TOWARD ROBUST NETWORK PAYLOAD INSPECTION

by

Randy David Smith

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2009

To my family

ACKNOWLEDGMENTS

Many people have played a role either directly or indirectly in shaping this thesis. I am indebted to my advisors Cristian Estan and Somesh Jha, who over the course of the last few years guided me throughout this work. Their insights, energy, and encouragement where invaluable to me.

I have had the privilege of collaborating with many great students on problems directly related to this work as well as problems that were more peripheral. I would like to express my thanks to Shijin Kong, Dan Gibson, Daniel Luchaup, Drew Davidson, and Lorenzo De Carli for the opportunity to work with them.

My parents, Johnson and Rhea Smith, have always provided encouragement and support, even when they could not understand and I could not explain some of the challenges of the moment. Over the past few years, my sister Tiffany Morgan always provided invaluable help, even when I was perhaps undeserving. Thank you. Mom, Dad, and Tiffany: you have helped to make this dissertation possible.

In August 2003, I entered graduate school at the University of Wisconsin with plans to finish in at most four years, and maybe in as few as three years. Six years later I am now putting the finishing touches on this dissertation. Throughout this experience, my wife Laura and our children Andrew, Benjamin, Mary, and Dallin have been constant and faithful companions. They have consoled me in the disappointments and joined with me in the triumphs common to graduate school. They have been patient, understanding, and accepting when ever-present deadlines and late nights have pulled me away from time with them that they were entitled to. Knowing that these words are wholly inadequate, I express my deepest thanks to them for bearing with me, for being so constant and supportive, and for giving me their faith and trust. It is to them that I dedicate this dissertation.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

ABSTRACT

A Network Intrusion Detection System (NIDS) resides at the edge of a network and is tasked with identifying and removing all occurrences of malicious traffic traversing the network. At its heart is a signature matching engine that compares each byte of incoming and outgoing traffic to signatures representing known vulnerabilities or exploits and flags or drops traffic that matches a signature.

Signature Matching is fundamentally a language recognition problem. Signatures are commonly represented as regular expressions, which can be matched simultaneously by finite automata. Unfortunately, standard nondeterministic finite automata (NFAs) and deterministic finite automata (DFAs) induce a space-time tradeoff when combined and are unsuitable for NIDS use. NFAs are small and compact but too slow, whereas DFAs are fast but consume too much memory. Other alternatives such as filtering induce a tradeoff between accuracy and execution time.

We argue that the tradeoffs associated with signature matching are not fundamental obstacles, but instead result from limitations in existing models. We posit that with richer, more complex matching models, we can devise mechanisms that obviate these tradeoffs and have acceptable memory and performance profiles.

In an analysis of an existing NIDS, we show that the use of filters induces worst-case behavior that is six orders of magnitude slower than the average case and can be invoked by an adversary to enable perpetual evasion. We present an enhanced model, with an acceptably small increase in memory, that brings the worst case to within one order of magnitude of the average case.

Next, for DFA-based matching we present a first-principles characterization of the state-space explosion and subsequent memory exhaustion that occurs when DFAs are combined, and we give conditions that eliminate it when satisfied. We then show that through the careful inclusion of auxiliary state variables, we can transform automata so that they satisfy these conditions and do not lead to memory exhaustion. With this enriched model, we achieve matching speeds approaching DFAs with memory footprints similar to NFAs.

A NIDS operates in a constrained, adversarial environment. This work makes contributions towards enabling robust signature matching in such environments.

Chapter 1

Introduction

Network Intrusion Detection Systems (NIDS) have become critical components of modern network infrastructure. Functionally, at the core of any NIDS there resides a signature matching or payload inspection engine that potentially compares every byte of incoming and outgoing traffic to signatures from a database containing known exploits or misuses. In practice, though, implementations must balance a number of conflicting demands. Network speeds, signature counts, and signature processing requirements continue to increase yet are bounded by the limits of raw processing speeds, relatively small memories, and the need to maintain wire-speed performance. Failure to address these demands opens the door to evasion and may compromise the efficacy of a NIDS.

This work concerns the development and analysis of techniques and models for performing regular expression-based payload inspection of streaming data such as network traffic. Network intrusion detection provides the motivation and context for the work, but the results are general and may be applicable to other domains as well [76]. My thesis is that richer computation models can help improve the capability, performance, scalability, and robustness of payload inspection.

1.1 Network Intrusion Detection Systems

The use of a NIDS to protect network resources is in part an artifact of the continuing evolution of the Internet away from its original intended use. Initially, the Internet was designed for military use with the overall objective of reliably interconnecting existing but disparate networks together. Within this overall objective, a ranked list of priority and design goals [22] guided the development of the architecture. Military use presumes hostile environments, and the highest stated priority was to maintain communication in the presence of lost hosts, links, or entire networks [10]. Accounting and management of resources, which includes addressing the security of those resources, was ranked lowest. The importance of and need for accountability was understood, but as Clark [22] points out, “During wartime, one is less concerned with detailed accounting of resources than with ... rapidly deploying them in an operational manner”.

Today, the Internet permeates many facets of society and has become a key communications conduit for activities ranging from banking and financial transactions, to public utility and critical infrastructure management, to recreation and entertainment. Further, it is arguably more common for computing devices and networks to be connected to the Internet in some way than not. Either directly or indirectly, the Internet serves as a gateway to considerable assets, to private or sensitive information, and to mechanisms that can be used to control, manage, or disrupt these assets and information. Thus, whether through intended malice or accident, there is significant risk to the disruption of network communication or the unauthorized access of connected networks and information.

Unfortunately, the overall Internet architecture still retains the traits of its original design. With the steady increase in both the frequency and sophistication of malicious activity on the Internet, support for security–including access control, authentication, nonrepudiation, and integrity–has significantly grown in importance. But, these capabilities are not intrinsic to the architecture and effectively must be patched in. Of course, one approach is to re-architect the Internet to reflect the change in priorities. The recently proposed Accountable Internet Protocol [5] among others, for example, attempts to do just that by placing accounting, and accountability, squarely as the highest priority. Approaches such as these may eventually yield fruit, but there are significant hurdles in the way. The current Internet has become part of our infrastructure: the costs of “replacing” it are high, the timescales are long, and proposals are still being put forth, evaluated, and refined. And, if history is a guide, such changes are likely to occur incrementally. For the time being, it seems that we must do our best with the current architecture.

In this environment, devices such as a NIDS have been developed to shoehorn security features into network functionality. At a high level, a NIDS is essentially a filter. Simply put, its task is to identify and discard malicious or unwanted traffic before such traffic reaches its intended target, without disrupting the flow of legitimate traffic. Regarding terminology, intrusion detection focuses on identifying and flagging unwanted traffic, whereas intrusion *prevention* additionally removes the traffic. From the perspective of this work, the difference between detection and prevention is simply a policy decision, though; we use the term intrusion detection generally to refer to both detection and prevention.

1.2 NIDS Requirements: Between a Rock and a Hard Place

Functionally, NIDS operation can be described very succinctly: pattern match all network traffic against the signature database as it traverses the NIDS, and report (or drop) matching payloads. The operational requirements are daunting though, and NIDS performance needs continue to grow at a faster pace than resource availability. Some of the reasons for this are as follows:

1. **Increase in signature counts.** Despite recent focus on developing and maintaining secure software, software continues to be produced with vulnerabilities, new vulnerabilities are discovered in existing software, and novel attacks and exploits continue to appear. This translates to new signatures that must be written and added to the database. Adding signatures is unfortunately all too common an occurrence. Over a two year period (from April 2005 to April 2007), for example, the number of signatures in the Snort rule database [91] increased almost three-fold, from 3,166 to 8,868. The direct consequence is that larger numbers of signatures must be matched against payloads, increasing the load on the NIDS.

2. **Increase in signature complexity.** Intrusion detection is in many respects a cat-and-mouse game. Network attacks continue to increase in complexity as intrusion detection systems evolve to respond to them. However, countering evasion [47, 86, 88], mutation [56], and other attack transformation techniques [92] requires carefully crafted signatures that cover large classes of attacks but still make fine enough distinctions to eliminate false positive

matches. Signature languages have thus evolved from simple exploit-based signatures to more complex session [93,102,117] and vulnerability-based [17,113] signatures, with higher processing costs.

3. **Increase in network speeds.** Organizations are relying more and more on intrusion detection devices to protect their networks, even while network speeds are increasing and NIDS are being placed deeper into the network (toward the core, away from the edges). Many mid-size or larger networks are routinely run at 100 Mbps, 1 Gbps, or even 10 Gbps. To be effective, a NIDS must be able to inspect packets at line rates against the signatures in the database, regardless of the underlying network speed. Failure to do so can lead to inadvertent denial-of-service attacks if a NIDS must drop packets due to CPU resource exhaustion[1]. Raw processor speed increases are leveling out[2]; rising network speeds and traffic loads ultimately result in fewer processor cycles per packet that can be devoted to matching signatures.

4. **Presence of an active adversary.** When performance is a driving metric, as it is for a NIDS, one frequently seeks to optimize the common case. In security-sensitive settings, however, one must assume the presence of an active adversary seeking to disrupt operation and induce worst case behavior. The extent of the disruption is bounded by the difference between the average and worst cases, but in some cases it is large enough to be used as a denial of service attack or to enable evasion [29,99]. In such an adversarial environment, a NIDS must perform packet inspection under the assumption that the worst-case *is* the average case.

It is fair to say that a NIDS resides between a rock and a hard place. Attacks and exploits have been growing in frequency and sophistication for many years and continue to do so. Traffic loads increase with rising network speeds and deployment of NIDS devices in larger and larger networks. On the other hand, raw processing speeds are leveling off and signatures require more processing power to match. Evasion and denial of service are two of the ever-present consequences of failing

[1]Alternatively, for intrusion *detection*, lapses may occur in which malicious packets enter the network undetected.
[2]Multicore processors can yield throughput gains, provided sufficient parallelization can be extracted and synchronization costs are small.

to satisfy these constraints. To remain effective under such demanding circumstances, a NIDS must combine sufficiently expressive signature languages with efficient matching techniques.

1.3 Payload Inspection, the heart of a NIDS

Intrusion detection is a multi-layered process involving many tasks. A NIDS resides in-network, but it must identify exploits and intrusions targeted at end-hosts. Thus, a NIDS must emulate the network-level behavior of the hosts it is protecting so that it can detect exploits as reconstructed by the protocol stack in the host. This behavior includes reassembling streams of data from (potentially) out-of-order, fragmented packets, performing traditional header-based packet classification [46, 107, 110], and normalizing alternative traffic encodings such as in HTTP URLs [38, 90].

Despite all this work, these tasks are just precursors to the actual process of matching signatures. In the lowest layer, at the heart, there resides in any NIDS a pattern matching engine that potentially compares every byte of incoming and outgoing traffic to signatures from a database containing known exploits or misuses. This process is generally referred to as *signature matching*, *deep packet inspection*, or *payload inspection*[3]. Payload inspection directly bears the weight of the conflicting demands described in the previous section and is the most processing-intensive component of intrusion detection. Measurements on deployed systems have shown [18] that payload inspection alone accounts for up to 75% of the total processing time; our own experiments are consistent with these results. Ideally, we would like the time-complexity of DFAs and the space-complexity of NFAs.

NIDS performance is limited to the speed at which network traffic can be matched against signatures. Thus, the *language* used to express signatures and, correspondingly, the data structures and procedures used to represent and match input to strings in that language have a tremendous impact on performance. To keep up with line speeds, signatures or portions thereof must be combined and matched simultaneously in a single pass over the input. String-based signatures, initially popular, have fast multi-pattern matching procedures [1, 109] but limited expressivity. Modern signatures commonly use the full capabilities of regular expressions, which are highly expressive yet

[3]We use all three terms interchangeably in this work.

compact. From the perspective of matching procedures, regular expressions are typically implemented as either deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). Like strings, DFAs are fast–requiring only a single table lookup per input symbol–and can be readily combined into a single automaton that recognizes the union of the component languages. However, DFAs often interact poorly when combined, yielding a composite DFA that is typically much larger than the sum of the sizes of the DFAs for individual signatures and often significantly exceeds available memory. Our own experiments in Chapter 6 show that DFAs corresponding to actual NIDS signatures require several gigabytes of memory.

At the other extreme, combined NFAs have a small memory footprint, but their matching time can be large. Because they are nondeterministic, NFA implementations must use some strategy for exploring possible state sequences while searching for a match. A "depth-first search" implementation requires backtracking over many possible nondeterministic paths when searching for an accepting path. On the other hand, a "breadth-first search" avoids backtracking but must simultaneously maintain and update a distinct pointer for each possible path in each component signature. For ever-expanding signature databases used in intrusion detection, these costs are not amenable to real-time inspection.

A payload inspection engine must be both compact and fast to support NIDS requirements. But, DFAs and NFAs induce a trade-off requiring either large matching times or large memory usage, both of which are unsuitable for high-speed network environments.

1.4 Thesis and Contributions

This work concerns the development and analysis of techniques and models for performing regular expression-based payload inspection of streaming data such as network traffic. The genesis of this work stems from the observation that in many cases, the time-space tradeoffs associated with signature matching are not the result of fundamental obstacles, but rather the consequences of limitations in existing models. We posit that with the introduction of richer, more complex models for signature matching, we can construct algorithms and data structures that bridge the gap between the tradeoff points and provide acceptable memory and performance characteristics.

For example, the disparity between worst-case and average-case performance for some NIDS, while demonstrably large, can be reduced to manageable levels by using memoization in some cases to avoid unnecessary computation at the expense of only a slight increase in memory usage. As another example, through the careful inclusion and manipulation of simple auxiliary state variables, we can devise automata-based matching mechanisms that avoid the time-inefficiencies of NFAs and the memory-inefficiencies of DFAs. From the DFA perspective, a slight decrease in matching performance yields a dramatic reduction in the memory footprint. These examples illustrate that by tolerating slightly degraded behavior on one side of the tradeoff (time or space), we can obtain significantly improved behavior on the other side. Manipulation and management of these tradeoffs is a fundamental theme of this work.

In evaluating this thesis, we perform detailed analyses of existing NIDS to guide our work and give insight into the practical constraints of payload inspection. In the process, we discover new attack techniques and threats targeted directly at a NIDS itself. We formally characterize the state-space explosion that occurs when DFAs are combined, and we propose and evaluate novel models that naturally extend DFAs yet avoid the memory consumption associated with state-space explosion. Finally, we develop novel methods for performing edge compression on automata to further reduce memory usage.

Ultimately, our goal is to develop the theory, models, and algorithms for making regular expression-based multi-pattern matching of streaming data practical at high speeds. However, this work is not the last word on this subject; rather, it is hoped that these contributions will lay the groundwork and prove useful to further research in automata-based inspection.

We briefly describe our contributions in the sections below and present them in detail in succeeding chapters.

1.4.1 Threat Models Against a NIDS

The purpose of a NIDS is to detect malicious or unauthorized activity present in network traffic, but as a network-attached device it is itself a potential target. Successfully attacking a NIDS enables evasion by allowing malicious or restricted data to pass through the NIDS and enter or leave the network undetected.

We explore NIDS evasion through the use of a denial-of-service mechanism known as an algorithmic complexity attack [29]. Given an algorithm whose worst-case performance is significantly worse than its average-case performance, an algorithmic complexity attack occurs when an attacker is able to trigger worst-case or near worst-case behavior. Two traffic streams are involved in such attacks. The first is the unauthorized or malicious payload, such as an attack, that needs to traverse the NIDS without being detected. The second is aimed squarely at the NIDS and serves as a cover by slowing it down so that incoming packets (including the true attack) are able to slip through undetected. Evasion is most successful when the true attack enters the network, and neither it nor the second attack is detected by the NIDS.

We present an algorithmic complexity attack that exploits worst-case signature matching behavior in the Snort NIDS. By carefully constructing packet payloads and sending them into the network, our attack forces the signature matcher to repeatedly backtrack during inspection, yielding packet processing rates that are up to 1.5 million times slower than average. We term this type of algorithmic complexity attack a *backtracking attack*. Our experiments show that hundreds of intrusions can successfully enter the network undetected during the 5-minute course of the attack. Further, a single attack packet sent once every three seconds is enough to perpetually disable a NIDS.

We follow with a countermeasure that uses memoization to store intermediate state that must otherwise be recomputed. This defense against the backtracking attack relies on the use of better algorithms that reduce the disparity between average and worst case without changing functionality. Empirical results show that this solution confines the processing times of attack packets to within one order of magnitude of benign packets. We note that although the countermeasure does not eliminate the disparity between average and worst-case, it significantly reduces the magnitude and practical potency of the attack.

This work had immediate practical value to those using the NIDS employed in our analysis. But, the long-term value is both more subtle and more instructive. First, it shows explicitly the danger of relying on average-case performance in adversarial environments. Second, the solution we propose solves this particular problem, but it is not necessarily generalizable to other matching architectures. Nevertheless, it highlights the inadequacy of standard mechanisms for multi-pattern signature matching, which insights lead directly to the next contribution.

1.4.2 DFA State Explosion and Extended Finite Automata

As stated earlier, DFAs have a fast matching procedure and can be easily combined, but DFA combination often leads to an untenable explosion in the state-space which can easily exhaust available memory. In some cases, the combined DFA state space is exponential in the sizes of the source DFAs. We examine this phenomenon in detail and propose mechanisms for obviating it. This work is divided into three parts.

First, we present a first-principles characterization of state-space explosion. We describe, formally, why it occurs and give ideal conditions that eliminate it when satisfied. We introduce the notion of *ambiguity* for DFAs, distinct from non-determinism, that captures these conditions and show how violation of this property leads directly to state-space explosion. We then illustrate how auxiliary state variables can be used to "factor out" the components of automata that lead to ambiguity. When these components are removed, automata can be freely combined without any state explosion. Intuitively, appropriately including auxiliary variables changes the shape of automata and restructures the state space so that computation state can be maintained more efficiently than by using explicit DFA states alone.

Second, we introduce a formal model, termed *Extended Finite Automata (XFAs)*, that extends the standard DFA model with (first) a finite set of auxiliary variables and (second) explicit instructions attached to states for updating these variables. The extension from DFAs is natural: a DFA is simply an XFA with no added auxiliary variables. Variables cannot affect state transitions, but they can influence acceptance. This model provides a formal framework for associating auxiliary variables to automata. The model is fully deterministic and yields combination and matching algorithms that are straightforward extensions to those for DFAs. We present algorithms for constructing XFAs, combining XFAs, and matching XFAs to input.

Third, and finally, a primary advantage of this model is that it allows for systematic analysis and optimization. When many individual XFAs are combined, the resulting automaton accumulates all the individual variables and may replicate instructions across many states. Even when no state-explosion occurs, this can lead to large per-flow state and processing times. Taking inspiration from common principles used in optimizing compilers [74], we devise optimization techniques for systematically reducing both the number of instructions and the number of variables. These techniques include exploiting runtime information and support, coalescing independent variables, and performing code motion and instruction merging.

In summary, Extended Finite Automata provide a model for enabling transformations that remove ambiguity from DFAs. In this model, individual XFAs can be combined together, and the combined XFA can be pattern matched at (fast) speeds approaching those of DFAs while at the same time retaining (small) memory footprints similar to those of NFAs.

1.4.3 Edge Compression

XFAs provide a framework for eliminating DFA state explosion when individual automata are combined. Nevertheless, individual automata states have a large memory footprint themselves. Specifically, each state in an automaton contains a transition table that holds the next-state transition for each possible input symbol. For example, a one-byte wide input alphabet has 256 input symbols. Using 4-byte state identifiers, a single transition table requires 256 symbols $\times$ 4 bytes/symbol totaling $1,024$ bytes of memory. A 2-byte wide input symbol can double the matching rate in principle, but the transition table grows in size to $262,144$ bytes of memory per state.

Many have observed that transition tables contain redundant information that can be removed through compression [1, 30, 55, 64, 84]. When entries in a transition table contain the same next-state identifier, those entries can be replaced by a single *default* transition that is followed for each removed symbol. Alternatively, some input symbols induce the same next-state transition in the transition tables at every state. These input symbols form an equivalence class, and there may be several classes of equivalent symbols. To reduce transition table size, symbols in an equivalence class can be replaced by a single designated symbol in that class and indexed during matching using an *alphabet compression table*.

Single alphabet compression tables for DFA state compression have been used extensively in compiler-writing tools such as LEX and YACC [2, 55, 84] and have been recently explored in the signature matching context as well [13]. We refine this technique by introducing *multiple* alphabet compression tables, built from the observation that many sets of input symbols have equivalent behavior for large numbers of states, but not necessarily all states. We develop heuristics for partitioning the set of states in an automaton and creating compression tables for each subset in a way that yields further reductions in memory usage.

Using compression tables does require more processing time, since the per-byte cost now includes lookups into these tables. However, experiments show that once the overhead of the first compression table has been paid for, inclusion of additional compression tables comes at no additional runtime cost. We conduct further experiments comparing the performance and memory usage of multiple alphabet compression to default-transition compression and to the combined compression scheme, and we show how to integrate alphabet compression and default-transition compression so that both are employed simultaneously.

1.5 Dissertation Organization

This dissertation is structured as follows. In Chapter 2 we survey prior work performed and provide general background and context to properly frame our own work. We give a brief history of intrusion detection, describe threats against a NIDS and mechanisms that circumvent them, and discuss proposed languages and models for performing signature matching at wire speeds.

In Chapter 3 we present the Backtracking Algorithmic Complexity Attack, highlighted in Section 1.4.1 as our first contribution. We provide details of the Snort architecture to which it applies, we describe mechanisms for crafting the attack input, and we present the memoization-based technique for countering the attack. For both the attack and the defense mechanism, we report measurement results obtained from live experiments in a controlled setting.

In Chapters 4, 5, and 6 we present the Extended Finite Automata model for matching regular expressions, our second major contribution. Chapter 4 gives formal matching semantics for regular expressions in the signature matching environment. Chapter 5 focuses on the formal underpinnings including state space explosion and matching models, whereas Chapter 6 contains algorithms for manipulating XFAs, including construction, combination, matching, and optimization.

We present our third major contribution, multiple Alphabet Compression, in Chapter 7. Finally, Chapter 8 concludes.

Chapter 2

Background and Related Work

In this chapter, we review previous work to give context and frame our own work. We begin in Section 2.1 with a brief history of intrusion detection, and we informally characterize and contrast the many facets of general intrusion detection.

Intrusion detection systems are themselves subject to attack. In Section 2.2 we survey general threats against a NIDS and describe work performed to deflect or eliminate the threats altogether. These threats include, in part, exploiting inherent ambiguities and triggering algorithmic complexity attacks. Finally, in Section 2.3 we survey work performed toward enabling efficient signature matching in a NIDS and contrast it with our own.

2.1 Intrusion Detection

In its most general sense, intrusion detection refers to the process of detecting unauthorized or malicious activity on any computing resource. Early work on automated intrusion detection can be traced back to almost 30 years prior to the time of this writing [54, 57]. In a 1980 report, Anderson [6] proposed using programs to automatically inspect audit logs that identify and track misuses and other anomalous behavior. Beginning in 1983 and spanning several years, Denning [31] developed an intrusion detection model intended to be “independent of any particular...type of intrusion” and therefore suitable as the basis for a general-purpose Intrusion Detection Expert System, or IDES, as it was termed at the time. Other products became available, although most were directed toward finding intrusions on individual hosts. The year 1990 marked the introduction of tools explicitly targeted toward network intrusion detection performed on local area networks [49], and increased commercial and government interest in intrusion detection technology soon followed. Since then, intrusion detection has grown to the point that it is now a ubiquitous component of modern computer and network systems [73, 115].

Intrusion detection systems can be characterized along many axes. We briefly consider two: host vs. network intrusion, and anomaly vs. misuse detection. A host-based IDS [42] resides directly on an individual host and monitors system processes and memory usage for the presence of malicious behavior. Exploit-based systems identify specific, known intrusions such as buffer overflows, heap overflows, format string attacks, and so forth. Model-based systems construct models of specific process behavior (such as typical system call sequences) and look for deviations from that behavior. A network IDS, on the other hand, resides in-network and monitors traffic to and from many hosts. Network-based and host-based systems serve complementary roles: a host-based system constructs detailed knowledge about the environment and behavior of a specific computer system, but it imposes monitoring overhead and has no knowledge of activity outside the machine itself. On the other hand, a network-based system can observe the activity of many hosts on a network, but its knowledge of any specific host is limited and often incomplete.

Looking at the other axis, anomaly detection relies on statistical techniques for performing intrusion detection. These systems construct a baseline model of normal behavior against which current behavior is compared. Significant deviations from the model are flagged as intrusions. Several techniques have been proposed for finding deviations, using methods such as maximum entropy estimation [45], dimensionality reduction through principle components analysis [66], wavelet-based signal analysis [11], and sketch-based change detection [61], to name a few. Misuse detection, on the other hand, compares traffic to instances of specific misuses. Misuses are characterized by specific signatures, expressed as strings or regular expressions, that are all compared to traffic as it passes through the detector. Traffic that does not match any misuse signatures is benign by definition. There are many commercial NIDS offerings along with popular open-source systems including Bro [85] and Snort [91].

As with host- and network-intrusion detection, anomaly and misuse detection are likewise complementary. Anomaly detectors can find new attacks and classes of intrusions, but the false positive rate is high. Significant changes in the nature of the traffic may need to be observed before a deviation is identified. Misuse detectors perform very detailed analysis of traffic, but detection is limited to the known signatures in their database; they cannot detect novel attacks without updated signatures. Recent work has investigated techniques for automating the signature construction process. Proposals such as EarlyBird [98], Autograph [58], and Polygraph [79] can be interpreted as techniques that in part bridge anomaly detection and misuse detection by first identifying anomalies and then automatically constructing misuse signatures from them.

The focus of this dissertation is misuse network intrusion detection. In particular, our work focuses specifically on the mechanisms for matching signatures to traffic at line rates. Nevertheless, our techniques may have value in other domains involving high-speed matching of streaming data [76].

2.2 Threats Against a NIDS

2.2.1 Adversarial Environments and Ambiguity

A NIDS operates in an adversarial environment; to be effective, it must be robust to adversarial activity designed to foil or bypass detection. However, achieving such resilience continues to be an elusive goal. In early foundational work, Ptacek and Newsham [88] outline inherent difficulties with network intrusion detection and describe three general categories of attacks: insertion, evasion, and denial of service. An *insertion* attack inserts data into the stream that the NIDS processes but an end host discards. Conversely, an *evasion* attack carries data that bypasses a NIDS but reaches the intended target. Both of these attacks rely on ambiguities or subtleties in network protocol specifications and subsequent differences in implementation to succeed. For instance, an attacker can send a packet whose Time-To-Live (TTL) field is set so that the packet reaches the NIDS but expires and is dropped before arriving at the destination. Failing to recognize the short TTL at the NIDS leads directly to an insertion attack opportunity. Finally, *denial of service* attacks are resource consumption attacks that exhaust memory, processing cycles, or network bandwidth to defeat the NIDS. Many techniques have been proposed to address these problems; each of them has limitations.

One approach to removing ambiguities is to introduce a *normalizer* [47, 71] that intercepts traffic and transforms it to a canonical form before undergoing NIDS analysis and entry into the network. Handley *et al.* [47] introduce a normalizer that identifies incorrect or ambiguous elements in IP, TCP, and UDP protocol headers and either corrects the error or drops the packet. Normalization may lead to a change in end-to-end semantics; Handley *et al.* consider this issue and describe other trade-offs involved in normalization. Rubin *et al.* [94] explore normalization of application layer data in packet payloads and propose a combined normalization and matching mechanism for HTTP traffic.

Ideally, a NIDS should faithfully emulate the relevant behavior of all hosts on the network so that its interpretation of traffic matches the hosts' interpretation. Unfortunately, some ambiguities cannot be resolved by a normalizer and stem from implementation differences in the hosts themselves both at the protocol level and the application level [90, 96]. In this case, adopting a specific normalization policy provides protection to only some of the hosts on the network. To address this, Shankar and Paxson [96] propose using a database of host profiles by which a NIDS can query the key traits of a host to resolve ambiguities and properly emulate the relevant host during matching. Active Mapping provides a sort of context-sensitivity to a NIDS; the authors argue that some form of context-sensitivity is required to resolve per-host ambiguities.

Finally, to examine protocols such as HTTP, a NIDS must reassemble application-level streams from distinct, out-of-order, possibly duplicated packets. *Stream Reassembly*, as the process is called, is also a source of ambiguity and can lead to evasion and denial of service. For instance, packets with identical sequence numbers and other stream information can contain distinct payloads. Only the destination end-host knows which fragment will be accepted; evasion is possible when the NIDS incorrectly guesses which fragment the end host will accept. Dharmapurikar and Paxson [33] examine the issues with stream reassembly in detail, show how adversaries can disrupt operation, and characterize the damage they can cause. They develop hardware-based mechanisms that are resilient to adversaries, and use techniques such as graceful degradation when under attack. Even so, they observe that the effects of an adversary cannot always be removed, and in some cases must be tolerated.

2.2.2 Algorithmic Complexity Attacks

Denial of Service attacks targeting a NIDS can also disrupt the detection process. One way to achieve denial of service is to exhaust resources such as available memory and processing cycles on the NIDS. For instance, to perform stream reassembly a NIDS must maintain connection state for each individual data stream. Dreger *et al.* [35] report experimental results showing that up to 4,000 new connections are created per second on a moderately large network of (presumably) benign traffic, and that the number of connections tends to grow over time and can quickly exhaust memory without some form of management. An attacker who establishes (and never closes) several streams can induce memory exhaustion by forcing the creation of excessive connection state [33, 88].

Denial of service can also be induced by triggering *algorithmic complexity attacks*, which occur when an attacker is able to induce worst-case behavior in an algorithm whose worst-case is significantly beyond its average case. These attacks exploit poorly designed and implemented algorithms rather than program correctness flaws or ambiguity. Typically, an attacker induces these attacks by crafting input to invoke the worst case, possibly over several iterations of the algorithm. Crosby and Wallach [29] demonstrate the efficacy of algorithmic complexity attacks by exploiting weaknesses in hash function implementations to effectively turn randomized $O(1)$ lookups into $O(n)$ linear scans. Using this technique, they reduce the packet processing rate of the Bro NIDS [85] from 1200 packets per second to only 24 packets per second. In preliminary follow-on work they illustrate how to achieve slowdown attacks against regular expression matching engines [28].

In Chapter 3 we describe a variant termed the backtracking algorithmic complexity attack. Through careful construction of attack payloads, our attack forces the signature matching component of the popular Snort NIDS [91] to repeatedly backtrack, yielding processing rates that are up to 1.5 million times slower than average. In live, controlled experiments, we use this attack as a cover to successfully and perpetually evade detection of other attacks.

The strength of these attacks draws on the difference between average- and worst- case behavior of algorithms. In principle, this class of attacks can be eliminated by avoiding algorithms with large performance gaps, or by designing explicitly for the worst-case. But in practice, worst-case performance of existing algorithms is often unacceptable, and one common approach is to employ architectures with fast, approximate techniques backed by slower, more involved algorithms [34, 63, 94]. For example, Dreger *et al.* [35] proposes an adaptive filtering technique to dynamically adjust for load changes. Snort [91], on the other hand, uses sequences of successively more refined matching predicates to reduce average-case execution time. It is precisely this architecture that opens the door to algorithmic complexity attacks.

Other techniques have been proposed to address the performance requirements of intrusion detection. Lee *et al.* [67] dynamically divide the workload among multiple modules, making adjustments as necessary to maintain performance. Load shedding is performed as necessary to distribute the load to different modules, or to lower its priority. Alternatively, Kruegel *et al.* [62] have proposed achieving high speed intrusion detection by distributing the load across several sensors, using a scatterer to distribute the load and slicers and reassemblers to provide stateful detection. Still other approaches seek to provide better performance by splitting up (and possibly replicating) a sensor onto multiple cores or processors [26, 111]. These approaches show that allocating more hardware can better protect large networks with large amounts of traffic, but they do not directly address worst-case complexity attacks.

Both [67] and [85] propose the use of monitors to track the resource usage and performance history of a NIDS. In [67], if a monitor discovers abnormally long processing times, the current operations are aborted and optionally transferred to a lower priority process. For [85], on the other hand, the monitor simply triggers a restart of the NIDS. In the general case, such techniques may provide a useful mechanism for ensuring guaranteed minimum performance rates at the cost of decreased detection accuracy. However, such mechanisms result in periodic lapses in detection capability.

The backtracking algorithmic complexity attack arises from the inadequacy of standard NFA-based regular-expression matching techniques for intrusion detection. DFA-based approaches have constant $O(1)$ complexity per byte and are thus immune to complexity attacks, but they have untenable memory requirements. In Chapters 5 and 6, we propose Extended Finite Automata as a model for addressing these weaknesses.

2.3 Signature Matching

Signature matching is at the heart of intrusion detection and is the focus of this work. At a high level, signature matching is simply a language recognition process in which input strings (the payload) are tested for membership in the languages defined by the set of signatures. Two complementary facets of languages shape the behavior and characteristics of signature matching mechanisms: the language used for writing signatures, and the matching model used for identifying and accepting elements of the language. Unfortunately, these facets also introduce conflicting tradeoffs; *e.g.*, more expressive language models typically require more complex and time-consuming matching procedures. We briefly describe related work on both facets below and highlight some of the tradeoffs involved.

We note that a related but distinct problem is the automaton intersection problem: given a set of DFAs $\mathcal{D}$ with common alphabet Σ, does there exist a string $x \in \Sigma$ such that for each $D \in \mathcal{D}$, $x \in L(D)$? In other words, the problem is to discover whether the intersection of the automata in $\mathcal{D}$ recognizes more than the empty language. This problem is pspace-complete [41]. In contrast, for signature matching, the string x is supplied as input and the problem is to efficiently find which automata in $\mathcal{D}$ accept the prefixes of x.

2.3.1 Signature Languages

Languages for expressing signatures have grown in complexity as attacks and evasion attempts have become more sophisticated. Simpler languages are more efficient to match, but have limited expressive power. Complex languages are more expressive but have a longer matching procedure.

2.3.1.1 Exploits and String-based Signatures

String-based signatures were initially popular for intrusion detection and still find some use today, most notably as fast pre-filters that guard more complex matching mechanisms [82]. Such signatures have two advantageous properties. First, they are very efficient. Common string matching algorithms [14, 59] use skiplists, suffix automata [78], and other heuristics [72] to complete in linear time or less. For these algorithms, each byte of the payload is examined at most once, and the per-byte cost is bounded and small.

Second, strings can be combined and matched simultaneously over a single pass of the input payload. In particular, the Aho-Corasick algorithm [1] produces a concise automaton-like structure linear in the total size of the strings with a fixed worst-case performance bound that matches all strings in a single pass. Other proposed multi-string matching techniques [25, 116] provide different tradeoffs [114] between matching speed and memory usage. Simultaneous multi-pattern matching is crucial to NIDS performance, allowing the number of signatures to grow over time without affecting matching performance.

A number of improvements have been made to the basic string matching paradigm, including parametrized matching [7, 8], approximate matching [75], matching with wildcards [3], and other methods [24, 39]. These techniques seek to retain the basic efficiency of string matching while increasing the expressivity of the underlying signatures. Still other work has focused on improving matching performance through the use of hardware [69, 104, 106] or through other algorithmic means. For example, Tuck *et al.* [109] apply path compression and bitmaps to eliminate some of the space costs inherent to the Aho-Corasick algorithm. They report memory requirements that are reduced by a factor of 50 without any decrease in performance.

Unfortunately, strings are fundamentally limited by their lack of expressivity. In adversarial settings, many transformation techniques [32, 56, 90, 93] are commonly employed to produce distinct variations in attack signatures; a string-based detector would need a distinct string for each of these variations, which can easily number into the thousands. As a simple example, a case insensitive version of “root” can be easily captured with the regular expression ([Rr][Oo][Oo][Tt]) but requires 16 distinct strings using a string-based approach. Thus, requirements for signature languages have evolved from simple exploit-based paradigms to those that can succinctly characterize classes of exploits, among other capabilities.

2.3.1.2 Vulnerabilities and Regular Expressions

String-based signatures are effective for performing pattern matching in search of specific exploits, but they are not sufficient for expressing more general patterns or vulnerabilities. Due to these limits, modern systems [20, 85, 91] have migrated towards richer models that provide greater flexibility than strings. Currently, regular expressions [53] have become the *de facto* standard for expressing signatures. Regular expressions are strictly more expressive than strings, but like strings they can be succinctly expressed and combined and matched simultaneously using automata. Indeed, Sommer and Paxson [102] argue that the increased expressivity of regular expressions combined with their efficient matching procedures (linear in input size) yields fundamental improvements in signature matching capabilities.

One advantage of regular expressions is their ability to model general vulnerabilities, such as a buffer overflow, as opposed to specific exploits that target the vulnerability. Brumley *et al.* [17] propose techniques for automatically constructing vulnerability-based signatures built from regular expressions. The principal advantages of this approach are the elimination of human error in signature construction and resilience to polymorphic attacks. Given a vulnerable application, input that produces an exploit, and a runtime trace leading up to the exploit, the approach constructs a regular expression signature that in principle captures all exploit variations. Our work is complementary to theirs. We focus on signature and regular expression matching, whereas they study signature creation.

In the Shield approach [113], “vulnerability-specific, exploit-generic” filters are inserted into a host’s network stack and executed on each packet that reaches the scripts. These filters determine whether a packet’s contents satisfy a vulnerability condition and blocks the packet or alerts the user if an exploit is detected. For text-based protocols, Shield uses regular expressions to express vulnerabilities, although the full capabilities of the filters are more general.

Regular expression signatures are also used to provide additional matching context. Sommer and Paxson [102] propose *contextual signatures*, which incorporate protocol event sequencing (*e.g.*, ordering of commands and responses) into string matching. This allows standard matching algorithms to be performed in the context of the protocol-level events that have been received. Rubin *et al.* [93] extend this notion with *session signatures*, which model entire attack sequences from connection initialization by the attacker to attack acknowledgment by the victim (to the attacker). Yegneswaran *et al.* [117] introduce *semantics-aware signatures* that incorporate session-level and application-level protocol semantics into signatures. Each of these techniques aims to reduce false matches by providing a rich context in which to perform signature matching.

All of the work discussed above in this subsection employs regular expressions either as a target for signature creation, or as the language used for filtering. In contrast, our work examined the models used for *matching* regular expressions to input, regardless of their construction or use.

2.3.1.3 Other Languages

With regard to expressive power, regular expressions are superior to strings. Thus, it is natural to ask whether even more expressive mechanisms such as context-free or context-sensitive languages would provide even further benefit. For these classes of formal languages the answer is in the affirmative, since they are both super-sets of regular languages [53]. However, to the best of our knowledge the practical consequences of employing richer language models is not fully understood with regard to the wire-speed performance requirements of matching payloads.

As a case in point, automata corresponding to regular expression signatures can be combined and matched simultaneously with little (if any) increase in runtime cost, but this may not be practically feasible or even possible for matching mechanisms of more expressive languages. For example, both Bro [85] and Shield [113] define custom scripting languages for expressing signatures, but it is not clear whether scripts in these or other languages could be fully combined and executed simultaneously as is done with finite automata. Further, matching procedures for more expressive languages may impose untenable runtime costs. For example, in a host intrusion detection context (where runtime performance is important but not as stringent), both Wagner and Dean [112] and Giffin *et al.* [43] report that context-sensitive models were prohibitively expensive and unsuitable for practical use.

It would be interesting to further explore the suitability of richer language models for network-based intrusion detection, but such exploration is beyond the scope of this work.

2.3.2 Matching Models for Regular Expressions

Whereas the choice of signature language determines the expressive power of a signature scheme, the matching model defines the runtime characteristics – the memory usage and performance – associated with determining whether input strings belong in the language. In this dissertation, our focus is on matching models for regular expressions.

