9: Large Classifiers Compressed # of Rules

The compression results of TUF using the ternary trie data structure are compared with Ternary Razor in Tables 4.7-4.9. For small classifiers, TUF Tern performs almost identically

to Ternary razor, producing output classifiers with the same number of rules. For medium classifiers, TUF Tern generally outperforms Ternary Razor, with up to 33% fewer rules to represent the same classifier. For large classifiers, TUF Tern performs similarly to Ternary Razor, having up to 9% fewer rules, but also sometimes needing 3% more rules. TUF Tern rarely outperforms TUF ACL, producing an average of 2% and a worst case of 21% more rules.

4.9.3 Efficiency

For prefix compression, TUF-based TCAM Razor is much faster than the original TCAM Razor. For small classifiers, TUF-based Razor can be more than 1000 times faster than the original TCAM razor. For larger classifiers, the speed difference is an average of twenty times faster, achieving the exact same level of compression in much less time.

Ternary compression algorithms take longer than prefix algorithms, as their search for commonalities throughout the classifier is more difficult. On a desktop PC with an AMD 3GHz Phenom CPU, these algorithms usually complete in under one minute. For some of our classifiers and some of our algorithms and some field permutations, the running time occasionally exceeds one minute; fortunately, these cases can be ignored as they almost always result in poor compression. That is, we can set a reasonable limit such as five minutes for running a compression algorithm and terminate the algorithm and discard the result for the given field permutation if it exceeds the given time limit. Furthermore, for many applications, a slow worst case compression time is acceptable as updates are performed offline.

Figure 4.12 shows a time comparison for TUF-based TCAM Razor on complete classifiers and on incomplete classifiers. For these tests, we took the same collection of classifiers and removed the last rule from each, making them into incomplete classifiers. We then compared

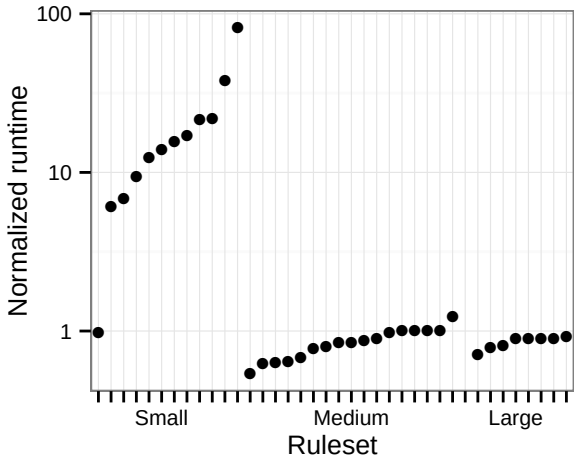


Figure 4.12: Incomplete compression time comparison

the time it takes to run Razor on the original classifiers with that of running the incomplete variant of TCAM razor on the incomplete classifiers. Each point in Figure 4.12 represents the running time of the incomplete Razor divided by the running time of complete Razor on the same classifier. For small classifiers, incomplete compression can take many times the time of complete classifier compression, as the number of rules used to represent a partial classifier can greatly increase as prefix expansion in one dimension is compounded by prefix expansion of other rules in the next dimension. For medium and large classifiers, the compression time is similar, with incomplete Razor having a smaller run-time, due to having fewer possible solutions to merge at each step of compression.

4.9.4 Ternary Redundancy Removal

To evaluate the variety of redundancy removal algorithms, we tested a variety of algorithms on a variety of classifiers. The algorithms tested are All-match redundancy removal ("All-match"), and 4 variants of completeness-based Projection/Division redundancy removal (PD). The three variants of PD (PD, PD-w, PD-wo) use a lazy FDD [81] construction/traversal to identify completeness. The difference between these variants is the transfor-

mation applied to their field structure. PD has no transformations applied to it; PD-w splits wide fields larger than 16 bits to 16-bit units, and PD-wo does both wide field splitting and bit ordering by stars. We chose 16-bit width as a reasonable compromise between creating too many fields and limiting the number of ranges needed to represent a ternary field. The classifiers we use as input are the raw results of TUF ACL, before its final pass of redundancy removal. We use these classifiers as they are representative of the classifiers that need this kind of ternary compression.

Cat.	Rules	Range	All-match	PD	PD-w	PD-wo
Med.	2660	3293	0.82	0.36	0.38	0.42
Med.	323	378	0.04	0.01	0.02	0.02
Med.	339	343	0.01	0.01	0.01	0.02
Med.	924	1063	0.03	0.06	0.06	0.08
Med.	442	486	0.02	0.02	0.02	0.03
Med.	200	219	0.01	0.01	0.01	0.01
Med.	2268	$3.8 * 10^9$	115.18	0.26	0.29	0.35
Med.	28	33	10.00	0.00	0.00	0.00
Med.	155	180	0.07	0.01	0.01	0.02
Large	1728	122236	11.16	80.87	2.04	16.70
Large	1505	1541792	86.97	0.11	0.13	0.16
Large	2444	1542837	timeout	0.28	0.31	0.35
Large	2159	4136	0.09	0.24	0.27	0.29
Large	2257	2649	0.09	0.28	0.29	0.35
Large	5344	8590	2.38	1.20	1.39	1.40
Large	3795	4780	0.25	0.66	0.72	0.75

Table 4.10: Run-times in seconds of Red. Rem. algorithms

Table 4.10 gives run-times of those classifiers that took the All-match redundancy removal algorithm more than 1/100s to process. It also gives the category of the original classifier, the number of ternary rules in the redundant classifier and the number of range rules that these are equivalent to. We note that the run-time of All-match is generally good, but there are a few outliers that take inordinate amounts of time to process, notably the Med. 2268, Large 1505 and Large 2444 classifiers. Switching to the basic Projection/division-based algorithm

(PD) greatly reduces the time taken for Med. 2268, Large 1505 and Large 2444, but greatly increases the time taken for Large 1728. The classifier transformations of splitting fields reduces runtime significantly, and reordering bits does and reordering bits does not seem to improve results significantly beyond the splitting baseline.

4.10 Problem Variants

There are a variety of problems very similar to the original problem where TUF leads to a very natural solution. Recall that the basic problem was to represent a classifier $C : \{0,1\}^w \to D$ as a TCAM classifier. This problem statement implies that C is a proper function, assigning a decision to every input. It is often the case that we don't care what the decision of the classifier is on some inputs. Alternately, we may need the classifier to not return a result for some inputs, i.e. for the classifier function to be partial. Finally, we may have the situation for which some inputs can be left without a decision, but if they are given a decision, it must be some particular $d \in D$. We will discuss how to solve problems that have these components by using the TUF framework.

Don't care: To compress a classifier for which some inputs can be assigned any decision, we modify the leaf construction of TUF for the affected inputs to produce a solution set that represents that any input can be assigned. One way to solve this is by adding a new decision to denote these regions and compressing as normal, then removing all rules with that new decision. It can also be done without the post-filter by changing how TUF creates leaves. To convert a "don't care" BDD leaf into a solution set, a EB with empty foreground is used, giving to the solution set $\left\{\frac{\emptyset}{\emptyset}\right\}$. This solution set can be processed as normal, with the EB combining with any background despite its foreground being incomplete. This produces a

small TCAM classifier with a decision for each "don't care" input chosen to minimize the overall classifier size.

Incomplete: The second variant to explore is compressing incomplete classifiers, that is, classifiers that don't have a decision for some inputs. The obvious approach of treating "incomplete" as a decision, compressing the classifier as normal, and then removing rules with "incomplete" decisions does not work. The rules with "incomplete" decision may be shadowing other rules, and when those other rules are exposed, their decision will be used, giving an incorrect decision. By adjusting TUF in a manner similar to how "don't care" is handled, a correct solution can be achieved. The solution set constructed for a leaf representing an "incomplete" decision is constructed to indicate that no decision can be given: $\left\{\frac{\emptyset}{Inc.}\right\}$. In this solution set, $Inc.$ is a new decision that is special only in that it must remain in the background. When merging a solution set for a complete subtree with the solution set for an incomplete subtree, the incomplete solution can only merge with the EB solution:

$$\left\{\frac{A}{X}, \frac{B}{Y}, \frac{C}{\emptyset}\right\} + \left\{\frac{D}{Inc.}\right\} = \left\{\frac{C+D}{Inc.}\right\}_I.$$

This eliminates other solutions from the other set, as they have nothing to merge with. Merging two incomplete solutions proceeds as normal. In this way, the incompleteness of the solution propagates while still allowing `LRMerge` to simplify the foregrounds.

Another solution for compressing incomplete classifiers involves using a weighted compression algorithm. The sections of the input space with no decision are given a decision with very large weight, to force the construction of a solution where the default rule has the 'incomplete' decision. Removing this default rule from that solution will correctly produce an incomplete classifier as desired. The TUF solution short-circuits the process of creating

other solutions with different backgrounds, greatly reducing the amount of work performed. It also seems a more natural solution to this problem variant than hoping that an overly-large weight for a particular decision will construct a ruleset with the expected structure.

Incomplete w/ dec.: The final problem variant is for scenarios when a particular input must have either a particular decision or have no decision. This can be useful to compress an extremely large classifier by compressing the first n rules, then compressing the next n rules, etc. The compressed version of the whole classifier is then the concatenation of the compressed partial classifiers. Compressing the last part is straightforward as it is a complete classifier, but the other parts are incomplete, so would normally need to be compressed using the incomplete classifier compression described above. In this scenario, we have additional information that can be used to attain better results, namely the decision from the full classifier.

Incomplete classifier compression is greatly hampered by the requirement that certain inputs have no decision. Requiring that each part but the last encode exactly the classifier specified by the rules making that part would result in reduced compression, whereas allowing some specific decision to be used allows some overlapping of rules. To accomplish this, we again modify TUF's leaf creation to convert a leaf marked as "d or nothing" and convert it to the solution set

$$\left\{ \frac{\emptyset}{\texttt{Singleton}(d)}, \frac{\texttt{Singleton}(d)}{\emptyset}, \frac{\emptyset}{Inc.} \right\}.$$

The first two values in this solution set correspond to the ways to encode this leaf having value d, while the third encodes that its value can be left unspecified. Note that the incomplete solution can be eliminated if its cost is greater than the EB solution. Whichever of these solutions works best is the one that will be kept by TUF, without any further modification to

the algorithm. Further, these variants can be mixed within a single classifier, with different parts of the input receiving different treatment simply by changing how the leaf nodes are generated.

4.11 Conclusions

In this paper, we propose a new framework for compressing TCAM-based packet classifiers and three new algorithms for implementing this framework. This framework allows us to look at the classifier compression problem from a fresh angle, unify many seemingly different algorithms, and discover many unknown compression opportunities. Our experimental results show that TUF gives insights that (1) significantly improve the speed of the existing TCAM Razor algorithm with no loss in compression and (2) lead to new algorithms that compress better than prior algorithms. More importantly, this framework opens a new direction for further work on TCAM-based packet classifier compression.

Chapter 5

Hashing

5.1 Introduction

Networking devices depend increasingly on hashing methods for performance. Hash tables are an important building block for forwarding plane requirements. The MAC table in core routers and the flow table in GGSN routers are commonly implemented as large hash tables. Hash is also essential for various proposals which use probabilistic approaches such as Bloom Filters for IP lookup, packet classification, load-balancing, routing protocols and network measurements. ([82,83] and etc). The pathological cases of these algorithms can have severe impact on the visible behavior of that device, and widespread abuse of these failure conditions could have severe consequences.

As networking applications emphasize worst case over average performance, using hash functions for performance is asking for trouble. As Knuth wrote in [84],

> ... we need a great deal of faith in probability theory when we use hashing methods, since they are efficient only on the average, while their worst case is terrible.

Hash tables with fixed-size buckets and Bloom Filters are two common building blocks in networking applications. The "terrible" worst cases for them are severely impaired capacity (could be close to 0) and severely increased false positive rate (could be 100%). Although the extreme worst case might never occur even for hash functions with low quality output, they

would are more likely to cause unexpected performance degradation, such as reduced capacity of hash tables and increased false positive rates. This is because all performance guarantees for hash-based algorithms are built upon the assumption of a hash function whose output is random-looking. Moreover, the choice of hash function can make it easier for attackers to trigger these behaviors [85].

The concept of Universal Hashing [86] is proposed to remedy the uncertainty of the hash function. With the help of universal hashing, many probability tail bounds could be strictly established. However, the randomness of universal hashing comes from the randomness of picking function from the family, specifically from the randomness of the seed. For a practical implementation, the random seed would be baked into the hardware. Some well-known universal hashing schemes such as the H_3 would perform poorly with seed hard-wired.

People would have much more confidence when using hash functions designed for crypto-purpose. However, those functions are out of reach of the resource limitations and timing requirements of network processors. This paper explores what is possible within these constraints.

In this paper, we present two contributions to the state of the art. We provide a statistics-based evaluation framework that tests the quality of these resource-limited hash functions to differentiate the performance of hash functions. We also demonstrate a family of hardware hash functions with performance similar to hash functions with much higher implementation cost.

5.1.1 Networking Hash Trends

We see trends towards small, wide, fast and general purpose hardware hash functions. Even with billions of gates fitting on a single chip, having a hash function take a small number of

gates is still important. The input and output should be wide, able to consume many bytes each cycle and return a large hash value. Speed is always a requirement for networking, but many applications of hash functions in networking require low latency as well. Finally, this hash function should be general purpose, not making any assumptions about its input or output, but doing reasonably well in a broad range of situations.

Wide Increasing traffic speeds combined with clock rates hitting their limit means the amount of work per cycle needs to increase. One dimension this increase happens is in bus size, with data buses growing to 128 bits and higher. With the wider deployment of IPV6, the packet header fields which are most commonly hashed have grown in size, thus our hash function should take a large input per cycle. At the same time, larger hash tables and other hash-based data structures such as Bloom Filter require a large hash output. A large input and output is important for our hash function to interact well with the rest of a networking system. For this paper, we will work towards a hash function with 128 bit input and 128 bit output.

Small Small hash functions also increase the system's ability to handle traffic. Parallel hash functions also allow more work per clock cycle. Networking processors are scaling to use tens to hundreds of MIPS, ARM or custom cores on the same die. To show how important size is, MIPS ships a whole CPU core as small as 0.65 mm^2 [87]. In the case that each processor gets its own hashing block, this level of duplication means the hash function's size is even more important. An area equivalent to about 4K gates is small enough to be duplicated everywhere it's needed, so we will use that as our target size.

Fast The timing of a hardware design affects both latency and throughput. High latency designs can often be pipelined to gain throughput, but this requires large flip-flops to carry state from one cycle to the next, increasing the size of the hash function. For some applications, in some of those kind of proposals, the hash function is not only on the critical path of forwarding, it is even in the feedback path of some state-machine. The state machine needs to run in a high speed with the result from hashing in every cycle. For those types of application scenarios, a low latency hash function is critical. Even in other situations, lower latency is important to avoid complex scheduling. We will focus our efforts on designs that take only one or two cycles.

General Purpose A general purpose hash function will be a more useful addition than a specialized one. Some network devices are custom built for a specific purpose, but more and more devices are engineered to be software controlled and hardware accelerated. This allows them to support new algorithms and protocols, extending the lifetime of the device. A specialized hash function designed for a hash table keyed by MAC addresses can be designed to be smaller and more efficient than a general purpose hash, but it will perform poorly when used by other algorithms. We should not make assumptions on either the structure of the input nor the use of the hash output, so our hash function is general purpose.

To sum up, industry is heading towards higher expectations for hash functions. In this paper, we will aim to produce a circuit with about 4K gates that hashes 128 bits into 128 bits in a single cycle. Sometimes the implementation cost of hash functions does not prevent their use in products, but using better methods to build hardware hashes will reduce implementation costs and enable better hash-based algorithms to be used in networking.

5.2 Related Work

Looking at existing solutions, we do not find any hash functions that satisfy both the quality requirements of coming network algorithms as well as the size and speed requirements of ASIC designs. Cryptographic hash functions cannot meet our gate count or timing restrictions. Other hardware hash functions are low quality or can't scale to larger bus sizes. High quality software hash functions cannot meet our size restrictions and are inefficient on a network processor.

5.2.1 Cryptographic hashes

Much work has been spent optimizing hardware implementations of cryptographic hashes. Satoh and Inoue [88] summarize many implementations and give improvements. All of the implementations shown are much over 4000 standard gates and nowhere near one cycle to run. These hash functions were designed to be efficient in both hardware and software. As such, they do not fully take advantage of the parallelism available in a hardware-only design. We will compare our design with MD5, but only to show how close to its output quality we can come with much more limited resources.

5.2.2 Non-cryptographic Hardware Hashes

Commonly used hardware hashes are either low quality or scale poorly in bus width. To our knowledge, only three general hash functions are widely deployed: CRC-32, PearsonHash [89] and BuzHash [90]. CRC was not designed as a hash function, but instead for detecting bit errors in transmitted data. As pointed out in many papers such as [85], the linear structure of hash functions such as CRC and H_3 [86] with fixed seed makes them vulnerable to at-

tacks. Attackers could invert the hash function and forge attack traffic to trigger "terrible" worst case behavior. Although some people think the design of router benchmarks should specifically avoid causing these behaviors [91], most would agree that all else being equal, a design that makes it difficult to trigger worst case behavior is a better design.

Pearson Hash and BuzHash are both designed for string hashing, taking input one byte at a time. Pearson Hash extends an 8-bit to 8-bit hash function (usually implemented as a lookup table) to $8 * n$ bit inputs by XORing the input byte with the internal state, hashing that to form a new internal state and repeating on the next byte. BuzHash works similarly, using an 8-bit to word-size hash function table to hash each input byte and then mixing these using rotations and XORs. Pearson Hash requires the result of mixing before being able to do the next table lookup, giving a hash that takes 128-bit words as input and latency of at least 16 memory operations + 16 mixing operations. BuzHash can accept wider input by using parallel lookup tables, but as each needs 8Kb of fast memory or 16Kb if we want 64-bit output, this quickly exceeds our size budgets. The final cost of a wide, fast implementation of these kind of hash functions exceeds our intended size constraints very quickly.

5.2.3 Software Hashes

Another source for inspiration in hardware hashes is software hash functions. Software hashes with high throughput and output quality include Bob Jenkin's hash [92] and the Murmur hash [93]. Jenkin's hash uses a sequence of rotations, additions and subtractions on three input values at a time to mix the input with its internal state. Murmur hash uses fast multiplications, rotations and XOR to do the core mixing. The commonly used rotate operations take no gates to implement, as they're just wire patterns. The rest of the operations seem reasonable in hardware, but further investigation reveals that the ALUs (Arithmetic Logic

Units) used in desktop CPUs are optimized for speed. These optimizations increase the size of the logic past our area constraints. Without these optimizations, the resulting hash function doesn't meet our latency constraints. This problem eliminates these successful software hash functions from hardware implementation in our context.

5.2.4 Existing Evaluation Methods

In the literature that discusses selection of hash functions, few tests are provided for general purpose, non-cryptographic hash functions. Special purpose statistical tests are used in some cases [94], but are not appropriate for selecting a general purpose hash. Here we review the most general tests, the Avalanche test and the Chi-squared uniformity test.

Avalanche test A strong test of hash functions is the avalanche test, first introduced in [95]. Intuitively, a small change in the input should trigger a large change in the output, as a small rock falling can trigger an avalanche. Given a hash function H and input x, denote $H(x) = y$. For x' different from x in one bit, a hash function that passes the avalanche test will have $H(x')$ very different from y. Optimally, every pair of inputs with hamming distance 1 (one bit difference) will hash to n-bit outputs with hamming distance $n/2$.

In practice, we test this by sampling the input space and for each sample, inspecting the change in output for each one-bit change in input. By tabulating which bits change, one can build up a model of bit interrelationships and the strength of the avalanche property. For example, if whenever the first input bit is flipped (no matter the value of the other bits), the first output bit always flips, that combination of bits has an avalanche measurement of 100%. Similarly, if a particular output bit doesn't ever change when a particular input bit is flipped, that pair of bits has 0% avalanche. A good avalanche result would be if the first

output bit changed 50% of the time when the first input bit was flipped, depending on the values of the other input bits as to whether or not it would flip. The closer all percentages are to 50%, the better the avalanche provided by the hash function.

Uniformity testing The Chi Squared (χ^2) test can quantify the uniformity of hash output values. For a hash function with output ranging from 0 to n, we hash m input values and count how often each output value occurs, calling these counts $c_i, 0 \leq i \leq n$. The Chi Squared test compares these counts with the counts that would occur if the output had a uniform distribution: $c_u = m/(n+1)$. The statistic

$$\chi^2 = \sum_{i=0}^{n} \frac{(c_i - c_u)^2}{c_u} \tag{5.1}$$

measures how close the c_i counts are to uniform - a smaller value indicates a more uniform distribution of c_i's. As we want hash results with a uniform distribution for packet sampling, this test is especially important in that context [96].

The Chi Squared test for uniformity is impractical for directly testing hash functions. Since most hash functions have a very large range of outputs a proper test would require computing the hash value of a very large number of inputs, otherwise most buckets would be empty due to lack of input. To resolve this problem, the test is usually performed on a variation of the hash function with reduced output range.