Matching models for regular expressions are based on finite automata. It is well-known that regular expressions and finite automata are inherently connected [53]. Specifically, a language is regular if and only if it is accepted by a finite automaton ([68], Theorem 2.5.1). Thus, for a regular expression R, we can determine whether any input sequence is a member of the language $L(R)$ defined by R by feeding it to a finite automaton $F(R)$ corresponding to R. If $F(R)$ accepts the input, then the input is a member of the language $L(R)$. Signature matching is fundamentally a language recognition problem, with the caveat that membership in multiple languages must be evaluated simultaneously.

Regular expressions are typically represented as either Nondeterministic Finite Automata (NFAs) or Deterministic Finite Automata (DFAs). Individual automata can be readily composed to form a single composite automata (we give an algorithm in Section 4.3), but they often do not scale when combined. As indicated in Chapter 1, NFAs can compactly represent multiple signatures but may require large amounts of matching time, since the matching operation needs to explore multiple fruitless paths in the automaton. DFAs are fast, requiring only a single table lookup per input symbol, but they exhibit polynomial or exponential growth in the state space when combined. Thus, in their basic form, NFAs and DFAs as used in intrusion detection lie at opposite extremes from a memory-performance perspective and are not suitable for high speed signature matching.

Using DFAs (for their speed) as a starting point, many techniques have been proposed to reduce the memory footprint of combined DFAs. Memory reduction is achieved in two ways: by reducing the total number of automaton states, and by reducing the footprint of individual states.

Since the total memory footprint is a function of the number of states, reducing the number of states yields the most significant reduction, and reducing the footprint of individual states provides second-order results. Nevertheless, both mechanisms do induce some increase in the matching time since additional computation is performed per byte. We examine these costs in detail later in this dissertation.

2.3.2.1 Reducing the Number of States

In early work, Yu *et al.* proposed *DFA Set-Splitting* [118], a state reduction technique that combines signatures into multiple DFAs (instead of a single DFA) such that total memory usage is below a supplied threshold. Regular expressions are heuristically selected for combination until the resulting automaton exceeds its "fair share" of memory, at which time it becomes immutable and a new combined automaton is begun. The process repeats until all expressions have been included. Set-splitting controls state-space explosion by capping the amount of memory a group of DFAs can consume when combined, at the cost of introducing additional DFAs to be matched. As a result, set-splitting traces a curve between NFAs and DFAs in a space-time plot: as the memory threshold is increased, the number of DFAs that must be simultaneously matched shrinks, decreasing the inspection time. The primary advantage of this technique is its simplicity, although large signature sets may require many DFAs to be matched in succession.

Becchi and Cadambi [12] propose State Merging, in which information about states themselves is moved into edge labels in a way that allows states to be combined. The algorithm employs a heuristic-driven iterative approach for selecting candidate states to be merged. The authors report total memory savings of up to an order of magnitude, but performance results are not given. We also reduce the number of DFA states, but our work seeks to eliminate the cause of state-space explosion in combined DFAs.

Lazy approaches to DFA evaluation [44] can reduce the effective number of DFA states, although the overall total number remains the same. Beginning only with an NFA (compact but slow) and a start state, lazy approaches dynamically build the DFA at runtime by constructing transitions and states only as needed, as determined by the input. Lazy construction draws on the fact that although the state space is large, the working set of visited or *hot* states is typically much smaller. Although in its current form this technique was first proposed for XML Stream Processing, recent work [70] has confirmed that working set sizes are relatively small for intrusion detection signatures as well. The cost, of course, is increased execution time when new transitions and states are constructed and exposure to algorithmic complexity attacks. This approach is orthogonal to our own and can be applied to any DFA-based matching scheme.

Auxiliary variables can be used to reduce the memory requirements of an automaton. This approach, common to software verification [9,52] and model checking [23], associates one or more variables with an automaton and uses them to track matching state more compactly than explicit states alone can do. But, to the best of our knowledge, prior techniques for including these variables are ad-hoc and not designed for high speed payload inspection. Developed concurrently with our own work, Kumar *et al.* [63] present heuristics that use flags and counters for remembering whether portions of signatures have been seen. Like us, they use auxiliary variables for reducing the state space, although there are some fundamental differences. First, their technique is heuristic and seeks only to reduce the number of states, whereas we begin with a formal characterization of state space explosion and then show how auxiliary variables can eliminate it. Second, the interaction between states, variables, and transitions is not formalized, and it is not clear how individual automata can be combined and manipulated. We provide an extensible formal model explicitly designed for this.

2.3.2.2 Reducing the Size of a State

The second way to reduce the overall memory size is to reduce the memory footprint of individual states, a process referred to as *edge compression*. Since pointers for the transitions leading out of a state constitute the bulk of a state's footprint, reducing the number of outbound pointers can reduce the size of a state significantly. Most edge compression techniques are variations of two themes: default transitions, and alphabet compression. These techniques trace their roots back to automata compression techniques developed for compiler-writing tools such as LEX and YACC [30,55,84]. We briefly describe some of them here.

A default transition replaces entries for multiple symbols in a transition table with a single entry that is followed whenever the removed symbols are reserved at the state. During matching, the default transition is followed to its destination state if in the current state there is no transition table entry for the current input symbol. Depending on the construction, it is possible that a single input symbol may require traversing chains consisting of several default transitions before the proper destination state is reached.

Johnson [55] introduced the use of default transitions for DFA compression during development of the YACC parser generator. He used auxiliary arrays and performed careful placement of state identifiers to achieve compression. The popular Aho-Corasick multi-pattern string matcher [1] also employed default transitions extensively, so that only transitions that made forward progress toward matching some pattern were distinctly represented.

In an intrusion detection environment, one of the central challenges with default transition schemes is quickly determining whether the current input symbol is contained in the state, or whether the default transition must be followed. Recently, Kumar *et al.* [64] proposed a hardware-based technique, D2FAs, with heuristics for construction that limit the maximum length of default transition chains that must be followed. During matching, they use hardware-supported hashing to quickly identify whether or not a default transition should be followed at a given state. In further work, the hardware dependency has been removed by assuming the availability of wide pointers [65] into which transition information is encoded, and the construction heuristics have been reformulated to reduce default transition chain lengths further [13]. Finally, Ficara *et al.* [37] propose an enhancement that follows only one default transition at the cost of continually copying transition table entries on each input symbol.

Alphabet compression is complementary to default transitions and draws on the fact that some input symbols induce the same behavior at all states in a DFA. Such input symbols form an equivalence class, and all but one symbol from each equivalence class can be removed from the transition tables of all states. During matching, an alphabet compression table maps symbols to their respective equivalence classes. Our experiments with this technique using the Flex scanner have yielded memory reductions of up to 50$\times$, although the reduction comes at a high execution time cost involving multiple memory accesses that are not suitable for high-speed matching. Further, tools like Flex perform matching using repeated invocations of the DFA and assume different semantics. Recently, Becchi and Crowley have examined single alphabet compression tables in the context of intrusion detection [13]. In our work on edge compression, in contrast, we explore the use of multiple alphabet compression tables that yield even further memory reductions without additional runtime cost.

2.3.2.3 Hardware and Other Approaches

Many hardware approaches have been proposed to circumvent the time-space tradeoff inherent to DFA and NFA matching. Hardware-based solutions can parallelize the processing required to achieve high performance by processing many signatures in parallel rather than explicitly combining them. Sidhu and Prasanna [97] provide a hardware-based NFA architecture that updates the set of states in parallel during matching. Sourdis and Pnevmatikatos [105] employ content-addressable memories (CAMs) to increase the performance further, and Clark and Schimmel [21] present optimization techniques (such as examining multiple bytes per clock cycle) and achieve regular expression matching at rates of up to 25 Gbps. Brodie *et al.* [15] also employ multi-byte transitions and apply compression techniques to reduce the memory requirements. These techniques show promise for high performance matching. However, replication of NFAs introduces scalability issues as resource limits are reached (Clark and Schimmel are able to fit only 1500 signatures on their prototype). In addition, hardware techniques in general lack the flexibility for evolving signature sets that is implicit to intrusion detection, and they restrict applicability to those instances where the hardware cost can be justified and custom hardware support is available. In general, our focus on mechanisms for signature matching neither requires nor precludes the use of custom hardware. Our work focuses on the inherent space complexity of combining automata and mechanisms for avoiding the space costs, independent of the target architecture. In our experiments, we assume only a generic computing platform consisting of a modern general purpose processor with a standard amount of memory (*e.g.*, a 3 GHz process with 4 GB of memory).

Moving beyond signature matching, other extensions to automata have been proposed in the context of information security. *Extended Finite State Automata (EFSA)* extend traditional automata to assign and examine values of a finite set of variables. Sekar and Uppuluri [95] use EFSAs to monitor a sequence of system calls. Extensions, such as EFSA, fundamentally broaden the language recognized by the finite-state automata, *e.g.*, EFSAs correspond to regular expression for events (REEs). On the other hand, XFAs can be viewed as an optimization of a regular DFA, but XFAs do not enhance the class of languages that can be recognized. It will be interesting to consider XFA-type optimizations to EFSAs.

Eckmann *et al.* [36] describe a language STATL, which can be thought of as finite-state automata with transitions annotated with actions that an attacker can take. The motivation for STATL was to describe attack scenarios rather than improve the efficiency of signature matching. Automata augmented with various objects, such as timed automata [4] and hybrid automata [51], have also been investigated in the verification community. For example, hybrid automata, which combine discrete transition graphs with continuous dynamical systems, are mathematical models for digital systems that interact with analog environments. As with EFSAs, these automata (which are usually infinite-state) fundamentally enhance the languages they recognize.

Time-space tradeoffs like that induced by DFA and NFA matching are pervasive to Computer Science. As illustrated with DFA Set Splitting [118], in such tradeoffs memory use can be reduced at the cost of slower program execution, or alternatively, the computation time can be reduced at the cost of increased memory use. In complexity theory, researchers investigate whether addition of a restriction on the space inhibits one from solving problems in certain complexity class within specific time bounds. For example, time-space tradeoff lower bounds for SAT were investigated by Fortnow [40]. Time-space tradeoffs have also been explored in the context of attacks [77, 83]. We are not aware of existing work on time-space tradeoffs in the context of signature matching for NIDS.

Chapter 3

NIDS Evasion via Backtracking Algorithmic Complexity Attacks

In this chapter, we examine algorithmic mechanisms for inducing denial of service in a NIDS. Many common algorithms have performance whose worst-case behavior is distinct from its average-case. For example, hash lookup has $O(1)$ (constant) complexity on average, but in the worst-case degenerates to an $O(n)$ linear traversal. Similarly, Quicksort operates in $O(n\ log\ n)$ time on average, but in its worst-case is $O(n^2)$ [27]. For both examples, the worst-case behavior is triggered by the runtime input to the algorithm. By design, typical inputs will exhibit average-case complexity. However, inputs can be supplied that exploit algorithm and/or implementation knowledge to trigger the worst-case. In hostile environments where the input is controlled at least partially by an adversary, this presents an opportunity for an attacker to induce degraded performance.

A NIDS has similar behavioral properties at both small and large scales. At the small scale, processing components such as classification, normalization, and stream reassembly require the use of algorithms with differentiated average- and worst-case. At the large scale, it is common for NIDS designers to employ fast but approximate matching techniques backed by slower, more detailed matching in order to meet performance requirements. Thus, the system itself may be architected such that average-case and worst-case are distinguished. Finally, engineers, software designers, and network operators can collectively control all inputs and parameters that affect NIDS performance except for one, the runtime traffic input.

An algorithmic complexity attack [29] is an attack in which carefully crafted inputs produce worst-case behavior that exhausts processing resources and degrades performance, leading eventually to denial of service. To use firewall terminology [19], under such conditions a “fail-open”

NIDS allows packets to pass unexamined, whereas a “fail-closed” NIDS drops packets. For example, as discussed in Chapter 2, Crosby and Wallach [29] showed how to induce worst-case hash function behavior to reduce performance in the Bro NIDS [85] to 24 packets per second (this flaw has since been removed in Bro).

In this chapter we introduce the Backtracking Algorithmic Complexity Attack. Through carefully crafted input patterns, our attack forces the large-scale signature matching component of the Snort NIDS [91] to repeatedly backtrack during inspection, yielding packet processing rates that are up to 1.5 million times slower than average.

Our countermeasure to the backtracking attack is an algorithmic, semantics-preserving enhancement to signature matching based on the concept of memoization. The core idea is straightforward: whereas the backtracking attack exploits the need of a signature matcher to evaluate signatures at all successful partial match offsets, a memoization table can be used to store intermediate state that must otherwise be recomputed. Our defense against the backtracking attack relies on the use of better algorithms that reduce the disparity between worst and average case without changing functionality. Even so, it does not eliminate the performance gap entirely, as we will show.

Our result applies directly to Snort [91], a popular open-source NIDS that provides both NIDS and IPS functionality and claims more than 150,000 active users. Snort uses a signature-based architecture in which each signature is composed of a sequence of operations, such as string or regular expression matching, that together identify a distinct misuse. In our experiments, we use Snort over both traces and live traffic. In addition, we provide a practical implementation of the defense by extending Snort’s signature matching functionality directly.

In summary, our contributions in this chapter are two-fold. First, we discuss NIDS evasion through algorithmic complexity attacks. We present a highly effective real attack, the backtracking attack, that yields slowdowns of up to six orders of magnitude and is feasible against the (estimated) tens of thousands of networks monitored by Snort. Second, we present an algorithmic defense, based on the principle of memoization, that confines the slowdown to less than one order

Predicate	Description	Type
`content`:$< str >$	Searches for occurrence of $< str >$ in payload	multiple-match
`pcre`:$/regex/$	Matches regular expression $/regex/$ against payload	multiple-match
`byte_test`	Performs bitwise or logical tests on specified payload bytes	single-match
`byte_jump`	Jumps to an offset specified by given payload bytes	single-match

Table 3.1: Subset of Snort predicates used for packet inspection. Multiple-match predicates may need to be applied to a packet several times.

of magnitude in general and to less than a factor of two in most cases. We provide a practical implementation of this solution and show its efficacy in a live setup.

We organize this chapter as follows. In Section 3.1 we describe Snort's rule-matching architecture. Sections 3.2 and 3.3 present the backtracking attack and the countermeasure, respectively. Section 3.4 details our experimental results, and Section 3.5 considers other types of complexity attacks. Section 3.6 concludes.

3.1 Rule Matching in Snort

Our work is performed in the context of the Snort NIDS. Snort employs a signature-based approach to intrusion detection, defining distinct signatures, or rules, for each misuse to be searched for. Each signature is in turn composed of a sequence of *predicates* that describe the operations that the signature must perform. Section 3.1.1 gives an overview of the language used to specify these rules. Section 3.1.2 describes the algorithm used to match rules against packets.

3.1.1 Expressing Rules in Snort

Snort's rules are composed of a header and a body. The header specifies the ports and IP addresses to which the rule should apply and is used during the classification stage. The body has a sequence of predicates that express conditions that need to succeed for the rule to match. A rule matches a packet only if all predicates evaluated in sequence succeed. Of the predicates that are

```
alert tcp $EXT_NET any -> $HOME_NET 99
 (msg:"AudioPlayer jukebox exploit";
  content:"fmt=";                 //P1
  pcre:"/^(mp3|ogg)/",relative;  //P2
  content:"player=";              //P3
  pcre:"/.exe|.com/",relative;   //P4
  content:"overflow",relative;   //P5
  sid:5678)
```

Figure 3.1: Rule with simplified Snort syntax describing a fictional vulnerability.

part of Snort's rule language, we focus on those used to analyze the packet payloads. Table 3.1 summarizes the relevant rules.

Figure 3.1 depicts a signature using a simplified version of Snort's rule language. The header of the rule instructs Snort to match this signature against all TCP traffic from external sources to servers in the home network running on port 99. The body of the rule contains three `content` predicates, two `pcre` [87] predicates, and two terms, `msg` and `sid`, used for notification and book-keeping. The rule matches packets that contain the string `fmt=` followed immediately by `mp3` or `ogg`, and also contain the string `player=`, followed by `.exe` or `.com`, followed by `overflow`.

Predicates have one important side effect: during rule matching a predicate records the position in the payload at which it succeeded. Further, when a predicate contains a `relative` modifier, that predicate inspects the packet beginning at the position at which the previous predicate succeeded, rather than the start of the payload. For example, if predicate P3 in Figure 3.1 finds the string `player=` at offset i in the payload, the subsequent `pcre` predicate (P4) succeeds only if it matches the packet payload after position i.

3.1.2 Matching signatures

When matching a rule against a packet, Snort evaluates the predicates in the order they are presented in the rule, and concludes that the packet does not match the rule when it reaches a predicate that fails. To ensure correctness, Snort potentially needs to consider all payload offsets at which `content` or `pcre` predicates can succeed. We term these *multiple-match* predicates.

```
Payload  fmt=aac player=play 000 fmt=mp3 rate=14kbps player=cmd.exe?overflow
Offset   01234567890123456789012345678901234567890123456789012345678901234567
                   1         2         3         4         5         6

                                                                     (P5,59,67)
                                                         (P4,51,59)  (P4,51,59)
                                             (P3,31,51)  (P3,31,51)  (P3,31,51)
           (P2, 4, f)            (P2,28,31)  (P2,28,31)  (P2,28,31)  (P2,28,31)
(P1, 0, 4) (P1, 0, 4) (P1, 0,28) (P1, 0,28)  (P1, 0,28)  (P1, 0,28)  (P1, 0,28)
```

Figure 3.2: Packet payload matching the rule in Figure 3.1 and corresponding stack trace after each call to `getNextMatch` on line 3 of Algorithm 3.1.

In contrast, predicates `byte_test` and `byte_jump` are *single-match*, meaning that any distinct predicate invocation evaluates the payload once.

In the presence of a multiple-match predicate `P`, Snort must also retry all subsequent predicates that either directly or indirectly depend on the match position of `P`. For example, consider matching the rule in Figure 3.1 against the payload in Figure 3.2. The caret (ˆ) in `P2` indicates that `P2` must find a match in the payload immediately after the previous predicate's match position. If Snort considers only `P1`'s first match at offset 4, then `P2` will fail since `P2` is looking for `mp3` or `ogg` but finds `aac` instead. However, if Snort also considers `P1`'s second match at offset 28, `P2` will succeed and further predicates from the rule will be evaluated. Snort explores possible matches by backtracking until either it finds a set of matches for all predicates or it determines that such a set does not exist.

Algorithm 3.1 presents a simplified version of the algorithm used by Snort to match rules against packets.[1] All predicates support three operations. When a predicate is evaluated, the algorithm calls `getNewInstance` to do the required initializations. The previous match's offset is passed to this function. The `getNextMatch` function checks whether the predicate can be satisfied, and it sets the offset of the match returned by calls to the `getMatchOffset` predicate. Further invocations of `getNextMatch` return true as long as more matches are found. For each of these matches, all subsequent predicates are re-evaluated, because their outcome can depend on the

[1]The Snort implementation uses tail calls and loops to link predicate functions together and to perform the functionality described in Algorithm 3.1. The algorithm presented here describes the behavior that is distributed throughout these functions.

```
MatchRule($Preds$):
1 $Stack \leftarrow (Preds[0]$.getNewInstance$(0))$;
2 while $Stack.size > 0$ do
3     if $Stack.top$.getNextMatch() then
4         if $Stack.size == Preds.size$ then return $True$;
5         $ofst \leftarrow Stack.top$.getMatchOffset();
6         Push($Stack, Preds[Stack.size]$.getNewInstance($ofst$));
7     else Pop($Stack$);
8 return $False$;
```

Algorithm 3.1: Rule matching in Snort. The algorithm returns $True$ only if all predicates succeed.

offset of the match. The rule matching stops when the last predicate succeeds, or when all possible matches of the predicates have been explored. Figure 3.2 shows the stack at each stage of the algorithm. Each stack record contains three elements: the predicate identifier, the offset passed to `getNewInstance` at record creation, and the offset of the match found by `getNextMatch` (`f` if no match is found). In this example, the algorithm concludes that the rule matches.

3.2 NIDS Evasion via Backtracking

The use of backtracking to cover all possible string or regular expression matches exposes a matching algorithm to severe denial of service attacks. By carefully crafting packets sent to a host on a network that the NIDS is monitoring, an attacker can trigger worst-case backtracking behavior that forces a NIDS to spend seconds trying to match the targeted rule against the packet before eventually concluding that the packet does not match. For the rule in Figure 3.1, `P2` will be evaluated for every occurrence of the string `fmt=` in the packet payload. Furthermore, whenever this string is followed by `mp3`, `P2` will succeed and the matcher will evaluate `P3`, and if `P3` succeeds it will evaluate `P4`. If `fmt=mp3` appears n_1 times, `P3` is evaluated n_1 times. If there are n_2 occurrences of `player=`, `P4` will be evaluated n_2 times for each evaluation of `P3`, which gives us a total of $n_1 \cdot n_2$ evaluations for `P4`. Similarly, if these occurrences are followed by n_3 repetitions of `.exe` or `.com`,

```
Payload  fmt=mp3fmt=mp3fmt=mp3player=player=player=.exe.exe.exe
Offset   0123456789012345678901234567890123456789012345678901234
                   1         2         3         4         5
```

Figure 3.3: A packet payload that causes rule matching to backtrack excessively.

`P5` is evaluated $n_1 \cdot n_2 \cdot n_3$ times. Figure 3.3 shows a packet that has $n_1 = n_2 = n_3 = 3$ repetitions. Figure 3.4 shows the evaluation tree representing the predicates evaluated by the algorithm as it explores all possible matches when matching Figure 3.1 against the payloads in Figure 3.2 and in Figure 3.3. Our experiments show that with packets constructed in this manner, it is possible to force the algorithm to evaluate some predicates hundreds of millions of times while matching a single rule against a single packet.

The amount of processing a backtracking attack can cause depends strongly on the rule. Let n be the size of a packet in bytes. If the rule has k unconstrained multiple-match predicates that perform $O(n)$ work in the worst case, an attacker can force a rule-matching algorithm to perform $O(n^k)$ work. Thus the following three factors determine the power of a backtracking attack against a rule.

1. *The number of backtracking-causing multiple-match `content` and `pcre` predicates k.* The rule from Figure 3.1 has $k = 4$ because it has 4 backtracking-causing multiple-match predicates (including `P5` which does not match the attack packet, but still needs to traverse the packet before failing). Note that not all `contents` and `pcres` can be used to trigger excessive backtracking. Often, predicates that have constraints on the positions they match cannot be used by an attacker to cause backtracking. An example of such a predicate is the first `pcre` from Figure 3.1, predicate `P2`, which has to match immediately after the first `content`.

2. *The size of the attack packets n.* We can use Snort's reassembly module to amplify the effect of backtracking attacks beyond that of a single maximum sized packet. The rule from Figure 3.1 is open to attacks of complexity $O(n^4)$. When Snort combines two attack packets into a virtual packet and feeds it to the rule-matching engine, n doubles, and the rule-matcher does 16 times more work than for either packet alone.

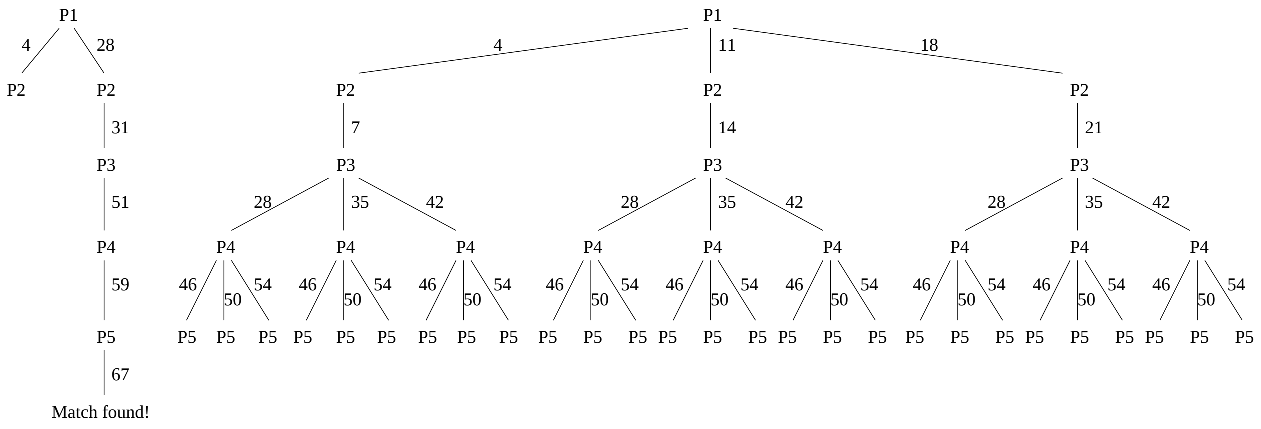


Figure 3.4: Predicate evaluation trees in Snort. The left tree represents the 6 predicate evaluations performed on the payload in Figure 3.2, and the right tree shows the 43 evaluations performed for the payload in Figure 3.3. Numbers on edges indicate payload offsets where a predicate matched.

3. *The total length of the strings needed to match the k predicates*. If these strings are short, the attacker can repeat them many times in a single packet. This influences the constants hidden by the O-notation. Let s_1,…,s_k be the lengths of the strings that can cause matches for the k predicates. If we make their contribution to the processing time explicit we can compute for each string the exact number of repetitions. If we divide the packet into k equal-sized portions, each filled with repetitions of one of these strings, we obtain $n_i = \lfloor \lfloor n/k \rfloor / s_i \rfloor$. The cost of the attack is $O(\prod_{i=1}^{k} n_i) = O(n^k/(k^k \prod_{i=1}^{k} s_i))$. Other factors such as the amount of overlap between these strings, the length of the strings needed to match predicates that do not cause backtracking, and the details of the processing costs of the predicates also influence the processing cost. These factors remain hidden by the constants inside the O-notation.

Approximately 8% of the 3800+ rules in our ruleset were susceptible to backtracking attacks to some degree. Our focus is on the most egregious attacks, which typically yielded slowdowns ranging from three to five orders of magnitude. We quantify the strength of these attacks experimentally in Section 3.4.

```
**MemoizedMatchRule(**$Preds$**)**:
1  $Stack \leftarrow (Preds[0].\texttt{getNewInstance}(0));$
2  $MemoizationTable \leftarrow \emptyset;$
3  while $Stack.size > 0$ do
4      if $Stack.top.\texttt{getNextMatch}()$ then
5          if $Stack.size == Preds.size$ then return $True$;
6          $ofst \leftarrow Stack.top.getMatchOffset();$
7          if $(Stack.top, ofst) \notin MemoizationTable$ then
8              $MemoizationTable \leftarrow MemoizationTable \cup \{(Stack.top, ofst)\};$
9              $\texttt{Push}(Stack, Preds[Stack.size].\texttt{getNewInstance}(ofst));$
10     else $\texttt{Pop}(Stack);$
11 return $False$;
```

Algorithm 3.2: The memoization-enhanced rule-matching algorithm. Lines 2, 7, and 8 have been added.

3.3 Memoization, a remedy for backtracking

As illustrated above, rule-matching engines are open to backtracking attacks if they retain no memory of intermediate results, which for Snort are predicate evaluations that have already been determined to fail. Thus, matching engines can be forced to unnecessarily evaluate the same doomed-for-failure predicates over and over again, as Figure 3.4 indicates.

Algorithm 3.2 shows our revised algorithm for rule matching that uses memoization [27, 89]. It is based on the observation that the outcome of evaluating a sequence of predicates depends only on the payload and the offset at which processing starts. The memoization table holds $(predicate, offset)$ pairs indicating for all predicates, except the first, the offsets at which they have been evaluated thus far. Before evaluating a predicate, the algorithm checks whether it has already been evaluated at the given offset (line 7). If the predicate has been evaluated before, it must have ultimately led to failure, so it is not evaluated again unnecessarily. Otherwise, the $(predicate, offset)$ pair is added to the memoization table (line 8) and the predicate is evaluated (line 9). Note that memoization ensures that no predicate is evaluated more than n times. Thus, if a

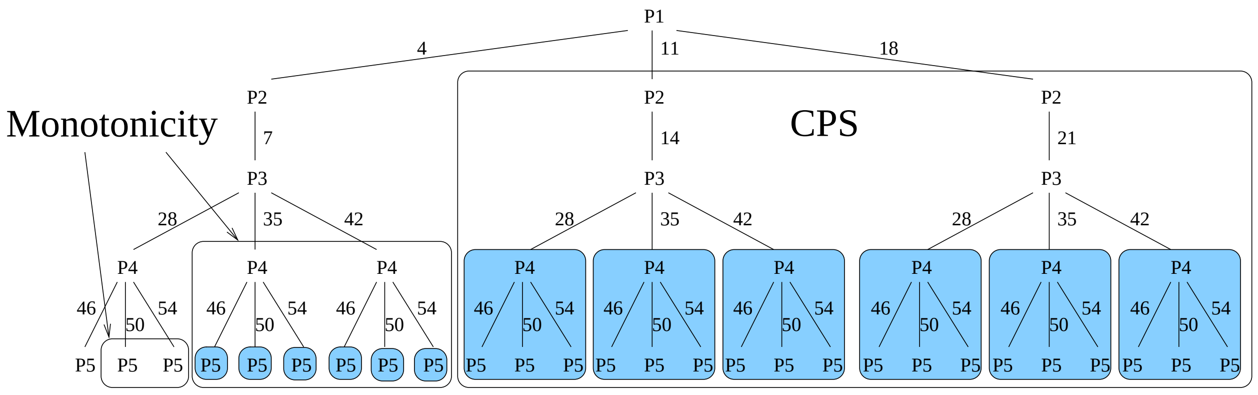


Figure 3.5: The memoization algorithm performs only 13 predicate evaluations instead of 43 as it avoids the grayed-out nodes. The CPS optimization reduces the number of predicate evaluations to 9, and the monotonicity optimization further reduces the evaluations to 5.

rule has k' predicates performing work at most linear in the packet size n, memoization ensures that the amount of work performed by the rule matching algorithm is at most $O(k' \cdot n \cdot n) = O(k'n^2)$. Figure 3.5 updates Figure 3.4 to reflect the effects of memoization. The greyed out nodes in the large tree from Figure 3.5 correspond to the predicates that would not be re-evaluated when using memoization. For the most damaging backtracking attacks against rules in Snort's default rule set, *memoization can reduce the time spent matching a rule against the packet by more than four orders of magnitude* (with the optimizations from Section 3.3.1, more than five orders of magnitude).

To implement memoization, we used pre-allocated bitmaps for the memoization table, with a separate bitmap for each predicate except the first. The size of the bitmaps (in bits) is the same as the size v (in bytes) of the largest virtual packet. Thus if the largest number of predicates in a rule is m, the memory cost of memoization is $v(m-1)/8$ bytes. In our experiments, memoization increases the amount of memory used in Snort by less than 0.1%.

A naive implementation of memoization would need to initialize these bitmaps for every rule evaluated. We avoid this cost by creating a small array that holds up to 5 offsets and an index into the array. When a rule is to be evaluated, only the index into the array needs to be initialized to 0. If the number of offsets a predicate is evaluated at exceeds 5, we switch to a bitmap (and pay the cost of initializing it). It is extremely rare that packets not specifically constructed to trigger backtracking incur the cost of initializing the bitmap.

3.3.1 Further optimizations

We present three optimizations to the basic memoization algorithm: detecting constrained predicate sequences, monotonicity-aware memoization, and avoiding unnecessary memoization after single-match predicates. The first two of these significantly reduce worst case processing time, and all optimizations we use reduce the memory required to perform memoization. Most importantly, all three optimizations are sound when appropriately applied; none of them changes the semantics of rule matching.

Constrained predicate sequences: We use the name *marker* for predicates that ignore the value of the offset parameter. The outcome of a marker and of all predicates subsequent to the marker are independent of where predicates preceding the marker matched. As a result, markers break a rule into sequences of predicates that are independent of each other. We use the name *constrained predicate sequence* (CPS) for a sequence of predicates beginning at one marker and ending just before the next marker. For example, `P3` in Figure 3.1 looks for the string `player=` in the entire payload, not just after the offset where the previous predicate matches because `P3` does not have the `relative` modifier. Thus the rule can be broken into two CPSes: `P1-P2` and `P3-P4-P5`.

Instead of invoking the rule-matching algorithm on the entire rule, we invoke it separately for individual CPSes and fail whenever we find a CPS that cannot be matched against the packet. The algorithm does not need to backtrack across CPS boundaries. Less backtracking is performed because the first predicate in each CPS is invoked at most once. For the example in Figure 3.5, detecting CPSes causes the algorithm not to revisit `P1` and `P2` once `P2` has matched, thus reducing the number of predicate invocations from 13 to 9.

Monotone predicates: Some expensive multiple-match predicates used by Snort have the monotonicity property that we informally define as follows: if the set of matches returned from predicate p at any offset o is always a subset of the set of matches returned from evaluation at an earlier offset, then predicate p is monotone. For these predicates we use the more aggressive *lowest-offset memoization*. In this optimization, we skip calls to a monotone predicate if it has previously been evaluated at an offset smaller than the offset for the current instance. For example, say we first evaluate a monotone `content` predicate starting at offset 100 that does not lead to a match of the entire rule. Later we evaluate the same predicate starting at offset 200. The second instance is guaranteed to find only matches that have already been explored by the first instance. With basic memoization, after each of these matches of the second instance we check the memoization table and do not evaluate the next predicate because we know it will lead to failure. But, the `content` predicate itself is evaluated unnecessarily. With monotonicity-aware memoization, we do not even evaluate the `content` predicate at offset 200.

The monotonicity property generalizes to some regular expressions too, and it can be defined formally as follows: let S_1 be the set of matches obtained when predicate p is evaluated at offset o_1, and S_2 the matches for starting offset o_2. If for all packets and $\forall o_1 \leq o_2$ we have $S_2 \subset S_1$, then p is monotone. In our example from Figure 3.1, all `contents` and `pcres` are monotone with the exception of the first `pcre`, `P2`, because it matches at most once *immediately after* the position where the previous predicate matched.

Lowest-offset memoization helps reduce worst case processing because for some predicates the number of worst-case invocations is reduced from $O(n)$ to 1. For the example in Figure 3.5, this optimization would have eliminated the second and third evaluations for predicates `P4`, and `P5` (and for `P3` also if CPSes are not detected). This further reduces the number of predicate instances evaluated from 9 to 5.

Unnecessary memoization: Basic memoization guarantees that no predicate is evaluated more than n times. For some rules with single-match predicates we can provide the same guarantee even if we omit memoizing some predicates. If we employ memoization before evaluating a single-match predicate, but not before evaluating its successor, we can still guarantee that the successor will not be evaluated more than n times (at most once for every evaluation of our single-match predicate). Also, if we have chains of single-match predicates it is enough to memoize only before the first one to ensure that none is evaluated more than n times. Thus, our third optimization is not to perform memoization after single-match predicates, such as `byte_test` and `byte_jump` (see Table 3.1), except when they are followed by a monotone predicate. For our rule set, this optimization reduces by a factor of two the amount of memory used for memoization.

Protocol	Rule ID	Processing time (seconds/gigabyte)				Slowdown w.r.t. avg traffic		Slowdown w.r.t. same protocol	
		Trace traffic	Backtracking attack						
			Original	Basic Memo.	Memo+Opt.	Original	Memo+Opt	Original	Memo+Opt
IMAP	1755	200.6	89,181	1,802	91.9	4,329×	4.46×	444×	0.46×
IRC	1382	14.6	1,956,858	1,170	87.6	94,993×	4.25×	134,031×	6.00×
MS-SQL	2003	119.3	18,206	715	140.4	884×	6.82×	152×	1.17×
NetBIOS	2403	729.7	357,777	57,173	122.0	17,368×	5.92×	490×	0.17×
Oracle	2611	110.5	6,220,768	3,666	174.0	301,979×	8.45×	56,296×	1.57×
SMTP	3682	132.8	30,933,874	2,192	126.4	1,501,644×	6.14×	232,936×	0.95×
SMTP 3682, w/o reassembly			1,986,624	903	103.1	96,438×	5.00×	14,960×	0.78×
SMTP	2087	132.8	175,657	5,123	164.5	8,527×	7.99×	1,323×	1.24×

Table 3.2: Strength of the backtracking attack and feasibility of the memoization defense. Columns 7-8 show the overall slowdown under attack when memoization is not and is used. Columns 9-10 show similar slowdowns with respect to the same protocol.

3.4 Experimental Results

We performed empirical evaluations with traces and in a live setting. In Section 3.4.1, we present measurements comparing backtracking attack packets with traces of typical network traffic. Our results show that three to six orders of magnitude slowdowns achieved with the backtracking attack are reduced to less than one order of magnitude slowdown under memoization. In Section 3.4.2, we show actual evasion using a non-memoized implementation, and the resulting recovery with the memoized version.

For our experiments we used the Snort NIDS, version 2.4.3, configured to use the Aho-Corasick [1] string matching algorithm. Snort is run on a 2.0 GHz Pentium 4 processor and is loaded with a total of 3812 rules. We instrumented Snort using cycle-accurate Pentium performance counters. When enabled, instrumentation introduced less than 2% overhead to the observed quantities of interest. We found that our measured observations were consistent with the instrumentation results collected by Cabrera *et al.* [18].

3.4.1 Trace-based Results

For benign traffic, we obtained two groups of three traces each captured on different days at distinct times. The first group of traces were captured on the link between a university campus and a departmental network with 1,200 desktop and laptop computers, a number of high-traffic servers (web, ftp, ntp), and scientific computing clusters generating high volumes of traffic. These traces are 7 minutes long and range in size from 3.1 GB to just over 8 GB. The second group of traces were captured in front of a few instructional laboratories totaling 150 desktop clients. They are also 7 minutes long and range in size from 816 MB to 2.6 GB.

We created attack traffic by generating flows corresponding to several protocols and supplying payloads that are constructed in a similar manner to the payload construction outlined in Section 3.2.

In the trace-based experiments, we fed the benign traffic and attack traffic traces into Snort and observed the performance. We performed these experiments with and without memoization enabled. Figure 3.6 shows the slowdowns experienced due to backtracking attacks targeting several rules and the corresponding defense rates. It summarizes the information in Table 3.2. In each group, the leftmost bar represents the cost of packet processing for the specified protocol relative to 20.6 s/GB, the combined average packet processing rate in all our traces. For Rule 1382 (IRC), the rate is less than 1, reflecting the fact that the average traffic processing time for IRC traffic is less than the baseline.

The central bar in each group shows the slowdown observed by packets crafted to target the specific rules indicated at the base of each group. The attacks result in processing times that are typically several orders of magnitude slower than the baseline, with the most egregious attack coming in at a factor of 1.5 million times slower. Finally, in the rightmost bar of each group we see the result of each attack repeated with the memoization defense deployed. In most cases, Snort performance when under attack is comparable to if not better than when not under attack.

Table 3.2 details the attacks and the defenses quantitatively for several different protocols. For each attack, Columns 1 and 2 give the protocol and the targeted Rule ID to which the attack belongs, respectively. Column 3 shows the average processing time for each protocol. Columns 4

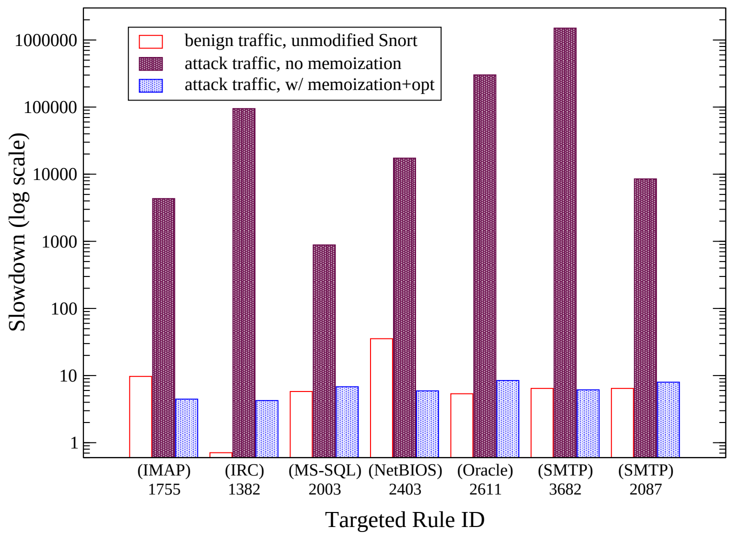


Figure 3.6: Relative processing times for benign and attack traffic, and attack traffic with memoization. Memoization confines the slowdown to less than one order of magnitude.

through 6 show the raw processing times for attack packets under an unmodified Snort, Snort with basic memoization, and Snort with fully optimized memoization. Columns 7-8 give overall slowdowns and Columns 9-10 supply the slowdowns on a per-protocol basis. The backtracking attack achieves slowdowns between 3 and 5 orders of magnitude for rules from many protocols. When memoization is employed, the overall slowdown is confined to within one order of magnitude. Per-protocol, memoization confines most attacks to within a factor of two of their normal processing time.

Rows 7 and 8 highlight the impact that reassembly has on the processing time. In this experiment, when reassembly is performed the size of the virtual packet fed to the rule-matching engine is only twice the size of a non-reassembled packet, but the processing time is almost 16$\times$ longer.

The effects of the three memoization optimizations can be seen by comparing Columns 5 and 6 in Table 3.2. The strength of the optimizations varies by protocol, ranging from just under a factor of 10 to just over a factor of 30, excluding the NetBIOS outlier. In the Snort rule set, NetBIOS rules contain many predicates that can be decomposed into constrained predicate sequences. These rules benefit considerably from the optimizations.

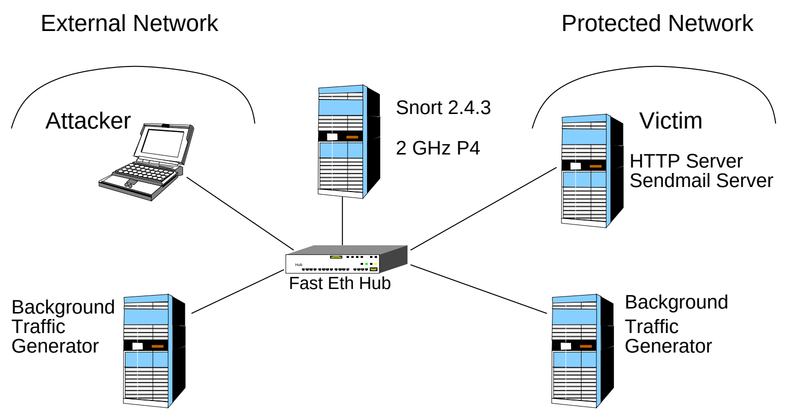


Figure 3.7: Live Snort evasion environment. Snort monitors a network composed of web and mail servers.

Recall that the attacks applied are all low-bandwidth attacks. Even though the overall slowdown rate using memoization is up to an order of magnitude slower, these rates apply *only* to the attack packets (which are few in number) and not to the overall performance of Snort. Under memoization, processing times for attack packets fall within the normal variation exhibited by benign packets.

In the rightmost column, slowdowns less than 1.0 indicate that with all the optimizations included, Snort was able to process backtracking attack packets more quickly than it could process legitimate traffic. In other words, our optimizations allowed Snort to reject these attack packets more quickly than it otherwise was able since fewer overall predicate evaluations are performed.

3.4.2 Evading a Live Snort

In this section we demonstrate the efficacy of the backtracking attack by applying it to a live Snort installation. We first show successful evasion by applying the attack under a variety of conditions. We then show that with memoization, all the formerly undetected attacks are observed.

Figure 3.7 shows the topology used for testing evasion for this experiment. To induce denial of service in Snort, we use an SMTP backtracking attack that connects to a Sendmail SMTP server in the protected network. We are using this attack to mask a Nimda [81] exploit normally recognized by Snort. Both the Nimda exploit and its SMTP cover are sent from the same attacking computer.

Test	Description of backtrack attack	Exploits detected	Required rate (kbps)
1	Control; no attack	300/300	N/A
2	*two* packets every 60 sec.	220/300	0.4
3	*two* packets every 15 sec.	6/300	1.6
4	*one* packet every 5 sec.	4/300	2.4
5	*one* packet every 3 sec.	0/300	4.0
6	*twenty* packets initially	0/300	0.8
7	*one* packet every 3 sec. (memoization enabled)	300/300	N/A
8	*twenty* packets initially (memoization enabled)	300/300	N/A

Table 3.3: Summary of live Snort experiments. Without memoization, 300 intrusions pass into the network undetected.

Each Nimda exploit is sent one byte at a time in packets spaced 1 second apart. To simulate real world conditions, we used the Harpoon traffic generator [103] to continuously generate background traffic at 10 Mbps during the experiments.

We measure the effectiveness of the backtracking attack by the number of malicious exploits that can slip by Snort undetected over various time frames. We initiated a new Nimda exploit attempt every second for 5 minutes, yielding 300 overlapping intrusion attempts. Table 3.3 shows the results. Test 1 is the control: when the backtracking exploit is not performed, Snort recognizes and reports all 300 exploits despite our fragmenting them. In Test 2, we sent two backtracking attack packets every 60 seconds for the duration of the experiment. Snort missed only one-third of the attacks, detecting 222 out of 300 intrusion attempts. In Test 3, we increased the frequency of the backtracking attacks to 2 packets every 15 seconds, dropping the detection rate to just 2% of the transmitted exploits. Test 4 decreased the detection rate even further, and in Tests 5 and 6 the attacker successfully transmitted all 300 exploits without detection. Aside from high CPU utilization during the attacks and an occasional, sporadic port scan warning directed at the SMTP attack, Snort gave no indication of any abnormal activity or intrusion attempt.

These experiments show that the transmission rate needed to successfully penetrate a network undetected is quite low, with both tests 5 and 6 requiring no more than 4.0 kbps of bandwidth.

Test 5, in particular, suggests that perpetual evasion can be achieved through regular, repeated transmissions of backtracking attack packets.

Tests 7 and 8 demonstrate the effectiveness of memoization. These tests repeat Tests 5 and 6 with memoization enabled (including all optimizations). With memoization, Snort successfully detected all intrusions in both tests.

In summary, these experiments validate the results of our trace-based experiments and illustrate the real-world applicability of the backtracking attack. Using carefully crafted and timed packets, we can perpetually disable a NIDS without triggering any alarms, using at most 4 kilobits per second of traffic. Correspondingly, the memoization defense can effectively be used to counter such attacks.

3.5 Discussion

Often, algorithmic complexity attacks and their solutions seem obvious once they have been properly described. Nevertheless, software is still written that is vulnerable to such attacks, which begs the question–how can a NIDS or IPS designer defend against complexity attacks that he has not yet seen? A possible first step is to explicitly consider worst-case performance in critical algorithms and to look at whether it is significantly slower than average case and can be exploited. For example, Crosby and Wallach [29] have shown that in the Bro NIDS, failure to consider worst-case time complexity of hash functions leads to denial of service. With this mindset, we briefly consider mechanisms employed by existing NIDS with an eye towards triggering the worst case.

- Deterministic finite automata (DFA) systems can experience exponential memory requirements when DFA's corresponding to individual rules are combined. In some cases, automata are built incrementally [102] to reduce the footprint of a DFA that cannot otherwise fit in memory. Because each byte of traffic is examined exactly once in a DFA, backtracking does not occur. However, it may be possible for an adversary to construct packets that trigger incremental state creation on each byte of payload, resulting in consistently increased computation costs and potentially leading to memory exhaustion.

- Nondeterministic finite automata (NFA) systems reduce the memory requirement costs of DFA systems by allowing the matcher to be in multiple states concurrently. In practice, this is achieved either through backtracking or by explicitly maintaining and updating multiple states. In the first case, algorithmic complexity attacks are achieved by triggering excessive backtracking. In the second, the attacker tries to force the NIDS to update several states for each byte processed.
- Predicate-based systems such as Snort can be slowed down if the attacker can cause more predicates to be evaluated than in the average case. We have presented an attack that forces the repeated evaluation of a few predicates many times. In contrast, attacks can be devised that seek to evaluate many predicates a few times. For example, Snort employs a multi-pattern string matcher [1] as a pre-filter to pare down the rules to be matched for each packet. Constructing payloads that trigger large numbers of rules can lead to excessive predicate evaluations.

We have performed preliminary work that combines the second and third observations above to yield packet processing times in Snort that are up to 1000 times slower than average. These results, combined with those of this paper, suggest that left unaddressed, algorithmic complexity attacks can pose significant security risks to NIDS.

3.6 Conclusion

Algorithmic complexity attacks are effective when they trigger worst-case behavior that far exceeds average-case behavior. We have described a new algorithmic complexity attack, the backtracking attack, that exploits rule matching algorithms of NIDS to achieve slowdowns of up to six orders of magnitude. When faced with these attacks, a real-time NIDS becomes unable to keep up with incoming traffic, and evasion ensues. We tested this attack on a live Snort installation and showed that the protected network is vulnerable under this attack, along with the tens of thousands of other networks protected by Snort.

To counter this attack, we have developed a semantics-preserving defense based on the principle of memoization that brings Snort performance on attack packets to within an order of magnitude of benign packets. In some cases, the techniques employed allow Snort to evaluate the packets even faster than average.

In general, it is not clear how to find and root out all sources of algorithmic complexity attacks. To do so requires knowledge of average- and worst-case processing costs. Without a formal model of computation, such knowledge is difficult to obtain and is often acquired in an ad-hoc manner. Mechanisms for formally characterizing and identifying algorithms and data structures that are subject to complexity attacks can serve as useful analysis tools for developers of critical systems, such as NIDS.

Chapter 4

DFA Matching Semantics

In our second main contribution, we present an alternative model to DFAs for performing signature matching. As a prelude to that work, in this short chapter we formally define the language specification and matching semantics for regular expressions and deterministic finite automata in the context of intrusion detection. We start by reviewing standard definitions for DFAs. We then extend DFAs as necessary to enable multi-pattern matching of streaming data, and we show that with these extensions, matching can be modeled precisely as a finite state transducer. Finally, we present an algorithm for combining multiple DFAs into a single DFA that matches all the component DFAs simultaneously.

4.1 Deterministic Finite Automata

A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- δ is a function from $Q \times \Sigma$ to Q,
- q_0 is a designated start state, and
- $F \subseteq Q$ is a set of accepting (or final) states.

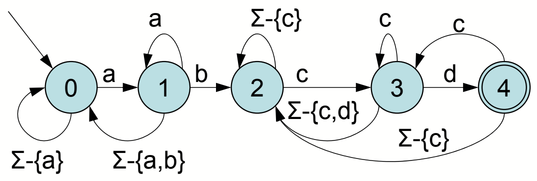


Figure 4.1: A DFA recognizing the regular expression /.*ab.*cd/. Starting in state 0, the input is accepted if the DFA is in state 4 after the last symbol is read.

The function δ is a total function over the states and alphabet that maps states in Q crossed with alphabet symbols in Σ back to states. For each state $q \in Q$ and each symbol $\sigma \in \Sigma$, there is a *transition* from q to some state $q' \in Q$ (note: $q' = q$ is possible).

When supplied an input sequence $x \in \Sigma^{\star}$, a DFA begins at starting state q_0 and moves from state to state as each symbol in x is read. If the DFA is in an accepting, or final, state when the last symbol of x is read, then we say that the DFA *accepts* its input. We use the term *current state* to refer to the state the DFA is in at any given point in the input sequence.

Pictorially, we can represent a DFA as a directed graph in which states are nodes in the graph, edges between nodes are transitions, and each edge is labeled with a symbol from the input alphabet Σ. Since δ is total, there are $|\Sigma|$ labeled edges out of each state to other states in the DFA. Figure 4.1 shows a DFA with $Q = \{0, 1, 2, 3, 4\}$ states, $\Sigma = \{a, b, c, d\}$, $q_0 = 0$, $F = \{4\}$, and δ as depicted. Note the back-arc from state 0 to itself labeled $\Sigma - \{c\}$ is a graphical shorthand representing all edges whose labeled transition is not c. We may alternatively express this as `[^c]`.

DFAs are inherently tied to regular expressions. A regular expression is a mechanism for concisely specifying classes of languages, some of which may be infinite. A DFA, on the other hand, is a language acceptor and is used for recognizing whether a string is a member of a language or not. Moreover, the class of languages accepted by finite automata is exactly the class of languages specifiable using regular expressions. In other words, for any regular expression R, there is a DFA D such that D accepts all strings in $L(R)$, the language described by R. The DFA in Figure 4.1, for example, accepts all strings in the language specified by the regular expression `/.*ab.*cd/`, read as “an arbitrary number of symbols, followed by the sequence `ab`, followed by an arbitrary number of symbols, followed by the sequence `cd`”. Note that throughout this work, as a notational convenience we delimit regular expressions with a forward slash (“/”) at the beginning and the end of the expression.

4.2 Streaming Data and Transducers

Streaming applications can be characterized by their long-lived streams, which can be viewed as a single sequence of input of indeterminate length. For these applications, which include intrusion detection, one is typically interested in finding matching patterns up to the currently-scanned byte in the input, rather than accepting (or rejecting) the entire stream as a whole. This change in semantics affects both the regular expressions used to specify the underlying languages as well as the structure of the automaton used for matching.

Recall that regular expressions are language specifiers. To accommodate streaming data, regular expressions need to be adjusted so that their described languages include the *entire* stream up to the current byte, even the previous portion not directly relevant to the signature. In practice, this is achieved by prefixing the regular expression with a Kleene closure over the full alphabet ($\Sigma^\star$), typically denoted as “`.*`”. This construct has the effect of prepending into the language (specified by the regular expression) all alphabet symbol sequences up to the occurrence of the language strings themselves. For example, the regular expression `/.*ab.*cd/` in Figure 4.1 contains this prefix already. On the other hand, `/ab.*cd/` specifies a language whose strings must begin with the sequence `ab` and is therefore not suitable for stream matching.

Automata semantics must be similarly adapted. Two properties must be amended. First, per the definition, a DFA accepts its input only if it is in an accepting state after all the input has been scanned. The adjustment is to change the matching procedure to acknowledge acceptance each time the DFA moves into an accepting state. This has the effect of allowing the automaton to accept the input up to the current byte without regard to the remainder of the input. Second, DFAs emit a binary “accept” or “reject” to indicate acceptance, but streaming applications typically match many regular expressions simultaneously, each of which has a distinct ID that must be emitted whenever accepted. Ignoring how we combine DFAs for the moment, we address this change by associating one or more IDs with each accepting state and emit them as appropriate.

From a strict definitional perspective, we can model DFAs with these two changes as deterministic finite state transducers [53]. A finite state transducer is a DFA augmented with an output alphabet and a function that maps states or edges to symbols from the output alphabet. Transducers that emit output symbols on states, as is the case here, are termed Moore machines; those that emit on edges are termed Mealy machines. During matching, transducers emit the output symbols associated with each state (or edge) each time it is visited.

In our formulation, Moore machines are the relevant model. Formally, a Moore machine is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where

- Q (states), Σ (alphabet), δ (transition function), and q_0 (start state) are as in DFAs,
- Δ is the output alphabet, and
- λ is a mapping from Q to Δ specifying the output associated with each state (which may be empty for some states).

Note that there is no set of final states in a Moore machine. The Moore machine model captures streaming data matching semantics precisely. To wit, starting with a DFA, we set Δ (the output alphabet) to be the set of all regular expression IDs, and we construct λ to map accepting states in the DFA to the output symbol in Δ corresponding to the proper ID. Thus, matching can be interpreted as the process of converting the input to a sequence of matching regular expression IDs.

Throughout this dissertation, we assume Moore machine transducer semantics for simultaneous matching of multiple regular expression patterns by making the simple changes to DFA matching described above. Nevertheless, the changes from standard DFA matching are minor. Thus, to remain consistent with accepted practice, we continue to refer to the process generically as DFA matching.

4.3 Combining Automata

Automata combination refers to the process of combining two or more distinct automata into a single, composite automaton which, when executed, is equivalent to executing all the individual automata simultaneously. This process is central to meeting the performance demands of signature matching. We give a formal description of DFA combination and present an algorithm for efficiently combining automata.

Definition 4.1 Let $D_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be two DFAs with common alphabet Σ. The *standard product construction* of D_1 and D_2 is the DFA $D = (Q, \Sigma, \delta, s, F)$, where

- for each $q_1 \in Q_1$ and $q_2 \in Q_2$, there is a distinct element $q = \langle q_1, q_2 \rangle \in Q$,
- $s = \langle s_1, s_2 \rangle$,
- for each $q_{1a}, q_{1b} \in Q_1$ and $q_{2a}, q_{2b} \in Q_2$ and $\sigma \in \Sigma$, if $\delta_1(q_{1a}, \sigma) \to q_{1b}$ and $\delta_2(q_{2a}, \sigma) \to q_{2b}$, then $\delta(\langle q_{1a}, q_{1b} \rangle, \sigma) \to \langle q_{2a}, q_{2b} \rangle$,
- for each $\langle q_1, q_2 \rangle \in Q$, if $q_1 \in F_1$ or $q_2 \in F_2$ (or both), then $\langle q_1, q_2 \rangle \in F$.

We denote the product construction D of automata D_1 and D_2 as $D = D_1 + D_2$. Further, $L(D) = L(D_1) \cup L(D_2)$. The standard product construction for Moore automata is defined similarly.

Definition 4.2 Let $M_1 = (Q_1, \Sigma, \Delta_1, \delta_1, \lambda_1, s_1)$ and $M_2 = (Q_2, \Sigma, \Delta_2, \delta_2, \lambda_2, s_2)$ be two DFAs with common alphabet Σ. Then, the standard product construction of of M_1 and M_2 is the Moore automaton $M = (Q, \Sigma, \Delta, \delta, \lambda, s)$ where

- Q, δ, and s are as defined for DFA combination above,
- for each $\rho_1 \in \{\Delta_1 \cup \epsilon\}$ and $\rho_2 \in \{\Delta_2 \cup \epsilon\}$, there is a distinct symbol $\langle \rho_1, \rho_2 \rangle \in \Delta$ (ϵ is a symbol not occurring in Δ_1 or Δ_2),
- for each $q_1 \in Q_1$ and $q_2 \in Q_2$, there are three cases to consider:

```
    Combine(MooreMachine first, MooreMachine second) :
 1  worklist WL
 2  MooreMachine c
 3  c.addState (⟨first.start, second.start⟩)
 4  ⟨first.start,second.start⟩.output_sym.append (first.start.output_sym)
 5  ⟨first.start,second.start⟩.output_sym.append (second.start.output_sym)
 6  c.setStart (⟨first.start, second.start⟩)
 7  WL = { ⟨first.start, second.start⟩ }
 8  while ( |WL| > 0 ) do
 9      ⟨s,t⟩ = WL.pop ()
10      foreach ($\beta \in \Sigma$) do
11          $s'$ = first.getNextState($s$, $\beta$)
12          $t'$ = second.getNextState($t$, $\beta$)
13          if ⟨$s'$,$t'$⟩ $\notin$ c.states then
14              c.addState (⟨$s'$,$t'$⟩)
15              ⟨$s'$, $t'$⟩.output_sym.append ($s'$.output_sym)
16              ⟨$s'$, $t'$⟩.output_sym.append ($t'$.output_sym)
17              WL.push ( ⟨$s'$,$t'$⟩)
18          c.addTrans (⟨$s, t$⟩,⟨$s', t'$⟩,$\beta$)
19  return c
```

Algorithm 4.1: Standard product construction for Moore machines.

i. if $\lambda_1(q_1) \rightarrow \rho_1$ and $\lambda_2(q_2)$ is undefined, then $\lambda(\langle q_1, q_2 \rangle) = \langle \rho_1, \epsilon \rangle$;

ii. if $\lambda_2(q_2) \rightarrow \rho_2$ and $\lambda_1(q_1)$ is undefined, then $\lambda(\langle q_1, q_2 \rangle) = \langle \epsilon, \rho_2 \rangle$;

iii. if $\lambda_1(q_1) \rightarrow \rho_1$ and $\lambda_2(q_2) \rightarrow \rho_2$, then $\lambda(\langle q_1, q_2 \rangle) = \langle \rho_1, \rho_2 \rangle$.

The construction above produces combined automata with $|M_1| \cdot |M_2|$ states, where $| \cdot |$ denotes the number of states. Nevertheless, some combined states may be unreachable; *i.e.*, there is no sequence of transitions from the start state leading to the unreachable state. Algorithm 4.1 gives a worklist-based algorithm for computing the product construction of Moore automata that avoids constructing unreachable states. In line 7, the combined start state initializes a worklist which is added to by each newly created state (line 17). In each iteration, the algorithm pops a state from the worklist, follows transitions out of it, and places new states on the worklist as necessary. Iteration continues until the worklist is empty, when all combined states have been created and processed. Since the number of states in the two input machines is finite, the algorithm must terminate.

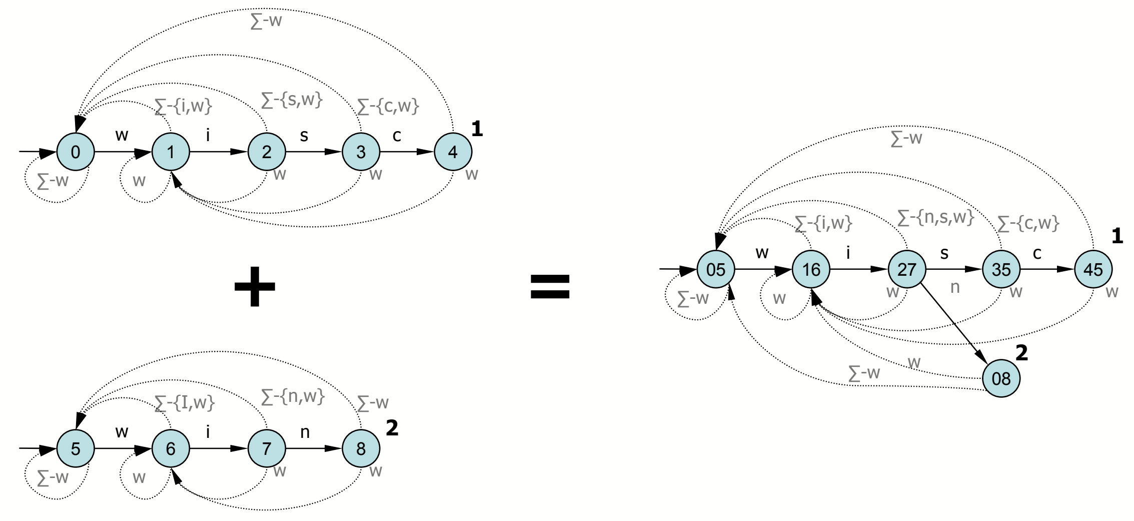


Figure 4.2: Standard product construction (right) for DFAs corresponding to `/.*wisc/` (upper left) and `/.*win/` (lower left).

Output symbols attached to states are represented as lists to which output symbols are appended. For each state $q = \langle s, t \rangle$ in the combined automaton, we simply copy the output symbols from s and t into q (Lines 15 and 16). The correctness of this follows from the fact that entering composite state q when matching is equivalent to entering states s and t simultaneously, implying that the symbols in both s and t need to be emitted. Alternatively, for DFA matching we replace Lines 15 and 16 with a statement that marks q as accepting if either s or t is accepting. Figure 4.2 illustrates the combination algorithm for two DFAs corresponding to the regular expressions `/.*wisc/` (upper left in the figure, with ID 1) and `/.*win/` (lower left, with ID 2).

Chapter 5

State-Space Explosion and Ambiguity

This chapter and the next are devoted to mechanisms for avoiding the time-space tradeoff associated with automata-based matching. We begin in this chapter with a first-principles characterization of state-space explosion that lays the groundwork for the techniques we develop to circumvent the tradeoff. We describe, formally, why it occurs and give ideal conditions that eliminate it when satisfied. When these conditions are met, automata can be freely combined without any state explosion. In preparation for the next chapter, we then illustrate how auxiliary state variables can be used to "factor out" the components of automata that violate these conditions.

5.1 Combining DFAs Considered Harmful

DFAs corresponding to NIDS signatures explode when combined. That is, the size of the state space, and hence the number of states, increases dramatically when DFAs are combined. The magnitude of DFA state space explosion depends strongly on the types of signatures being matched. For simple string-based regular expressions of the form $/.*\mathtt{s}/$, where $\mathtt{s} \in \Sigma^*$ is a sequence of alphabet symbols, the number of states required to recognize n signatures is bounded above by the total size of the strings, or $O(n)$ if we bound the size of the largest string. Other types of patterns can cause the number of states to increase quadratically and exponentially in the number of signatures.

Consider "counting" regular expressions of the form $\mathtt{/.*\backslash n s_1 [\hat{}\backslash n]\{k\}/}$, read as "newline followed by the sequence $\mathtt{s_1}$ followed by k non-newline characters". Signatures of this form are commonly used to identify and stop exploits aimed at triggering buffer overflow vulnerabilities.

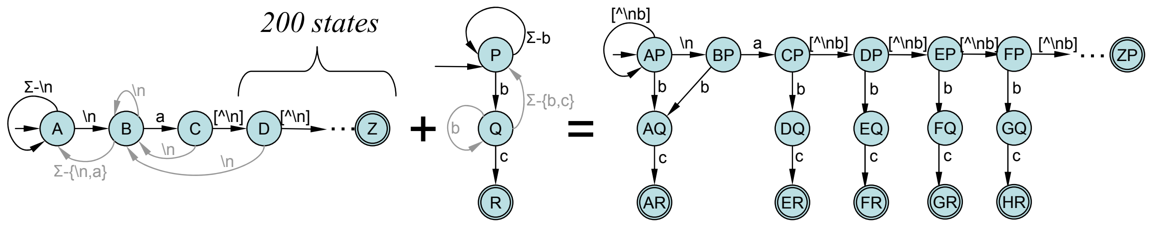


Figure 5.1: The combined DFA for corresponding to the expressions `/.*\na[^\n]{200}/` and `/.*bc/` replicates the second expression 200 times, once for each counting state (for clarity, some edges have been omitted).

The left-hand automaton in Figure 5.1 shows the DFA for a signature of this form. Note that the counting range specifier `{200}` is a form of syntactic sugar that replaces 200 repetitions of `[^\n]`. In this case, there are 200 states used solely for counting the number of successive non-newline symbols observed.

By itself, this DFA is relatively innocuous, since the number of states is linear in the number of symbols in the expression. However, when combined with other DFAs, even those for simple string-based regular expressions, the total number of states multiplies. The right-hand side of Figure 5.1 shows the DFA resulting from the cross product of the counting DFA with a simple string-based automaton. For clarity, some edges have been removed. In general, the combined DFA needs to concurrently track the independent matching progress of both source DFAs. For this example this means that the DFA corresponding to `/.*bc/` is replicated at each of the counting states in the first DFA. This requires $O(nk)$ states for tracking a single such counting automata and n strings and $O(n^2k)$ states for tracking n such signatures and n strings.

Other signature patterns can lead to an exponential growth in the state space when their DFAs are combined. This occurs, for example, with signatures of the form `/.*`$\texttt{s}_1$`.*`$\texttt{s}_2$`/`, read as "sub-pattern $\texttt{s}_1$ followed by an arbitrary number of characters followed by sub-pattern $\texttt{s}_2$". Figure 5.2 shows the result of combining the DFAs for the signatures `/.*ab.*cd/` and `/.*ef.*gh/`. For each signature of this type, the combined DFA needs to remember whether it has already seen the first sub-pattern so that it knows whether or not to accept the second sub-pattern. For example, in Figure 5.2 the DFA is in state PV when neither `ab` nor `ef` has been observed. Similarly, it is in state RV when `ab` but not `ef` is seen, state PX when `ef` but not `ab` is seen, and state RX when both

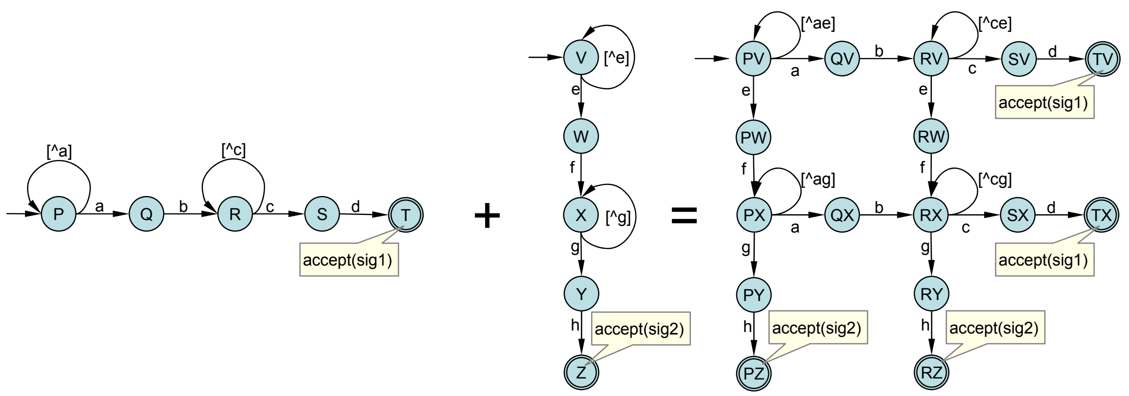


Figure 5.2: The combined DFA for expressions `/.*ab.*cd/` and `/.*ef.*gh/` must use states to "remember" which subpatterns have occurred (for clarity, some edges have been omitted).

`ab` and `ef` have been seen. In general, to remember n independent bits of information, the DFA needs at least 2^n distinct states. For this case, an analysis of the generalized example shows that if the strings are of length l, then the actual number of states used by the combined DFA is $O(nl2^n)$.

Both of these examples illustrate the growth in the state space that occurs when DFAs are combined. Intuitively, we can generalize this phenomenon as follows. When a set of DFAs are combined, the combined automaton tracks the matching progress of the individual automata simultaneously. As alluded to by the state names in Figures 5.1 and 5.2, states in the combined automaton are equivalent to tuples of individual states from the source automata. Thus, in the combined DFA, there is a distinct state for each reachable combination of states in the source DFAs. As shown above, signatures often overlap or (partially) subsume each other, leading to interleaved matching progress with many distinct combinations of reachable states.

Beyond this, DFA states typically require 1,024 bytes each, so that large numbers of states can quickly exhaust memory. In a later section, we informally present a mechanism for restructuring the state-space so that even though state-space explosion still occurs, it does not affect the number of automaton states.

5.2 Understanding State-Space Explosion

In this section we formally characterize state space explosion and give sufficient conditions for guaranteeing that such explosion will not occur. We show how incorporating auxiliary state variables can be used to transform automata so that they satisfy these conditions and eliminate such explosion in automata states directly. This characterization provides the underlying foundation that motivates our work on extended finite automata.

5.2.1 State and Path Ambiguity

State-space explosion centers around the notion of ambiguity, which we define as follows. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA with states Q, input symbols Σ, transition function δ, start state q_0, and accepting states $F \subseteq Q$. For state $q \in Q$ we define $paths(q)$ to be the set of paths from q_0 to q. In the presence of cycles, *paths*(q) may be infinite. Since D is deterministic, we can uniquely represent each path $\pi \in paths(q)$ by the corresponding sequence of input symbols $\sigma(\pi)$.

We say that state q is *unambiguous* if and only if the following conditions hold:

1. there exists a finite sequence $x_q \in \Sigma^\star$ such that for each path $\pi \in paths(q)$, $\sigma(\pi) = y \cdot x_q$ for $y \in \Sigma^\star$;
2. for some $\pi \in paths(q), \sigma(\pi) = x_q$ (*i.e.*, $y = \epsilon$).

In other words, q is unambiguous if and only if all paths to q have the same suffix x_q and at least one path to q is specified solely by x_q.

A DFA D is *unambiguous* if and only if all states in D are unambiguous and the following conditions also hold:

3. for each $y \in \Sigma^\star$, $\exists f \in F$ such that $y \cdot x_f \in paths(f)$;
4. let $m(f)$ be the path corresponding to x_f for state $f \in F$. Then, for each $q \in Q$, $q \in m(f)$ for some $f \in F$.

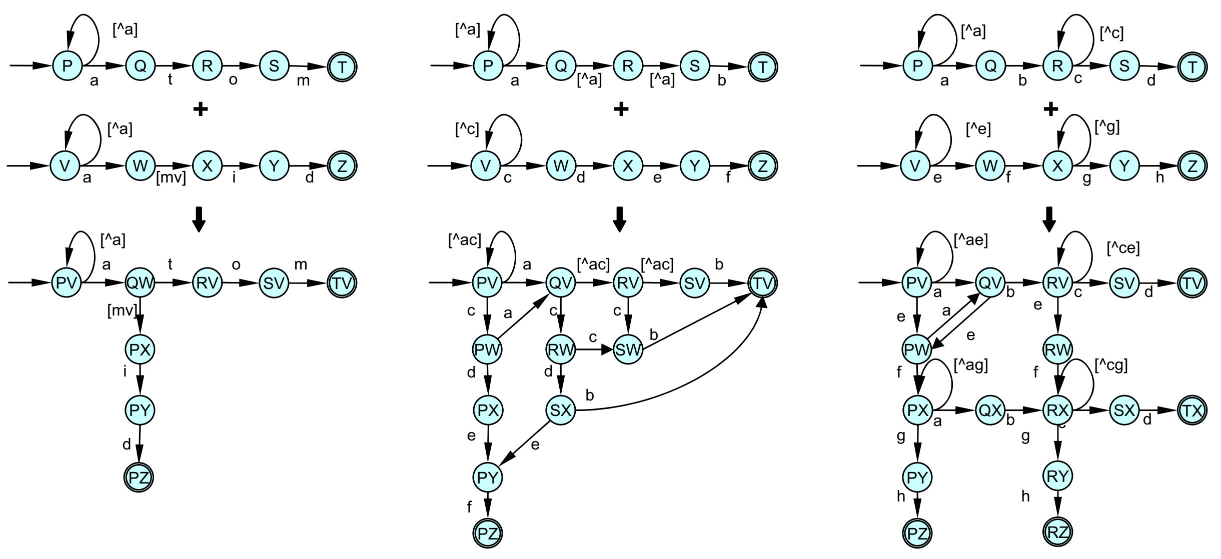
For an unambiguous automaton D, the first three conditions ensure that all strings in the language accepted by D are of the form $.*x_f$ where $f \in F$. The fourth condition ensures that there are no superfluous states that do not advance matching progress toward acceptance and is unnecessary if the DFA has been minimized. Note that ambiguity is different from nondeterminism; *i.e.*, an ambiguous state may be reached by many distinct sequences, but the succession of states is still deterministic in the input. Finally, we say that a path $\pi \in paths(q)$ is ambiguous if there is an ambiguous state in π.

5.2.2 Combination and State Explosion

State-space explosion results from the interaction between states in ambiguous and unambiguous paths when automata are combined. During combination, unambiguous states in the prefix of a path from one automaton get replicated when combined with ambiguous states in a path in another automaton. This phenomenon occurs because the combined automaton must now track progress in matching both the unambiguous path and, independently, the ambiguous path. Of course, the amount of replication observed depends on how extreme and pervasive the ambiguity is in the two source automata and how much interaction occurs between them. Automata with limited levels of ambiguity introduce comparatively small amounts of replication, whereas a path of infinite length can cause an entire automaton to be copied and leads directly to exponential replication.

To illustrate, consider the examples in Figure 5.3. In this figure and in most others, we show all states but for clarity eliminate many transitions. In Figure 5.3a, automata for the expressions `/.*atom/` and `/.*a[mv]id/` are combined. Only the first automaton is unambiguous, but the ambiguity in the second automaton is limited to allowing only an `m` or a `v` in the transition between the two states. When combined, the unambiguous and ambiguous paths do not interact, and no state replication occurs in this case. In general, though, the replication is limited to a few states.

Figure 5.3b describes the case in which the regular expression `/.*a[^a][^a]b/` (read as: "an `a` followed by two non-`a` characters, followed by `b`") is combined with the expression `/.*cdef/`. In the first automaton, paths to States R, S, and T are all ambiguous (the path to T is ambiguous because no path $p = yx$ where $x =$ `b` and $y = \epsilon$ exists). In the combined automaton shown in the figure, a full copy of both original automata is required so that both expressions can be matched. However, states in the prefix of the single unambiguous path in `/cdef/` must also be partially replicated so that the combined automaton can properly track the progress in matching both `/cdef/` and the "don't care" transitions in the first automaton. In this case, the number of paths to ambiguous states is finite, but additional unambiguous paths in the first regular expression



(a) `/.*atom/` and `/.*a[mv]id/` (b) `/.*a[ˆa][ˆa]b/` and `/.*cdef/` (c) `/.*ab.*cd/` and `/.*ef.*gh/`

Figure 5.3: Depending on the structure of the underlying automata, the combined automaton sizes may be linear (left), polynomial (middle) or exponential (right) in the limit (some edges removed for clarity).

would be partially replicated along these as well, so that in practice a large number of additional states may need to be created.

Figure 5.3c depicts the case in which both regular expressions contain a Kleene closure (`.*`) in the middle of the expression. This introduces ambiguous paths of infinite length since the closure can consume an infinite number of symbols. When combined with another automaton A, the closure effectively replicates A in many cases. When the two automata in the figure are combined, the result is similar to a cross-product of states, since the two automata are heavily interleaved and states must be created that track each possible position in the first automaton with each possible position in the second. When n expressions of this form are combined, the number of required states in the combined automata is exponential in n.

5.2.3 Eliminating Ambiguity Using Auxiliary Variables

From a systematic perspective, we can eliminate state-space explosion by first identifying the conditions in which it cannot occur, and second, specifying transformations that translate offending automata into automata that satisfy the conditions without changing semantics. In this context, ambiguity in automata as defined above provides a sufficient set of conditions, and we relate them to state space explosion by the following theorems.

Theorem 5.1 Let D_1 and D_2 be DFAs with $D_1 + D_2$ their standard product combination. If D_1 and D_2 are unambiguous, then $|D_1 + D_2| < |D_1| + |D_2|$, where $|D|$ is the number of states in D.

Theorem 5.2 If D_1 and D_2 are unambiguous, then $D_1 + D_2$ is unambiguous.

We provide a brief sketch of a proof. As described in Section 5.2.1, an unambiguous DFA $D = (Q, \Sigma, \delta, q_0, F)$ recognizes languages of the form $\{.{*}x_f | f \in F\}$. Consequently, the language $L(D)$ can be expressed as $\Sigma^\star(\sum_{f \in F} x_f)\Sigma^\star$. But, this has the same structure as languages recognized by Aho-Corasick-constructed DFAs (see [1, section 8]). Thus, unambiguous DFAs are equivalent to Aho-Corasick automata. Now, combining Aho-Corasick automata is equivalent to taking the strings from one automaton and inserting them into the other. Moreover, the number of states in an Aho-Corasick automaton is bounded above by $\sum_{i=1}^{k} |y_i|$, where $|y_i|$ denotes the length of the string y_i. From this, Theorem 1 is established and Theorem 2 immediately follows.

Theorem 5.1 simply places a bound on the number of automaton states that are produced by the combination process. Theorem 5.2 states that unambiguity is closed under standard combination.

We define state-space explosion formally as a pairwise phenomenon that occurs whenever $|D_1 + D_2| \geq |D_1| + |D_2|$ for two automata D_1 and D_2. Theorem 1 is overly restrictive since in reality a larger class of expressions than strings of the form $. * s$ can be combined without any appreciable blowup (Figure 5.3a, for example). Further, a combined automaton that exhibits a modest increase in the number of states beyond the additive sum of its component DFAs is perfectly acceptable in many cases. Despite these restrictions, Theorem 5.1 *is* sufficient for the purposes of characterization and provides an ideal: if, as in string matching, we can ensure that the additive sum of states always dominates the combined sum for any automata, then state-space explosion can never occur.