This variation is usually to treat the original output as an integer and to apply a final division and use the remainder as the hash value. This means that the test results often don't apply directly to the hash function to be tested. Dividing by a prime number takes many non-uniform distributions and converts them to a much more uniform distribution, which is

why many software hash tables use prime sizes. Dividing by a power of 2 throws away the top bits of the result, only testing part of the hash output. Test results on the modified hash functions are optimistic, so the user of this test must be aware that uniform results on the reduced output variation do not imply uniformity of the original function.

5.3 Design Methodology

Section 5.3.1 presents the major considerations in building a resource constrained, general purpose hardware hash function. This is followed by the specifics of the design in four parts. Section 5.3.2 outlines the general framework of the family of hash functions we explored. The following two sections give two implementations of the mixing primitive: XOR stages and S-box stages. Finally, Section 5.3.5 explains the complexities of routing bit permutations and gives a lower-cost design.

5.3.1 Design considerations

We identify the following rules of thumb in designing a good hash. They are not the only way to build a good hash function. They are design principles that led us to our current design, so understanding them will improve understanding of our design.

As a framework for this discussion, model the input and output of a hash function as vectors of bits. For a given type of input, each of the input bits has a fixed entropy, or unpredictability. For example, MAC addresses use a bit to indicate whether this address is globally unique or locally administered. In most situations, this bit will always be set for globally unique, so it has low entropy because it is very predictable. Since we don't know

the use for the output values, we should strive to have them each maximally unpredictable so that the output resembles uniformly distributed random numbers.

The first guideline is to thoroughly "mix" each input bit into all the output bits. A hash function with n-bit output can be viewed as n binary functions of the input value, one binary function producing each output bit. If some of the output bits don't depend on some of the input bits, that output bit is more predictable than if it depends on the value of every input bit. Thus, we want to use every input bit as part of the "recipe" for producing each output bit.

A perfect hash function with n-bit input and n-bit output values would be a $2^n \rightarrow 2^n$ permutation that assigns each input value a distinct, random output value. In this situation, the output would be predictable only knowing both the input value and the hash function, and having partial information on either or both would make the output very unpredictable. If any input values mapped to the same output value, that output value would be more common than others, and some output value would not be mapped to, giving a more uneven result and a less random looking output.

Cryptographers call the device that performs this kind of permutation a S-box. As large S-boxes are impossible to build, we emulate them with a sequence of operations that "shuffle" the input values. The permutation built into the S-box can be decomposed into a sequence of simpler mappings that are each feasible to compute.

Within this context, our second guideline is to prefer invertible mappings. It is important to map similar inputs to very different outputs, as real-world input data often has inputs with small differences. If the first stage uses a non-invertible simple mapping, similar inputs are likely to collide in this stage and be mapped to identical outputs by later stages. After a series of invertible stages, similar inputs should be mapped to very different values without

any collisions. If the hash function produces output smaller than its input, the required non-invertible transformation should be delayed until the last stage to reduce the chance of similar input collisions caused by the simplicity of this transformation. Using invertible mappings avoids output collisions of similar inputs.

Building a hardware hash function is quite different from a software hash. The level of control of individual bits is much higher, but arithmetic operations like integer multiplication are more expensive. Even the measures of performance are different.

Hardware hash performance is measured in area and timing. The area of a circuit depends on how much wire and how many gates are needed. The timing of a circuit depends only on the longest path from a source bit to an output bit. The timing of a circuit does not depend on how many operations are performed in total, but on how long it takes for all the output bits to be valid. This is very different from software hash functions, which execute instructions mostly sequentially, whose timing is determined by the number of operations performed. Parallel operations are common in hardware, with the time of a complex operation measured by the longest path through logic gates needed to produce any output bit.

Hardware hashes have direct control of values at the bit level, and have access to simpler building blocks. In hardware, bits are just signals on wires, so shuffling the bits of a value is simply routing the wires representing that value to new locations. Bit-wise operations like XOR are cheap, and can be done in parallel on many bits simultaneously. Arithmetic operations, even addition, are much more expensive, because a single carry can ripple through all the bits being added, creating a very long path from the input to output bits. This can be optimized, but at the cost of much more area. All this considered, hardware hash functions will have a very different design from hash functions designed for efficient software implementation.

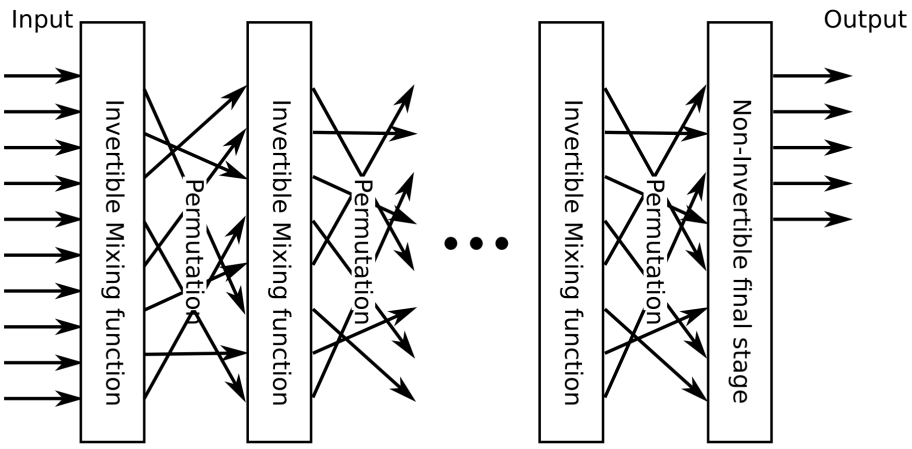


Figure 5.1: Framework of hash function

5.3.2 The framework

We propose a modular hash function design, with multiple stages of alternating mixing and permutation, as shown in Figure 5.1. This design is similar to a cryptographic substitution-permutation network without key bits being merged at each stage. Each component is designed to be invertible, so we automatically avoid bias and cannot have any collisions. We attempt to maximize the efficiency, and use many stages to produce a high quality result.

Instead of building each stage as one large mixing function, much area is saved by using many small mixing functions to mix adjacent bits. By permuting the bits in between stages, the input bits' influence will cascade to affect every bit of the inter-stage state. To mix the input bits into the entire n-bit inter-stage state, at least $\log_b n$ stages are needed, where b output bits are affected by an input bit in each stage. Using many stages that mix bits in small clusters reduces cost significantly.

We suggest two possibilities for mixing stages, a linear XOR stage and a non-linear S-box stage. For the permutation stages, we provide an efficient hardware implementation of very large bit permutation functions.

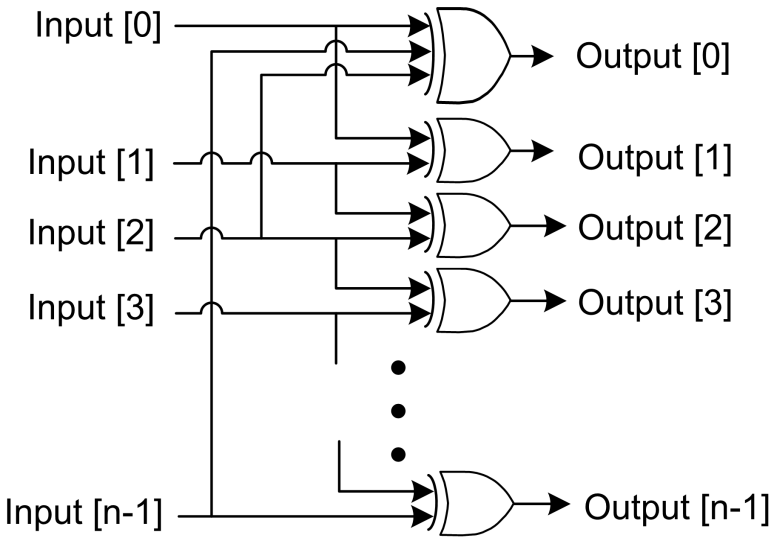


Figure 5.2: One stage of XOR Arrays

$$Y = \begin{bmatrix} 1 & & 1 & & \ldots & & & 1 \\ 1 & 1 & & & \ldots & & & \\ & 1 & 1 & & \ldots & & & \\ & & 1 & 1 & \ldots & & & \\ & & & \ldots & \ldots & . & & \\ & & & & \ldots & 1 & 1 & \\ & & & & \ldots & & 1 & 1 \end{bmatrix} X$$

Figure 5.3: Equivalent matrix equation

5.3.3 XOR stage

To produce an invertible stage, we start with a structured transformation. The easiest way to make an invertible function is by a linear function, representable by a bit matrix. Each row of this matrix specifies which input bits to XOR together to produce an output bit. If the matrix is invertible, the linear function it represents is also invertible. The cost of this function is proportional to the number of ones in the bit matrix, so we want a function whose matrix is also sparse.

Our XOR stage implements a very sparse, invertible, matrix multiplication as the equation in Figure 5.3. This is implemented by XORing adjacent bits as shown in Figure 5.2.

We include one 3-input XOR gate in each stage so the resulting mapping is invertible. If we include only 2-input XOR gates, any combination we produce will be non-invertible.[1] If we use a 1-input XOR gate (also known as a wire), that input bit will only affect one output bit. We would need to be careful that its poor mixing doesn't carry to the last stage.

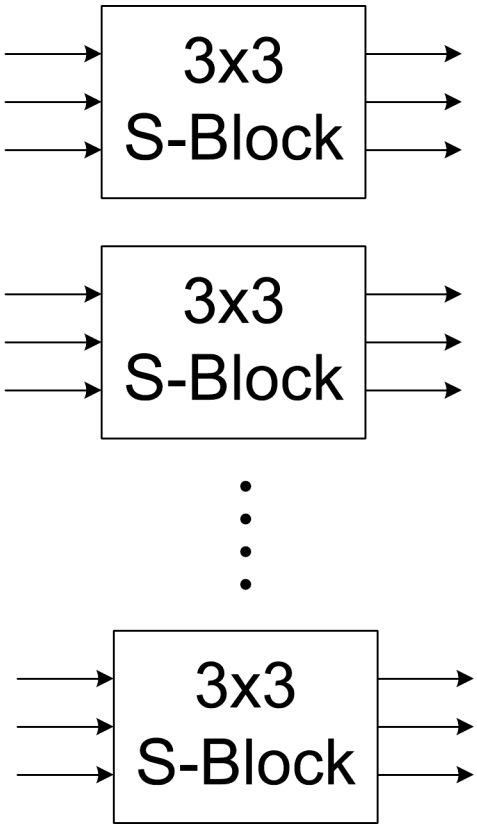


Figure 5.4: One stage of S-boxes

This design is very inexpensive and mixes bits efficiently. Using 3-input XOR gates would allow more mixing, but the gate size would go up by about 2x and the gate delay as well by about 60% [97]. The complexity of routing wires would be increased as well, as we would likely use more non-adjacent bits. We would get similar cost and better mixing from two smaller stages of 2-input XOR gates.

[1] Using row-operations on the matrix cannot change the property that each row has an even number of 1's. Thus Gauss-Jordan elimination cannot terminate, and the matrix is non-invertible.

5.3.4 S-box stage

Linear functions can provide good mixing, but we also need non-linear mappings to add additional complexity to the output to reduce its predictability. Without non-linear mappings, there will be no avalanche, and each output bit will be a simple XOR of some of the input bits. By including non-linear mappings, the output bits can each depend on all of the input bits, each in different ways.

Our solution to this is to use an array of S-boxes as shown in Figure 5.4. These S-boxes will be implemented using direct combinatorial logic. This implementation is area efficient for small S-boxes and produces its result much faster than a lookup table. The smallest S-box that provides a non-linear result is a 3×3 S-box, which takes three bits of input and returns three bits of output. Among all possible S-boxes, we select the following one (and its isomorphic equivalents):

$$[a, b, c] \xrightarrow[(3\text{-}bit\ S\text{-}box)]{Permutation} [Q_a, Q_b, Q_c]$$

$$\{0, 1, \ldots, 7\} \rightarrow \{1, 4, 7, 5, 2, 0, 3, 6\}$$

$$\begin{aligned} Q_a &= \bar{a}b + \bar{a}c + bc \\ Q_b &= ab + a\bar{c} + b\bar{c} \\ Q_c &= \bar{a}b + \bar{a}\bar{c} + b\bar{c} \end{aligned}$$

The S-box we select provides the nonlinear property that with randomly distributed input, flip any one bit or two bits, all three output bits would be flipped with probability 50%. Undesirably, when all three input bits are inverted, all three output bits are also

inverted. This is a downside of using such a small S-box, but it still provides the necessary non-linearity in our framework.

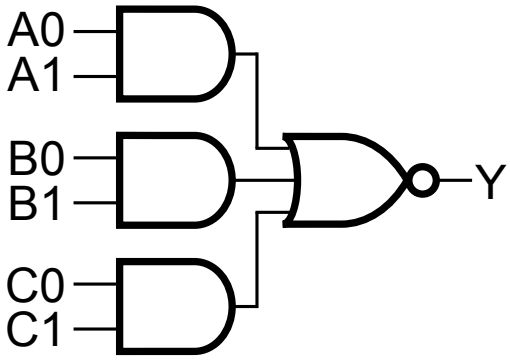


Figure 5.5: AOI222 gate

To implement the formula $\bar{a}b + \bar{a}c + bc$, we use a single AOI222 gate shown in Figure 5.5. This gate is about the same size as a 2-input XOR gate. Using this gate, the S-box shown above could be implemented by only 3 gates and 3 inverters. Overall, the cost of a 3×3 S-box stage is only a little larger than an XOR stage.

For comparison, an example of 4×4 S-box is the following:

$$
\begin{aligned}
Q_a &= a\bar{b}\bar{c}d + \bar{a}b + \bar{a}\bar{c}\bar{d} + bc \\
Q_b &= a\bar{b}d + \bar{a}c\bar{d} + b\bar{c} \\
Q_c &= a\bar{b}c + a\bar{b}\bar{d} + \bar{a}b\bar{d} + \bar{a}c\bar{d} + bcd \\
Q_d &= a\bar{b}c + \bar{a}b\bar{c} + ab\bar{d} + \bar{a}cd
\end{aligned}
$$

This 4×4 S-box produces a more consistent non-linear transformation but the cost is much higher. AOI2222 gates exist in some standard cell libraries, but even this wouldn't be able to compute Q_c above using a single large gate. The only option for larger S-boxes is

to use multiple gates, which results in a much larger cost in area and timing. Using 3×3 S-boxes as the basic element seems the best trade-off between unit cost and mixing ability.

5.3.5 Permutation stage

The optimal permutation stage is a 128-bit P-box, which permutes all 128 bits arbitrarily. The cost of wiring is usually ignored in ASIC design, but it turns out that this construction's cost is not ignorable. With a wide data path, the distance from bit one to bit 128 is long enough to add to our total delay, impacting timing as well.

Even more than wire delay, the real difficulty in doing arbitrary permutations of the input bits is the cost of crossing wires. As integrated circuits are constructed in layers, in order to swap two wires, one must connect vertically across layers. Laying out an arbitrary permutation in silicon requires much more area than just connecting straight through from one set of gates to the next. A constrained permutation can still allow full mixing with much lower cost.

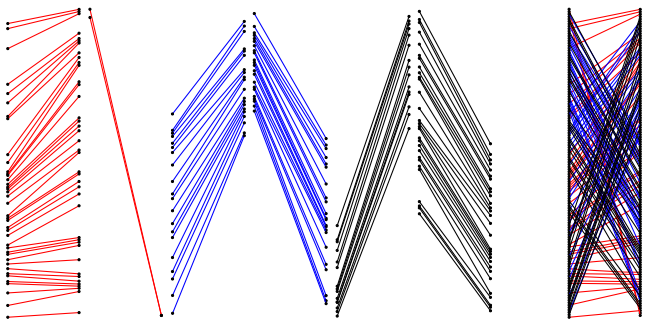


Figure 5.6: Example permutation with 6 layer separated and then combined

The following solution with constant wire crossing cost, is illustrated in Figure 5.6. We start with the input points and output points in parallel columns and produce a matching across these points as follows. All input points are divided into k groups. The output connec-

tions are divided into the same groups, ensuring the same number of input as output wires in each group. Within a group, the first input point is randomly connected to an output point, and the next input point to the next output point, returning to the first output point after reaching the last. If we label the input and output points with integers 0 to $m-1$, we choose $r = \text{rand}(m)$ and connect input point i to output point $o = (i + r) \mod m$, $0 \leq i < m$. Doing this for each group requires two layers per group, one for the wires $i < o$ and the other for $i > o$. Figure 5.6 shows a 3-group permutation with red, blue and black rotations shown in separate layers, together with the combined permutation resulting from their combination. In some of our tests, we use two rounds of this kind of permutation, and will discuss the effects of this in Section 5.6.

5.4 Evaluation

In this section, we propose three hash testing criteria: generalized uniformity test, generalized universality test and generalized Avalanche test. For convenience of discussion, we also present evaluation results in this section.

In our experiments, we use data from CAIDA's Anonymized 2010 Internet Traces Dataset [98]. We use the Src IP, Dst IP, Src port, Dst port and sequence numbers from these traces as hash input, totaling 128 bits per packet. It should be noted that we removed the duplicated tuples which are generally caused by TCP retransmission. Failing to do this would introduce additional non-uniformity into the results, with duplicate inputs leading to duplicate outputs. TCP sequence numbers are designed to be unpredictable as they're required to resist data insertion attacks, so including these adds significantly to the entropy of the input. Moreover, although CAIDA's anonymization is prefix-preserving, this might introduce more

randomness into the set of input values than would be expected on a production router. This gives poor hash functions an advantage, as having more input variation requires less processing to produce widely varying output. Even with this entropy advantage, our tests can still differentiate levels of output quality.

5.4.1 Hash functions for Comparison

The hash functions to be compared against our function are the following: CRC-32 (IEEE 802.3), two H_3 functions [86] with pre-generated random matrices, Pearson hash [89], Buz-Hash function [90], Bob Jenkins's hash [92], Hsieh's SuperFastHash [99] and MD5 [100]. As discussed in Section 5.2, only CRC and H_3 come close to fitting our size and latency restrictions, and all other hash functions would not meet our design restrictions. The following comparisons are only on output quality without considering implementation cost.

All functions were configured to take 128 bits of input. Some functions could only return 32 bits of result, hence for a fair comparison, we only compare the lower 32 bits. Since our constructions produce equally unpredictable output bits, using only a subset of them still fairly represents the original.

For convenience of comparison to a uniform distribution, we also included a random number generator (MT19937), which generates uniform numbers independent of the input. The results of its tests should show optimal results to within experimental error.

As the methodology to build our hash functions is quite flexible, we present a family of hash function constructions. The general methodology is to cascade XOR and 3x3 S-box stages. For convenience of notation, we use x to denote XOR stage and t to denote the 3x3 S-box stage (t for three). If more than one stage of the same type are cascaded, we put the repetition count after the stage identifier. For example, the configuration name $tx4t$ means

one S-Box stage, four XOR stages, and another S-box stage, with a permutation block in between each stage. In Section 5.3.5, we have discussed the method of using more regular connections between stages to simplify the routing. The suffix "L6" denotes a hash function using the (3 group) 6-layer permutation block demonstrated in Figure 5.6 and "RL" indicates a completely random permutation of all 128 bits.

We select $tx4tx3t$,$tx4tx4t$, $tx3tx3tx3t$ and $tx5tx5t$ as possible configurations of our hash function based on size and output quality. The configurations $t15$ and $x15$ are included only for comparison purposes, despite their impracticality.

5.4.2 Generalized Uniformity Test

We could characterize the problem of evaluating hash functions in the following way: For an arbitrary set S, we want to test whether $h(S)$ is uniformly distributed across all output values. This means we want to see whether $h(S)$ is statistically the same as independently, identically distributed numbers generated from a uniform distribution with the same range.

However, the range of output values is too large for any direct test. The natural solution is to group the output values and then perform the balls-and-bins test on the groups. We can think of this as projecting a sparse mass in a high-dimensional space into a low-dimension space. The mapping proposed by us is to randomly select a subset of the output bits, and group hash outputs based on their value of these bits. Still using the analogy of projection, the mapping proposed by us is to select some axes on the space and collapse all other dimensions. Then we use those mapped results to perform the uniformity test.

Another reason for this method of grouping is that we want to test the dependence among bits and want to reveal possible correlations. This kind of projection would reveal the bias or correlation among the chosen output bits. For example, if the 13^{th} bit is correlated with

18^{th}bit, when we keep those two axes, we'll find a non-uniformity in the resulting data, as all four combinations of those two bits won't appear equally often. As people assume that hash function output is indistinguishable from random numbers, they often divide the result into multiple values. This testing is important to know whether those values depend on each other.

To perform our generalized uniformity test, for each of M rounds, fix K output bits by sampling without replacement. Each round consists of hashing N input values, projecting out the K output bits for that round and counting the frequency of each of the 2^K possible outcomes. We use M on the order of 10 to 100, K from 8 to 12 and depending on K, N from 400 to 10^6, with larger N for larger K.

Compared to the previous methods proposed in [96] and the common ones that could be found in the Internet, the major difference here is that our number of bins is a power of 2 so we're not mixing any more during the test, and we choose bits other than the low order bits.