Given these conditions, the next task is to identify a mechanism for transforming automata. As stated earlier, by augmenting DFAs with auxiliary variables we can represent the state space more

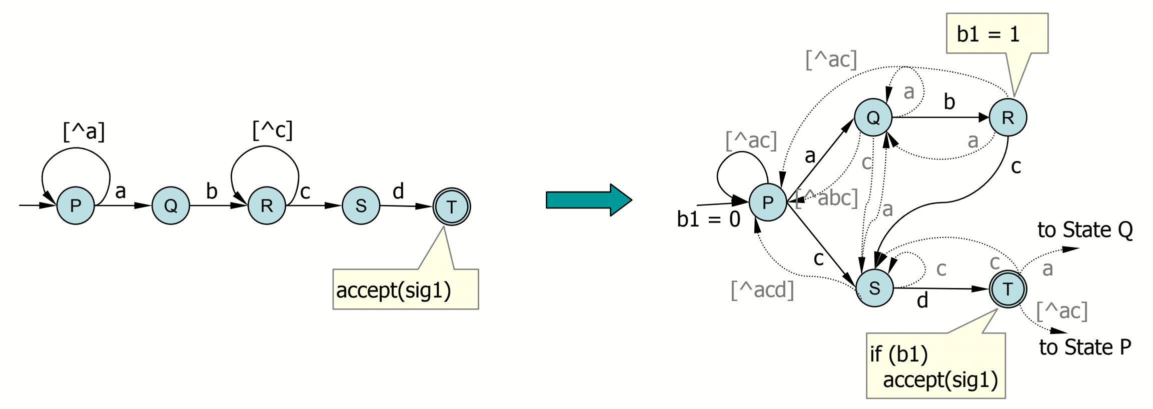


Figure 5.4: Adding a bit to `/.*ab.*cd/` transforms the automaton so that it is not ambiguous. Dotted lines show less-important edges.

compactly than explicit states alone can do. Intuitively, incorporating auxiliary variables changes the "shape" of an automaton since part of the computation state is now stored in the variables. By carefully controlling how these variables are incorporated and manipulated, we can in turn transform an ambiguous DFA into an equivalent automaton with less ambiguity or none at all.

As an example, consider again `/.*ab.*cd/`, whose DFA is ambiguous (state R is ambiguous). In addition, assume that we can associate a single bit with this expression that can be freely manipulated (set, reset, and tested). Ignoring the method of construction for the time being, we can use this bit to "remember" whether the first substring has been observed or not. In so doing, the shape of the automaton itself is transformed as illustrated in Figure 5.4. When constructed appropriately, the new automaton along with the bit preserves the semantics of the regular expression. Most importantly, in the new automaton all states are unambiguous and the automaton satisfies the condition for avoiding state-space explosion.

Next, consider `/\na[^\n]{200}/` whose DFA was shown in Figure 5.1. The DFA for this expression contains 200 ambiguous states whose sole purpose is to count the distance in the input from the sequence `\na` in which a newline is not observed. As shown, when combined with other DFAs, unambiguous paths are partially or fully replicated at each of these "counting states."

To eliminate the ambiguity in this automaton, we introduce a simple counter whose value can be set (initialized to a value), reset (indicating the counter value should be ignored), decremented, and compared to zero. The transformation incorporating the counter is given in Figure 5.5. The state variable replaces the 200 counting states, leading to a sharp reduction in the size of the automaton. Most importantly, the

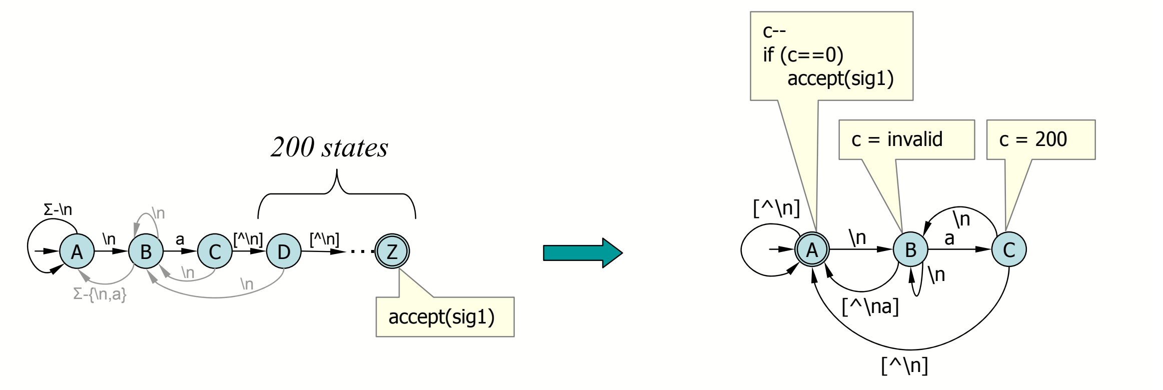


Figure 5.5: Adding a counter to `/.*\na[^\n]{200}/`. The resulting automaton is unambiguous.

careful inclusion of the counter has yielded an automaton whose states are unambiguous and satisfies the unambiguity condition. Note that the counter is decremented on the start state. For this counter we assume a semantics in which the variable is *inactive* until initialized. We discuss this property in more detail in Section 6.5.

In both of these examples, we have in essence “factored out” the ambiguity of the DFAs and placed it into auxiliary state variables that manipulate some aspects of the matching state more compactly than explicit DFA states can. Figures 5.6 and 5.7 show the same combination operation as Figures 5.1 and Figures 5.2, respectively, except that the ambiguous DFAs in the figures are replaced by their semantically equivalent unambiguous XFA counterparts. In both figures, the input XFAs are unambiguous, and the resulting combined XFAs are also unambiguous, as per Theorem 5.2. Theorem 5.1 is also easily checked.

In general, the amount of auxiliary state we introduce and its effect on the underlying automaton is very fluid. At one extreme, DFAs corresponding to strings have unambiguous forms that require no auxiliary state. At the other end, we can reduce an automaton to a single state (with transitions to itself) by incorporating an appropriate combination of possibly many different types of state variables. Of course, the number of state variables may then be very large, and updating them may be time consuming. Transforming an ambiguous automata to an equivalent unambiguous form may introduce auxiliary state that lies somewhere between these extremes.

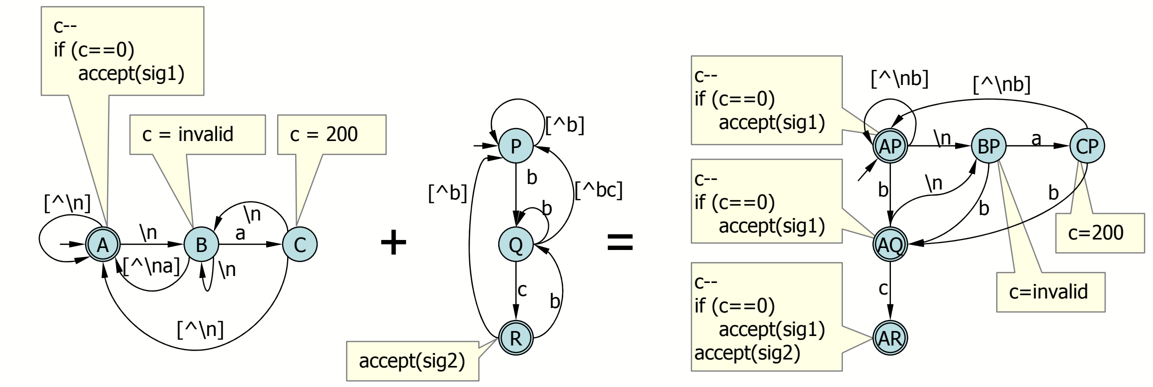


Figure 5.6: The combined automaton corresponding to the expressions `/.*\na[^\n]{200}/` and `/.*bc/`. The inputs on the left are unambiguous as is their combination on the right.

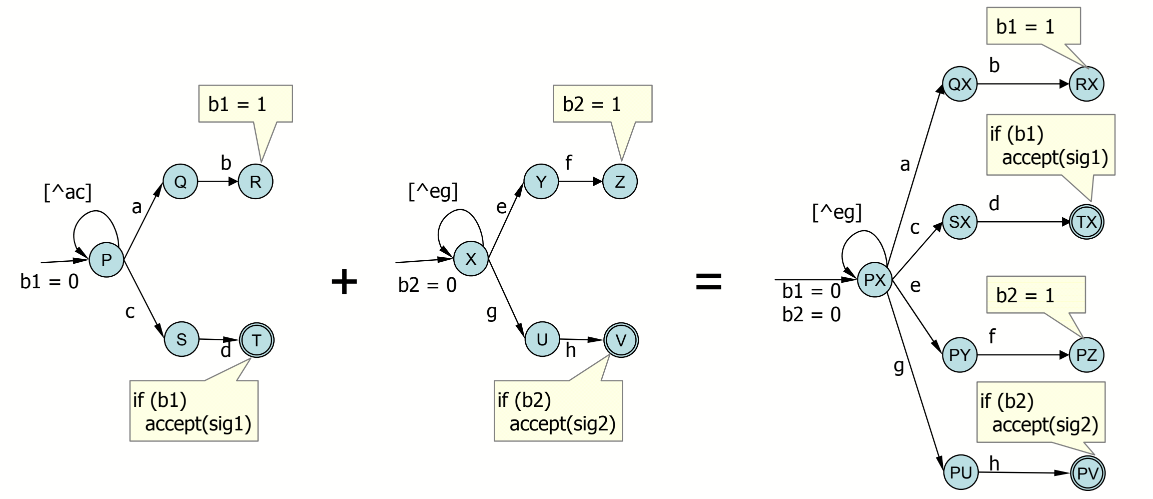


Figure 5.7: The combined automata for expressions `/.*ab.*cd/` and `/.*ef.*gh/`. All automata are unambiguous.

5.2.4 Generalizing Ambiguity

The conditions we have given to support these conclusions are very strict. Here, we take first steps toward generalizing the notion of ambiguity to better characterize polynomial state replication as illustrated in Figure 5.3b. A language $L \subseteq \Sigma^\star$ is *finite* if and only if the number of sequences $|L|$ in L is finite. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We say that a state $q \in Q$ is *finitely unambiguous* if and only if $L_D(q) = \Sigma^\star R_q$, where R_q is a finite set of strings. In this case, all suffixes of paths to a finitely unambiguous state q belong to the set R_q. Thus, finite unambiguity generalizes our earlier definition. As before, DFA D is finitely unambiguous if all of its states are finitely unambiguous.

Consider two finitely unambiguous DFAs $D_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$. The languages L_1 and L_2 accepted by D_1 and D_2, respectively, have the following form:

$$L_1 = \sum_{q \in F_1} \Sigma^\star \cdot R_q \qquad L_2 = \sum_{q \in F_2} \Sigma^\star \cdot R_q$$

Hence, the language accepted by $D_1 + D_2$ is given as $\Sigma^\star \cdot (\sum_{q \in F_1 \cup F_2} R_q)$ and the size of the DFA $D_1 + D_2$ is bounded above by $\sum_{q \in F_1 \cup F_2} \sum_{\sigma \in R_q(q)} |\sigma|$. Thus, we can bound the amount of replication that occurs.

To summarize, we have presented a formal framework for characterizing state space explosion and have shown that in this framework, auxiliary variables can be used to eliminate explosion. In the next section, we formalize the ideas presented here into an explicit model that specifies how auxiliary variables are incorporated into automata.

Chapter 6

Extended Finite Automata

In the previous chapter we gave a formal characterization of state-space explosion and showed by example how auxiliary variables can be employed to remove the ambiguity from automata. In this chapter, we present a formal model for incorporating variables into automata and explore its consequences for signature matching applications. This model, termed *Extended Finite Automata* [100, 101], or XFAs for short, extends the standard DFA model with auxiliary state variables and instructions for manipulating them, yet at the same time retains many of the advantageous traits of DFAs. For example, the model is fully deterministic in the states and input, and enables combination and matching algorithms that are straightforward extensions to those for DFAs.

This chapter proceeds by first giving the formal model for XFAs (Section 6.1), and then discussing algorithms for constructing XFAs (Section 6.2), combining XFAs (Section 6.3), and matching XFAs (Section 6.4). Following that in Section 6.5 we present a set of optimizations inspired from compiler construction principles that further reduce memory usage and increase performance. In Section 6.6, we present a series of experimental results showing the behavior of XFAs, and we conclude with a discussion in Section 6.7.

6.1 Formal Models

XFAs generalize DFAs to include variables along with instructions for manipulating those variables. Practically, variables are stored in an auxiliary memory associated with an automaton. Subject to certain constraints, instructions for manipulating variables can be attached to either edges or states.

We formally represent the space of auxiliary variable values used by Extended Finite Automata as a finite set termed the *data domain*, denoted as D. Each distinct setting of the variables is represented as a distinct data value d in the data domain. We associate a data value d_i with the current state c_i, whose values change as the automaton processes its input. Together, these extend the notion of a current state to a current *configuration* (c_i, d_i). We similarly extend starting states and accepting (or final) states in DFAs to starting and accepting configurations in XFAs. With each transition or state we associate an *update function* $U : D \to D$ (or for non-deterministic XFAs an *update relation* $U \subseteq D \times D$) which specifies how d is to be updated. For the common case in which the data domain is not being updated on a transition, we associate the identity function with the transition.

Below, we present two equivalent models for XFAs: one with instructions attached to edges, and one with instructions attached to states. Each serves a distinct purpose.

Definition 6.1 An edge-based nondeterministic extended finite automaton (NXFA) is described by the 7-tuple $(Q, \Sigma, \delta, D, U, QD_0, F)$, where

- Q is the set of states,
- Σ is a finite set of symbols (the alphabet),
- $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation,
- D is the finite set of values in the data domain,
- $U : Q \times (\Sigma \cup \{\epsilon\}) \times Q \to 2^{D \times D}$ is the per transition *update relation* which defines how the data value is updated on every transition,
- $QD_0 \subseteq Q \times D$ is the set of initial configurations consisting of initial states paired with initial data domain values,
- $F \subseteq Q \times D$ is the set of accepting configurations.

XFAs add two additional components (a data domain and an update relation) to the classical non-deterministic finite automaton (NFA). Further, as described above, XFAs change the initial state and the acceptance criteria to include elements of the data domain.

As stated above, a configuration is a tuple (q, d) where $q \in Q$ and $d \in D$. Configurations extend the notion of state in DFAs to include a data domain value. Similarly, operations manipulating states in DFAs are extended to manipulate configurations in XFAs. Thus, for XFAs, there is a transition from configuration (q, d) to (q', d') on an input symbol $a \in \Sigma$ (denoted by $(q, d) \xrightarrow{a} (q', d')$) if and only if $(q, a, q') \in \delta$ and $(d, d') \in U(q, a, q')$. Further, a string $a_1 a_2 \cdots a_k$ is accepted by an XFA $\mathcal{X}$ if and only if there is a sequence of transitions $(q_0, d_0) \xrightarrow{a_1} (q_1, d_1) \cdots (q_{k-1}, d_{k-1}) \xrightarrow{a_k} (q_k, d_k)$ such that $(q_k, d_k) \in F$. The set of strings accepted by $\mathcal{X}$ is the language $L(\mathcal{X})$.

An XFA $\mathcal{X}$ is *state deterministic* if the transition relation δ is a function from $Q \times \Sigma$ to Q. $\mathcal{X}$ is *data deterministic* if for all $(q, a, q') \in Q \times \Sigma \times Q$, $U(q, a, q')$ is a function from D to D. $\mathcal{X}$ is *deterministic* if it is both state and data deterministic. For deterministic XFAs, we provide a tighter definition for δ and U as follows:

Definition 6.2 An edge-based deterministic extended finite automaton (XFA) is described by the 7-tuple $(Q, D, \Sigma, \delta, U_\delta, (q_0, d_0), F)$, where

- Q is the set of states,
- Σ is a finite set of symbols,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- D is the finite set of values in the data domain,
- $U : Q \times \Sigma \times D \to D$ is the per transition *update function*,
- (q_0, d_0) is the initial configuration,
- $F \subseteq Q \times D$ is the set of accepting configurations.

In the deterministic definition, both δ and U are functions instead of relations. The transition function δ has the same type as for DFAs. For the update function U, a configuration (state and data value) and an input symbol uniquely determine the updated data value.

The above definitions attach instructions to transitions, but instructions can alternatively be attached to states. From a language perspective, edge-based and state-based XFAs are equivalent: for any edge-based XFA $\mathcal{X}_e$ there is a state-based XFA $\mathcal{X}_s$ such that $L(\mathcal{X}_e) = L(\mathcal{X}_s)$ and vice versa. We define a state-based XFA as follows.

Definition 6.3 A state-based deterministic extended finite automaton (XFA) is described by the 7-tuple $(Q, \Sigma, \delta, D, U, (q_0, v_0), F)$, where

- Q is the set of states,
- Σ is a finite set of symbols,
- $\delta : Q \times \Sigma \to Q$ is the transition function,

- D is a finite set of values,
- $U : Q \times D \rightarrow D$ is the per-*state* update function
- (q_0, d_0) is the initial configuration,
- $F \subseteq Q \times D$ is the set of accepting configurations.

Formally, the only difference between edge-based and state-based XFAs is the type of the update function U. In the latter case, U is a function from only states and domain values to domain values ($U : Q \times D \rightarrow D$). Even so, each model has its advantages, and the algorithms we present for manipulating XFAs employ both models. XFAs are constructed from regular expressions using the edge-based model (Section 6.2). At the same time, combination, matching, and optimization algorithms are more efficient for state-based XFAs (Sections 6.3–6.5). Algorithms for transforming a constructed edge-based XFA to a state-based XFA are straightforward and given later.

According to the definitions, all auxiliary state is maintained in principle using a single (possibly composite) variable, although in practice we can have many distinct variables without any loss of generality. Further, although XFAs are formally defined in terms of abstract data domains, in most cases we can map data domains and update functions to fairly common high-level data types such as bits, counters, and bitmaps. Combination routines then automatically combine them further. Note also that according to the definition, a standard DFA is simply an XFA with a data domain containing only one element.

We argue that the XFA model, whether edge-based or state-based, has fundamental advantages for incorporating variables and in some sense is the most natural extension for adding variables. As with DFAs, transitions are a function of states and input symbols only and are not influenced by variable values. Similarly, variable update functions are a function of states and variable values only (for state-based XFAs). This distinct separation – decoupling transition behavior from variable values – is one of the key enabling features of the model. On the one hand, retaining DFA-like transitions allows us to adapt and use common DFA operations with only slight modification in most cases. In particular, XFAs can be constructed individually and later combined using standard techniques. Matching is also more efficient, since variable values do not need to be queried prior to following transitions. On the other hand, the use of explicit instructions provides fertile ground for systematically applying optimizations and analysis techniques common to compiler construction.

Finally, and also with reference to the suitability of the XFA model, distinctions between edge-based and state-based XFAs are analogous to the distinctions between Mealy and Moore automata, respectively, under certain assumptions. Recall that Mealy and Moore machines extend DFAs with an output alphabet and a mapping function from edges (for Mealy automata) or states (for Moore automata) to the output symbols. If from an automaton's perspective we view update functions as opaque objects, then distinct instructions can be interpreted as distinct output symbols that an XFA emits during traversal[1]. Thus, XFA models share the same characteristics as Mealy and Moore models. For example, to recognize the same language, state-based XFAs require more states than corresponding edge-based XFAs.

6.1.1 Cost Models for XFAs

DFAs are fast and efficient, requiring only a single table lookup per byte along with a test for acceptance at each state. The complexity is thus $O(1)$ per byte and $O(l)$ for a stream of l bytes. Let ϕ be the execution time cost of an individual table lookup. Then, DFA cost is ϕ per byte, or $\phi \cdot l$ for a stream of l bytes.

For XFAs, the cost model is slightly more complex, since instruction execution must also be accounted for. Let $|U(q,a)|$ denote the number of high-level instructions for manipulating variables that are attached to a transition for edge-based XFAs, and $|U(q)|$ the number attached to state q for state-based XFAs. Let $|F(q)|$ denote the number of acceptance conditions to check at state q. Lastly, let $m_i =$max$\{|U(q,\alpha)||q \in Q, \alpha \in \Sigma\}$, and $n_i =$max$\{|U(q)| + F(q)|q \in Q\}$.

With these definitions, we can model the execution time cost of edge-based XFAs as $\phi + \psi \cdot (m_i + F)$ per byte, and the execution time cost of state-based XFAs as $\phi + \psi \cdot n_i$ per byte, where ψ is the cost of executing a single instruction. Note that acceptance conditions are implemented at the high level using simple instructions and are modeled identically as update functions from a cost perspective.

These models assume that state(s) with the maximum number of instructions are exclusively traversed for each byte of the input and thus reflect worst-case behavior. In fact, the average number of executed instructions per byte may be much smaller. Nevertheless, the worst-case cost model given above provides a lower bound.

[1]Equivalency conditions regarding the output symbol on the start state of Moore machines are relevant for XFAs as well [53], but also treated analogously.

6.2 Constructing XFAs

The steps for compiling a regular expression to an XFA are similar to those for constructing a DFA from a regular expression using the standard Thompson construction [108]: parsing the regular expression and constructing a non-deterministic automaton with ϵ-transitions, removing ϵ-transitions, determinizing the states, and minimizing the automaton. For XFAs, we need to provide algorithms corresponding to these four steps. The key difference is the inclusion of the abstract data domain. Parsing is modified to initially populate the data domain and introduce update relations that manipulate values in the domain. Each of the remaining steps is extended to transform these elements appropriately as the transitions and states are determinized and minimized, yielding an edge-based XFA with abstract variable values and update functions. Finally, the last step maps these values and functions to concrete data types such as bits, counters, or their combinations, with edge-based instructions for manipulating them. We illustrate the techniques in this section with a running example that constructs an XFA from the regular expression `/.*ab.*cd/`.

The construction techniques described here add data domain values to transitions between states. Thus, they employ the edge-based XFA model. After construction is complete and an XFA is produced, we may optionally transform the edge-based XFA to a state-based XFA.

It is important to note that all variables are fully specified at construction. Unlike some models, there is no dynamic creation or destruction of variables. For example, signatures of the form `/.*x.{n}y/` need $n+1$ bits to track the n positions (which may themselves include an `x`) between `x` and `y`. Thus, an XFA for this expression would include an $n+1$-bit bitmap; no dynamic variable creation is employed.

6.2.1 Extending Regular Expressions

Transforming a regular expression into an XFA requires striking a balance between using states and transitions on one hand and executing instructions that manipulate variables on the other. At one extreme we can produce a (possibly large) DFA which uses no variables and at the other extreme a (possibly slow) program that does not rely on state information at all. There are regular expressions for which the best XFA lies at one of these extremes. For expressions such as `/.*s/`, where `s` is a string, a simple DFA with no variables is ideal. At the other extreme, the example from Figure 6.2 which recognizes `/.{n}/` gives an XFA that is effectively just a program: there is a single state which does not influence at all how

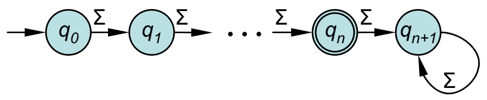


Figure 6.1: The DFA for `/.`$\{n\}$`/` (n arbitrary symbols).

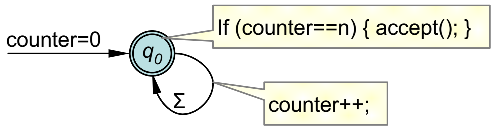


Figure 6.2: An XFA recognizing `/.`$\{n\}$/.

the auxiliary memory is updated or when acceptance happens. Prior to construction, we augment regular expressions with new or re-interpreted operators to control where the resulting XFA lies along this spectrum.

We expand the grammar of regular expressions with an additional operator that introduces data domain values and changes the shape of the resulting XFA. We also re-interpret an operator to introduce data domain values. We call regular expressions with these extensions *domain-augmented regular expressions*. The set of domain-augmented regular expressions (denoted RE_Σ) over an alphabet Σ is recursively defined as follows:

- $\emptyset \in RE_\Sigma$.
- $\epsilon \in RE_\Sigma$.
- $\forall a \in \Sigma, a \in RE_\Sigma$.
- if $E_1 \in RE_\Sigma$ and $E_2 \in RE_\Sigma$, then $E_1|E_2 \in RE_\Sigma$.
- if $E_1 \in RE_\Sigma$ and $E_2 \in RE_\Sigma$, then $E_1 \cdot E_2 \in RE_\Sigma$.
- if $E \in RE_\Sigma$, then $E^\star \in RE_\Sigma$.
- if $E_1 \in \Sigma$ and $E_2 \in \Sigma$, then $E_1 \# E_2 \in RE_\Sigma$. (parallel concatenation)
- if $E \in RE_\Sigma$, integer $n \geq 0$, integer $m \geq 0$, and $n \leq m$, then $E\{n, m\} \in RE_\Sigma$. (integer ranges)
- if $E \in RE_\Sigma$, then $(E) \in RE_\Sigma$.

Next, we define the language corresponding to this augmented grammar. Given an expression $E \in RE_\Sigma$, let $L(E) \in \Sigma^\star$ denote the language corresponding to E. We define $L(E)$ recursively as follows:

- $L(\emptyset) = \emptyset$.
- $L(\epsilon) = \{\epsilon\}$.

ID	Signature
2667	`.*[/\\]ping\.asp`
3194	`.*bat"#.*&`
2411	`.*\nDESCRIBE\s#[^\n]{300}`
3466	`.*\nAuthorization:\s*Basic\s#[^\n]{200}`
1735	`(.*new XMLHttpRequest#.*file://)|(.*file://#.*new XMLHttpRequest)`

Table 6.1: Snort signatures for HTTP traffic annotated with the parallel concatenation operator ‘#’.

- $\forall a \in \Sigma, L(a) = \{a\}$.
- if $E = E_1 | E_2$, then $L(E) = L(E_1) \cup L(E_2)$.
- if $E = E_1 \cdot E_2$, then $L(E) = \{x_1x_2 | x_1 \in L(E_1)$ and $x_2 \in L(E_2)\}$.
- if $E = E'^{\star}$, then $L(E) = \{x_1x_2 \cdots x_k | 0 \leq i \leq k, x_i \in E'\}$.
- if $E = E_1 \# E_2$, then $L(E) = \{x_1x_2 | x_1 \in L(E_1)$ and $x_2 \in L(E_2)\}$.
- if $E = E'\{n, m\}$, then $L(E) = \{x_1x_2 \cdots x_k | 0 \leq n \leq k \leq m$ for $1 \leq i \leq k, x_i \in E'\}$.

In the definitions above, we introduce two operators: parallel concatenation ($E_1 \# E_2$), and integer range constraints ($E\{n, m\}$). Parallel concatenation is semantically equivalent to standard concatenation, and integer ranges produce a subset of the strings produced by $E^{\star}$. All other operators have behavior identical to their standard regular expression counterparts. Thus, neither operator changes the underlying language.

During NXFA construction, these two operators introduce values from the data domain into the nondeterministic XFA. Integer ranges, a form of syntactic sugar, are already present in the signatures (although we do re-interpret them to introduce a counter). Thus we only need to decide where to use the parallel concatenation operator ‘#’. Currently, this is a partly manual step.

The previous chapter showed that string-based automata are unambiguous and therefore not vulnerable to state-space explosion. The purpose of the parallel concatenation operator is to break up a regular expression, or parts of one, into string-like subexpressions that are individually suitable for string matching. For example, we annotate `/`$.*\mathrm{s}_1.*\mathrm{s}_2$`/`, where s_1 and s_2 are strings, as `/`$.*\mathrm{s}_1\#.*\mathrm{s}_2$`/`. Put another way, we add the ‘#’ operator right before subexpressions such as ‘`.*`’ and `[^\n]{k}` that repeat characters from either the whole input alphabet or a large subset thereof. Table 6.1 shows examples of regular expressions representing actual NIDS signatures from our test set annotated with ‘#’. Note that for signature 2667 we have not used any parallel concatenation as the expression is sufficiently string-like already. This signature will be compiled to an XFA without any added variables. For signature 3466, we do not insert a ‘#’ in front of `\s*` because the character class `\s` contains few characters (the white spaces). For signatures such as 1735 which is a union of sub-expressions we apply the rules for inserting ‘#’ to the sub-expressions of the union separately.

6.2.2 Parse Trees and NXFAs

The first step in XFA construction is to parse the domain-augmented regular expression. This step is straightforward and only minimally changed from that for standard regular expressions. Parallel concatenation is left-associative and has precedence immediately below standard concatenation. Integer range constraints have the same precedence as Kleene Closure (E^*). All other operators are unchanged.

After parsing, the next step is to construct a non-deterministic XFA with epsilon transitions via a bottom-up traversal of the parse tree. Parallel concatenation and integer range operators introduce formal versions of a bit and a counter to the constructed NXFA, respectively. Other operators produce NXFA constructs structurally identical to those used for standard regular expressions, modified to incorporate identity update functions along their edges.

We give recursive construction definitions for each operator below. Figure 6.3 shows two generic nondeterministic XFAs, NXFA 1 and NXFA 2, upon which the recursive definitions are built. In the descriptions, NXFA 1 and NXFA 2 corresponding to arbitrary regular expressions E_1 and E_2, respectively. Without loss of generality, we assume that there is a single start state and a single accepting state in each automaton, although there may be multiple initial and accepting domain values. NXFA 1 has initial configuration domain values d_{i1} and accepting domain values d_{f1}. Initial and accepting domain values for NXFA 2 are defined analogously. We further assume that domain values between the two NXFAs are distinct, since if not domains can simply be remapped to new values if necessary. Unless otherwise noted, when NXFA 1 and NXFA 2 are combined using one of the operators below, the resulting domain D is simply the union of the respective domains D_1 and D_2.

1. **basis** ($R = \alpha \in \Sigma$). The basis case applies to occurrences of individual symbols[2] in the alphabet parsed from the regular expression. In this case, a two-state automaton is created with a single

[2]character classes such as `[aeiou]` or `[^\n]` also fall into this category.

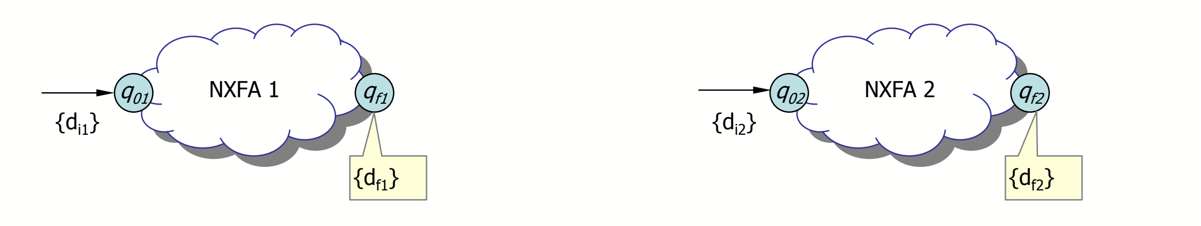


Figure 6.3: Source NXFAs used for recursive construction definitions for concatenation, union, and so forth. The NXFAs have starting configurations and accepting configurations as shown.

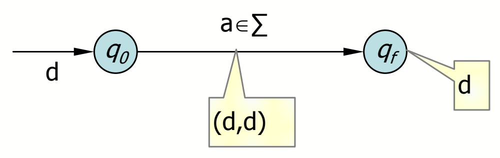


Figure 6.4: The basis case for NXFA construction. A single transition with an attached symbol has one domain value d. Its starting configuration is (q_0, d) and its accepting configuration is (q_f, d).

transition between them labeled with the observed symbol, as shown in Figure 6.4. This automaton has a domain $D = \{d\}$ consisting of a single element, which serves as the initial domain value and the accepting domain value. The identity function (d, d) attached to the edge links the initial and accepting configuration values.

2. **concatenation** $(E_1 \cdot E_2)$. Figure 6.5 depicts the construction process corresponding to concatenation. From the state perspective, as with standard concatenation an ϵ-transition links the accepting state of the left-hand automaton to the starting state of the right most automaton. For domain values, the initial domain values are the same as the left-hand automaton, and the accepting domain values are the same as for the right-hand automaton. The update relation attached to the ϵ-transition maps accepting domain values from the left-hand side to initial domain values in the right-hand side.

3. **alternation** $(E_1|E_2)$. The construction corresponding to alternation is given in Figure 6.6 and also mirrors the DFA construction definition. Here, we arbitrarily pick the initial domain values of the first alternative (NXFA 1) to be initial domain values for the new NXFA. In the ϵ-transitions from the start state q_0 we supply update relations that map them as appropriate to the correct alternative. We

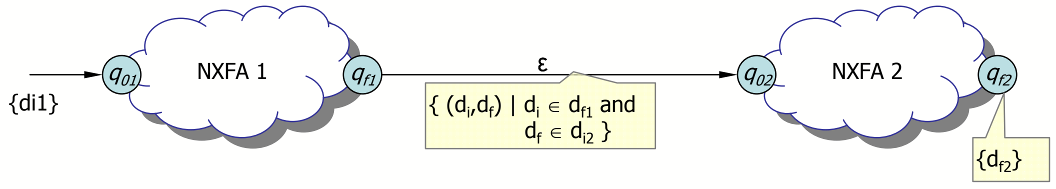


Figure 6.5: NXFA construction for concatenation. The epsilon transition links accepting domain values on the left hand side to initial domain values on the right hand side.

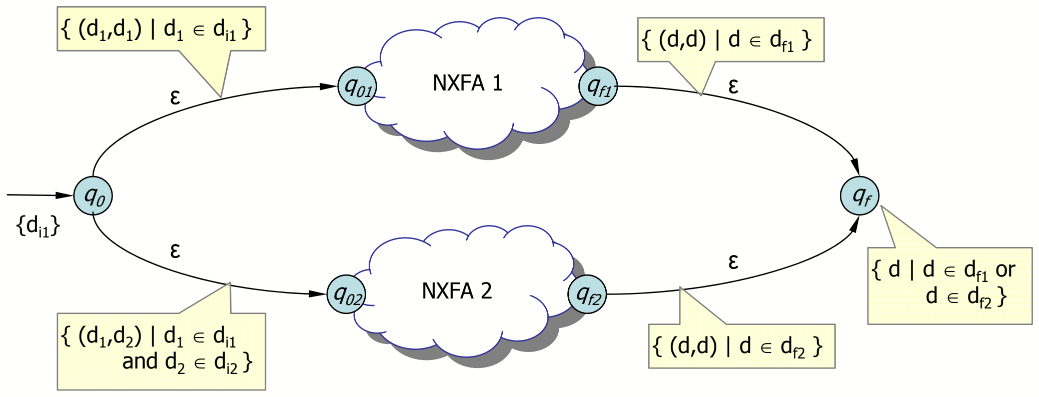
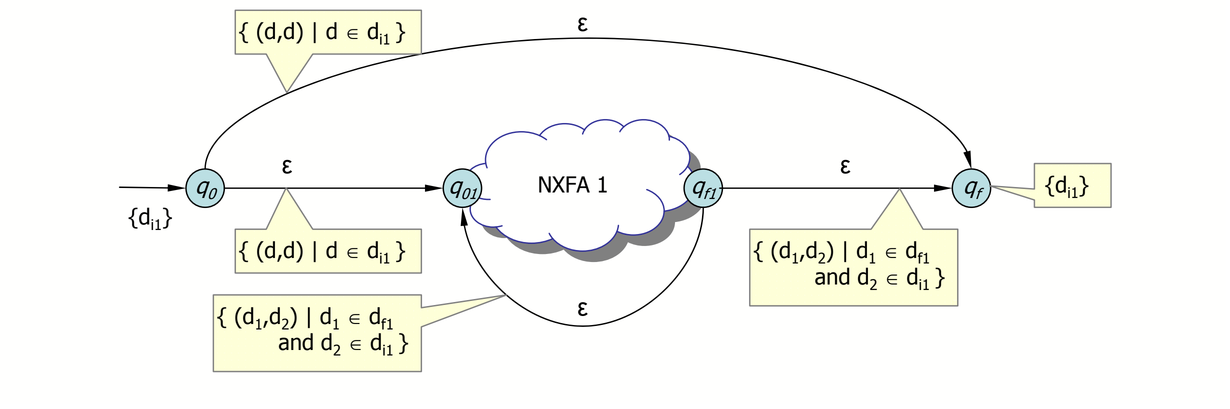


Figure 6.6: NXFA construction for the alternation operation. The accepting domain values are the union of the accepting values in either source NXFA.

attach identity relations to the ϵ-transitions connecting the accepting states in the source automata to the new accepting state q_f. Finally, the accepting domain values for q_f are the union of the accepting domain values in states q_{f1} and q_{f2}.

4. **Kleene closure** ($E_1^\star$). Kleene closure is depicted in Figure 6.7. In this construction, the new start state has initial domain values $\{d_{i1}\}$ corresponding to the initial domain values for state q_{01}. The accepting domain values are also the set $\{d_{i1}\}$, which follows since the evaluation of NXFA 1 is optional. Update relations attached to ϵ-transitions serve the purpose of mapping accepting domain values (from q_{f1}) to initial domain values.

5. **parallel concatenation** ($E_1 \# E_2$). Parallel concatenation has the same semantics as standard concatenation but is structurally distinct. Whereas standard concatenation as described above connects



{ (d,d) | d ∈ d_i1 }
ε
ε
NXFA 1
ε
{d_i1}
q_0
q_01
q_f1
q_f
{d_i1}
{ (d,d) | d ∈ d_i1 }
{ (d_1,d_2) | d_1 ∈ d_f1 and d_2 ∈ d_i1 }
ε
{ (d_1,d_2) | d_1 ∈ d_f1 and d_2 ∈ d_i1 }

Figure 6.7: Kleene Closure construction for NXFAs.

two NXFAs together in sequence by linking the accepting states of one to the starting states of another, parallel concatenation uses ϵ-transitions to link all states in the first NXFA to the start state of the second NXFA. Thus, parallel concatenation is structurally a superset of standard concatenation. We use appropriately constructed update relations to ensure that semantics remain the same as for standard concatenation despite the structural change.

Figure 6.8 shows the construction. As the figure shows, ϵ-transitions are added from every state in NXFA 1 to the start state of NXFA 2. The initial conditions are those of NXFA 1, and the accepting conditions are the same as for NXFA 2. Update relations are added as follows:

(a) For each of the added ϵ-transitions, we attach update relations that map domain values in D (recall, D is the union of domains D_1 and D_2) to the starting domain values of D_1.

(b) To each transition in NXFA 2 we add tuples in the update relation mapping NXFA 1's starting domain values to themselves.

(c) For each ϵ-transition out of an accepting state in NXFA 1 into a starting state of NXFA 2, we add tuples to the update relation that map accepting domain values in NXFA 1 to starting values in NXFA 2.