For each round, we use the standard Chi-square test of statistics (Eq. 5.1) as follows:

$$S = \sum_{i=1}^{2^K} \frac{(\#\text{Balls in Bin}_i - ExpectedLoad)^2}{ExpectedLoad},$$

where $ExpectedLoad = N/2^K$.

For each round, we could have a p-value $\alpha = F^{-1}_{\chi^2}(S)$. Here $F^{-1}_{\chi^2}$ is the inverse cumulative distribution function of χ^2 distribution with $2^K - 1$ degree of freedom. Then $1 - \alpha$ is the tail probability to observe S if the hash values are truly uniformly distributed. If the hash values are sampled from a uniform distribution, α of each round should be uniformly distributed within $[0, 1]$.

Since there are multiple rounds, we need to summarize the results of all rounds for comparison purposes. Here we use the average and the maximum of α values to get two values: α_{mean} and α_{max}, which serve as the final metrics of a hash function. The p-values for α_{mean} and α_{worst} can be derived as follows:

$$\alpha_{mean} = F_{us}^{-1}(\frac{1}{M}\sum_{i=1}^{M}\alpha_i),$$

where F_{us} is the distribution of the sum of M uniformly distributed numbers on $[0, 1]$ and

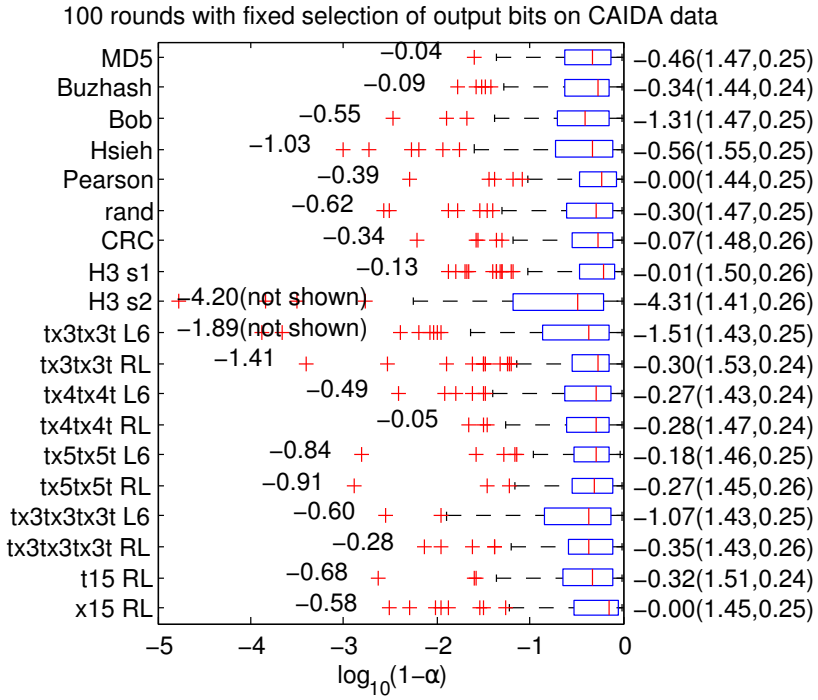
$$\alpha_{worst} = (\max\{\alpha_i\})^M.$$

The two formulas above hold only when the data sources of each round are independent.

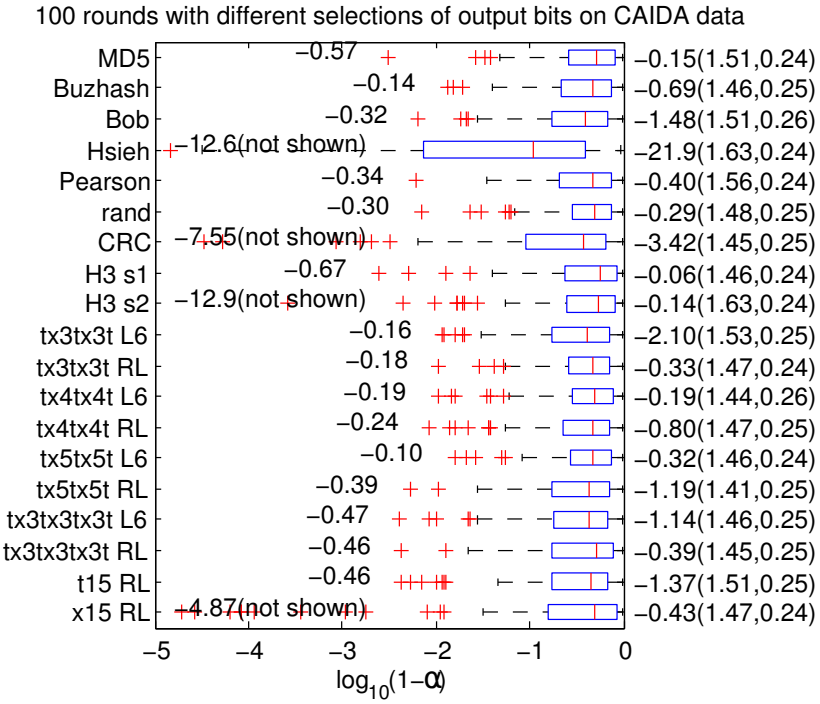
It is worth pointing out that here we use the one-sided p-value. A smaller α indicates more uniform hash values. If α is too close to 0, it means the hash values are more "uniformly distributed" than what we would expect from independent random choices. Such a hash function would be suspicious, but should not be rejected by a uniformity test. Hence we only use the one-side upper-tail p-value $1 - \alpha$ across all statistics.

It is also worth pointing out that multiple rounds should use different non-overlapping segments of data sources, otherwise the hash results would be correlated. This would compromise our ability to summarize the results of multiple rounds, no matter which function was used.

An example of testing result is shown in Figure 5.7 (a) and (b). This box-and-whisker plot shows the α values generated in 100 rounds with 10^5 balls per round hashed into 1024 bins. For convenience of comparison, the numbers are shown as $\log_{10}(1-\alpha)$. The numbers in



(a) Fixed output bits



(b) Dynamic bit selection

Figure 5.7: Box-and-whisker plot of the results of Uniformity tests on real trace
Along the right axis is α_{mean} (maximum load ratio, minimum load ratio) across all rounds. The load ratio is load of some bucket divided by average load. Inside the figure is the α_{worst}.

(a) are generated by a fixed selection of bits, while the αs in (b) are generated by a changed selection for each round.

We also printed the two p-values that summarize the results of all rounds, i.e. $\log_{10}(1 - \alpha_{worst})$ and $\log_{10}(1 - \alpha_{mean})$, inside the figure and on its right axis separately. [2] It is customary to declare the observations as non-uniform only when the p-value is less than 5% or 1%.

If we fix our choice of bits across rounds, we get the results shown in Figure 5.7 (a), showing that most functions do well. The exceptions are $H_3(s2)$ and $tx3tx3tL6$. $H_3(s2)$ has one worst case $10^{-6.2}$ (cut off by the figure boundary) with probability $10^{-4.2}$ to observe this kind of worst case happens in 100 rounds, although on average it performs well. Our hash function $tx3tx3t\ L6$ is slightly bad, having a worst case α of $10^{-1.89} \approx 1.29\%$. All other numbers indicate no significant flaws in the hash functions.

However, if we dynamically change the selection of outputs of hash function for each round, as shown in Figure 5.7 (b), we see that some combinations of bits aren't as uniform as others. The Hsieh's SuperFastHash starts to show poor worst case results (some points out of boundary are not shown in the figure) and its average α is also the worst. Testing CRC and H_3 with two different fixed seeds, we find many occurrences of very low performance, while their average cases are still good.

We also print the value of the maximum/minimum load ratio of the bins across all rounds at the right of the figure, inside parentheses. We observe that it is hard to summarize the

[2] It is worth noting that the α_{mean} and α_{worst} are not as simply related as the mean and the minimum of a dataset. The tail probability of an "average performance" is α_{mean}. If it is very small, it means the mean α is uncommon and the hash output is often of low quality. The tail probability of a "worst-case performance" is α_{worst}. If it is very small, it means the worst α observed is very extreme, meaning that the quality of this hash function is at least volatile (it might be consistently of very low quality, or sometimes very low, sometimes acceptable). With this in mind, it is possible for either of these two values to be larger than the other. They are just two scores from slightly different perspectives.

results of those maximum/minimum load ratios since they are almost all in the same range. There are only two exceptions: the results of the Hsieh's and $H_3(s2)$ in Figure 5.7 (b), which are 1.63, whose α_{worst} is very bad. This shows that Chi-square test is powerful for testing uniformity of the output values. Low chi-square results might not necessarily lead to a bad maximum load of hash bins. The results for $H_3(s2)$ in Figure 5.7 (a), show a poor chi-square result while its max/min load is still good. A hash function with a more uniform distribution should still be more useful for other purposes than a hash table.

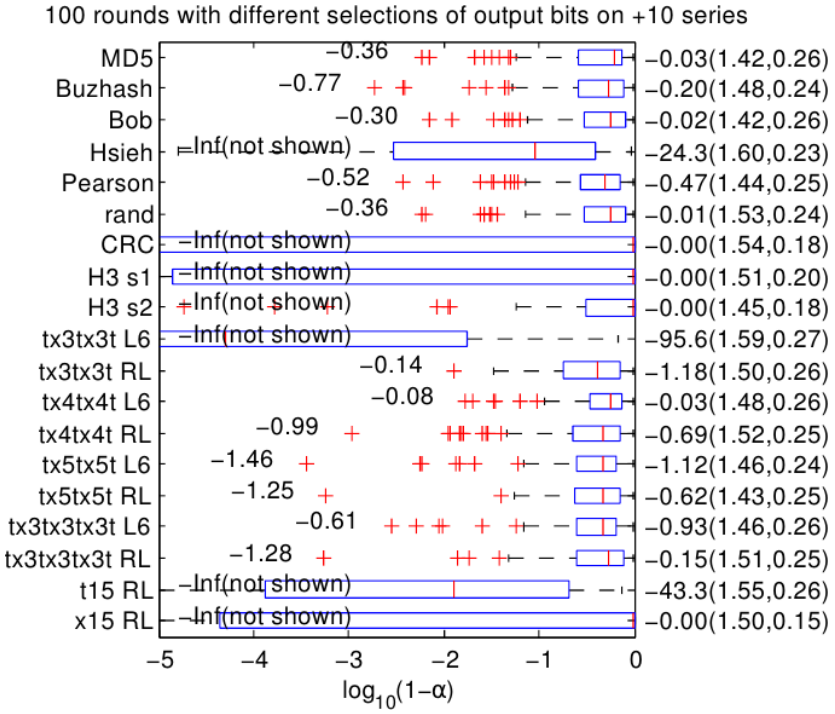


Figure 5.8: Uniformity results for "+10" series

We also run experiments using linear input sequences, which is also a common way to test networking devices [91]. Figure 5.8 shows the results of having input data of the form $x + 10i$ for a constant x and $0 \leq i < N$, with different output bits chosen each round of testing. The linear hash functions CRC and H_3 degrade in performance significantly on this test, while most other hash functions tested show consistent performance.

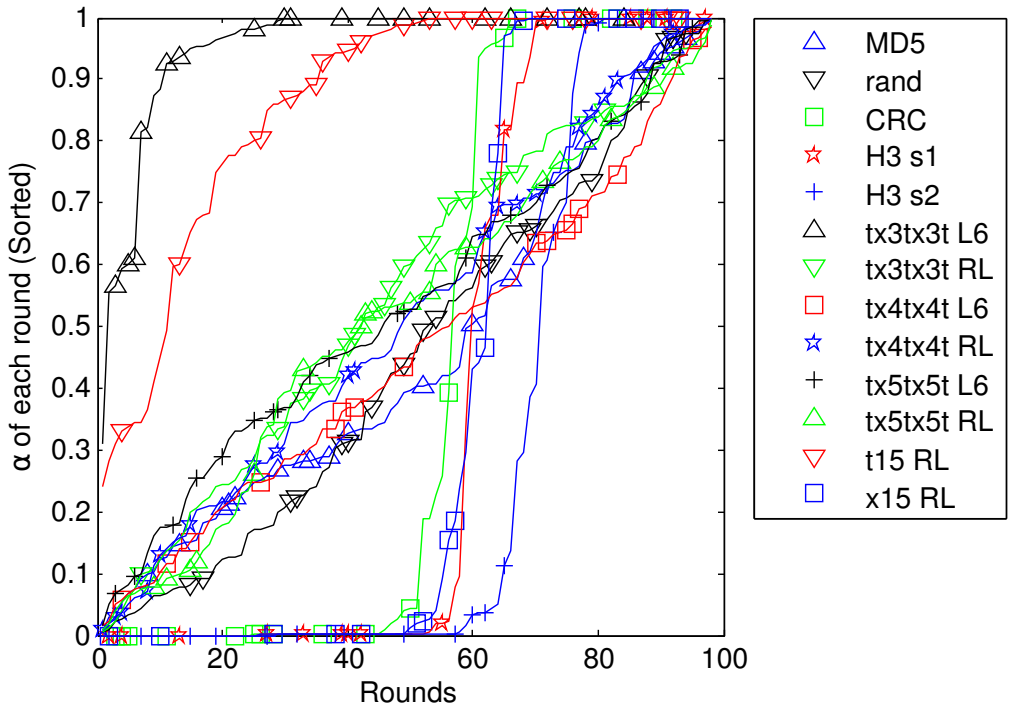


Figure 5.9: Q-Q plot for "+10" series *Many functions which perform well are removed from the figure.*

We also show the q-q plot of the α values for the linear sequence test. By sorting all the p-value αs of each round of each hash function in Section 5.9, we can visualize how often a test had a certain range of results. For an ideal hash function, this plot would show a diagonal line, since α would follow uniform distribution in $[0, 1]$. As most functions meet this expectation, we remove many functions from the figure for clarity. The two worst performing functions on the top of the plot are $t15$ and $tx3tx3tL6$. They have mostly large α values, indicating regularly non-uniform results. The outliers on the bottom of the plot are the hash functions with linear structure, including CRC and H_3. These give unusually frequently uniform results and too frequent non-uniform output, indicating uneven output bit quality.

5.4.3 Avalanche Test

Our Avalanche test is similar to the ones used in [101]. Instead of always testing on random input data, we use lower entropy input sets. We do this because we believe strong hash functions should have the avalanche property even for test sequences such as the "+1" test sequences.

A different variation on the Avalanche test is given by Castro et. al [102], where functions are tested to see if the hamming distances between hash values follow a binomial distribution. Given $H(x, y)$ as the hamming distance from x to y and n-bit output values they test whether

$$H(f(x), f(y)) \sim Bin(\frac{1}{2}, n)$$

This gives a single value for how accurately the given hash function passes the avalanche test, but obscures the valuable information of where the bit dependencies are. This information is valuable for improving a hash function in development, as changes to the mixing functions can remove these dependencies.

In Table 5.1, we present results of testing Avalanche Properties on two input sets. The first is composed of 10^5 random numbers and the second is the integers from 1 to 10^5 in order. For each input value, we test the 128 hash outputs generated by inverting each input bit. In total we have 128×32 input-output pairs and we track how frequently each output bit inverts as a result of each input bit being inverted. The $\pm 1\%$ and $\pm 5\%$ columns are the percentage of pairs that fall into $[49\%, 51\%]$ and $[45\%, 55\%]$ categories. The max $\pm\%$ column is the largest deviation from 50% among all the pairs.

Because of the size of our input, even the random number generator had some bits that weren't evenly distributed, as shown in the *max* column 0%. The hash function closest to

Hash	Real Trace % in ±2%	Real Trace % in ±5%	Real Trace max ±%	Sequential input % in ±2%	Sequential input % in ±5%	Sequential input max ±%
MD5	100.0	100.0	0.6	100.0	100.0	0.7
Bob	96.6	99.2	12.5	96.1	98.7	14.1
Buzhash	28.6	72.6	12.9	3.7	8.7	50.0
Hsieh	99.2	99.6	14.0	99.0	99.7	14.2
Pearson	95.2	98.1	15.5	36.0	75.0	18.8
rand	100.0	100.0	0.6	100.0	100.0	0.6
tx3tx3t L6	18.9	18.9	50.0	28.8	30.1	50.0
tx3tx3t RL	41.1	41.1	50.0	50.8	52.2	50.0
tx4tx4t L6	71.2	71.2	50.0	74.8	75.4	50.0
tx4tx4t RL	94.5	94.5	25.2	96.0	96.1	25.0
tx5tx5t L6	97.3	97.3	12.9	97.9	97.9	12.9
tx5tx5t RL	100.0	100.0	6.6	100.0	100.0	6.1
tx3tx3tx3t L6	96.5	99.6	7.9	95.2	99.4	6.7
tx3tx3tx3t RL	100.0	100.0	0.9	100.0	100.0	1.5
t15 RL	60.3	97.8	12.4	71.4	93.4	25.6

Table 5.1: Avalanche with real trace and +1 sequence

random is MD5, then is our $tx3tx3tx3tRL$. It is also interesting to observe that, although the performance of most hash functions are consistent across the input sets, Buzhash and Pearson differ between random and sequential input. This is likely caused by the way they process input byte-by-byte.

5.4.4 Universality Test

In practice, people usually want a hash function that gives very different results by changing its seed. A common way to accomplish this is to design one hash function $H(x)$, and reserve part of the input as the seed. We write seeded hash functions as $H(x, s) = y$. [3]

Formally, for any arbitrary seed s_1, s_2 and hash value x, we want to test whether $H(s_1, x)$ and $H(s_2, x)$ are independent. We will assume that hash values $H(s_1, x)$ and $H(s_2, x)$ are

[3]It should be noted that, when you want to have the 128-bit input and 16-bit seed, the only correct way is to build it based on a 144-input hash function. It is easily shown to be nearly equivalent if the seed is simply XORed with the input.

uniformly distributed in the space, i.e. H is a good function. Then we can test for the uniformity of $H(s_1, x)\&H(s_2, x)$, where & is the concatenation of the two hash outputs. Since two random variables X_1 and X_2 uniformly distributed in $[0, d-1]$ are independent if and only if $X_1 \times d + X_2$ is uniformly distributed in $[0, d^2 - 1]$, this test will determine the independence of the hash function with different seeds. The space to be tested is very large, so we recommend using the projection method from 5.4.2 to reduce the test complexity.

The testing procedure follows the following steps: We first divide up the input bits into three groups: fixed, seed and variable. The fixed bits we keep at an arbitrary, fixed value through testing we call f. Each round, we use two different seed values for the seed bits, s_1 and s_2. The remaining bits give the variation; within a round we test all combinations of those bits, denoted x. Each output value is reduced to p bits by removing the same bits from each. On each round, we select s_1 and s_2 and count the frequencies of all 2^{2p} outcomes possible from the projection of $H(f\&s_1\&x)\&H(f\&s_2\&x)$ for all possible x. This process is summarized in Figure 5.10. One important detail of this process is that if seed pairs (s_1, s_2) and (s_1, s_3) have been tested, (s_2, s_3) should not be tested, as the results of these tests wouldn't be independent.

This test could be formulated with a larger seed instead of holding bits fixed. We use the current configuration of test since big changes in seed would give the hash function more unpredictability on its inputs to produce unpredictable outputs with. We reduce the variation in seed values to stress the hash function by changing fewer bits in seeds to better determine its output quality.

For each round, we observe the largest and lowest load of the bins, and perform the chi-square test for uniformity on the distribution of the load of bins. We summarize multiple rounds using the method in Section 5.4.2.

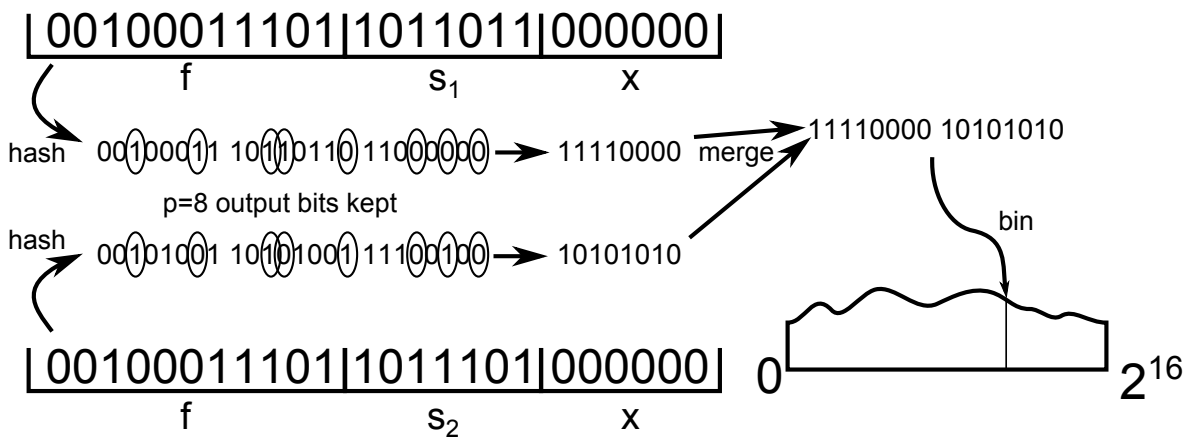


Figure 5.10: Procedure for testing universality

Before we come to present the results of this test, we connect this test with the idea of universal hashing. Can we test whether a function employed in practice has the universality property? We know that commonly used hash functions aren't proven to be in a universal hashing function family. There may be a statistical method to test if a function is "approximately" universal.