Together, these update relations ensure that an overall accepting configuration can be reached if and only if an accepting configuration for NXFA 1 is first observed followed by an accepting configuration for NXFA 2. Specifically, the proper initial domain value for NXFA 2 is only set by following

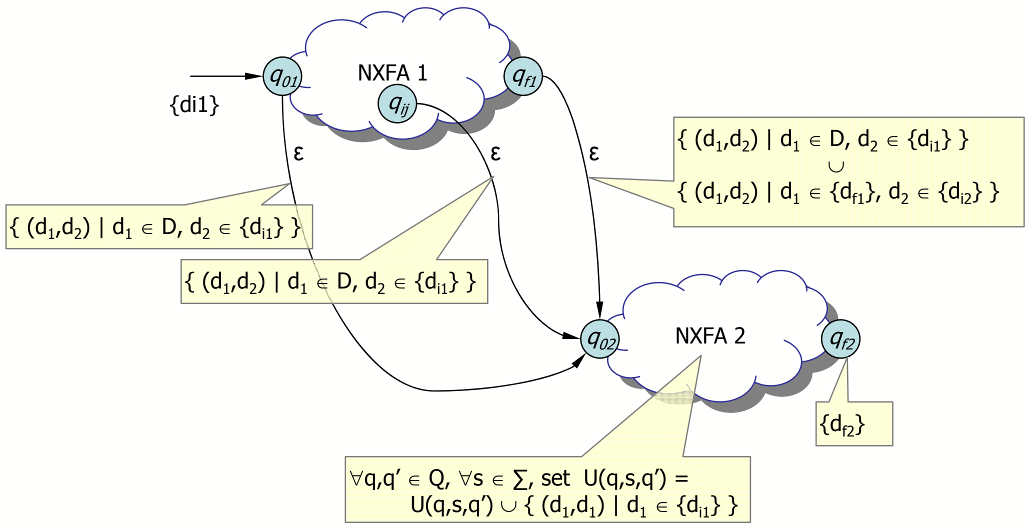

NXFA 1
NXFA 2
{di1}
ε
ε
ε
{ (d1,d2) | d1 ∈ D, d2 ∈ {di1} }
{ (d1,d2) | d1 ∈ D, d2 ∈ {di1} }
{ (d1,d2) | d1 ∈ D, d2 ∈ {di1} } ∪ { (d1,d2) | d1 ∈ {df1}, d2 ∈ {di2} }
{df2}
∀q,q' ∈ Q, ∀s ∈ ∑, set U(q,s,q') = U(q,s,q') ∪ { (d1,d1) | d1 ∈ {di1} }

Figure 6.8: Parallel Concatenation of two NXFAs. The update relation in the ϵ-transition from q_{f1} to q_{02} contains tuples that link the domain values in the accepting configuration of NXFA 1 to the initial configuration of NXFA 2.

the ϵ-transition from an accepting configuration for NXFA 1 (Item 5c). All other ϵ-transitions from NXFA 1 to NXFA 2 move the domain to a "poison" value with regard to NXFA 2 (Items 5a and 5b: an initial domain value for NXFA 1) that can only be reset by first accepting NXFA 1.

The effect of this construction is that both NXFAs can be matched "in parallel" by introducing the formal version of a bit that uses the data domain to enforce the sequencing requirements of concatenation. When R_2 has a leading `.*` as is the case for strings, this construction eliminates the ambiguity associated with the `.*` at the beginning of R_2 that is present with standard concatenation.

6. **Integer Ranges** ($E_1\{n,m\}$). Integer ranges of the form $E_1\{n,m\}$ are used to indicate that E_1 must occur at least n times and no more than m times in immediate succession, where $0 \le n, n \le m$, and $m \le \infty$.[3] Integer ranges already exist as a form of syntactic sugar for regular expressions. For XFAs, we re-interpret them to perform the repetition counting in the domain values rather than in explicit states. When ambiguous counting states are fully replaced by counting domain values, the associated ambiguity can be reduced or even eliminated (ref. Figures 5.1 and 5.6).

[3]Note $n < \infty$ or else the resulting automaton is not finite.

Figure 6.9 illustrates the construction process for integer ranges. Since counting is performed in the domain, the construction is structurally identical to the Kleene closure construction[4]. To properly count repetitions in the domain, we must construct a domain D that is the cross product of the domain D_1 for NXFA 1 with the counting range $0..m$ (we consider the case where $m = \infty$ below). This follows from the fact that NXFA 1 needs to manipulate its own domain values in each iteration of the counting. By crossing the domain values we can ensure that NXFA 1 manipulates its domain independently while at the same time the counting state is also preserved. Converting to the new domain is a two step process:

(a) First, we introduce a simple mapping, termed $M(\cdot,\cdot)$ to translate from the ordered pair produced by the cross product to a single domain value in the new domain D. Since both D_1 and the counting range are finite, $M(\cdot,\cdot)$ is also finite and can be pre-computed.

(b) Second, we translate all update relations in NXFA 1 to the new domain as shown in the figure. Since the update relations in NXFA 1 must exist for each counting value, we compute the cross product (and translate using $M(\cdot,\cdot)$) for each possible counting value.

Conceptually, NXFA 1 manipulates the first component of the ordered pair produced by the cross product, whereas the counting domain manipulates the second component. The mapping function M simply translates the ordered pair to a new single value and exists as a notational and implementation convenience. In the figure, the ϵ-transitions $q_0 \xrightarrow{\epsilon} q_{01}$ and $q_0 \xrightarrow{\epsilon} q_f$ map the initial domain values for NXFA 1 associated with a counter value of 0 to their translated counterparts in D. In the transitions $q_{f1} \xrightarrow{\epsilon} q_{01}$ and $q_{f1} \xrightarrow{\epsilon} q_f$, the counting domain values are incremented and the accepting domain values for NXFA 1 are mapped back to the initial domain values (as with Kleene closure). Finally, the accepting conditions for the entire automaton are the cross of the acceptable counting range $n..m$ with the initial domain values for NXFA 1.

Figure 6.9 shows that counters are incremented up to a value `max`, which is computed as follows: if $m = \infty$, then `max` $= m$, otherwise `max` $= m + 1$. Further, when $m = \infty$, the accepting counting values are fixed at n rather than running over the range $n..m$ in the accepting conditions attached to state q_f. Finally, we also add update relations to the $q_{f1} \xrightarrow{\epsilon} q_f$ transition that map accepting domain values in NXFA 1 to the starting domain values in NXFA 1 (and thus the accepting conditions of the

[4] In fact, Kleene closure is a special case of integer ranges in which $n = 0$ and $m = \infty$.

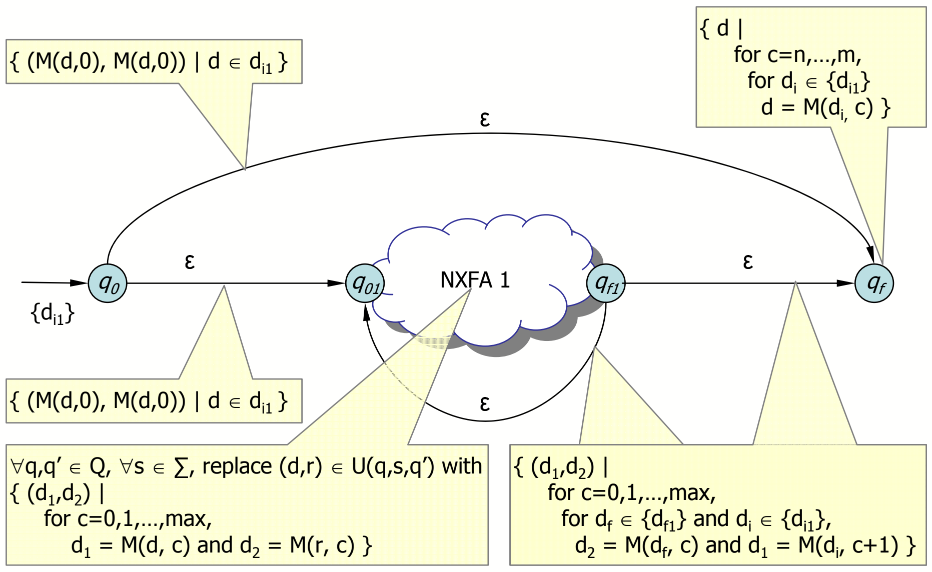

{ (M(d,0), M(d,0)) | d ∈ d_i1 }
{ d | for c=n,…,m, for d_i ∈ {d_i1} d = M(d_i, c) }
ε
ε
ε
ε
q_0
q_01
NXFA 1
q_f1
q_f
{d_i1}
{ (M(d,0), M(d,0)) | d ∈ d_i1 }
∀q,q′ ∈ Q, ∀s ∈ ∑, replace (d,r) ∈ U(q,s,q′) with { (d_1,d_2) | for c=0,1,…,max, d_1 = M(d, c) and d_2 = M(r, c) }
{ (d_1,d_2) | for c=0,1,…,max, for d_f ∈ {d_f1} and d_i ∈ {d_i1}, d_2 = M(d_f, c) and d_1 = M(d_i, c+1) }

Figure 6.9: Integer range construction for NXFAs, giving the construction for $E_1\{n, m\}$. The value `max` is set depending on the values of n and m, and $M(\cdot, \cdot)$ maps to a new domain incorporating counter values.

newly constructed NXFA), crossed with the value `max`. Formally, these update relations are expressed as follows:

$$\{(M(d_1, max), M(d_2, max))|d_1 \in d_{f1}, d_2 \in d_{i1}\}$$

These changes ensure that when $m = \infty$, the automaton will accept whenever at least n repetitions of NXFA 1 have occurred.

For the expression `/.*ab.*cd/` in the running example, we insert a parallel concatenation operator after the `ab` sub-expression to yield `/.*ab#.*cd/`. Figure 6.10 depicts the corresponding parse tree, and Figure 6.11 shows the NXFA with ϵ-transitions constructed from a traversal of the parse tree. Looking at the NXFA, one can observe that the only way to reach the accepting configuration $(s, d) = (15, 5)$ is by first reaching state 7 with a domain value of 2, and then following the ϵ-transition out to state 10, which nondeterministically moves the current domain value to 3.

6.2.3 Determinization and Minimization

Once a non-deterministic XFA has been constructed, the next steps involve ϵ-elimination, state determinization and data-domain determinization, and minimization of states and data domains. These steps are

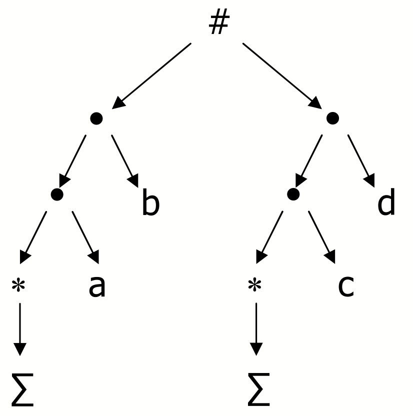


Figure 6.10: Parse tree produced from the augmented regular expression `/.*ab#.*cd/`.

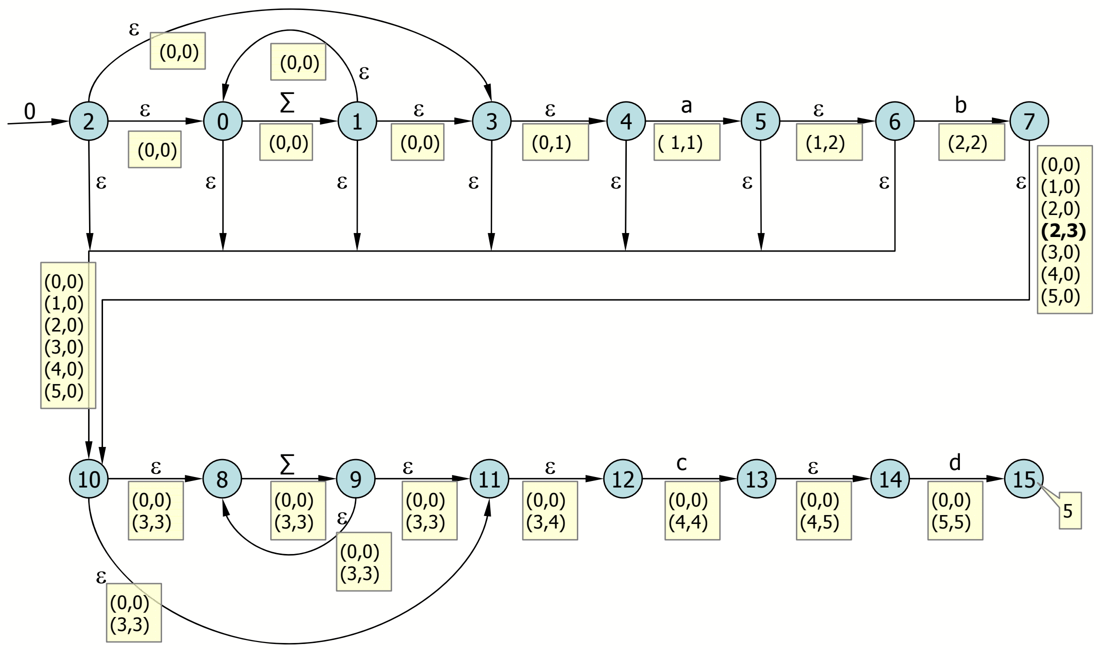


Figure 6.11: Basic NXFA for `/.*ab#.*cd/` constructed from the parse tree in Figure 6.10.

similar to those used for determinizing standard NFAs, but they introduce the additional complication of needing to appropriately manipulate the relations (and functions) that update the data domain.

Epsilon Elimination. The first step in the determinization process is to eliminate epsilon edges; *i.e.*, edges with no associated symbol, also known as ϵ-transitions. We first introduce definitions for ϵ-reachability, relational composition, and ϵ-closure, and we extend update relations to paths and sets of paths. With these definitions, we then describe how to eliminate ϵ-transitions. For each of the definitions below, assume we have constructed a nondeterministic XFA with ϵ-transitions $\mathcal{X} = (Q, D, \Sigma, \delta, U, QD_0, F)$.

Definition 6.4 A state $q' \in Q$ is ϵ*-reachable* from another state $q \in Q$ if there is a path from q to q' consisting exclusively of ϵ-transitions.

Definition 6.5 Given two relations $U_1 \subseteq D \times D$ and $U_2 \subseteq D \times D$, the *relational composition* $U_2 \circ U_1$ is given as follows: for $d_1, d_2 \in D$, $(d_1, d_2) \in U_2 \circ U_1$ if and only if $\exists d \in D$ such that $(d_1, d) \in U_1$ and $(d, d_2) \in U_2$.

Note that Definition 6.5 is the standard definition for relational composition. Next, we extend update relations to paths and sets of paths. Consider a path $\pi = q_1 \xrightarrow{\epsilon} q_2 \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} q_{k+1}$ from q_1 to q_{k+1} consisting only of ϵ-transitions. Then, the update relation $U(\pi)$ corresponding to path π is $U(q_k, \epsilon, q_{k+1}) \circ U(q_{k-1}, \epsilon, q_k) \circ \cdots \circ U(q_2, \epsilon, q_3) \circ U(q_1, \epsilon, q_2)$. Generalizing, the update relation corresponding to a *set* of paths $\{\pi_1, \cdots \pi_k\}$ is given by $\bigcup_{i=1}^{k} U(\pi_i)$.

Finally, we define ϵ-closure for nondeterministic XFAs.

Definition 6.6 The ϵ*-closure* of state q is the set of tuples constructed as follows: $(q', U') \in \epsilon$-closure(q) if and only if there exists an ϵ-reachable path from q to q', where $U' = \bigcup_{\pi \in paths(q,q')} U(\pi)$, and $paths(q, q')$ is the set of ϵ-paths from q to q'.

Once the ϵ-closure(q) has been computed for all states $q \in Q$, we can construct a non-deterministic XFA $\mathcal{X}' = (Q, D, \Sigma, \delta', U', QD'_0, F')$ with ϵ-transitions removed as follows:

- For $a \in \Sigma$, a transition $(q, a, q') \in \delta'$ if and only if
  - $(q, a, q') \in \delta$, or
  - $\exists q_1 \in Q$ such that q' is ϵ-reachable from q_1 and $(q, a, q_1) \in \delta$.

```
EliminateEpsilon($Q, D, \Sigma, \delta, U_\delta, QD_0, F$):
1  $\delta' \leftarrow \emptyset$;
2  $U'_\delta \leftarrow \emptyset$;
3  foreach $(q_i, s, q_f) \in \delta \cap Q \times \Sigma \times Q$ do
4      foreach $(d_i, d_f) \in U_\delta(q_i, s, q_f)$ do
5          foreach $(q_{reachable}, d_{reachable}) \in$ ComputeEpsilonReachable $(q_f, d_f)$ do
6              $\delta' \leftarrow \delta' \cup \{(q_i, s, q_{reachable})\}$;
7              $U'_\delta \leftarrow U'_\delta \cup \{((q_i, s, q_{reachable}), (d_i, d_{reachable}))\}$;
8  $QD'_0 \leftarrow \emptyset$;
9  foreach $(q_0, d_0) \in QD_0$ do
10     $QD'_0 \leftarrow QD'_0 \cup$ ComputeEpsilonReachable $(q_0, d_0)$;
11 return $(Q, D, \Sigma, \delta', U'_\delta, QD'_0, F)$;

ComputeEpsilonReachable($q, d$)          :
12 $Result \leftarrow \{(q, d)\}$;
13 foreach $(q_i, d_i) \in Result$ do
14     foreach $q_f \in \{q | (q_i, \epsilon, q) \in \delta\}$ do
15         $Result \leftarrow Result \cup \{q_f\} \times \{d_f | (d_i, d_f) \in U_\delta(q_i, \epsilon, q_f)\}$;
16 return $Result$;
```

Algorithm 6.1: ϵ-elimination for NXFAs.

- $U'(q, a, q')$ is equal to the following relation:

$$U(q, a, q') \cup \bigcup_{(q_1, U_1) \in \epsilon-closure(q')} U_1 \circ U(q, a, q').$$

- Let $G \subseteq QD_0$ be the set of configurations such that $\forall g \in G$, $\exists (q, d) \in F$ in which (q, d) is ϵ-reachable from g. Then, $F' = F \cup G$.

- For $(q', d') \in Q \times D$, $(q', d') \in QD'_0$ if and only if (q', d') is ϵ-reachable from some initial configuration $(q, d) \in QD_0$.

The last item above states that the start configurations QD'_0 in $\mathcal{X}'$ are precisely those configurations that are ϵ-reachable from the start configurations in $\mathcal{X}$. Algorithm 6.1 gives the procedure for removing ϵ-transitions. Informally, the algorithm extends standard ϵ-elimination by composing update functions along chains of "collapsed" ϵ-transitions from the original NXFA and places these new relations into the appropriate transition in the ϵ-free NXFA. These composed functions keep track of the possible changes to the data domain value along the collapsed edges. In the running example, Figure 6.12 shows the NXFA from Figure 6.11 after epsilon elimination has completed. After running Algorithm 6.1, we then remove states from the NXFA that are not accepting and have no paths leading to accepting states (*i.e.*, dead states). Figure 6.13 shows the result of this process. Note that in Figure 6.13, there are four starting configurations: states 0, 8, and 12, each paired with domain value 0, and state 4 paired with domain value 1.

Determinization. Epsilon elimination produces an ϵ-free automaton that is nondeterministic in both its states and its domain. We perform determinization in two stages, determinizing transitions first and update relations (yielding update functions) second. Both steps extend the classic subset construction [53] for computing deterministic automata. Let $\mathcal{X} = (Q, D, \Sigma, \delta, U, QD_0, F)$ be a nondeterministic XFA with ϵ-transitions removed, and let $\mathcal{X}' = (Q', D', \Sigma, \delta', U', QD'_0, F')$ be the eventual state-determinized XFA, where q_0 is a starting state in QD'_0. As with the subset construction, states in $\mathcal{X}'$ correspond to sets of states in $\mathcal{X}$. Consider first a state determinization scheme constructing $\mathcal{X}'$ as follows:

- $q'_0 = \{q_1, q_2, ..., q_k\}$, where for each state $q_i \in \{q_1, q_2, ..., q_k\}$, q_i is a starting state in $\mathcal{X}$.
- $D' = D$.
- $d'_0 = \bigcup\{d_i\}$, where $\{d_i\}$ are the sets of starting domain values in $\mathcal{X}$.
- Let $q' = \{q_i, ..., q_m\}$ and $p' = \{p_j, ..., p_n\}$. $q', p' \in Q'$ if and only if $\exists \alpha \in \Sigma$ such that $\forall q_k \in q', q_k \xrightarrow{\alpha} p_l$ (or q_k has no transition on α)[5], where $p_l \in p'$ and p' contains only states that are the targets of transitions on α from states in q'.
- $\{p_i, ..., p_n\} \in \delta'(\{q_i, ..., q_m\}, \alpha)$ if and only if $\exists q_k \in \{q_i, ..., q_m\}, p_l \in \{p_j, ..., p_n\}$ such that $p_l \in \delta(q_k, \alpha)$.
- $U'(\{q_i, ..., q_m\}, \alpha, \{p_1, ..., p_n\}) = \bigcup_{k=i}^{m} \bigcup_{l=j}^{n} U(q_k, \alpha, p_l)$.
- $(\{q_i, ..., q_m\}, d) \in F'$ if and only if $\exists q_k \in \{q_i, ..., q_m\}$ such that $(q_k, d) \in F$.

This formulation computes the subset construction independently of the domain values and constructs update relations using a simple set union operation. Unfortunately, this construction is flawed, and $\mathcal{X}'$ may not accept the same language as $\mathcal{X}$. To see why, suppose that in $\mathcal{X}'$ we have a state $\{q_1, q_2\}$ and there are transitions $q_0 \xrightarrow{\alpha} q_1$ and $q_0 \xrightarrow{\alpha} q_2$ in $\mathcal{X}$. In this construction, the update relation U associated with $(\{q_0\}, \alpha, \{q_1, q_2\})$ is the union of the update relations U_1 and U_2 associated with (q_0, α, q_1) and (q_0, α, q_2), respectively. Assume that $(d_0, d_1) \in U_1$ and $(d_0, d_2) \in U_2$ but $(d_0, d_2) \notin U_1$ and $(d_0, d_1) \notin U_2$. Since

[5]for at least one $q_k \in \{q_i, ..., q_m\}$, we must have $q_k \xrightarrow{\alpha} p_l$

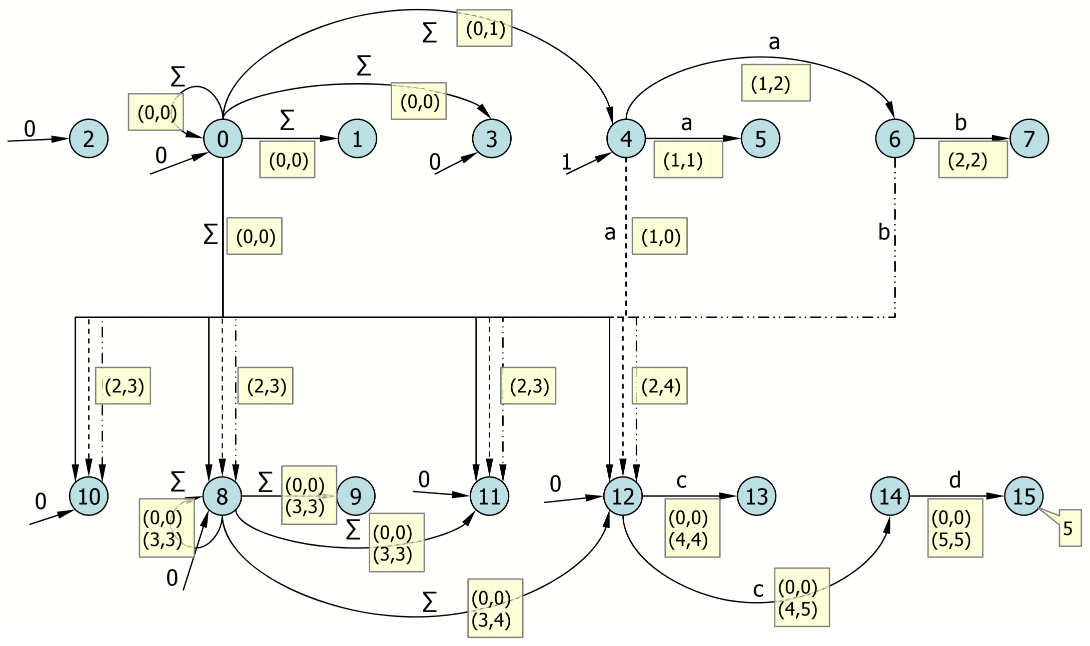


Figure 6.12: The NXFA from Figure 6.11 corresponding to `/.*ab#.*cd/`, after ϵ-transitions have been removed.

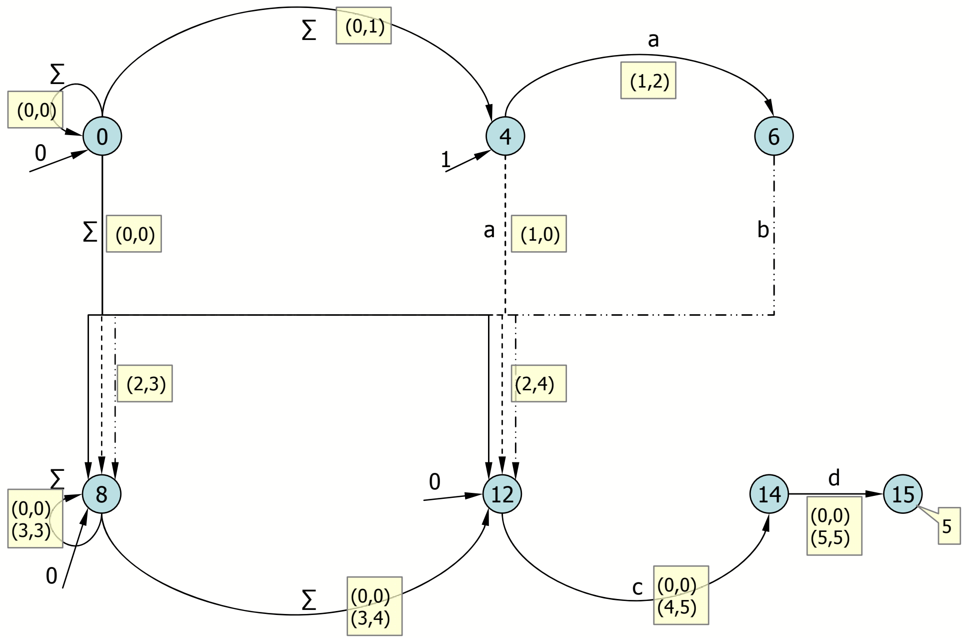


Figure 6.13: The ϵ-free NXFA corresponding to `/.*ab#.*cd/` after dead states have been removed.

(state,domain)→D	(state,domain)→D	(state,domain)→D
⟨0,0⟩→0	⟨14,2⟩→14	⟨4,0⟩→28
⟨0,1⟩→1	⟨14,3⟩→15	⟨4,0⟩→29
⟨0,2⟩→2	⟨14,4⟩→16	⟨8,0⟩→30
⟨0,3⟩→3	⟨14,5⟩→17	⟨8,0⟩→31
⟨0,4⟩→4	⟨12,0⟩→18	⟨8,0⟩→32
⟨0,5⟩→5	⟨12,0⟩→19	⟨8,0⟩→33
⟨15,0⟩→6	⟨12,0⟩→20	⟨8,0⟩→34
⟨15,1⟩→7	⟨12,0⟩→21	⟨8,0⟩→35
⟨15,2⟩→8	⟨12,0⟩→22	⟨6,0⟩→36
⟨15,3⟩→9	⟨12,0⟩→23	⟨6,0⟩→37
⟨15,4⟩→10	⟨4,0⟩→24	⟨6,0⟩→38
⟨15,5⟩→11	⟨4,0⟩→25	⟨6,0⟩→39
⟨14,0⟩→12	⟨4,0⟩→26	⟨6,0⟩→40
⟨14,1⟩→13	⟨4,0⟩→27	⟨6,0⟩→41

Table 6.2: The mapping from ⟨state, domain⟩ pairs in $Q \times D$ to the new domain used in Figure 6.14.

$(d_0, d_1) \in U'$ and $(d_0, d_2) \in U'$, the configurations $(\{q_1, q_2\}, d_1)$ and $(\{q_1, q_2\}, d_2)$ are both reachable in $\mathcal{X}'$ from the configuration $\{\{q_0\}, d_0\}$. However, configurations (q_1, d_2) and (q_2, d_1) are not reachable from (q_0, d_0) in $\mathcal{X}$. This can lead to extra accepting paths in $\mathcal{X}'$.

One way to remedy this is for each state $q \in Q$ to have its own copy of the relevant update relations to manipulate. Thus, we set $D' = Q \times D$ and update the construction above as follows:

- Let $q'_0 = \{q_1, q_2, ..., q_k\}$ as above. Then, $QD'_0 = \{(q'_0, \langle q_i, d_i\rangle) | q_i \in \{q_1, q_2, ..., q_k\}$ and $(q_i, d_i) \in QD_0\}$.
- For $q' = \{q_i, ..., q_m\}$ and $p' = \{p_j, ..., p_n\}$ in Q', $(\langle q', d_1\rangle, \langle p', d_2\rangle \in I'(q, \alpha, p')$ if and only if $(d_1, d_2) \in U(q_r, \alpha, p_s)$ for some $q_r \in \{q_1, q_2, ..., q_k\}$ and $p_s \in \{p_j, ..., p_n\}$.
- $(q', \langle q_r, d\rangle) \in F'$ if and only if $(q_r, d) \in F$ for some $q_r \in \{q_1, q_2, ..., q_k\}$.

By transforming to domain $Q \times D$, we preserve the dependencies in the update relations that existed in XFA $\mathcal{X}$. We give the algorithm for performing state determinization in Algorithm 6.2. In the running example, the nondeterministic XFA in Figure 6.13 has states $Q = \{0, 4, 6, 8, 12, 14, 15\}$ and domain $D = \{0, 1, 2, 3, 4, 5\}$. Thus, the size of domain $Q \times D$ is 42. Figure 6.14 shows the state-deterministic XFA that results from applying Algorithm 6.2 to the NXFA in Figure 6.13. For clarity, we map each domain element in $Q \times D$ to a distinct integer using the mapping shown in Table 6.2.

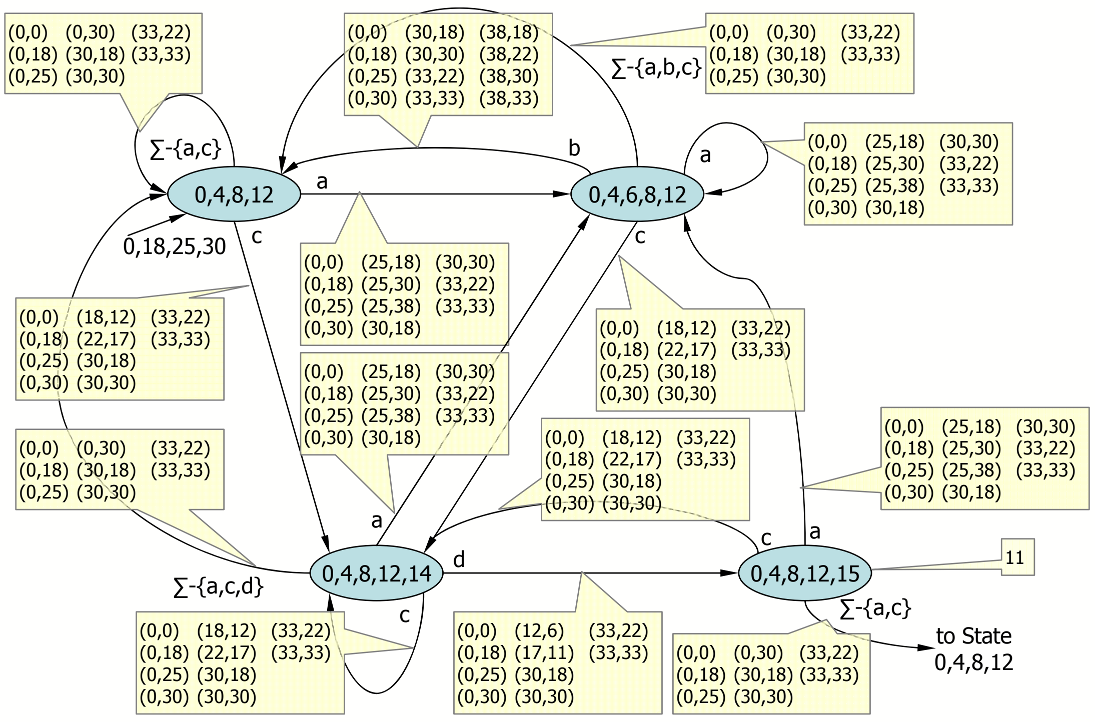


Figure 6.14: The state-deterministic XFA for `/.*ab#.*cd/` constructed with Algorithm 6.2. Update relations over the domain $Q \times D$ arise from the mapping in Table 6.2.

```
DeterminizeTransitions($Q, D, \Sigma, \delta, U_\delta, QD_0, F$):
1  $D' \leftarrow Q \times D$;
   // Data value in new NXFA = configuration in old
2  $D'_0 \leftarrow QD_0$ ;
   // New states are sets of old states
3  $q'_0 \leftarrow \{q_0 | \exists d_0 \in D.(q_0, d_0) \in QD_0\}$ ;
4  $Q' \leftarrow \{q'_0\}$;
5  $\delta' \leftarrow \emptyset$;
6  $U'_\delta \leftarrow \emptyset$;
7  foreach $q'_i \in Q'$ do
8      foreach $s \in \Sigma$ do
9          $q'_f \leftarrow \{q_f | \exists q_i \in q'_i.(q_i, s, q_f) \in \delta\}$;
10         $Q' \leftarrow Q' \cup \{q'_f\}$ ;                    // Accum. reachable sets of old states
           // New states have 1 trans. per symbol
11         $\delta' \leftarrow \delta' \cup \{(q'_i, s, q'_f)\}$ ;
12         $U \leftarrow \{((q_i, d_i), (q_f, d_f)) | q_i \in q'_i \wedge q_f \in q'_f \wedge$
13                                 $(d_i, d_f) \in U_\delta(q_i, s, q_f)\}$;
           // Update relations preserve semantics
14         $U'_\delta \leftarrow U'_\delta \cup \{(q'_i, s, q'_f)\} \times U$ ;
15 $F' \leftarrow \{(q', (q, d)) | q' \in Q' \wedge q \in q' \wedge (q, d) \in F\}$ ;
16 return $(Q', D', \Sigma, \delta', U'_\delta, \{q'_0\} \times D'_0, F')$;
```

Algorithm 6.2: Algorithm for determinizing transitions.

```
DeterminizeData($Q, D, \Sigma, \delta, U_\delta, \{q_0\} \times D_0, F$):
1  $d'_0 \leftarrow D_0$ ;                                   // New data values = sets of old data values
2  $D' \leftarrow \{d'_0\}$;
   // $QD$ accumulates all reachable configurations
3  $QD \leftarrow \{(q_0, d'_0)\}$ ;
4  $U'_\delta \leftarrow \emptyset$;
5  foreach $(q_i, d'_i) \in QD$ do
6      foreach $s \in \Sigma$ do
7          $q_f \leftarrow \delta(q_i, s)$;
8          $d'_f \leftarrow \{d_f | \exists d_i \in d'_i.(d_i, d_f) \in U_\delta(q_i, s, q_f)\}$;
9          $D' \leftarrow D' \cup \{d'_f\}$ ;                 // Accum. reachable sets of old values
10         $QD \leftarrow QD \cup \{(q_f, d'_f)\}$;
           // Build deterministic update functions
11         $U'_\delta \leftarrow U'_\delta \cup \{((q_i, s), (d'_i, d'_f))\}$ ;
12 $F' \leftarrow \{(q, d') | (q, d') \in QD \wedge \exists d \in d'.(q, d) \in F\}$;
13 return $((Q, D', \Sigma, \delta, U'_\delta, (q_0, d'_0), F'), QD)$;
```

Algorithm 6.3: Algorithm for determinizing NXFA data domains.

New Domain Value	Domain Value Sets from Domain Determ.
0	{0,18,25,30}
1	{0,18,25,30,38}
2	{0,12,18,25,30}
3	{0,6,18,25,30}
4	{0,18,22,25,30,33}
5	{0,18,22,25,30,33,38}
6	{0,12,17,18,22,25,30,33}
7	{0,6,11,18,22,25,30,33}

(a) Sets of domain values→new domain value

New State ID	State Sets from State Determin.
P	{0,4,8,12}
Q	{0,4,6,8,12}
S	{0,4,8,12,14}
T	{0,4,8,12,15}

(b) Sets of states→new state ID

Table 6.3: The simplifying mapping from sets of domains and sets of states in Figure 6.15 to single domain values and single state IDs used to produce Figure 6.16.

Data domain determinization is the second half of the determinization process. Algorithm 6.3 accepts as input a state-deterministic XFA and produces an XFA as output that is both state- and data-deterministic. Data-determinization applies the same notion of subset construction for determinizing states to update relations in the data domain. Let $\mathcal{X} = (Q, D, \Sigma, \delta, U, QD_0, F)$ be the state-deterministic input XFA and $\mathcal{X}' = (Q, D', \Sigma, \delta, U', QD'_0, F')$ be the corresponding deterministic XFA. In principle, for the new domain $D' = 2^D$. However, the algorithm uses a worklist that adds new domain values to D' as necessary so that only the domain values that are reachable from the starting configuration are included. Algorithm 6.3 also outputs the set QD (Line 13) that contains all configurations that are reachable from the starting configuration. This set is used at a later stage.

Figure 6.15 shows the deterministic XFA resulting from applying Algorithm 6.3 to the state-deterministic XFA in Figure 6.14. Analogous to state determinization, update relations are now update *functions* defined over domain values in D' that correspond to sets of domain values in D. Note that although the update functions associated with transitions U' are not defined on the entire data domain D', they are defined on all data values $d' \in D'$ that can occur in any state q. Admittedly, the use of sets of domain values and sets of states makes the figure hard to read and obscures the structure of the update functions. In Figure 6.16 we show the same XFA as Figure 6.15 with the sets of domains and sets of states replaced by the correspondence given in Table 6.3. The Domain value set $\{0, 18, 25, 30\}$ (the initial domain value) in Figure 6.15 corresponds to value 0 in Figure 6.16, and so forth.

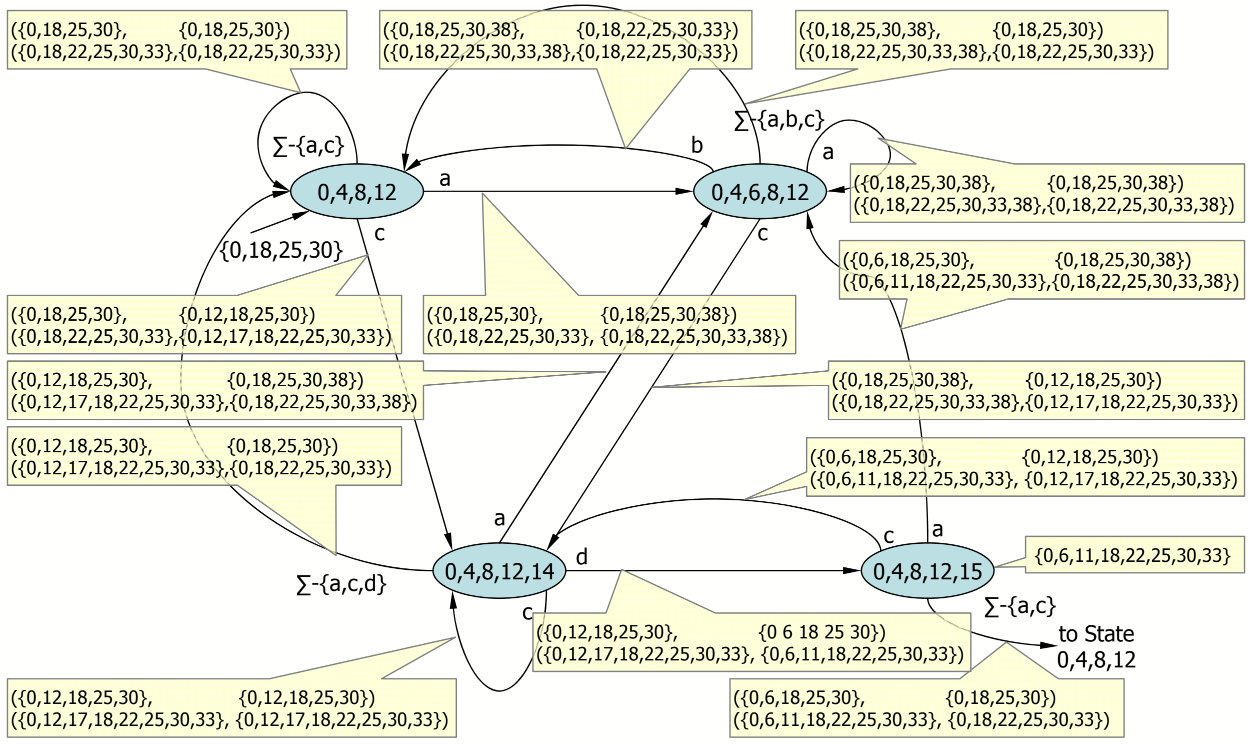


Figure 6.15: The state- and domain-deterministic XFA for `/.*ab#.*cd/` after domain determinization has completed. Each domain value in the resulting XFA corresponds to a set of domain values in the input state-deterministic XFA.

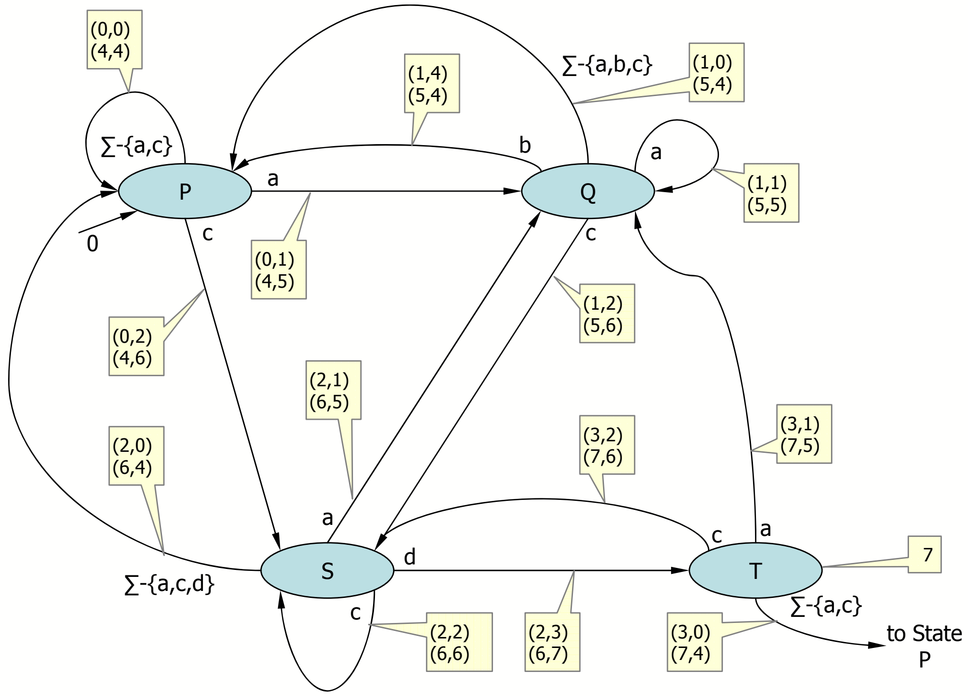


Figure 6.16: The state- and domain-deterministic XFA for `/.*ab#.*cd/` with state and domain values replaced according to the map in Table 6.3.

Minimization. Once the XFA has been determinized, the next step is to perform minimization. The minimization process for XFAs is also split into two parts: data domain minimization and state minimization. Each component of minimization employs the kernel of the standard Hopcroft-Ullman minimization algorithm [53] for finding equivalent data domain elements and states, respectively.

Consider a DFA $M = (Q, \Sigma, \delta, q_0, F)$. Minimization of M finds the coarsest equivalence relation $R \subseteq Q \times Q$ that satisfies the following conditions:

- $(q_1, q_2) \in R$ implies that $q_1 \in F \leftrightarrow q_2 \in F$.
- $(q_1, q_2) \in R$ implies that $\forall a \in \Sigma, (\delta(q_1, a), \delta(q_2, a)) \in R$.

Let $\mathcal{R}$ be the set of all relations satisfying the conditions above. We say that $R_1 \in \mathcal{R}$ is coarser than $R_2 \in \mathcal{R}$ if and only if $R_2 \subseteq R_1$. Algorithms for computing the coarsest equivalence relation are given in [53]. Once the coarsest equivalence relation R is computed, all the states in Q in the same equivalence class can be merged.

Consider a deterministic XFA $\mathcal{X} = (Q, D, \Sigma, \delta, U, (q_0, d_0), F)$. Analogous to the above, assume we find the coarsest equivalence relation $R \subseteq (Q \times D) \times (Q \times D)$ that satisfies the following conditions:

- $((q, d), (q', d')) \in R$ implies that $q = Q'$ (configurations with different states are never equivalent).
- $((q, d), (q, d')) \in R$ implies that $(q, d) \in F \leftrightarrow (q, d') \in F)$.
- $((q, d), (q, d')) \in R$ implies that $\forall a \in \Sigma\ ((q', U(q, a, q')(d)\ ), \quad (q', U(q, a, q')(d'))\ ) \in R$ (note: $q' = \delta(q, a)$).

In this case, if for a state q and two data values d_1 and d_2 we have $((q, d_1), (q, d_2)) \in R$, then configurations (q, d_1) and (q, d_2) can be merged. Recall that during determinization, the data domain was expanded from D to $Q \times D$. The equivalence relation R above allows us to merge data values for each state $q \in Q$. Algorithm 6.4 gives the procedure for minimizing the data domain.

Minimizing states follows a similar formulation. For a state $q \in Q$, we define the set $A \subseteq D$ such that $\forall d \in D$, if $(q, d) \in F$, then $d \in A$. Next, recall that $U(q, \alpha)$ is the set of tuples comprising the update function attached to transition α leading out of state q. We define $U_\Sigma = \{(\alpha, U(q, \alpha)) | \alpha \in \Sigma\}$. U_Σ is the set of all update functions out of q, grouped by alphabet symbol. We then define $\texttt{effects}(q) = (A, U_\Sigma)$. Intuitively, $\texttt{effects}(q)$ captures the update functions and accepting data values associated with state q.

We find the coarsest equivalence relation $R \subseteq Q \times Q$ that satisfies the following conditions:

```
**ReduceDataDomain(**$(Q, D, \Sigma, \delta, U_\delta, (q_0, d_0), F), QD$**)**:
/* $P_Q$ Holds for each state the finest partition of possible data values known to
   be necessary.                                                              */
$P_Q \leftarrow \emptyset$ $Workset \leftarrow \emptyset$;
**foreach** $q \in Q$ **do**
    $partition \leftarrow \{\{d|(q,d) \in F\}, \{d|(q,d) \in QD - F\}\} - \{\emptyset\}$;
    $P_Q \leftarrow P_Q \cup \{(q, partition)\}$;
    **if** $|partition| > 1$ **then** $Workset \leftarrow Workset \cup \{q\}$;
**while** $Workset \neq \emptyset$ **do**
    $q_f \leftarrow Workset[0]$;
    $Workset \leftarrow Workset - \{q_f\}$;
    **foreach** $(q_i, s) \in \{(q,z)|((q,z), q_f) \in \delta\}$ **do**
        **if** $\exists d'_i \in P_Q(q_i).\nexists d'_f \in P_Q(q_f).\forall d_i \in d'_i.U_\delta(q_i,s)(d_i) \in d'_f$ **then**
            $newpartition \leftarrow \emptyset$;
            **foreach** $d'_i \in P_Q(q_i)$ **do**
                **foreach** $d'_f \in P_Q(q_f)$ **do**
                    $d' \leftarrow \{d_i|d_i \in d'_i \wedge U_\delta(q_i,s)(d_i) \in d'_f\}$;
                    **if** $d' \neq \emptyset$ **then** $newpartition \leftarrow newpartition \cup \{d'\}$;
            $P_Q \leftarrow P_Q - \{(q_i, P_Q(q_i))\} \cup \{(q_i, newpartition)\}$;
            $Workset \leftarrow Workset \cup \{q_i\}$;
$QD' \leftarrow \{(q,d')|q \in Q \wedge d' \in P_Q(q)\}$;
$D' \leftarrow \{d'|\exists q \in Q.(q,d') \in QD'\}$;
$d'_0 \leftarrow \{d'|(q_0,d') \in QD' \wedge d_0 \in d'\}[0]$;
$F' \leftarrow \{(q,d')|(q,d') \in QD' \wedge (q, d'[0]) \in F\}$;
$U'_\delta \leftarrow \{((q_i,s),(d'_i,d'_f))|(q_i,d'_i) \in QD' \wedge (\delta(q_i,s),d'_f) \in QD' \wedge U_\delta(q_i,s)(d'_i[0]) \in d'_f\}$;
**return** $(Q, \Sigma, \delta, F, D', q_0, d'_0, U'_\delta, A', QD')$;
```

Algorithm 6.4: Reducing the size of the data domain by combining the equivalent configurations.

```
    ReduceStates($Q, D, \Sigma, \delta, U_\delta, (q_0, d_0), F$):
    /* $P$ Holds the finest partition of states known to be necessary.    */
1   $P \leftarrow \emptyset$;
2   foreach $q \in Q$ do
3       if $\exists EC \in P$ such that GetStateEffect($q$)=GetStateEffect($EC[0]$) then
4           $P \leftarrow P - \{EC\} \cup \{EC \cup \{q\}\}$;
5       else $P \leftarrow P \cup \{\{q\}\}$;
6   while $\exists EC \in P$ such that $\exists q_1 \in EC, q_2 \in EC, s \in \Sigma . \nexists EC_f \in P . \{\delta(q_1, s), \delta(q_2, s)\} \subseteq EC_f$ do
7       $NewECs \leftarrow \emptyset$;
8       foreach $q \in EC$ do
9           if $\exists EC' \in NewECs$ such that $\forall s \in \Sigma . \exists EC_f \in P . \{\delta(q, s), \delta(EC'[0], s)\} \subseteq EC_f$ then
10              $NewECs \leftarrow NewECs - \{EC'\} \cup \{EC' \cup \{q\}\}$;
11          else $NewECs \leftarrow NewECs \cup \{\{q\}\}$;
12      $P \leftarrow P - \{EC\} \cup NewECs$;
13  $Q' \leftarrow P$;
14  $\delta' \leftarrow \{((q'_i, s), q'_f) | q'_i \in Q' \wedge q'_f \in Q' \wedge \delta(q'_i[0], s) \in q'_f\}$;
15  $q'_0 \leftarrow \{q' | q' \in Q' \wedge q_0 \in q'\}[0]$;
16  $U'_\delta \leftarrow \{((q'_i, s), U_\delta(q'_i[0], s)) | q'_i \in Q' \wedge s \in \Sigma\}$;
17  $F' \leftarrow \{(q', d) | q' \in Q' \wedge (q'[0], d) \in F\}$;
18  return $(Q', D, \Sigma, \delta', U'_\delta, (q'_0, d_0), F')$;

    GetStateEffect($q \in Q$)                :
19  $A \leftarrow \{d | (q, d) \in F\}$;
20  $U_\Sigma \leftarrow \{(s, U_\delta(q, s)) | s \in \Sigma\}$;
21  return $(A, U_\Sigma)$;
```

Algorithm 6.5: Reducing the size state space by combining the equivalent states.

- $(q_1, q_2) \in R$ implies that `effects`(q_1) = `effects`(q_2);
- $(q_1, q_2) \in R$ implies that $\forall \alpha \in \Sigma, (\delta(q_1, \alpha), \delta(q_2, \alpha)) \in R$.

Then, with the equivalence relation R computed, two states in the same equivalence class can be merged into one. The procedure for minimizing states is given in Algorithm 6.5. Minimization has no effect on the XFA in the running example.

6.2.4 Finding Efficient Implementations

For performance reasons, an XFA does not explicitly use U and F to manipulate and test data values. Instead elements of U and F are appropriately mapped to high-level data types that are more efficiently computed and more easily managed. Thus, the last step in the compilation process is to map abstract data domain operations to efficient, concrete instructions for manipulating data values. Intuitively, this step replaces domain values and update functions with higher-level variables and instructions that manipulate them.

To perform the mapping, the compiler makes use of generic data-type templates that formally relate domain values and update functions to high-level types. When a suitable mapping is found, as described below, the compiler substitutes the appropriate components of the template into the XFA. We call these templates Efficiently Implementable Data Domains (EIDDs) and define them as follows:

Definition 6.7 An *efficiently implementable data domain (EIDD)* is a 6-tuple (D, d_0, E, U_E, C, A_C), where

- D is the finite set of values in the data domain,
- d_0 is the initial data domain value,
- E is a set of symbolic names for efficient-to-compute update functions,
- $U_E : E \rightarrow D^D$ is a mapping from these names to fully defined (deterministic) update functions on D that can be associated with XFA transitions,
- C is a set of symbolic names for efficient-to-check acceptance conditions,
- and $A_C : C \rightarrow 2^D$ is a mapping from these names to acceptance conditions that can be associated with XFA states.

To illustrate, Figure 6.17 shows an EIDD for a simple bit formatted according to the grammar we have defined in our implementation. In the figure, the domain has two elements and an initial value of 0. E holds the high level names of the operations applied to the bits: `noop`, `set`, and `reset`. `Ue` maps these names to the corresponding update functions. Note that update functions are total over the domain. For example, the update function for `set`–$\{$`(0,1),(1,1)`$\}$–implements the total function $0 \rightarrow 1, 1 \rightarrow 1$ over the domain $D = \{0, 1\}$. Continuing, `C` and `Ac` specify acceptance condition names and relate them to specific domain values. The `unconditional` acceptance condition will accept on any value in D, whereas `conditional_hi` accepts only if the data value is 1 during matching. Finally, although not part of the EIDD itself, items `CGe` and `CGa`[6] map the high-level operations to the low-level instructions that are executed. In this case, the instructions are in the format used by our prototype interpreter.

Although Definition 6.7 specifies that the update functions in E must be "efficient to compute" and the acceptance conditions from C "efficient to check", we cannot give a single definition for what it means to

[6] `CGe` and `CGa` stand for "Code Generation–edges" and "Code Generation–acceptance", respectively.

```
eidd_t  bit_full  =  {

  D    = {0..1  },
  d    = 0,
  E    = {noop,  set,  reset},
  Ue   = {
             (noop,       {(0,0),  (1,1)}),
             (set,        {(0,1),  (1,1)}),
             (reset,      {0,0),   (1,0)})
         },
  C    = {  nonaccepting,  conditional,  unconditional  },

  Ac   = {
             (  nonaccepting,      {}  ),
             (  conditional_lo,    {0}  ),
             (  conditional_hi,    {1}  ),
             (  unconditional,     {0,1})
         },

  CGe  = {
             (noop,     ``''),
             (set,      ``[%i,bit,set()]''),
             (reset,    ``[%i,bit,reset()]'')
         },

  CGa  = {
             (nonaccepting,       ``''),
             (conditional_lo,     ``[%i,bit,test,0,([%i,accept])]''),
             (conditional_hi,     ``[%i,bit,test,1,([%i,accept])]''),
             (unconditional,      ``[%i,accept]'')
         }
 };
```

Figure 6.17: an EIDD specifying the mapping for a simple bit.

be efficient as this depends strongly on the platform XFAs run on. For example, on some platforms we may define efficiency as the use of five or fewer machine code instructions to perform the update or to check the condition, on others we may use different definitions.

Algorithm 6.6 presents the basic procedure for mapping to EIDDs. Note the use of two unconventional notations. First, for some sets A we use $A[0]$ to denote an arbitrary element of the set; the correctness of the algorithm does not depend on which element gets chosen and whenever we use this notation we know that $A \neq \emptyset$. Second, the conditions of some while loops and if-statements are of the form $\exists a \in A$, and in these cases we assume that inside the body of the loop or the if-statement a is bound to one of the elements of A. As above, it is not important for the correctness of the algorithm which element is chosen.

Given an XFA $(Q, D, \Sigma, \delta, U_\delta, (q_0, d_0), F)$ and an EIDD $(D', d'_0, E, U_E, C, A_C)$, this algorithm computes a mapping that consists of three components:

1. $D'_{QD} : Q \times D \rightarrow D'$ maps all configurations from QD (the set of all reachable configurations, produced by the Algorithm 6.3–Data Determinization) to values in the new data domain D' specified by the EIDD,
2. $E_\delta : Q \times \Sigma \rightarrow E$ maps all transitions to efficient update functions, and
3. $C_Q : Q \rightarrow C$ maps all states to efficient acceptance conditions.

The mapping D'_{QD} for data domain values is from QD to D' rather than from D to D', thus mapping only those configurations that are reachable to the EIDD's domain. Note that the EIDD domain D' can be much smaller than D (and typically is) because different values of D can map to the same value of D' without affecting semantics, as long as there is no state where multiple values from D can occur simultaneously. In the running example, for instance, the data domain size is reduced from 8 to 2. Below are the conditions that a valid mapping (D'_{QD}, E_δ, C_Q) satisfies to ensure that it preserves the semantics of the XFA.

$$\begin{array}{rll}
\forall q \in Q, \exists c \in C & s.t. & (q,c) \in C_Q \\
\forall (q,d) \in QD, \exists d' \in D' & s.t. & ((q,d),d') \in D'_{QD} \\
\forall (q_i,s) \in Q \times \Sigma, \exists e \in E & s.t. & ((q_i,s),e) \in E_\delta \\
 & & D'_{QD}(q_0,d_0) = d'_0 \\
\forall (q,d) \in F & & D'_{QD}(q,d) \in A_C(C_Q(q)) \\
\forall (q,d) \in QD - F & & D'_{QD}(q,d) \notin A_C(C_Q(q)) \\
\forall ((q_i,d_i),s) \in QD \times \Sigma & & D'_{QD}(\delta(q_i,s), U_\delta(q_i,s)(d_i)) = \\
 & & \quad U_E(E_\delta(q_i,s))(D'_{QD}(q_i,d_i))
\end{array}$$

Algorithm 6.6 finds a mapping if one exists or declares failure by returning an empty mapping. The loop at line 11 expands D'_{QD} when it finds situations in which a transition $t = \delta(q_i, s)$ has already been mapped in E_δ and a configuration (q_i, d_i) of the source state for t also has been mapped in D'_{QD}, but the configuration resulting from applying the update function for t, $(q_f, d_f) = (\delta(q_i, s), U_\delta(q_i, s)(d_i))$, has not been mapped in D'_{QD} yet. In this case (q_f, d_f) can be mapped to the value from D' which is the output of $E_\delta(q_i, s)$ for input $D'_{QD}(q_i, d_i)$. E_δ is expanded by choosing an unmapped transition on line 15 and by trying all possible mappings for it in the loop on line 16. Some mappings for edges can lead to conflicting mappings for certain configurations; the `FindInconsistency` function detects such mappings.

The recursive calls in `FindValidMapping` continue until all transitions are labeled with a symbolic update function. When this happens (or even earlier) the loop at line 11 will assign a mapping in D'_{QD} to all configurations that are reachable from (q_0, d_0). Thus if the function ever returns on line 14, all transitions from δ have a mapping in E_δ, and all the configurations from QD have a mapping in D'_{QD}. Since the loop on line 16 tries all possible update functions, we know that if there is a mapping from transitions to update functions that leads to a valid mapping of configurations to values from D', the algorithm will find it. Otherwise, it will signal failure by returning $(\emptyset, \emptyset)$.

In the running example, Algorithm 6.3 returns a state deterministic XFA with the following set of reachable configurations (using the mapping in Table 6.3):

$$QD = \{(P,0),(P,4),(Q,1),(Q,5),(S,2),(S,6),(T,3),(T,7)\}.$$

```
MapXFAtoEIDD($(Q, D, \Sigma, \delta, U_\delta, (q_0, d_0), F), QD, EIDD$):
1  $(D', d'_0, E, U_E, C, A_C) \leftarrow EIDD$;
2  $E_\delta \leftarrow \emptyset$;
3  $D'_{QD} \leftarrow \{((q_0, d_0), d'_0)\}$;
4  $(D'_{QD}, E_\delta) \leftarrow$FindValidMapping $(D'_{QD}, E_\delta)$;
5  if $(D'_{QD}, E_\delta) = (\emptyset, \emptyset)$ then return $(\emptyset, \emptyset, \emptyset)$ ;
6  $C_Q \leftarrow \emptyset$;
7  foreach $q \in Q$ do
8      $c_{correct} \leftarrow \{c \in C | \forall((q, d), d') \in D'_{QD}.d' \in A_C(c) \iff (q, d) \in F\}[0]$;
9      $C_Q \leftarrow C_Q \cup \{(q, c_{correct})\}$;
10 return $(D'_{QD}, E_\delta, C_Q)$;

FindValidMapping($D'_{QD}, E_\delta$)                                   :
11 while $\exists(s, ((q_i, d_i), d'_i)) \in \Sigma \times D'_{QD}.\nexists d'_f \in D'.((\delta(q_i, s), U_\delta(q_i, s)(d_i)), d'_f) \in D'_{QD} \wedge E_\delta(q_i, s) \in E$ do
12     $D'_{QD} \leftarrow D'_{QD} \cup \{((\delta(q_i, s), U_\delta(q_i, s)(d_i)), U_E(E_\delta(q_i, s))(d'_i))\}$;
13 if FindInconsistency $(D'_{QD}, E_\delta)$ then return $(\emptyset, \emptyset)$;
14 if $|E_\delta| = |Q| \cdot |\Sigma|$ then return $(D'_{QD}, E_\delta)$ ;
15 $trans \leftarrow \{(q_i, s) | (q_i, s) \in Q \times \Sigma \wedge \nexists e \in E.((q_i, s), e) \in E_\delta\}[0]$;
16 foreach $e \in E$ do
17     $Result \leftarrow$FindValidMapping $(D'_{QD}, E_\delta \cup \{(trans, e)\})$;
18     if $Result \neq (\emptyset, \emptyset)$ then return $Result$;
19 return $(\emptyset, \emptyset)$ ;

FindInconsistency($D'_{QD}, E_\delta$)                                  :
20 foreach $(q_i, s, d_i) \in Q \times \Sigma \times D$ do
21     if $\exists d'_i \in D', d'_f \in D'.((q_i, d_i), d'_i) \in D'_{QD} \wedge ((\delta(q_i, s), U_\delta(q_i, s)(d_i)), d'_f) \in D'_{QD}$ then
22         if $\exists e \in E.((q_i, s), e) \in E_\delta \wedge (d'_i, d'_f) \notin U_E(e)$ then
23             return true ;
24 foreach $q \in Q$ do
25     if $\forall c \in C.\exists((q, d), d') \in D'_{QD}.\neg d' \in A_C(c) \iff (q, d) \in F$ then return true ;
26 return false ;
```

Algorithm 6.6: Basic algorithm for finding a mapping of an XFA to a given EIDD.

With the XFA in Figure 6.16 as input along with QD given above and the EIDD in Figure 6.17, Algorithm 6.6 produces the following mappings:

$D'_{QD}(Q \times D \to D')$:

$$\begin{array}{cccc} (P,0) \to 0 & (Q,1) \to 0 & (S,2) \to 0 & (T,3) \to 0 \\ (P,4) \to 1 & (Q,5) \to 1 & (S,6) \to 1 & (T,7) \to 1 \end{array}$$

$E_\delta(Q \times \Sigma \to E)$:

$(P, a) \to$ `set`
all other $Q \times \Sigma \to$ `noop`

$C_Q(Q \to C)$:

$P \to$ nonaccepting
$Q \to$ nonaccepting
$S \to$ nonaccepting
$T \to$ conditional_hi

Figure 6.18 shows the constructed XFA with domain values replaced by the D'_{QD} mapping above. Figure 6.19 shows fully constructed XFA with high-level variables after all mapping substitutions have been performed ("C"-like instructions are used rather than those used by the prototype interpreter).

In principle, the use of EIDD templates is just an optimization. We could alternatively provide a fully generic template and let a matching algorithm construct the high-level types without any guidance (or restriction) from the template. But in practice, the matching algorithm as presented has $O(|E|^{|\delta|})$ worst-case complexity and thus would not scale well. One improvement, which cuts down unnecessary exploration, is to greedily pick the transitions for which the number of possible symbolic functions that can be mapped to without leading to inconsistencies is minimal. We also perform pre-computation to rule out symbolic functions that cannot map to given transitions because of mismatches in the number of input values mapped to an output value. These optimizations are sound; neither of them can cause the algorithm to miss an existing solution.

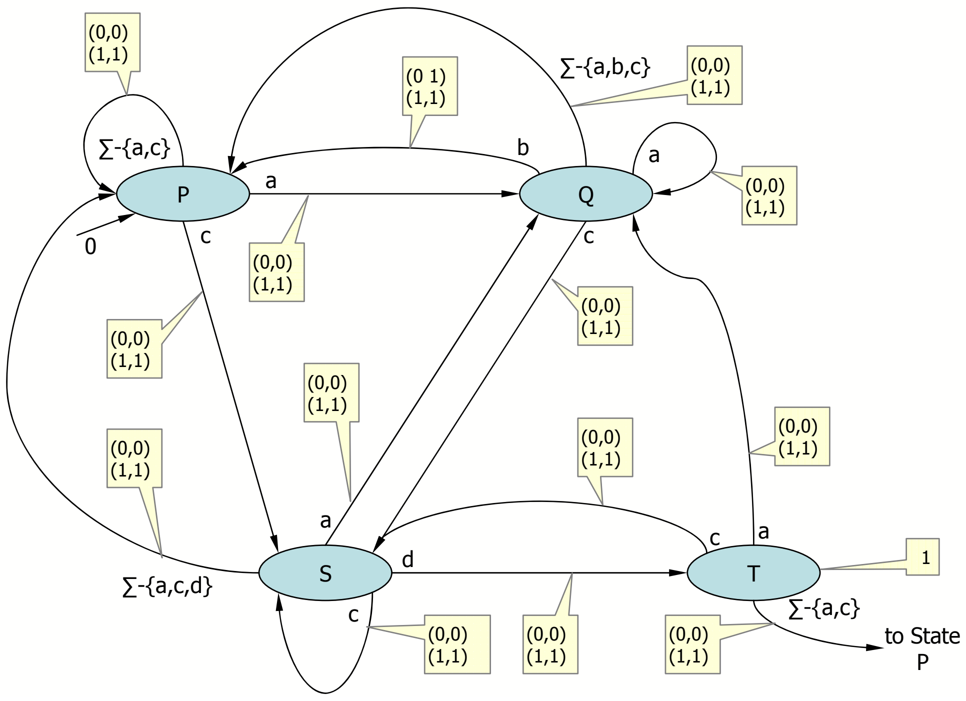


Figure 6.18: The XFA in Figure 6.16 with domain values replaced by a successful mapping to D' in the EIDD (D'_{QD} contains the mapping).

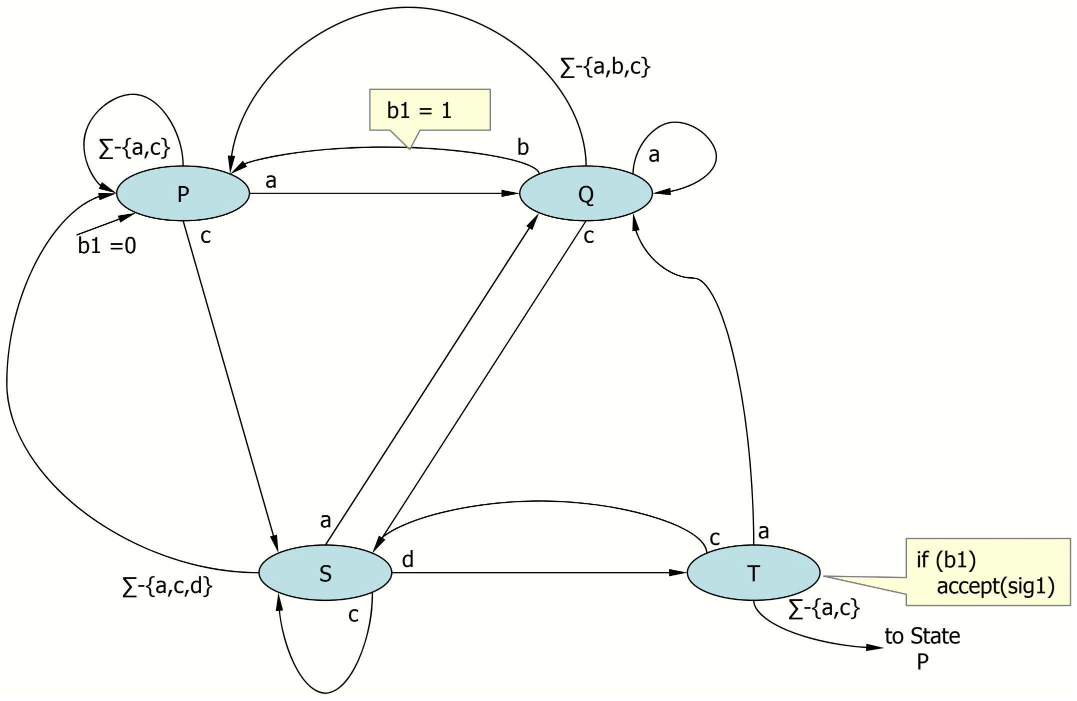
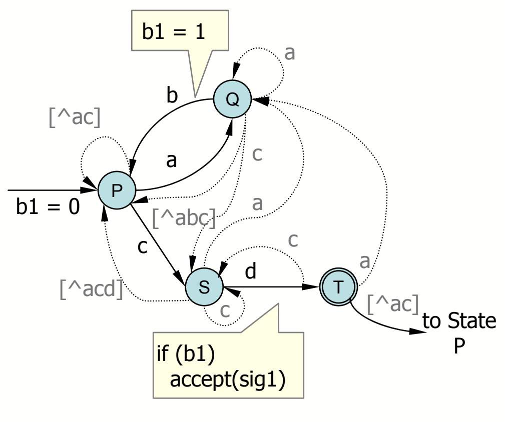
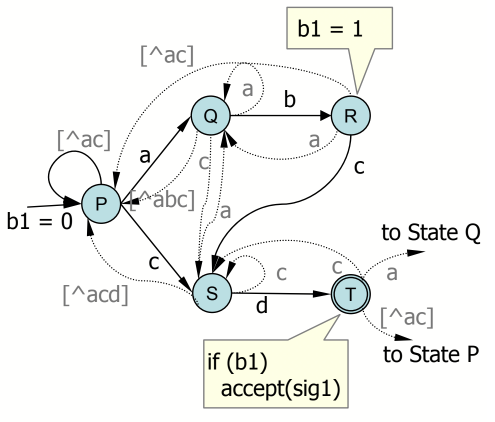


Figure 6.19: The fully constructed XFA with high-level mappings.



(a) edge-based XFA



(b) state-based XFA

Figure 6.20: Edge-based and State-based XFAs corresponding to the augmented regular expression `/.*ab#.*cd/`.

6.2.5 Edge-based to State-based XFA Transformations

The construction process as described produces edge-based XFAs from regular expressions. Although edge-based XFAs have fewer automaton states than state-based XFAs, they are operationally inefficient. For example, instructions on edges is cumbersome both for human analysis and for maintaining in code. Algorithms for combining, matching, and optimizing XFAs are also more involved. Fortunately, unlike construction, these algorithms operate on high-level auxiliary variables rather than low-level data domains. Further, conversion to state-based XFAs is straightforward. We sketch the procedure briefly: for every state S, we create a copy of S (along with its outgoing transitions) for each incoming transition to S that has a distinct set of instructions on that transition[7]. We then move these instructions to the corresponding copies of S and retarget the incoming transitions appropriately. Figure 6.20 shows edge-based and state-based XFAs corresponding to the regular expression `/.*ab.*cd/`, which after being annotated, has the form `/.*ab#.*cd/`. In Figure 6.20a, state P has an incoming transition from state Q on symbol `b` with a bit set instruction. State P is copied, renamed to state R, and the bit set instruction is attached to it. We employ the state-based XFA model for all the remaining operations we describe in this chapter.

[7]If all incoming transitions to S have the same instructions attached to them, no replication is necessary.

6.3 Combining XFAs

Although state space blowup occurs when DFAs are combined, recall that the fault lies in the “shape” of the source automata and in their violation of the conditions given in Chapter 5, not in the combination process itself. XFA combination is a straightforward extension to DFA combination. This is a consequence of the XFA model: transitions are fully deterministic in states and input only, and variables are separable [52] up to renaming.

Algorithm 6.7 gives the procedure for combining two XFAs. Note that this algorithm is virtually identical to Algorithm 4.1 given in Chapter 4 for combining DFAs. The chief difference is that instead of appending output symbols, we now append instructions to combined states.

Here, lines 15 and 16 add instructions to combined states from their original counterparts. For combined states $q = \langle s, t\rangle$, we copy the instructions from s and t into q. As before, correctness follows from the fact that entering state q is equivalent to entering states s and t simultaneously, so that instructions in both s and t need to be executed. Figure 6.22 shows the results of combining the XFAs in Figures 6.21a and 6.21b. For illustration purposes, names of states in the figure contain the source states from which they are composed. Note that this automaton has only 15 states, whereas the combined DFA (not shown, for brevity) requires 2,194 states.

Recall from Section 6.1.1 that the worst-case execution time cost model for XFAs is $\phi + \psi \cdot n_i$ per byte, where ϕ is the cost of a table lookup, ψ is the cost of executing a single instruction, and n_i is the maximum number of instructions at any state. For a combined automaton, ϕ and ψ remain unchanged, but n_i may increase since states in the combined XFA accumulate instructions from both source XFAs. Thus, combination does not affect the state lookup time, but it may affect the instruction execution time. In Section 6.5, we propose optimizations aimed at reducing n_i, the maximum number of instructions per state.

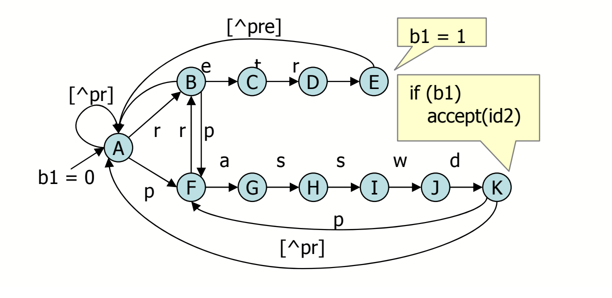
Combining many XFAs is an incremental process: new signatures can be combined with an existing automaton as necessary without needing to reconstruct entirely from scratch. One implicit precondition is that the variable value in the starting configuration be the same in each automaton. In practice, the last phase of the construction process ensures this when mapping to high-level types and instructions.

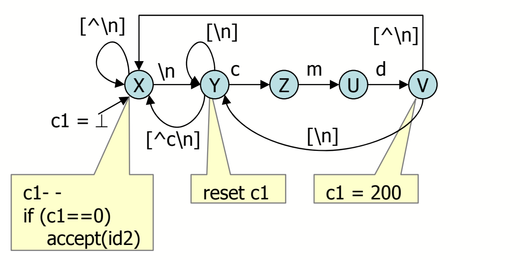
```
**Combine(XFA** first**, XFA** second**)** :
worklist WL
XFA c
c.addState ($\langle$first.start, second.start$\rangle$)
$\langle$first.start,second.start$\rangle$.instrs.append (first.start.instrs)
$\langle$first.start,second.start$\rangle$.instrs.append (second.start.instrs)
c.setStart ($\langle$first.start, second.start$\rangle$)
WL = { $\langle$first.start, second.start$\rangle$ }
while ( $|$WL$| > 0$ ) do
    $\langle$s,t$\rangle$ = WL.pop ()
    foreach ($\beta \in \Sigma$) do
        $s'$ = first.getNextState($s$, $\beta$)
        $t'$ = second.getNextState($t$, $\beta$)
        if $\langle s'$,$t'\rangle \notin$ c.states then
            c.addState ($\langle s'$,$t'\rangle$)
            $\langle s', t'\rangle$.instrs.append ($s'$.instrs)
            $\langle s', t'\rangle$.instrs.append ($t'$.instrs)
            WL.push ( $\langle s'$,$t'\rangle$)
        c.addTrans ($\langle s, t\rangle$,$\langle s', t'\rangle$,$\beta$)
return c
```

Algorithm 6.7: XFA combination. Instructions are copied from source states to “paired” states.

```
**MATCH(XFA** M**, uchar*** buf**, int** len**)** :
state curState = M.start
execInstrs ( curState.instrs)
for i $\leftarrow$ 0 to len do
    curState = curState.nextState(buf [i])
    execInstrs ( curState.instrs)
```

Algorithm 6.8: Algorithm to match an XFA against an input buffer.





(a) XFA for `/.*retr.*passwd/` (b) XFA for `/.*\ncmd[^\n]{200}/`

Figure 6.21: When combined, instructions get replicated at many states.

6.4 Matching to Input

XFA matching employs the same Moore-machine semantics as DFA matching, described in Chapter 3. That is, accepting states output a unique symbol from an output alphabet when reached, regardless of the position of the input. This straightforward extension to DFA matching is also a consequence of the XFA model.

XFA matching, given in Algorithm 6.8, simply extends this model by executing programs attached to states when they are reached. In our framework, and as indicated by the figures, acceptance conditions are implemented as instructions. Thus, no special acceptance tests are needed. An indication of an accepted expression is emitted and processed identically to any other instruction.

6.5 Optimization

The conditions and model in Sections 5.2 and 6.1 allow XFAs to be independently constructed and easily combined without blowup, but this flexibility comes at a cost. As XFAs are combined, variables and their instructions from the source XFAs accumulate in the combined XFA. For a combined XFA composed from several individual automata, many auxiliary variables must be maintained (increasing per-flow state size), and states may contain many instructions to execute (increasing execution time).

In this section, we present a set of optimization techniques that systematically reduce both program sizes and per-flow state requirements of combined XFAs. Taking inspiration from techniques developed for compiler construction [74], we present three distinct optimizations: exploiting runtime information

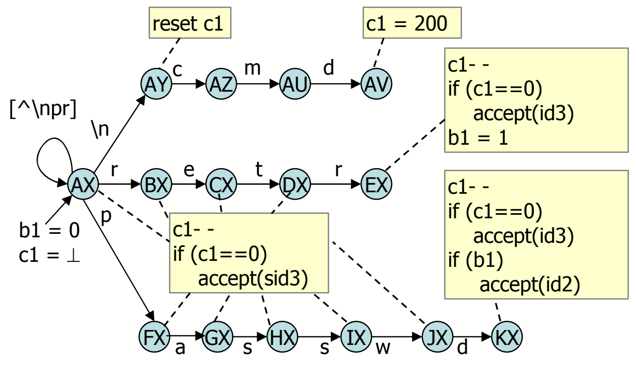


Figure 6.22: XFA produced by combining the XFAs in Figures 6.21a and 6.21b.

and support, combining independent variables, and moving and merging instructions. The first and last techniques reduce instruction counts, whereas the second reduces both per-flow state and instruction counts.

6.5.1 Exposing Runtime Information

Some regular expressions, such as `/.*\ncmd[^\n]{200}/`, induce counters that are decremented after every byte once initialized. For example, when the XFA in Figure 6.21b is combined with other automata, the decrement and test instructions get replicated to most of the states, as shown in Figure 6.22, even though no state explosion occurs. When many such automata are combined, distinct decrement instructions get propagated among all states. Executing these instructions at every state can significantly impact processing times during matching.

Once initialized, the counter in this example will be decremented on all states except those that follow a reset instruction. Thus, when the counter is initialized at a given payload offset, the offset at which it would reach 0 is also known. By maintaining this offset directly, we can eliminate the decrement instruction altogether. This highlights our first optimization, which is to provide runtime support for replacing (and eliminating) common or expensive operations.

Continuing, we extend the runtime environment with a sorted list holding the payload offsets at which the counter would reach 0 along with a pointer to the instructions to be executed when it does. After each symbol is read, the offset value at the head of this *offset list* is compared to the current payload offset, and the consequent instructions are executed on equality. In the automata, initialization and reset instructions are replaced with those that insert into and remove from the offset list, respectively. This does increase the processing overhead slightly, but the optimization replaces explicit updates of (potentially) many counter variables with a single $O(1)$ check after each byte read.