The 2-universal hashing is defined for a hash function f with seed space S as

$$\forall x_1, x_2, y_1, y_2, P[f(x_1) = y_1 (and) f(x_2) = y_2] \approx \frac{1}{|S|^2}.$$

The natural way to statistically test for universality is to select a fixed pair of x_1 and x_2, traverse all possible S, and get the distribution of the pair of y_1 and y_2. Since the space of y and S is large, bit projections are needed to make the computation feasible. Surprisingly, the procedure of this test is the same as the universality test already described. For this reason, we call our test as Universality Test. The nature of this test is to discover more into the "2-D" structure of the hash values.

Here we present an example of the result of this test in Figure 5.11. We use 18-bits of x, 5-bits of seed, and project out 6 bits from each hash result, resulting in 1M balls in 64K bins.

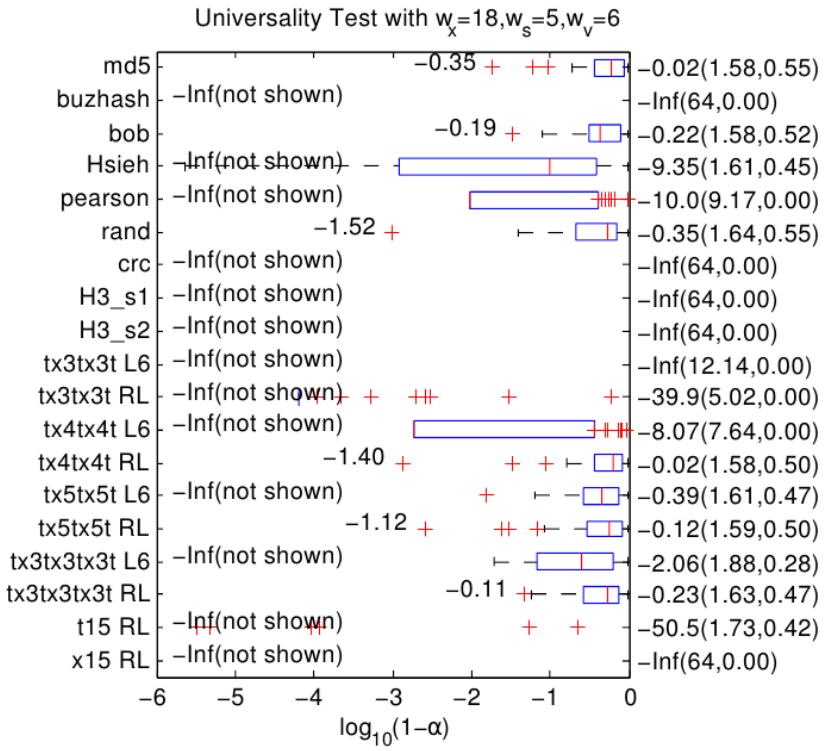


Figure 5.11: Universality testing results

Actually all results are similar and we don't select this configuration for a specific reason. The selections of the 18-bits x and 5-bits s from the input bits are randomly pre-configured. In total we could have $2^5 - 1$ pairs of seeds, since pairs should share at most one common seed as discussed above. Hence there are 31 rounds in total.

The results are shown in Figure 5.11, the meaning of all numbers are the same as the ones in generalized uniformity test. It is clear that much more functions fail this test. Not surprisingly, CRC and the linear families fail the test because of their linear structure. BuzHash also performs poorly on this test, with the worst possible max/min load ratios. Looking more closely at BuzHash's bin size distribution, we observe that BuzHash still has more variation in its bin size than linear hash functions, despite the same reported load ratios.

It's also interesting to observe that even the Pearson hash fails this test both in having a very non-uniform seed pairing as well as having very poor worst case bucketing. Many of our

constructions fail this test, with only those using full random permutations passing. Even many of those that failed still had reasonable worst case bucket sizes.

5.5 Future Work

It may be possible to improve the output quality with a different staging pattern. Since the 3×3 S-box stage is only a little bigger than an XOR stage (although the delay is much larger), we may be able to improve hash quality by trading XOR stages for S-box stages. Although we have only done a small exploration of the possibilities of this framework, we are sure that both are important, play different roles in our architecture, and should be deployed in a balanced way. Using only XOR stages, $x15$, or only S-box stages, $t15$, results in poor performance in most of the tests. As combining XOR stages produces a more complex but still linear XOR stage, the results for $x15$ are unsurprising.

There is more that can be done to discover better hash functions within our general framework, mainly in the realm of increasing the performance per unit cost of the permutation rounds. We expect our evaluation methods to be useful in guiding this optimization and leading to better hash functions.

5.6 Conclusion

In the light of current networking trends, we investigated existing hash functions and found them lacking in either output quality or implementation cost or latency in hardware. Also, we found that current evaluation methodologies are ineffective for the quality range we were investigating. In this paper, we developed a generalized evaluation framework to quantify the randomness of hash functions for those applications and evaluated various hash functions. We

also investigated strategies for building hash functions and present a family of hash functions that is not only small and fast to implement, but also scores high in our evaluations.

To summarize the hash quality results above, Bob Jenkin's hash is the only previously known non-cryptographic hash that passes all our tests. Among our candidate hash functions, only $tx3tx3tL6$ fails any uniformity test. Our function $tx3tx3tx3tRL$ performs nearly as well on the Avalanche test as $MD5$. The constructions using RL permutations (full random mixing) do well on the universality test.

Regarding the implementation costs of our proposed hash functions, we have performed a preliminary synthesis. This synthesis uses 45nm technologies with clock speeds at 1GHz, meaning the logic costs for these designs are much higher than the designs in Satoh and Inoue [88]. Based on our current synthesis results, the random permutation uses around 5% more cell area than the 6-layer simplified permutation. Constraining the synthesis so that our hash function completes in a single cycle, a 128×128 random XOR matrix, needs around 3.3K cell areas. A comparable construction is the pattern $tx3tx3tRL$, which needs around 3.9K cell areas. This 9-stage hash function gives good uniformity results, moderate avalanche, and good universality results, and is able to hash data at a rate of 16GB/s with a single cycle of latency.

Achieving more than this with single cycle latency increases the cell count significantly, as long chains of dependent gates have to be replaced by more complex constructions with shorter dependency length. The 13-stage $tx5tx5tRL$ needs over 8K cell areas to complete in a single cycle. We provide a methodology to strike a tradeoff between hashing quality and cost, not only optimized results for the 4K gate count.

Hardware hash design for networking systems is polarized. One style of hash design is exemplified by MD5 and SHA-1. This world focuses on very high output quality crypto-

graphic hashes that are mandated by external requirements and used for security purposes. The opposite end of hash design is not constrained by external standards, and has mediocre output quality in exchange for simple implementations. We expect this work to open new exploration in the field of hardware hash functions to bridge the gap with high output quality, low cost hash functions. Success in this area will help new high speed routers keep pace with bandwidth demands with a minimum of pathological cases for a highly reliable Internet.

REFERENCES

REFERENCES

[1] "Simple object access protocol (soap), www.w3.org/tr/soap/."

[2] "XML-RPC, http://www.xmlrpc.com/spec."

[3] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: vulnerability-driven network filters for preventing known vulnerability exploits," in *Proceedings SIGCOMM*, 2004.

[4] Z. Li, L. Wang, Y. Chen, and Z. Fu, "Network-based and attack-resilient length signature generation for zero-day polymorphic worms," *IEEE International Conference Network Prorocols (ICNP)*, pp. 164–173, 2007.

[5] N. Schear, D. R. Albrecht, and N. Borisov, "High-speed matching of vulnerability signatures," in *Proceedings International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," *IEEE Symposium Security and Privacy*, 2007.

[7] R. Pang, V. Paxson, R. Sommer, and L. Peterson, "binpac: a yacc for writing application protocol parsers," in *Proceedings ACM Internet Measurement Conference (IMC)*, 2006.

[8] N. Borisov, D. J. Brumley, and H. J. Wang, "A generic application-level protocol analyzer and its language," in *Proceedings Network and Distributed System Security Symposium (NDSS)*, 2007.

[9] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier, "An FPGA-based coprocessor for the parsing of context-free grammars," in *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2000, p. 236.

[10] ——, "An FPGA-based syntactic parser for real-life almost unrestricted context-free grammars," in *Proceedings 11th International Conference on Field-Programmable Logic and Applications (FPL)*, 2001, pp. 590–594.

[11] A. Koulouris, N. Koziris, T. Andronikos, G. Papakonstantinou, and P. Tsanakas, "A parallel parsing VLSI architecture for arbitrary context free grammars," in *Proceedings International Conference on Parallel and Distributed Systems (ICPADS)*, 1998, p. 783.

[12] Y. H. Cho and W. H. Mangione-Smith, "High-performance context-free parser for polymorphic malware detection," in *Proceedings Advanced Networking and Communications Hardware Workshop*, 2005.

[13] J. Moscola, Y. H. Cho, and J. W. Lockwood, "Reconfigurable context-free grammar based data processing hardware with error recovery," in *Proceedings 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[14] J. Moscola, J. W. Lockwood, and Y. H. Cho, "Reconfigurable content-based router using hardware-accelerated language parser," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 2, pp. 1–25, 2008.

[15] Y. H. Cho, J. Moscola, and J. W. Lockwood, "Context-free-grammar based token tagger in reconfigurable devices," in *Proceedings ACM/SIGDA 14th International symposium on Field programmable gate arrays (FPGA)*, 2006, pp. 237–237.

[16] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv, "NetShield: Massive semantics-based vulnerability signature matching for high-speed networks," in *Proceedings SigCOMM*, 2010.

[17] "Ethereal OSPF protocol dissector buffer overflow vulnerability. http://www.idefense.com/intelligence/vulnerabilities/display.php?id=349."

[18] "Snort TCP stream reassembly integer overflow exploit, http://www.securiteam.com/exploits/5bp0o209ps.html."

[19] "tcpdump ISAKMP packet delete payload buffer overflow. http://xforce.iss.net/xforce/xfdb/15680."

[20] "Symantec multiple firewall NBNS response processing stack overflow. http://research.eeye.com/html/advisories/published/ad20040512a.html."

[21] C. Shannon and D. Moore, "The spread of the witty worm. http://www.caida.org/research/security/witty/."

[22] A. Kumar, V. Paxson, and N. Weaver, "Exploiting underlying structure for detailed reconstruction of an internet-scale event," in *Proceedings ACM Internet Measurement Conference (IMC)*, 2005.

[23] S. C. Johnson, "Yacc - yet another compiler-compiler," Bell Laboratories, Technical Report 32, 1975.