6.5.2 Combining Independent Variables

Some logically distinct state variables can be reduced to a single actual variable. For example, if one counter is active in some set of states and another counter is active in a disjoint set, then the two counters can share the same memory location without interference, leading to reduced memory and smaller programs. This scenario is similar to the register assignment problem faced by a typical compiler: multiple variables can share the same register as long as they cannot be simultaneously "live."

Thus, the goal of this optimization is to automatically identify pairs of variables that are compatible at each state in an XFA. We achieve this goal through a two-step process: a dataflow analysis first determines the states at which a variable is active, and a compatibility analysis uses this information to iteratively find and combine independent variables. These techniques apply to many kinds of state variables, although for presentation purposes we focus on a fairly simple decrementing counter. To aid the discussion, we depict instructions in the format used by our interpreter, rather than the C-like language used thus far, which we describe briefly. Instructions have the form `[instr id,args]`. Initialization instructions set an initial value and also point to the instructions to be executed when the counter reaches 0. Consequently, *decrement* and *test* instructions are combined into a single instruction that decrements a counter and compares it to 0, executing the previously supplied instructions if so. For example, the instruction `[ctrSET 1,200,[ALT 3]]` initializes counter 1 to 200. When the counter reaches 0, the instruction `[ALT 3]` signals that signature 3 has matched. Finally, *reset* instructions make a counter invalid, or inactive (see below), so that it will not be manipulated until another *set* instruction occurs.

We illustrate with the running example in Figure 6.23. The leftmost XFAs correspond to expressions `/\na[^\n]{200}/` and `/\nb[^\n]{150}/` that are combined to give the XFA in the middle of Figure 6.23 (the "clouds" have meaning at a later stage and can be ignored now). In the end, optimization finds that the two counters in the combined automaton are independent and reduces them to one counter.

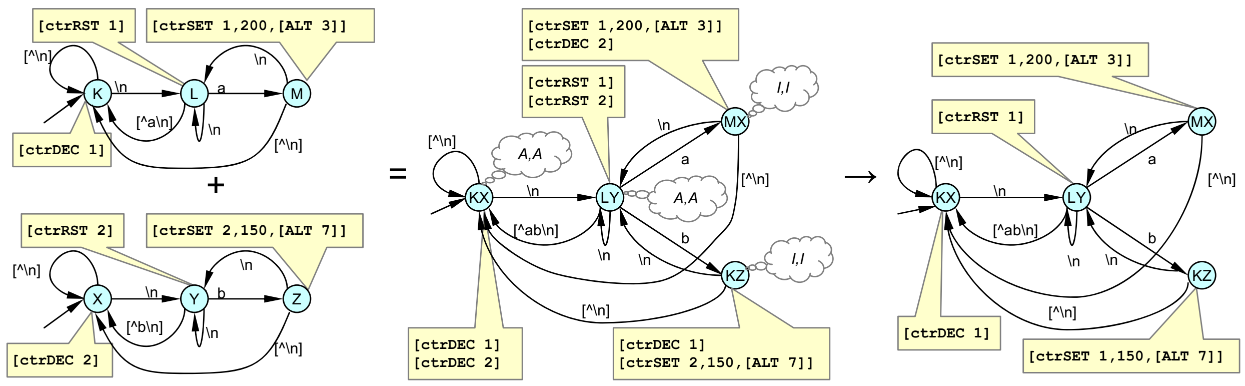


Figure 6.23: The counter minimization process applied to automata for signatures `/.*\na[^\n]{200}/` and `/.*\nb[^\n]{150}/`. The optimization results in the elimination of one of the original counters.

6.5.2.1 Dataflow Analysis

As informally described in Section 5.2.3, counters are initially inactive with status changes occurring whenever initialization or reset instructions are executed. The goal of this step is to determine the activity of each counter at each state in the combined automaton, even for those states without instructions. This requires a precise definition of active and inactive counters, given as follows:

Definition 6.8 Let Q be the set of states containing a `set` operation for counter C. Then, C is active at state S if there is a path from a state in Q to S in which no state in the path contains a `reset` operation for C. Otherwise C is inactive.

In other words, C is active at S if and only if there exists at least one input sequence ending at S containing a `set` but no subsequent `reset` for C. The term *activity* refers to the active or inactive status of a counter. Operations applied to an inactive counter are effectively a no-op.

To calculate activity, we define a dataflow analysis that fits into the classic monotone dataflow framework [74, 80]. Static dataflow analyses comprise techniques used at compile time to produce correct but approximate facts about behavior that arises dynamically at runtime. During execution, different input may yield different behavior depending on that input; static techniques must therefore produce correct (if approximate) results for all possible inputs. Dataflow analyses and their applicability to program optimization are well-studied and at the foundation of many common compiler optimizations including register allocation, constant propagation, and partial subexpression elimination [74]. To the best of our knowledge, ours is the first work that extends the technique to finite automata.

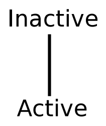


Figure 6.24: The value lattice that orders abstract counter facts. *Inactive* is the initial value.

The first step in an analysis is to identify the abstract values, or *facts*, that the counter can assume and order them in a lattice structure. Here, the values *active* and *inactive* are arranged in the lattice given in Figure 6.24. Second, a directed graph with a designated start node is supplied by the XFA itself. Third, flow functions define the effects that instructions have on each possible value in the lattice. For a counter $\mathcal{C}$ with `set`, `reset`, and `decr-and-test` instructions, the flow functions are defined as follows:

$$\begin{array}{lclclcl} \mathtt{f}_{\mathtt{set}}(\mathcal{C}) & \rightarrow & \text{Active} & \quad & \mathtt{f}_{\mathtt{decr-and-test}}(\mathcal{C}) & \rightarrow & \mathcal{C} \\ \mathtt{f}_{\mathtt{reset}}(\mathcal{C}) & \rightarrow & \text{Inactive} & \quad & \mathtt{f}_{\mathtt{preserve}}(\mathcal{C}) & \rightarrow & \mathcal{C} \end{array}$$

For `set` and `reset`, $\mathcal{C}$ becomes active and inactive, respectively. `decr-and-test` does not change $\mathcal{C}$'s value, and `preserve` is the identity function used when there is no instruction at a state.

These components define a standard forward-flow “may have” analysis. The analysis algorithm propagates facts for each counter among the states, applying flow functions whenever they are encountered. It terminates when the facts have converged to a single value per state. Upon completion, a counter is marked as inactive at a state S if and only if $\mathcal{C}$ is definitely inactive on all paths leading to S. Conversely, if there is any path to S in which $\mathcal{C}$ may be active, then $\mathcal{C}$ is active at S. Hence, the results are correct but approximate.

In Figure 6.23, the clouds in the middle XFA show the activity of each counter at each state prior to instruction execution as computed by the analysis. The counters are inactive at state MX because all paths to MX pass through LY, which resets both counters. Similarly, the counters are active in KX because there is a path from MX that sets counter 1 (making it active) and a path from KZ that sets counter 2.

6.5.2.2 Compatibility Analysis

Two counters can be reduced to one if they are *compatible* at all states in the automaton. At a single state, two counters are compatible if their operations and activity status can be combined without changing the semantics of either counter. We determine compatibility by computing the cross product of operations and activity status and pairwise comparing each element. The compatibility matrix in Figure 6.25a contains

		Inactive		Active			
		r,d,p	**set**	**reset**	**set**	**decr**	**pres**
Inact	**r,d,p**	r,d,p	set	reset	set	decr	pres
	set	set	–	set	–	–	–
Active	**reset**	reset	set	reset	set	–	–
	set	set	–	set	–	–	–
	decr	decr	–	–	–	decr	–
	pres	pres	–	–	–	–	pres

(a) Counter Compatibility

		Inactive		Active			
		r,t,p	**set**	**reset**	**set**	**test**	**pres**
Inact	**r,t,p**	r,t,p	set	reset	set	test	pres
	set	set	set	set	–	test	–
Active	**reset**	reset	set	reset	set	–	–
	set	set	set	set	set	–	–
	test	test	–	–	–	–	–
	pres	pres	–	–	–	–	pres

(b) Bit Compatibility

Figure 6.25: Compatibility matrices for counters and bits, specifying which operations are compatible at a state along with the surviving operation.

this information for the simple counters in this example. As with the dataflow analysis, activity at state S refers to the activity of the counter upon entrance to S, prior to instruction execution.

In the matrix, the *preserve* column handles the cases in which a counter has no instruction at the state in question. *r,d,p* coalesces the entries for the *reset*, *decrement*, and *preserve* operations, which have identical behavior for inactive counters. If two operations are compatible, the corresponding entry holds the operation that survives if the counters are combined. A dash indicates that operations are not compatible. Operations to active counters are incompatible with most other operations, but inactive operations are mostly compatible. The exception is an inactive `set`, which transitions a counter to the active state and is therefore mostly incompatible. The lower half of the rightmost column specifies the cases in which a state has instructions for only one counter, but the dataflow analysis determines that a second counter is also active. Combining the two counters and using the operation of the counter present at the state could change semantics of the second active counter, so the counters are in fact not compatible.

Algorithm 6.9 shows the process for identifying and reducing equivalent counters. For each pair, the algorithm cycles through all states and compares the pair using the `areCompat` function, which extracts activity status and operations for c_1 and c_2 at state s and invokes the counter compatibility matrix. Lines 8-10 perform the actual reduction for a pair of counters that are compatible at all states. When a reduction results in the elimination of one or more instructions at a state, the operation that remains is returned from the compatibility matrix via a call to the `getReduced` function. Note that compatibility is not transitive; when a pair of counters has been reduced, the resulting compatibility between this *new* counter and other counters must be re-established. This is satisfied by Line 11, which causes the algorithm to fall out to the outermost loop after a reduction has been performed. In the running example, the rightmost automaton

```
FIND_EQUIVALENT(XFA  M)        :
1  do
2    foreach pair of counters ($c_1$,$c_2$) do
3      compatible = true
4      foreach state $s \in$ M.states do
5        if  areCompat($s$, $c_1$, $c_2$) == FALSE then
6          compatible = false ; break
7      if compatible then
8        foreach state $s \in$ M.states do
9          $op$ = getReduced($s$, $c_i$, $c_j$)
10         combine counters $c_i$ and $c_j$, keeping operation $op$
11       break;
12 while compatible = true
```

Algorithm 6.9: Counter compatibility. Two counters are equivalent and can be reduced to one if they are compatible at each state.

shows the results after compatibility analysis has determined that counters 1 and 2 are compatible. All references to counter 2 are replaced by a reference to counter 1, and irrelevant `reset` and `decr` operations are removed.

In our experiments, this optimization completes quickly, despite the $O(n^3)$ worst-case runtime of the dataflow and compatibility analysis. With one exception (which contained 172 bits) the procedure completed in less than one minute per test set.

6.5.2.3 Compatibility for Optimized Counters and Bits

The techniques described here apply directly to optimized counters (produced from the first optimization) as well as to simple bits. Recall that counters optimized with the first optimization do not have explicit decrement operations. To compensate, we first insert a fake decrement instruction in each state and for each of these optimized counter. We then perform the analysis and coalesce counters if possible, after which the fake instructions are removed.

For bits, the compatibility matrix varies slightly from the counter compatibility matrix supplied, reflecting the fact that the consequent for a bit test is supplied in the `test` instruction itself rather than in the `set` instruction as is the case for counters. We present an updated compatibility matrix for bits in Figure 6.25b. In the matrix, *r,t,p* combines the entries for *reset*, *test*, and *preserve*. Here, all `set` options are compatible.

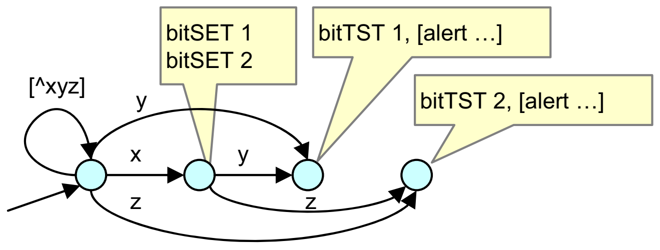


Figure 6.26: Combined automata for `/.*x.*y/` and `/.*x.*z/`. A different dataflow analysis can eliminate a bit.

We conclude the discussion of this optimization with the observation that other dataflow analyses can be designed that identify further reduction opportunities that this analysis misses. For example, Figure 6.26 shows a combined XFA for expressions `/.*x.*y/` and `/.*x.*z/` that share a common prefix and use one bit each. A dataflow analysis that uses more than just activity could determine that a single bit is sufficient for both of these expressions.

6.5.3 Code Motion and Instruction Merging

Many expressions yield automata that set or reset a single bit. When they are combined, individual states may contain many such *bit assignment* instructions. However, the cost of updating a single bit is the same as that for an entire word; by coalescing bit operations whose bits fall within the same word we can shorten the number of instructions in programs and simultaneously reduce the number of writes to memory.

This optimization operates on each state independently. The basic mechanism is to move bit assignment instructions so that those belonging to the same word are adjacent. Such sequences are then replaced by a composite one-word mask and an instruction that applies the mask when executed. There are subtleties, though. First, there are data hazards [50]: bit assignment instructions cannot be moved across other instructions that use or manipulate the bit values without changing semantics. As an example, in the sequence `[bitSET 2],[bitTST 4,([alert,42])],[bitRST 4]`, instruction 3 cannot move left because bit 4's value is used by instruction 2. Second, merged instructions should combine bits belonging to the same word only. Thus, the task is to move and merge as many instructions as possible while satisfying both conditions.

In practice, we use a simple greedy heuristic that identifies many opportunities for merging. The heuristic first identifies all bit assignment instructions that belong to the same word. Next, it looks for data hazards between neighboring pairs of assignments. When a pair with a hazard-free movement direction is found, the instruction is moved along this direction to its neighbor. The process repeats until no more moves are performed. For each word, the optimizer merges adjacent bits, constructs the mask, and replaces the instructions with a single bit mask instruction. This optimization is performed last of all, after the dataflow analysis.

6.6 Experimental Evaluation

6.6.1 Toolset

We have developed a fully-functional evaluation prototype that implements the algorithms described in this chapter and performs matching of XFAs and other automata to network traffic. Our prototype suite is divided into four separate applications: *re2xfa*, *xfa_manip*, *combine*, and *trace_apply*. We describe each of these tools as follows:

1. *re2xfa* - implements all of the XFA construction algorithms described in this chapter, producing XFAs from annotated regular expressions supplied as input.
2. *xfa_manip* - manipulates existing XFAs, including performing optimizations and edge-based to state-based transformations.
3. *combine* - performs cross-product combination of two or more XFAs.
4. *trace_apply* - performs XFA matching. When given an XFA and a tcpdump-formatted trace, this tool extracts and feeds payloads into the XFA and reports matching signatures.

In addition, for comparison purposes we have also implemented a DFA compiler that builds minimized DFAs from regular expressions. These tools produce DFAs with the same general format as XFAs and can be freely used in tools in the XFA toolset (*e.g.*, *combine* and *trace_apply*) for like comparisons. Both XFA and DFA implementations assume a 256-symbol alphabet (one byte symbols) and can parse all regular facets of pcre-style [87] regular expressions[8]. Thus, our implementation can faithfully represent the kinds of signatures seen in commercial NIDS systems.

For XFAs, instructions are executed using an interpreter built into *trace_apply*. Some experiments also use compiled instructions, which we describe below. Finally, since our primary goal is to study the

[8]Some pcre extensions to regular expressions include matching constructs that are not regular. We do not implement those.

feasibility of XFAs, standard NIDS operations such as defragmentation and normalization are beyond the scope of our experiments and not performed here.

6.6.2 Initial Feasibility Experiments

We first report early experiments testing the feasibility of XFAs. These experiments use only bits (no counters) as auxiliary memory with instructions attached to edges. For our test set we used a Snort signature set obtained in March 2007. We gathered traces of live traffic gathered at the edge of the University of Wisconsin's Computer Sciences Department network and collected at different times, with each trace containing between 17,000 and 86,000 HTTP packets. We measure performance as the number of CPU cycles expended per byte of payload. All experiments were performed on a standard Pentium 4 Linux workstation running at 3 GHz with 3 GB of memory.

6.6.2.1 Constructing XFAs

In this section we describe the steps used to construct our test set. First, we used the Snort2Bro tool (included in the Bro [85] software distribution) to do an initial parsing and conversion of Snort's HTTP signatures into Bro format, which we then passed through scripts that created the individual regular expressions. These scripts also inserted the parallel concatenation operator into approximately 97% of the applicable signatures. We indiscriminately gathered both client-side and server-side signatures, yielding 1556 signatures in total. We eliminated 106 signatures for reasons discussed below, giving us a signature set size of 1450.

In Step 2, we manually selected the appropriate instruction template (EIDD) and added the remaining parallel concatenation operators where necessary. In many cases, this process required just a few seconds per signature and was aided by the fact that many signatures have similar formats. Some signatures required the construction of a new EIDD when observed, which typically induced a one-time cost of up to an hour or two. In total, we spent approximately two days on this phase, not including EIDD creation time. Table 6.4 breaks down the signatures according to their general type and gives the number of variables (bits) needed per signature.

Next, we fed each signature and its matching EIDD to the *re2xfa* application, which produced an XFA. XFA construction time varied by EIDD: some completed within seconds whereas others require an hour or more, as summarized by Table 6.5. In our test set, 85% of the signatures completed within 10 seconds each.

Examples (some simplified)	# Sigs	EIDD name	Variables
`.*calendar(|[-_]admin)\.pl`	814	null	nothing
`.*cmd"#.*&`	5	set-only bit	1 bit
`.*<OBJECT#[^>]*classid=11cf-9377`	341	bit	1 bit
`.*<\0O\0B\0#([^>]\0)*c\0l\0s=\01\0c\0-\09\03\0`	213	bit plus parity	2 bits
`(.*[\\/]cgi60#.*auth)|(.*auth#.*[\\/]cgi60)`	56	two set-only bits	2 bits
`(.*/st\.cgi#.*\.\./)|(.*\.\./#.*/st\.cgi)`	21	2 bits plus overlap	3 bits

Table 6.4: Signature types and their mappings to XFAs.

Run time (seconds)	# of signatures.
< 1	37.1%
$1..10$	48.1%
$10..100$	0.1%
$100..1,000$	1.2%
$1,000..10,000$	13.5%

Table 6.5: Distribution of XFA construction times.

Finally, in step 4 we combined each of the XFAs produced in the previous step using the incremental combination algorithm outlined in Section 6.3. Combination of all individual XFAs into a single equivalent XFA required just over 10 minutes. Table 6.6 characterizes the number of instructions on edges and states in the combined XFA. 95% of the transitions have exactly one instruction, and 98% of the states have at most one instruction. The final XFA had 41,994 states (requiring 43 MB), used 193 bits (25 bytes) of auxiliary memory, and required 3.5 MB of instruction memory.

In general, the most manual-labor-intensive aspect of this process occurs when EIDDs are selected for regular expressions. For existing signature sets this is a one-time process, and our experience indicates that when new signatures are produced, a security expert (*i.e.*, someone who writes the initial signatures) familiar with our approach could easily annotate a regular expression, produce an XFA, and add it to an existing combined XFA within a matter of minutes, depending on the XFA construction time in Step 3. Even if a novel signature requires a new EIDD to be defined[9], this is also a one-time cost.

Signatures were removed from this test set for two reasons. First, some complex signatures compose bits and counters in ways that are prohibitively time-consuming to map to EIDDs using our prototype. Second, there are some signatures whose individual DFAs consume exponential amounts of memory and for which

[9]EIDDs are declarative and parsed by our prototype. They can be supplied at runtime and do not require a recompile.

# Instrs	0	1	2	3	4	5	6	7	8	9	10	11	12
% Edges	1.0	94.8	2.7	0.47	0.80	–	–	–	0.13	–	–	–	0
% States	78.9	20.0	0.9	0.03	–	–	–	–	0.05	0.05	–	–	–

Table 6.6: Distribution of instructions on edges and states. Entries marked '–' contribute less than .01%.

our construction algorithms also run out of memory, even though a compact XFA does exist. Signatures of the form `/.*a.`$\{n\}$`b/` among others fall into this category, for example. In both cases, the difficulties arise from using signatures that are not necessarily designed for deterministic automata. Thus, although many signatures with counters are straightforward to compile and map to EIDDs, for this experiment we eliminated all counter-based signatures from our test set.

In summary, these results demonstrate that XFAs can be readily constructed for large numbers of real-world signatures. We produced XFAs for 93% of Snort's HTTP signatures. Construction of this set required a day of manual effort, but admittedly, this process drew heavily on our own experience. When being performed by people with less experience, construction may take longer. Nevertheless, our experience suggests that new XFAs can be quickly constructed and incorporated in many cases.

6.6.2.2 Performance and Memory Usage

We compared XFAs to traditional DFAs and to multiple DFA-based solutions [118], using the same 1450 signatures for each of these techniques that were used for XFA construction. Our attempt to build a single, combined DFA for all signatures failed after only 88 out of 1450 signatures had been processed, at which time over 15 GB of memory was needed for the partial automaton.

Recall that in the Set-Splitting approach to multiple DFA matching, an upper memory bound is given and DFAs are heuristically grouped and combined into as few composite DFAs as possible such that the total memory usage is less than the supplied bound. We implemented the Set-Splitting heuristics and produced multiple DFAs (mDFAs) for several memory limits ranging from 66 MB (the smallest memory size that could hold all signatures) to 512 MB.

Table 6.7 summarizes the performance and memory usage individually for each of the techniques. DFAs, if realizable, would have the best performance but the largest memory consumption; the reported execution time was obtained using the largest partially combined DFA that could be fit into our test machine's

Automata Type	Total Memory	Num Automata	Exec Time (cycles/byte)
XFA	43MB + 3.5MB	1	226.8
DFA	$>$ 15GB	n/a	$\sim$34.8
mDFA	432 MB	67	4,374
	397 MB	107	8,071
	277 MB	147	11,341
	191 MB	346	25,735
	98 MB	587	44,671
	66 MB	786	62,601

Table 6.7: Machine size and execution times for XFAs, DFAs, and Multiple DFAs for several memory settings. XFAs approach DFA performance yet retain small memory sizes.

memory. The six mDFA points shown exhibit the tradeoffs between increased memory vs. increased time, with their execution time being largely a function of the number of created automata. The combined XFA compares favorably as these results show: compared to the next-best data point (the penultimate mDFA entry), the XFA requires 10$\times$ less memory *and* is 20$\times$ faster. On average, the XFA executed 1.12 instructions per byte, roughly consistent with the data in Table 6.6.

Figure 6.27 compares the mDFAs to XFAs graphically. In the plot, the y-axis reflects total memory usage and for XFAs includes both instruction memory and auxiliary memory (46.5 MB). Both axes are on a logarithmic scale. Entries toward the bottom left require reduced resources (either space or time). The plus marks ('+') in the plot show the points for several multiple DFA instances and in a sense represent the true cost of realizable DFA-based approaches. The points hint at the tradeoffs obtained through pure DFA approaches and suggest lower bounds given specific time or memory requirements. The DFA point, if we could plot it, would reside close to the left edge, several orders of magnitude beyond the extent of the graph.

The XFA result, represented by a star, is below and to the left of the curve suggested by the DFA-based approaches, indicating that XFAs require fewer resources overall. The XFA yields superior results as compared to mDFAs both in memory usage and performance.

6.6.3 Optimizations and State-based Experiments

In the second set of experiments we move beyond simple feasibility by examining the effects of optimization, state-based matching performance, and other characteristics applied to several sets of signatures. We use the same test environment as before for these experiments.

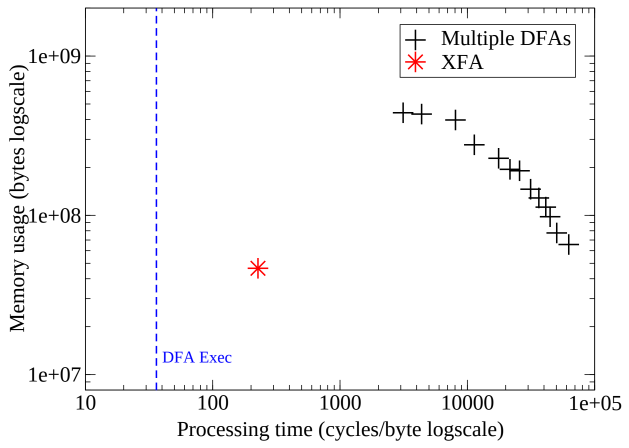
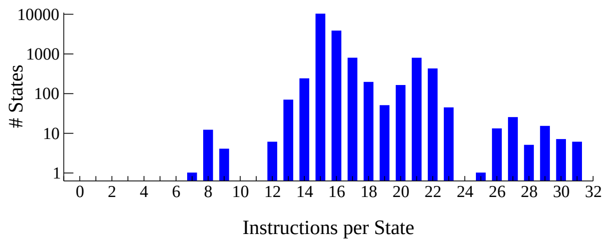


Figure 6.27: Memory vs. run-time for mDFAs and XFAs. XFAs are both smaller and faster than mDFAs for many memory ceilings.

We evaluated XFAs on FTP, SMTP, and HTTP signatures from Snort [91] and Cisco Systems [20]. We used the same procedure as before to produce edge-based XFAs from regular expressions. We then converted to state-based XFAs and combined the XFAs together per-protocol. We also built standard DFAs for each of the regular expressions and combined these per protocol as well.

Table 6.8 summarizes properties of the combined XFAs, showing the number of states, the types and quantities of variables, along with the memory requirements. In each test set, the top row describes the automaton before any optimizations are performed. Columns 3 and 4 give the number of states in the combined DFA and XFA, respectively, and illustrate the magnitude of the savings when state-space explosion is eliminated. In some cases, the combined DFA size may be a gross underestimate: Cisco FTP, for example, exhausted memory after only 23 DFAs were combined. Columns 5 and 6 show the number of variables used by each test set, Columns 7 and 8 give the maximum and average number of instructions per state, and Columns 9 and 10 give the amount of auxiliary memory needed for storing mutable variables and immutable programs. We used two-byte counters when computing the variable memory requirements.

We applied the three optimizations in Section 6.5 in consecutive order and show relevant results in Tables 6.9a, 6.9b, and 6.9c. In Table 6.9a and all subsequent tables, we use a forward slash to separate generic and implicit counters. As the table shows, a large fraction of generic counters were converted to an implicit form. Since these new counters require no explicit decrement instruction, the average number of instructions per state is considerably reduced as shown in Columns 3 and 5. Table 6.9b shows the effect of



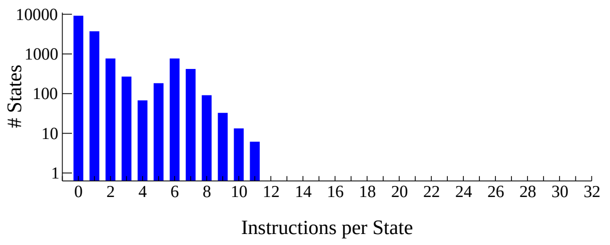


Figure 6.28: Instructions per state for Snort HTTP, before (left) and after (right) optimization.

the analyses for coalescing independent variables. In most datasets, the analysis discovers that a significant percentage of generic and implicit counters can be coalesced. Note that variables must have the same *type* to be considered. For example, generic counters can be coalesced with other generic counters but not with implicit counters. For bits, the reduction opportunities are more modest. We believe that improved results can be obtained with a more refined analysis. Finally, Table 6.9c reports the results of code motion and instruction merging applied to bit instructions. Not surprisingly, the largest reductions come from the sets with the most bits.

Table 6.8 summarizes the cumulative effect of the optimizations in the bottom row of each set. Figure 6.28 shows histograms of the number of instructions per state for Snort HTTP before and after optimization. Note the log scale on the y-axis. After optimization, just over half of all states have no instructions, and all remaining states have 11 or fewer instructions. Histograms for other sets are similar.

6.6.3.1 Memory Usage and Performance

Next, we analyze the memory and runtime performance of XFAs when applied to traces of live traffic. We wrote a translator that converts instructions on states to C source code (with a distinct function for each state) and compiled the code to a shared library whose functions are linked to the appropriate state during initialization. During inspection, programs are executed after the input symbol is read and the state transition is complete. Support for runtime information, as is used in Optimization 1, is compiled into the library as well.

For comparison purposes, we again evaluate multiple DFAs along with the D^2FAs [64] edge-compression scheme, which we briefly described in Chapter 2. Note that D^2FAs employ multiple DFAs to reduce the total number of states. For multiple DFAs, we supplied memory ceilings ranging from 4K total states to

Rule set	Num Sigs	# States DFA	# States XFA	Variables # bits	Variables # ctrs	Instrs per state max	Instrs per state avg	Aux memory (bytes) variables	Aux memory (bytes) program
Snort FTP	72	>3.1M	769	8	46	50	38.67	93	1336K
optimized				8	2/2	5	0.66	9	44K
Snort SMTP	56	>3.1M	2,415	11	31	37	21.48	64	2211K
optimized				6	4/6	21	0.69	21	114K
Snort HTTP	863	>3.1M	15,266	172	15	31	15.91	52	7445K
optimized				171	0/6	11	1.03	34	1008K
Cisco FTP	38	>3.1M	527	11	12	19	12.35	26	271K
optimized				10	0/3	4	0.33	8	16K
Cisco SMTP	102	>3.1M	3,879	8	3	10	5.20	7	806K
optimized				8	0/2	7	0.28	5	76K
Cisco HTTP	551	>3.1M	11,982	13	10	17	10.48	22	4907K
optimized				12	0/2	7	0.42	5	515K

Table 6.8: Combined automata for several protocols, before and after optimization.

Rule set	No-Opt ctrs	No-Opt Inst/state	Opt 1 ctrs gen/imp	Opt 1 Inst/state
Snort FTP	46	38.67	8/38	4.18
Snort SMTP	31	21.48	10/21	1.59
Snort HTTP	15	15.91	0/15	1.24
Cisco FTP	12	12.35	0/12	2.65
Cisco SMTP	3	5.20	0/3	0.34
Cisco HTTP	10	10.48	0/10	0.69

(a) Opt 1: Exploit runtime information

Opt 1 bits	Opt 1 ctrs	Opt 2 bits	Opt 2 ctrs
8	8/38	8	2/2
11	10/21	6	4/6
172	0/15	171	0/6
11	0/12	10	0/3
8	0/3	8	0/2
13	0/10	12	0/2

(b) Opt 2: Coalesce independ. vars

Opt 2 Inst/State max	Opt 2 Inst/State avg	Opt 3 Inst/State max	Opt 3 Inst/State avg
7	0.81	5	0.66
21	0.73	21	0.69
16	1.09	11	1.03
7	0.46	4	0.33
9	0.33	7	0.28
8	0.55	7	0.42

(c) Opt 3: Instruction merging

Table 6.9: Consecutively applying optimizations 1, 2, and 3.

512K total states. During runtime we matched multiple DFAs by modifying our matching code to maintain multiple state pointers. For the D^2FA evaluation, we applied the D^2FA edge compression algorithm to each combined DFA in each mDFA group. The D^2FA proposal requires custom hardware to hash an input symbol to the correct compressed transition entry. To adapt to a software-based environment, we used a simple bitmap-based structure to identify the next transition. This makes the hash function as fast as possible (simulating the hardware assist) with only a minor cost in memory usage.

As before, execution time tests were performed on 10 GB traces captured on the link at the edge of the University of Wisconsin Computer Sciences departmental network at varying times. Runtime measurements were collected using cycle-accurate performance counters and are reported as average cycles per payload byte. During execution, each automaton is applied only to packets belonging to its respective protocol.

Figure 6.29 gives space-time comparisons for each test set[10]. In all plots, the x-axis (processing time) and y-axis (memory usage) increase on a log scale. The dashed vertical line gives the runtime for the largest subset of DFAs that we could combine and fit into memory. Multiple DFAs (mDFAs) trace out a curve showing the trade-offs between memory usage and processing time. Each plus mark ('+') in the figures corresponds to a distinct memory ceiling from which mDFA groups were created.

D^2FAs build on mDFAs and follow a similar curve with a reduced memory footprint. We performed D^2FA edge compression for each multiple DFA point in the figures and show the results as diamonds in the graph. D^2FA edge compression decreases the memory footprint but increases the access time and generally follows the Multiple DFA curve. For XFAs, we plot the combined automata along with the cumulative effects of each optimization, leading toward the lower left corner. In the figures, Optimization 1 exhibits the largest visible improvement. By eliminating instructions on many states, both memory and runtime are reduced by up to an order of magnitude. In general, the second optimization also achieves significant reductions, although here they are largely subsumed by optimization 1. Optimization 3 reduces memory but has a negligible effect on performance.

[10] An early version of this work that appeared in SIGCOMM 2008 [101] contained a calculation error that improperly plotted XFA results in Figure 6.29. That error is corrected here.

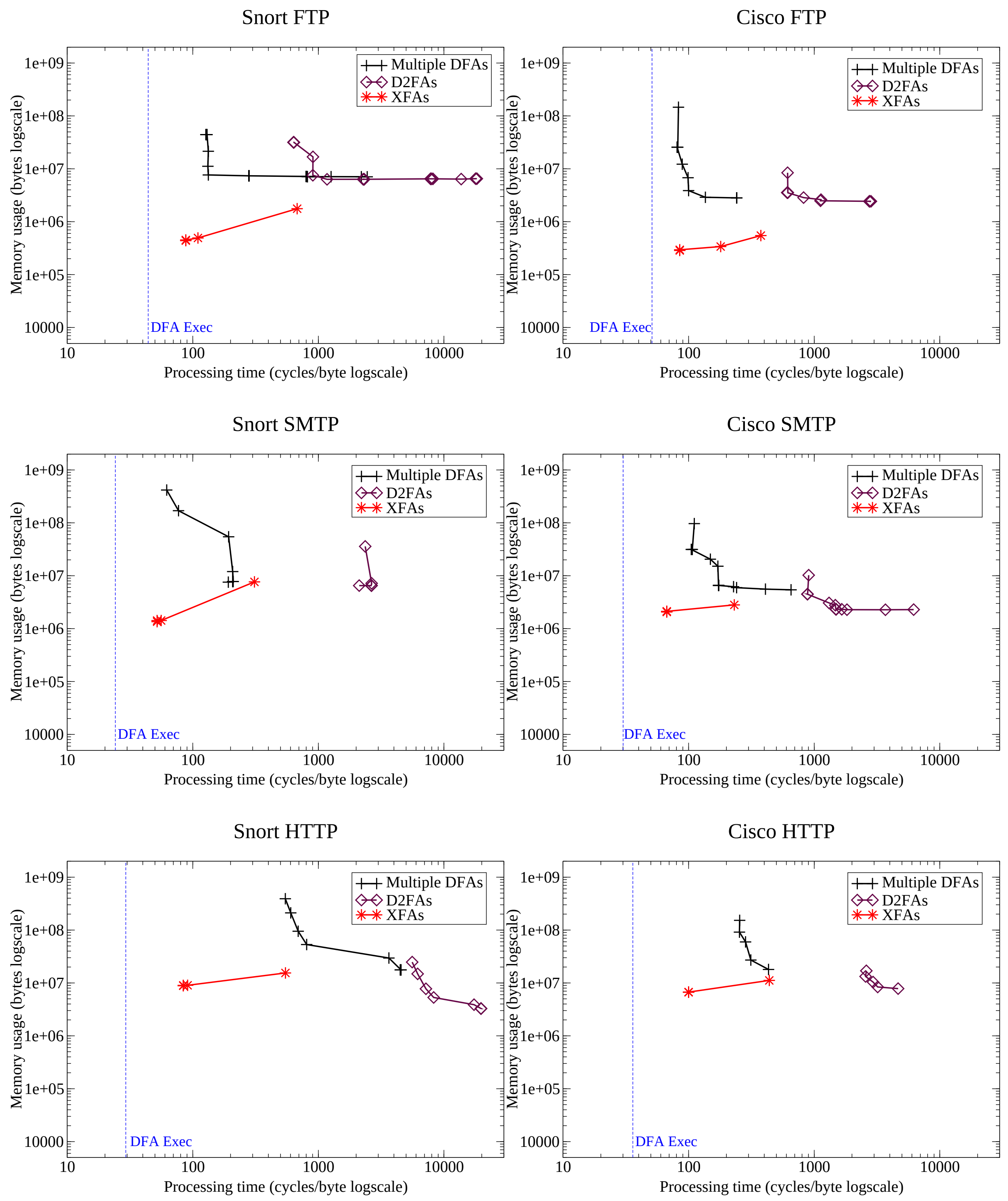


Figure 6.29: Memory versus run-time trade-offs for Multiple DFAs, D2FAs, and XFAs.

	XFA	mDFA/D^{2}FA					
Rule set		8K States		64K States		512K States	
Snort FTP	11	(4)	8	(2)	4	(2)	6
Snort SMTP	23	(11)	22	(11)	22	(4)	12
Snort HTTP	36	(77)	154	(41)	82	(27)	81
Cisco FTP	10	(4)	8	(2)	4	(2)	6
Cisco SMTP	7	(6)	12	(3)	6	(3)	9
Cisco HTTP	8	(23)	46	(14)	28	(8)	24

Table 6.10: Per-flow state in bytes for XFAs and mDFAs at various memory ceilings. Parentheses hold the number of mDFAs at each setting.

6.6.3.2 Per-flow State

A network link typically carries many streams of traffic simultaneously. Each stream is called a *flow*, and packets from many flows are multiplexed one after the other on a link. When a packet is received off the link, it must first be associated with its proper flow. This is the flow reassembly process described in Chapter 2.

Payload inspection for a flow occurs in a disjointed manner as packets are received. A NIDS maintains *per-flow state* that preserves the necessary matching context between two packets that are logically adjacent in a stream, but in fact may be physically separated by packets from many flows. When matching for a packet p_i from flow $\mathcal{F}$ has completed, the per-flow state for $\mathcal{F}$ is updated to reflect the full matching context. Later, when packet p_{i+1} has been received, the per-flow state is first retrieved so that matching can resume at the proper point.

For large links with many flows, the per-flow state requirements can be significant. Table 6.10 depicts the per-flow state for mDFAs/D^{2}FAs at various memory ceilings and for XFAs. mDFAs require a distinct current state pointer for each automaton in a group, and D^{2}FAs have these same requirements. We assume 2-byte state pointers for 8K and 64K ceilings and 3-byte pointers for 512K ceilings. XFA per-flow state contains a state pointer along with all the variables that must be maintained. We quantify this by adding a 2-byte state pointer to each of the optimized variable memory entries (column 9) in Table 6.8. Reductions in per-flow state for XFAs are a direct result of optimization 2. As Table 6.8 indicates, per-flow state can be reduced by up to a factor of six. In Table 6.10, per-flow state for XFAs is comparable to mDFAs in all cases. For large test sets, XFA state can be much smaller, depending on the mDFA memory ceiling.

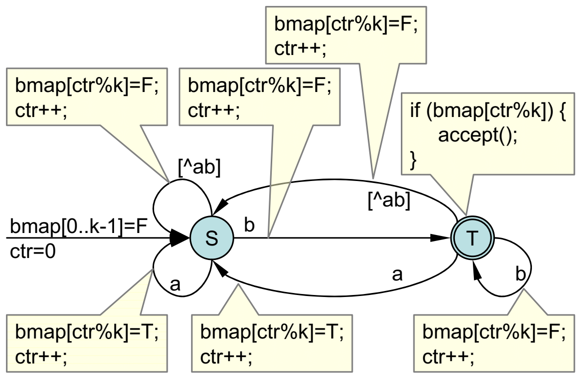


Figure 6.30: An XFA recognizing `/.*a.`$\{n\}$`b/`. In the instructions, $k = n + 2$.

6.7 Limitations and Discussion

The XFA model provides a framework that extends DFAs for incorporating auxiliary variables yet extends DFAs in a natural manner. Even so, there is still much work remaining. We briefly describe a subset of the open issues below.