[24] T. T. J. Parr and R. R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software, Practice and Experience*, vol. 25, 1995.

[25] "Darpa intrusion detection evaluation data set," www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1998data.html, 1998.

[26] "Harvestman," http://code.google.com/p/harvestman-crawler/, 2013.

[27] "Tcmalloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2011.

[28] R. Smith, C. Estan, and S. Jha, "Xfa: Faster signature matching with extended automata," in *Proceedings IEEE Symposium on Security and Privacy*, 2008, pp. 187–201.

[29] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *Proceedings SIGCOMM*, 2008, pp. 207–218.

[30] L. Yang, R. Karim, V. Ganapathy, and R. Smith, "Fast, memory-efficient regular expression matching with NFA-OBDDs," *Computer Networks*, vol. 55, no. 55, pp. 3376–3393, 2011.

[31] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings SIGCOMM*, 2006, pp. 339–350.

[32] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2006, pp. 93–102.

[33] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of ACM CoNEXT*. ACM, 2007.

[34] ——, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings CoNEXT*, 2008, pp. 1–12.

[35] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proceedings ACM/IEEE ANCS*, 2007, pp. 155–164.

[36] M. Bando, N. Artan, and H. Chao, "Scalable lookahead regular expression detection system for deep packet inspection," *Networking, IEEE/ACM Transactions on*, vol. 20, no. 3, pp. 699 –714, june 2012.

[37] J. Kořenek and V. Košař, "Nfa split architecture for fast regular expression matching," in *Proceedings ANCS*. New York, NY, USA: ACM, 2010, pp. 14:1–14:2. [Online]. Available: http://doi.acm.org/10.1145/1872007.1872024

[38] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on fpga," in *Proceedings ANCS*, 2008, pp. 30–39. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477948

[39] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on fpga," in *Proceedings DATE*, 2006, pp. 12–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1131355.1131359

[40] I. Bonesana, M. Paolieri, and M. D. Santambrogio, "An adaptable fpga-based system for regular expression matching," in *Proceedings DATE*, 2008, pp. 1262–1267. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403681

[41] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling pcre to fpga for accelerating snort ids," in *Proceedings ANCS*. New York, NY, USA: ACM, 2007, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/1323548.1323571

[42] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," in *Proceedings Field-Programmable Logic and Applications*, 2003, pp. 956–959.

[43] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines FCCM*, 2001, pp. 227–238.

[44] C. Meiners, J. Patel, E. Norige, E. Torng, and A. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *Proceedings 19th USENIX Security*, 2010.

[45] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based dfa deflation for fast and scalable regular expression matching using tcam," in *Proceedings ANCS*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 24–35. [Online]. Available: http://dx.doi.org/10.1109/ANCS.2011.13

[46] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[47] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings 13th Systems Administration Conference (LISA), USENIX Association*, November 1999, pp. 229–238.

[48] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999. [Online]. Available: citeseer.ist.psu.edu/paxson98bro.html

[49] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings ACM/IEEE ANCS*, 2007.

[50] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proceedings IEEE IISWC*, 2008.

[51] "Us army itoc research cdx 2009 trace," http://www.itoc.usma.edu/research/dataset/index.html, 2009.

[52] "A guide to search engines and networking memory," *http://www.linleygroup.com/pdf/NMv4.pdf*.

[53] D. E. Taylor, "Survey & taxonomy of packet classification techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, 2005.

[54] B. Agrawal and T. Sherwood, "Modeling TCAM power for next generation network devices," in *Proceedings IEEE International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 120– 129.

[55] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 488–500, 2012.

[56] C. Lambiri, "Senior staff architect IDT, private communication," 2008.

[57] P. C. Lekkas, *Network Processors - Architectures, Protocols, and Platforms*. McGraw-Hill, 2003.

[58] A. X. Liu and M. G. Gouda, "Complete redundancy removal for packet classifiers in tcams," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, in press.

[59] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary CAMs can be smaller," in *Proceedings ACM Sigmetrics*, 2006, pp. 311–322.

[60] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 490–500, April 2010.

[61] A. X. Liu and M. G. Gouda, "Complete redundancy detection in firewalls," in *Proceedings 19th Annual IFIP Conference on Data and Applications Security, LNCS 3654*, August 2005, pp. 196–209. [Online]. Available: http://www.cs.utexas.edu/users/alex/publications/Redundancy/redundancy.pdf

[62] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *Proceedings 27th Annual IEEE Conference on Computer Communications (Infocom)*, April 2008.

[63] H. Acharya and M. Gouda, "Firewall verification and redundancy checking are equivalent," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 2123–2128.

[64] R. Draves, C. King, S. Venkatachary, and B. Zill, "Constructing optimal IP routing tables," in *Proceedings IEEE INFOCOM*, 1999, pp. 88–97.

[65] S. Suri, T. Sandholm, and P. Warkhede, "Compressing two-dimensional routing tables," *Algorithmica*, vol. 35, pp. 287–300, 2003.

[66] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *Proceedings ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2007.

[67] R. McGeer and P. Yalagandula, "Minimizing rulesets for tcam implementation," in *Proceedings IEEE Infocom*, 2009.

[68] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, "Towards efficient implementation of packet classifiers in sdn/openflow," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 153–154.

[69] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal in/out tcam encodings of ranges," 2014.

[70] T. Mishra and S. Sahni, "Petcam–a power efficient tcam architecture for forwarding tables," *Computers, IEEE Transactions on*, vol. 61, no. 1, pp. 3–17, 2012.

[71] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst-case tcam rule expansion," *IEEE Transactions on Computers*, 2012.

[72] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *Proceedings 11th IEEE International Conference on Network Protocols (ICNP)*, November 2003, pp. 120– 131.

[73] R. Wei, Y. Xu, and H. Chao, "Block permutations in boolean space to minimize tcam for packet classification," in *INFOCOM, 2012 Proceedings IEEE*, march 2012, pp. 2561 –2565.

[74] Y. Chang, C. Lee, and C. Su, "Multi-field range encoding for packet classification in tcam," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 196–200.

[75] A. Bremler-Barr, D. Hay, D. Hendler, and B. Farber, "Layered interval codes for TCAM based classification," in *Proceedings of IEEE Infocom*, 2009.

[76] A. Bremler-Barr and D. Hendler, "Space-efficient TCAM-based classification using gray coding," in *Proceedings 26th Annual IEEE Conference on Computer Communications (Infocom)*, May 2007.

[77] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Sax-pac (scalable and expressive packet classification)," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 15–26.

[78] K. Kogan, S. Nikolenko, P. Eugster, and E. Ruan, "Strategies for mitigating tcam space bottlenecks," in *High-Performance Interconnects (HOTI), 2014 IEEE 22nd Annual Symposium on*. IEEE, 2014, pp. 25–32.

[79] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 1231–1239.

[80] D. Maier, "The complexity of some problems on subsequences and supersequences," *J. ACM*, vol. 25, no. 2, pp. 322–336, Apr. 1978. [Online]. Available: http://doi.acm.org/10.1145/322063.322075

[81] M. G. Gouda and A. X. Liu, "Firewall design: consistency, completeness and compactness," in *Proceedings 24th IEEE International Conference on Distributed Computing Systems (ICDCS-04)*, March 2004, pp. 320–327. [Online]. Available: http://www.cs.utexas.edu/users/alex/publications/fdd.pdf

[82] A. Z. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, 2003.

[83] J. Xu, "Tutorial on network data streaming," *ACM Sigmetrics*, 2008.

[84] D. Knuth, *The Art of Computer Programming 3: Sorting and Searching*. Addison-Wesley, 1968.

[85] S. Goldberg and J. Rexford, "Security vulnerabilities and solutions for packet sampling," in *Sarnoff Symposium, 2007 IEEE*. IEEE, 2008, pp. 1–7.

[86] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143 – 154, 1979. [Online]. Available: http://www.sciencedirect.com/science/article/B6WJ0-4B55K9J-D/2/036439eff8b0d54d7974c2d5d6997669

[87] MIPS Technologies, Inc., "Mips32 4ke family," Nov 2010, http://www.mips.com/products/cores/32-64-bit-cores/mips32-4ke/.

[88] A. Satoh and T. Inoue, "Asic-hardware-focused comparison for hash functions md5, ripemd-160, and shs," *Integration, the VLSI Journal*, vol. 40, no. 1, pp. 3–10, 2007.

[89] P. K. Pearson, "Fast hashing of variable-length text strings," *Commun. ACM*, vol. 33, pp. 677–680, June 1990. [Online]. Available: http://doi.acm.org/10.1145/78973.78978

[90] R. Uzgalis and M. Tong, "Hashing myths," Technical Report 97, Department of Computer Science University of Auckland, July 1994.

[91] D. Newman and T. Player, "Hash and Stuffing: Overlooked Factors in Network Device Benchmarking," RFC 4814 (Informational), Internet Engineering Task Force, Mar. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4814.txt

[92] R. Jenkins, "The hash," Nov 2010, http://burtleburtle.net/bob/hash/doobs.html.

[93] A. Appleby, "Murmurhash 2.0," Nov 2010, http://sites.google.com/site/murmurhash/.

[94] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for multipoint measurements," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 39–50, 2008.

[95] A. F. Webster and S. E. Tavares, "On the design of s-boxes," in *CRYPTO '85: Advances in Cryptology*. London, UK: Springer-Verlag, 1986, pp. 523–534.

[96] M. Molina, S. Niccolini, and N. Duffield, "A comparative experimental study of hash functions applied to packet sampling," in *International Teletraffic Congress (ITC-19), Beijing*. Citeseer, 2005.

[97] Artisan Components, "Tsmc 0.18mm process 1.8-volt sage-xtm standard cell library databook," Nov 2010, http://www.ece.virginia.edu/~mrs8n/cadence/SynthesisTutorials/tsmc18.pdf.

[98] P. H. kc claffy, Dan Andersen, "The caida anonymized 2010 internet traces - mar 25, 2010," http://www.caida.org/data/passive/passive_2010_dataset.xml.

[99] P. Hsieh, "Hash functions," Nov 2010, http://www.azillionmonkeys.com/qed/hash.html.

[100] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321 (Informational), Internet Engineering Task Force, Apr. 1992, updated by RFC 6151. [Online]. Available: http://www.ietf.org/rfc/rfc1321.txt

[101] B. Mulvey, "Hash functions," Nov 2010, http://bretm.home.comcast.net/~bretm/hash/.

[102] J. Castro, J. Sierra, A. Seznec, A. Izquierdo, and A. Ribagorda, "The strict avalanche criterion randomness test," *Mathematics and Computers in Simulation*, vol. 68, no. 1, pp. 1–7, 2005.