6.7.1 Mapping to EIDDs

The basic procedure for mapping an XFA with abstract data domains to an appropriate EIDD, given in Algorithm 6.6, uses a backtracking algorithm that we have enhanced to aggressively identify and prune fruitless searches. Even so, some mappings require an hour or more of computation time to complete. Further, each EIDD must specify all the high-level variable types (typically just bits and counters in various forms) to be used by an XFA. Common expressions that simply need one or more bits or counters have standard patterns and can be mapped quickly. However, complex regular expressions in which bits and counters are composed into complex data types require equally complex EIDDs. These are difficult to specify. In principle, we could define a fully generic EIDD that provides many compositions of bits and counters from which Algorithm 6.6 selects only those that it needs. But in our prototype, the resulting mapping times would be infeasible. More work needs to be done to make this mapping faster and to reduce human involvement. Alternative construction procedures may also be worth considering.

6.7.2 Expressions with Exponential State

Some signatures require exponential amounts of space during the construction process, even though they have a compact XFA representation. For example any deterministic automaton recognizing `/.*a.`$\{n\}$`b/` needs to remember which of the previous $n+1$ bytes in the input were '`a`' so that it knows to accept if it sees a '`b`' in the next $n+1$ input characters. DFAs require at least 2^{n+1} states for this case. Similarly, during construction an XFA also needs at least 2^{n+1} distinct configurations, although ultimately these can be contained partially in auxiliary memory rather than only in explicit automaton states. For example, an XFA corresponding to this regular expression, given in Figure 6.30, needs only two states, a counter, and a bitmap with $k = n+2$ bits of auxiliary memory. The number of configurations is exponential, but the number of distinct states is small. For small values of n, we can annotate the regular expression (as `/.*a#.`$\{n\}$`b/`), construct an EIDD, and build the XFA in Figure 6.30. However, since the number of configurations is exponential in n, we quickly run out of memory during construction as n grows. We found dozens of such regular expressions among Snort's web rules, such as rule 3519, which recognizes the regular expression `/.*wqPassword=[^\r\n&]{294}/`.

Fortunately, XFAs are not an exclusive solution and can be easily combined with other techniques to achieve full generality. For instance, we may use substring-based *filters* [91, 94] that identify only subparts of signatures and invoke full signature evaluation using DFAs, NFAs, or other techniques when the subparts are matched. Alternatively, multiple DFAs [118] may also be used.

In general, we observe that signatures are written with an understanding of the underlying matching engine's capabilities. Signatures that are written for an NFA-based engine (such as `/.*a.`$\{n\}$`b`/) are not necessarily appropriate for a deterministic engine and vice-versa. As shown, signatures that can be represented compactly for nondeterministic automata may require exponential state for deterministic automata. In many cases, small changes to a regular expression turn it into something we can build XFAs for efficiently. For example, it is possible to recognize `/.*a[^a]`$\{n\}$`b/` as an XFA with two states and a data domain of size $n+2$ used essentially as a counter. Of course, whether such changes are possible without changing the intent of the rule requires human judgment and is best performed by the signature writer.

6.8 Summary and Conclusion

The Big Bang Theory [48] asserts that a compact, highly compressed mass exploded into a mostly empty universe, leaving scattered pockets of organized matter. This is not too dissimilar from combined DFAs, which experience explosive growth yet are full of redundancy. Our running hypothesis in this XFA work is that the systematic use of auxiliary variables and optimizations provides a practical mechanism for deflating explosive DFAs.

In this and the previous chapter we presented a formal characterization of state-space explosion and showed how auxiliary variables can be used to eliminate it. We presented XFAs, a formal model that extends standard DFAs with auxiliary variables and instructions for manipulating them. We defined optimizations over this model that significantly improve performance and decrease per-flow state.

Many research problems remain open. Our treatment of state-space explosion is preliminary, and stronger results may allow us to better predict and control it. A better understanding of the interplay between protocol parsing and signature matching may yield simpler automata and better performance. But, even with our current prototype, measurements show large improvements over previous solutions. We are optimistic that in the end, XFAs will yield a fast, scalable mechanism for deep packet inspection.

Chapter 7

Edge Compression with Alphabet Compression Tables

The memory footprint of a DFA can be reduced in two ways: by reducing the number of states, and by reducing the size of each state. In the previous chapters we introduced Extended Finite Automata and showed how they could be used as a framework for reducing or eliminating state space explosion. This led directly to smaller numbers of states in an automaton. In this chapter, we look at the second mechanism for reducing memory–reducing the sizes of individual states.

We propose a lightweight compression technique for reducing the memory requirements of states in a DFA. We start from the observation that for NIDS signatures, distinct input symbols often have identical behavior in their DFAs. In these cases, an *Alphabet Compression Table (ACT)* can be used to map such groups of symbols to a single symbol that is retrieved by a table lookup. Alphabet compression tables were first proposed for use in compiler-writing tools such as YACC [2, 55] and have been recently explored in the signature matching context as well [13]. We refine this technique by introducing *multiple* alphabet compression tables [60]. Specifically, we develop heuristics for partitioning the set of states in a DFA and creating compression tables for each subset in a way that yields further reductions in memory usage.

As with XFAs, the use of compression tables or other edge compression techniques does increase the execution time since the appropriate transition needs to be "decompressed" before it can be followed. Fortunately, for alphabet compression tables our experiments show that the inclusion of additional compression tables beyond the first introduces no additional performance overhead. In essence, *multiple* alphabet compression comes for free.

This chapter is organized as follows. In Section 7.1 we start by reviewing algorithms for single alphabet compression tables. We then introduce multiple alphabet compression tables and present algorithms for constructing them. Many mechanisms for compressing the state memory have been proposed in the literature. We surveyed many of them in Chapter 2. In Section 7.2, we describe how compression tables

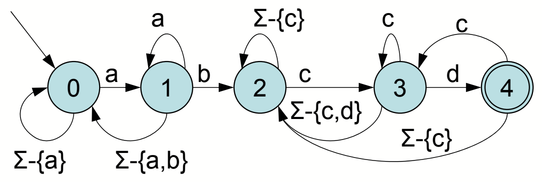


Figure 7.1: A DFA recognizing the regular expression (.*)ab(.*)cd. Starting in state 0, input is accepted whenever state 4 is reached.

interact with two other techniques, D^{2}FAs [64] and Set Splitting [118], which are used extensively in our evaluation. Our experimental results are contained in Section 7.3. Finally, Section 7.4 concludes.

7.1 Alphabet Compression Tables

Recall that a DFA is a directed graph with labeled edges used for efficiently matching regular expressions to input. Nodes are termed *states*, edges between nodes are called *transitions*, and each edge is labeled with a symbol from the input alphabet Σ. For each state S in the DFA, there is an edge for each input symbol in Σ from S to some state S' in the DFA. The set of transitions out of S is referred to as the *transition table* for S, and each state has its own table. A non-empty subset of the states are marked as *accepting*, and there is a distinct starting state s_0. Figure 7.1, reproduced from Figure 4.1, shows a DFA that recognizes the regular expression `(.*)ab(.*)cd`.

The DFA matching procedure keeps a *current state* variable that is initialized to state s_0. During matching, the DFA reads input characters one at a time and updates the current state by following the appropriate transition out of the current state to the destination state. Reaching an *accepting* state indicates that the input thus far is a string in the language defined by the regular expression.[1] Figure 7.2 depicts this procedure at a specific state.

Alphabet compression tables for DFAs arise from the observation that for any given transition table, there are often many input characters that lead to the same next state. Such identical behavior forms a binary relation between input symbols and partitions them into equivalence classes. Individual transition tables can then store a single entry for an entire equivalence class, and a shared lookup table can be used

[1]In the more traditional definition, a DFA signals a match only if it is in an accepting state after reading the last input character. All the results we present apply to that definition as well.

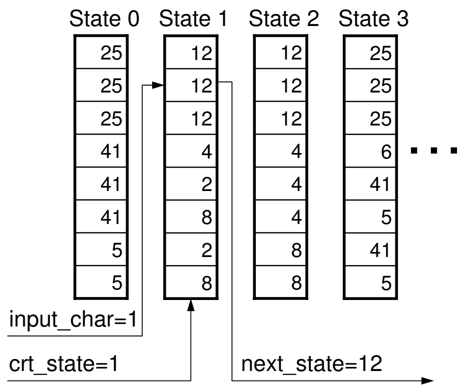
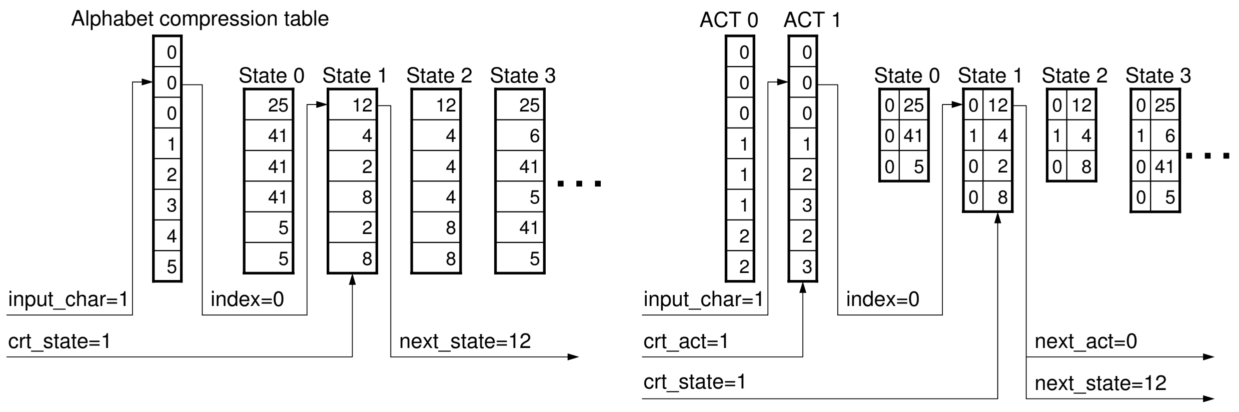


Figure 7.2: At its core, DFA matching involves looking up the next state given the current state and an input symbol in a DFA.

to map from the observed input character to the appropriate equivalence class entry in the compressed transition table (Figure 7.3a). Since this alphabet compression table (ACT) is shared by all states, it will be accessed for every input character, and thus likely reside in the cache of the processor. Therefore, while alphabet compression adds one extra lookup to the per-byte processing, it does not have a significant negative performance impact as there is no need for an extra off-chip memory access.

Before discussing the algorithm for building alphabet compression tables, we clarify some of the notation used in the algorithms in this chapter. Our notation relies heavily on the use of sets whose elements can be characters, states, or other sets (with all elements of a set being of the same type). We use the standard definition for set equality: $\{1, 2\} = \{2, 1\}$, but $\{\{1, 2\}, \{3, 4\}\} \neq \{1, 2, 3, 4\}$ (actually two such sets would never even get compared by our algorithms since their elements are of different types). As usual, the size of a set S given by $|S|$ only counts the number of elements in the top-level set, and does not give a recursive count of all atomic elements. For sets A and B, the statement $A \cup = B$ is shorthand for $A = A \cup B$. Finally, we represent hash tables as associative arrays and use standard notations (*e.g.* $hashtable[key]$) for performing lookups, using sets both as keys and values in some cases. To simplify the algorithms, we introduce the convention that for hash tables whose values are sets, looking up a non-existent value returns the empty set rather than explicitly signaling failure.

We say that a state *distinguishes between* two characters if the transitions corresponding to those characters go to different states. Thus in Figure 7.1, characters `b` and `d` are distinguished between by each of



(a) DFA using a single ACT (b) DFA using multiple ACTs

Figure 7.3: Basic DFA lookup augmented with one or more alphabet compression tables.

states 1 and 3, but not by 0, 2 and 4. On the other hand, characters `e` and `f` are not distinguished between by any state. When using a single compression table there is a unique partition of the symbols in the input alphabet that minimizes the total memory usage. Algorithm 7.1 gives the procedure that computes this partition in a single traversal of the states of the DFA. Starting with a partition of size one whose single entry is the full set of input characters, the algorithm progressively refines the partition to account for distinctions between input characters that manifest themselves as transitions to distinct states out of the same source state. Upon completion, the algorithm finds the smallest number of sets σ of input symbols where all the elements in each set induce the same sequence of traversed states in the automata. Per-state transition tables are correspondingly reduced from $|\Sigma|$ to σ entries. Conversely, for any two characters that are in different sets, there is at least one state that has transitions to different states for these two characters. Given the output of Algorithm 7.1, building the actual alphabet compression table and the compressed transition tables is straightforward. Note that the complexity of this algorithm is $O(n|\Sigma|)$ where n is the number of states and $|\Sigma|$ is the size of the input alphabet.

7.1.1 Multiple Alphabet Compression Tables

It is often the case that many characters behave identically for a large fraction of states $\mathcal{S}$ but are individually distinguished between by a small (perhaps overlapping) set of states. When using a single ACT for all states as in the previous section, individual characters of such groups will need separate entries in each of the compressed transition tables, limiting the memory savings that can be achieved. If instead

```
SingleAlphabetPartition(StateSet $States$):
1  $CrtBestPartition$ = $\{\Sigma\}$
2  foreach state $s \in States$ do
3      $NextPartition$ = $\{\}$
4      foreach character group $g \in CrtBestPartition$ do
5          $NextToChars$ = EmptyHashTable
6          foreach character $c \in g$ do
7              $NextToChars[s.next[c]] \cup= \{c\}$
8          foreach state $n \in NextToChars$.keys() do
9              $NextPartition \cup= \{NextToChars[n]\}$
10     $CrtBestPartition$ = $NextPartition$
11 return $CrtBestPartition$
```

Algorithm 7.1: Compression algorithm that finds the partition of the input alphabet Σ with the smallest number of equivalence classes.

we compute an ACT to apply only to the large subset of states $\mathcal{S}$, the transition tables are smaller since the groups of characters treated identically are larger and fewer. Thus, further reductions in memory usage can be obtained by using multiple ACTs, each over a disjoint subset of states.

To build a DFA with m ACTs, we first divide the states of an automaton into m subsets (discussed below) and then compute a separate ACT for each subset. During matching, the lookup function needs not only the current state and current input symbol but also the identity of the correct ACT to use (in the range $\{1..m\}$). Thus, in the transition table we don't just encode the next state but also the corresponding ACT. Figure 7.3b shows the matching process extended for multiple ACTs. Since the number of ACTs is small (up to 8 in our experiments), for all currently feasible configurations a 32-bit word can encode both the ACT number and the pointer to the next state so that sizes of transition table entries are not increased. Since entries of the transition tables are decoded efficiently and all the ACTs are typically cached, the matching process is not significantly slower than in the case of a single ACT.

In constructing multiple alphabet compression tables, we must first divide the states into subsets that will be covered by the same ACT. For any of these subsets, we can then use Algorithm 7.1 to build the corresponding ACT. If there are no restrictions on the number of ACTs we can use, the partition that minimizes the total size of the transition tables is the one in which all states that distinguish between the same input symbols use the same ACT. Algorithm 7.2 finds this best partition of the set of states in $O(n|\Sigma|)$ time. Unfortunately, for practical automata, the number of ACTs required to achieve the optimum is unfeasibly

```
BestStatePartition(StateSet $States$):
1  $AlphaPartToStates$ = EmptyHashMap
2  foreach state $s \in States$ do
3      $Groups$ = SingleAlphabetPartition ($\{s\}$) // Alg. 1
4      $AlphaPartToStates[Groups] \cup= \{s\}$
5  $Result = \{\}$
6  foreach partition $ap \in AlphaPartToStates$.keys() do
7      $Result \cup= AlphaPartToStates[ap]$
8  return $Result$
```

Algorithm 7.2: Algorithm to partition a set of states so that the sum of the sizes of transition tables is minimized when the number of ACTs is unlimited.

```
PartitionStates(StateSet $States$, Int $m$):
1  $SPrt$ = BestStatePartition ($States$)
2  foreach state set $ss \in SPrt$ do
3      $APrt[ss]$ = SingleAlphabetPartition ($ss$)
4  while $|SPrt| > m$ do
       /* Combine the two subsets that give the least increase in memory.   */
5      $min\_incr = \infty$
6      for $i = 0$ to $|SPrt| - 1$ do
7          for $j = i + 1$ to $|SPrt| - 1$ do
8              $prtsz = |(SPrt[i]| + |SPrt[j])|$
               /* Combine defined in Algorithm 7.4   */
9              $cmbsz$ = |Combine ($APrt[SPrt[i]]$, $APrt[SPrt[j]]$)|
10             $firstsz = |SPrt[i]| \cdot |APrt[SPrt[i]]|$
11             $secndsz = |SPrt[j]| \cdot |APrt[SPrt[j]]|$
12             $incr = prtsz \cdot cmbsz - (firstsz + secndsz)$
13             if $incr < min\_incr$ then
14                 $min\_incr = incr$
15                 $besti = i$
16                 $bestj = j$
17     $NewSet = SPrt[besti] \cup SPrt[bestj]$
18     $APrt1 = APrt[SPrt[besti]]$
19     $APrt2 = APrt[SPrt[bestj]]$
20     $APrt[NewSet]$ = Combine ($APrt1$, $APrt2$)
21     $SPrt \cup= \{NewSet\}$ - $\{SPrt[besti], SPrt[bestj]\}$
22 return $SPrt$
```

Algorithm 7.3: Bottom-up heuristic algorithm for partitioning a set of states into m subsets such that when ACTs are computed separately for each subset, total memory usage is low.

large, so we need algorithms that can guarantee that the number of state subsets produced by the partition of the states is bounded above by a given m.

```
Combine(AlphaPart $Set1$, AlphaPart $Set2$):
/* Structurally, AlphaPart is a set of sets having the form {{a,b,c},{d,e},...} for input symbols a,b,c,d,e,... */
1 $CombAlphaPart = \{\}$
2 foreach set $s_1 \in Set1$ do
3     foreach set $s_2 \in Set2$ do
4         if $s_1 \cap s_2 \neq \emptyset$ then
5             $CombAlphaPart \cup = s_1 \cap s_2$
6 return $CombAlphaPart$
```

Algorithm 7.4: Algorithm for constructing the coarsest alphabet partition such that equivalent symbols in the input partitions are respected when possible.

There are $S(n,m)$ ways to partition n elements into m disjoint subsets, where $S(n,m)$ is a Stirling number of the second kind [16], given as:

$$S(n,m) = \frac{1}{m!}\sum_{i=0}^{m}(-1)^i\binom{m}{i}(m-i)^n$$

Note that $S(n,m)$ is bounded above by $m^n/m!$. We found no criterion for easily determining the optimal partition and instead focused on heuristic techniques. We instead devised and evaluated two heuristics for partitioning the states into m subsets.

First, a "bottom-up" approach starts with the partition produced by Algorithm 7.2 and combines subsets until the total number of subsets is reduced to m. The combination routine iteratively combines subsets two at a time, selecting at each iteration the two subsets that yield the smallest increase in total memory usage. Algorithm 7.3 depicts this process.

The bottom-up algorithm computes a partition of size at most m of the set of states constructed from the larger optimal partition computed by Algorithm 7.2. As long as the partition is too large (*i.e.*, there are more than m subsets), the algorithm greedily picks the two subsets that can be combined with the smallest amount of increase in the total memory usage and combines them. Central to the operation of the bottom-up algorithm is the `Combine` function, shown in Algorithm 7.4, which combines two alphabet partitions such that the combined partition is as coarse as possible while simultaneously respecting the symbol distinctions of the input partitions. For example, for two alphabet partitions $\{\{a,b,c,d,e\},\{f,g,h\}\}$ and $\{\{a,b,c\},\{d,e,f\},\{g,h\}\}$, Algorithm 7.4 produces the partition $\{\{a,b,c\},\{d,e\},\{f\},\{g,h\}\}$. `Combine` operates in $O(|\Sigma|^2)$ time.

Since the alphabet partition produced by `Combine` is at least as large as the largest of the two initial alphabet partitions, combining the two subsets can increase (but never decrease) the sizes of the transition

```
FastPartitionStates(StateSet $States$, Int $m$):
1  $CrtPartition$ = {}
2  $RemainingStates$ = $States$
3  for $i$ = 1 to $m - 1$ do
4      $StatesCovered$ = $RemainingStates$
5      $TargetSize$ = $|RemainingStates|/2$
6      $Groups$=SingleAlphabetPartition ($StatesCovered$)
7      while $|StatesCovered| > TargetSize$ do
           /* Choose the pair of groups distinguished between by the fewest states, discard the states, combine the
              groups.                                                                                          */
8          $StatesCut$ = $StatesCovered$
9          for $j$ = 0 to $|Groups| - 1$ do
10             for $k$ = $j + 1$ to $|Groups| - 1$ do
11                 $Candidates$ = {}
12                 foreach state $s \in StatesCovered$ do
13                     if $s.next[Groups[j][0]] \neq s.next[Groups[k][0]]$ then
14                         $Candidates \cup= \{s\}$
15                 if $|Candidates| < |StatesCut|$ then
16                     $StatesCut$ = $Candidates$
17                     $bestj$ = $j$
18                     $bestk$ = $k$
19         if $|StatesCut|$ == $|StatesCovered|$ then
20             return $CrtPartition \cup \{RemainingStates\}$
21         $NewGroup$ = $Groups[bestj] \cup Groups[bestk]$
22         $Groups$ = $Groups \cup \{NewGroup\}$ - $\{Groups[bestj], Groups[bestk]\}$
23         $StatesCovered$ = $StatesCovered$ - $StatesCut$
24     $CrtPartition \cup= \{StatesCovered\}$
25     $RemainingStates$ = $RemainingStates$ - $StatesCovered$
26 return $CrtPartition \cup \{RemainingStates\}$
```

Algorithm 7.5: Fast top-down heuristic algorithm for partitioning a set of states into m subsets such that when ACTs are computed separately for each subset, total memory usage is low.

tables for the states in the two subsets. Taken together, the overall complexity of Algorithms 7.3 and 7.4 is $O(n^3|\Sigma|^2)$. By storing the results of the computation on line 9, it is possible to reduce the running time to $O(n^2 \log(n)|\Sigma|^2)$, at the cost of increasing the storage requirements to $O(n^2)$. While this algorithm is not guaranteed to give us the best partition of size m, our experiments for feasible values of m yield memory costs that are close to the optimum achievable without constraints on the number of ACTs.

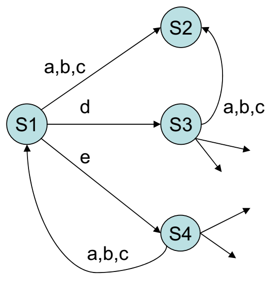
Unfortunately, the bottom-up algorithm's high running time is costly. We found that for large rule sets typical of those found in signature matching, this algorithm was unacceptably slow (each run required over a day to complete) with results no better than those from the top-down approach described below. Thus, we focus instead on the top-down approach presented below.

An alternative method to producing a partition with m subsets is to use a "top-down" approach that starts with a single set and iteratively subdivides until m subsets are produced. In Algorithm 7.5, we present such a top-down approach that completes in only $O(mn|\Sigma|^3)$ time. At each step, the algorithm sets a target for the size of the subset to remove (line 5: we found that setting this target to half the remaining states works well) and finds a subset that is large enough and has a small ACT. To do so it uses a greedy heuristic (lines 7-23) that starts with the set of all remaining states and removes states from the set until the desired size is reached.

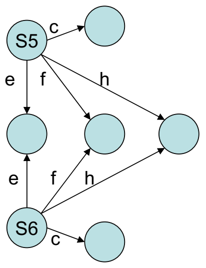
The greedy heuristic implemented by the loop between lines 7 and 23 tries to find a large set of states with an ACT that results in small transition tables. Each iteration of the loop reduces the size of the transition table by one by removing all states that distinguish between two groups of characters. To remove the fewest possible states, the nested inner loops (lines 9 to 18) go through all pairs of groups of characters and pick the two groups that can be combined by removing the fewest states. Note that if at each step of the outermost loop we chose the smallest subset larger than the target size (as opposed to the largest below it), the complexity of the algorithm would reduce to $O(n|\Sigma|^3)$. In practice the difference between the sizes of the two sets is not significant, and the actual running times do not depend much on m since as m increases the loop in lines 7 to 23 works with exponentially smaller sets of states and the processing requirements are dominated by the cost of the first few iterations through the outermost loop.

7.2 ACTs with Other Techniques

Our evaluation shows that using multiple ACTs can provide significant reductions in memory usage with little runtime cost. One of the benefits of ACTs is that it is not restricted to stand-alone operation and can be incorporated with other methods to achieve additional savings. In this section, we present methods for combining multiple ACT compression with other recently proposed techniques for DFA-based signature matching. We first contrast ACTs with another approach, D^{2}FAs [64], and follow this with a description of a hybrid approach that combines both techniques. We then describe how to incorporate ACTs into DFA Set Splitting [118] heuristics.



(a) Symbols behave identically



(b) Many common next-states

Figure 7.4: Different kinds of transition redundancy in DFAs. ACTs can eliminate redundancy in 7.4a; D^{2}FAs are best for eliminating that in 7.4b.

7.2.1 D^{2}FAs

Kumar *et al.* proposed Delayed Input DFAs (D^{2}FAs) [64] as another solution for compressing the transition tables used by DFAs. Whereas alphabet compression exploits the fact that for a given state the transitions for many input characters point to the same next state, D^{2}FAs build on the fact that for a given input character, many states transition to the same next state. Figure 7.4 illustrates the two distinct kinds of redundancy in transitions. In Figure 7.4a, many symbols behave identically and can be compressed with Alphabet Compression Tables. Conversely, in Figure 7.4b distinct symbols at many states lead to common next-states. For example, in States S5 and S6 symbols 'e', 'f', and 'h' each have common next-states with regard to the symbol. This second kind of redundancy can be effectively addressed with D^{2}FAs.

If two states have the same transitions for many characters as in Figure 7.4b, one can reduce memory by storing for one of the states only the transitions that differ. Default transitions that consume no input link states with elided transitions to states that contain the proper transition table entry. As shown in Figure 7.5, if the transition table entry for the input symbol is not stored in the current state, the default transition points to the state whose transition table should be consulted. Multiple states can have default transitions pointing to the same state, and one may need to follow multiple default transitions when processing a single input character. Following chains of default transitions comes at a processing cost, so the maximum length of default transition is given and fixed during construction.

Kumar *et al.* show that D^{2}FAs lead to large reductions in memory usage, but there are two limitations to consider when using D^{2}FAs alone. First, memory savings achieved by D^{2}FAs can vary widely among different kinds of signatures. Figure 7.6 shows the signature `.*\na[^\n]{50}` (read as an arbitrary number

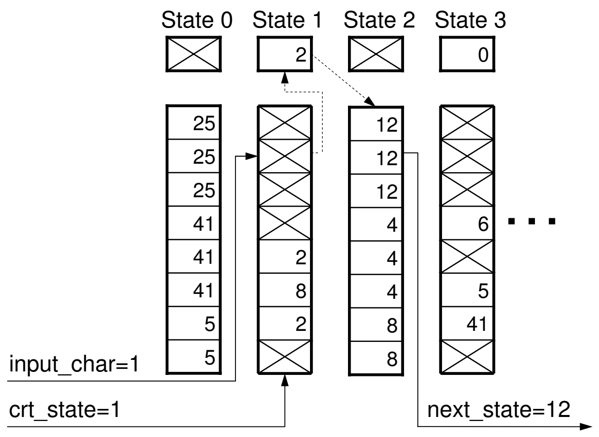


Figure 7.5: For each state, D^{2}FAs employ a default transition that is followed whenever its transition table contains no entry for the current symbol.

of characters followed by a newline and an a followed by 50 non-newline characters) for which D^{2}FAs cannot achieve significant memory savings, but ACTs can. Such signatures are commonly used to detect buffer overflow attacks. States 2 to 52 have very similar transition tables: for the newline character the next state is 1, and for all others the next state is the state with the next number. Applying an ACT for these states can reduce the size of their transition tables to 2, but D^{2}FAs cannot produce significant memory reductions since most of transitions are to distinct next states.

Second, software implementations of D^{2}FAs can be slow. The original D^{2}FA proposal is targeted to custom hardware environments where content addressable memories can be used. Software implementations must use a hash table-like data structure to compress transition tables, but without careful design this can result in unacceptable run-time and memory overheads resulting from computing hash functions and handling collisions.

To adapt to software-based environments, we designed a solution that combines a bitmap and an array to achieve good performance and low memory overhead. Each state has a bitmap as large as the alphabet (256 bits or 8 words) to indicate whether the transition corresponding to a given input character is stored or not, and an array to store the actual transitions. To determine the position of the transition in this array during matching, we need to count the number of bits set to 1 in the bitmap prior to the position of the bit corresponding to the input character. For our signature sets this solution uses between 0.1% and 148% more memory compared to an idealized solution that has no memory overhead. Compared to an idealized

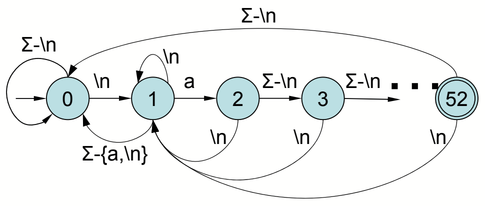


Figure 7.6: A DFA recognizing the signature `.*\na[^\n]{50}`.

solution that performs array lookups instead of hashed lookups, the runtime is between 2.6 and 6 times larger.

7.2.2 Hybrid D^{2}FAs and ACTs

Since ACTs and D^{2}FAs exploit different kinds of redundancy in DFA transitions, it is natural to ask whether it is possible to combine them into a solution that benefits from the strengths of both approaches. We evaluated a straightforward hybrid of the two methods that applies alphabet compression to D^{2}FAs. We perform D^{2}FA compression to a DFA first, followed by multiple alphabet compression applied to the results of the D^{2}FA transformation. To adapt ACT construction algorithms to a D^{2}FA data structure, we extend the transition tables to include a "not handled here" symbol that can be compressed along with symbols, and we extend the default transition entry to also include the appropriate compression table reference. With these extensions, we can directly apply our procedures for building the alphabet compression tables to D^{2}FA-compressed automata. Our experiments in Section 7.3 show that for some signature sets this hybrid solution results in the most compact automata.

Performing a state lookup involves traversing the auxiliary structures inherent to both ACTs and D^{2}FAs. Figure 7.7 illustrates the state lookup process for the hybrid approach. As in Figure 7.3b, the entries in the transition table indicate the next state and the ACT to use. Also, as with D^{2}FAs, the algorithm may need to follow multiple default transitions when processing an input character. In the example, the lookup process begins with an input symbol value of 1, current compression table 1, and current state 1. The first step is to perform a table lookup to apply the appropriate compression table. The resulting index value (index = 0) is then used to look up the next state in transition table for State 1. Here, the transition has been removed due to D^{2}FA compression. Thus, the default transition for State 1 is consulted, which links to State 2 with compression table 0. Continuing, we repeat the process: we map the original input symbol through ACT 0

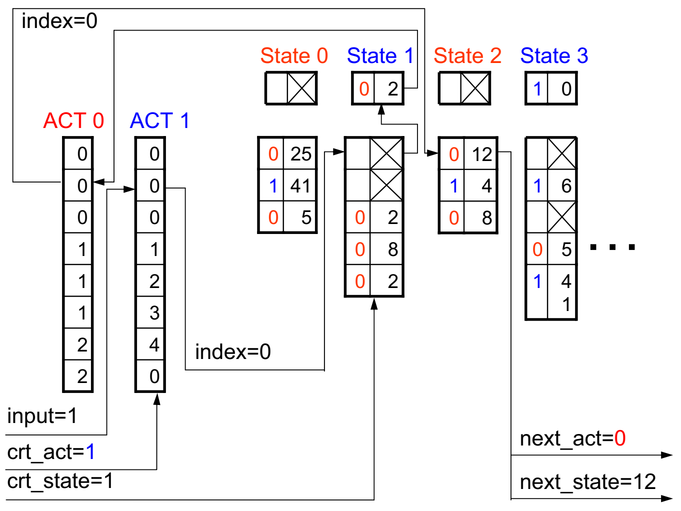


Figure 7.7: Depiction of the state lookup process when both multiple ACTs and D^{2}FAs are employed.

and then look up the resulting index value in State 2. In this state, the transition table contains an entry for the symbol, yielding a next-state of 12 and a next-ACT of 0.

7.2.3 ACT-aware DFA Set Splitting

DFA Set Splitting [118] is an early technique devised to reduce the memory footprint of combined DFAs. In this approach, an upper memory bound is specified, and signatures are partitioned into multiple groups of DFAs (instead of a single DFA) such that the total memory usage is below the supplied threshold. Increasing the threshold reduces the execution time, since fewer combined DFAs need to be created to fit in memory; decreasing the threshold has the opposite effect.

In principle it is easy for edge-compression techniques such as ACTs or D^{2}FAs to be applied in conjunction with Set Splitting. Applying edge compression to each of the constructed DFAs is one simple way to accomplish this. However, such independent interaction exposes sub-optimal memory and execution-time behavior. The Set Splitting heuristics partition the set of DFAs so that the overall *uncompressed* memory usage is within the supplied bound. But, edge compression reduces the effective memory footprint, so that in practice the set of combined DFAs may be an order of magnitude or more below the memory threshold. This excess memory could have been used toward producing fewer combined DFAs and reducing the overall execution time.

We propose a straightforward ACT-aware extension to Set Splitting that incorporates ACT-based edge compression into the heuristic used for partitioning signatures. The Set Splitting heuristic partitions signatures by measuring the *interactivity* between individual signatures and greedily choosing those signatures with the least interactivity. Signatures are then combined until a "fair share" of memory is used, at which time the combined DFA is frozen and another partition is begun. We extend this process by measuring the memory share with ACT compression applied; thus, the memory estimate used for constructing combined DFAs reflects that in actual use. This process is time consuming, since ACT compression must be re-applied at each intermediate stage. Nevertheless, as shown in Section 7.3, we are able to observe a significant reduction in the number of combined DFAs produced.

7.3 Experimental Results

We performed a comparative evaluation using multiple signature sets to better understand the behavior of ACTs in practice. We extracted regular expressions from the FTP, HTTP, and SMTP signatures from the Snort and Cisco IPS rule sets and grouped them by protocol, collecting 1550 regular expressions in total. In addition to the algorithms and techniques described in this chapter, we also implemented the DFA Set Splitting algorithm [118] (termed *mDFA* here, for "Multiple DFA") for combining a set of signatures to a group of DFAs. Finally, our comparative evaluation of D^2FAs was performed using the D^2FA source code obtained from its authors. Modifications discussed in Section 7.2.1 were built upon this as well. Test results involving execution time were obtained using a 10GB trace collected on the edge of a university departmental network. All experiments were performed on a Linux workstation with a 3.0 GHz Pentium IV processor and 3.4 GB of memory that was otherwise idle. We used cycle-accurate performance counters to measure the number of cycles required by the matching operations.

7.3.1 Multiple Alphabet Compression Tables

The first set of experiments looks at the behavior of ACTs as the number of compression tables is increased. For each of our rule sets, we combined a subset of the regular expressions and converted them to a large, single DFA. We then repeatedly invoked Algorithm 7.5 with values of m (the number of alphabet compression tables) increasing from 0 to 8. Table 7.1 presents the memory requirements, execution time,

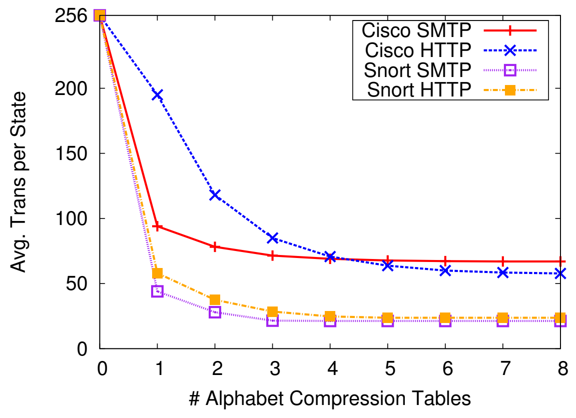
Signature Set	# of ACTs	Memory (KB)	Exec Time cycles/byte	Trans. per state
Cisco SMTP	0	630,669	46.3	256
	1	231,573	50.6	94
	8	165,234	48.8	67
Cisco HTTP	0	106,771	43.0	256
	1	81,329	54.7	195
	8	24,124	52.0	57
Cisco FTP	0	308,602	43.3	256
	1	39,780	49.9	33
	8	24,535	49.6	20
Snort SMTP	0	810,711	22.2	256
	1	139,340	30.0	44
	8	67,761	29.8	21
Snort HTTP	0	163,114	38.6	256
	1	36,955	46.1	58
	8	15,150	43.9	23
Snort FTP	0	1,386,340	35.5	256
	1	167,877	43.6	31
	8	93,815	42.9	17

Table 7.1: Measuring the cost of multiple compression tables. The biggest reductions come after the first table is employed, but additional tables yield further memory reductions.

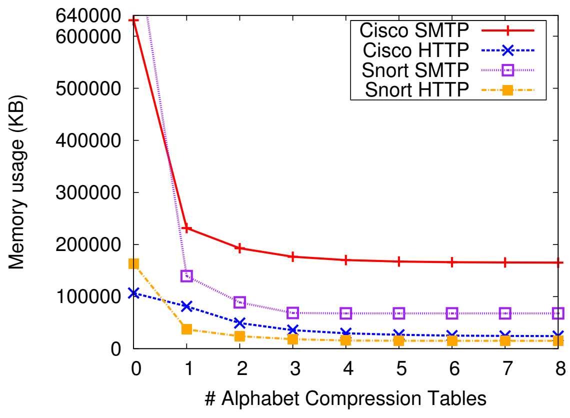
and average transitions per state for 0, 1, and 8 compression tables. Figure 7.8 presents the results graphically for all tested values of m. For clarity of presentation, we show detailed results for only four of the six data sets. The omitted data sets have similar behavior.

The case $m = 0$ is the combined DFA without any alphabet compression applied and serves as the baseline for comparison. Consequently, the number of transitions per state is 256, the size of the alphabet. As m is increased, the memory requirements (also counting the memory used by the compression tables themselves) decreases. With 8 tables, the Cisco signature sets exhibit approximately a $4\times$ reduction in memory usage, whereas for the Snort signature sets a $12\times$ to $15\times$ reduction is observed. As Figures 7.8a and 7.8b show, the memory usage experiences the largest decreases after the first alphabet compression table is applied, but using multiple ACTs reduces memory requirements further.

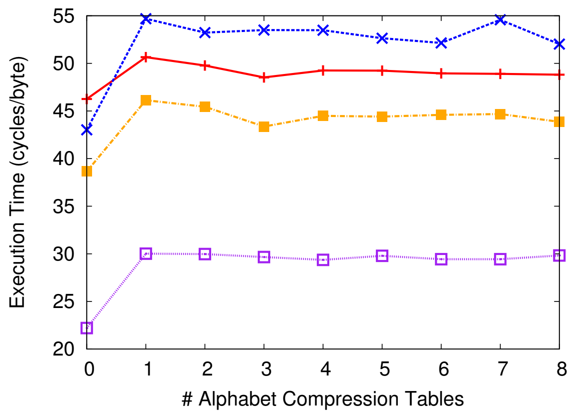
ACTs do carry an increased execution cost, adding 5 to 10 cycles per byte to the execution time on average. Fortunately, in Figure 7.8c we see that this cost is incurred only when the first alphabet compression table is introduced; adding multiple ACTs does not incur significant additional run-time costs. Thus, even



(a) Average transitions per state



(b) ACT memory requirements



(c) Execution time

Figure 7.8: Effect of multiple ACTs on memory usage and matching execution time. Incorporating ACTs induces an initial runtime cost, but subsequent increases in the number of tables is free.

though we observe diminishing returns in memory savings as the number of ACTs increases, the increased savings come for free, essentially, after the initial cost of including compression tables has been paid. For the remainder of the experiments, we use $m = 8$ ACTs.

7.3.2 ACTs, D^{2}FAs and Uncompressed DFAs

Next, we compare ACTs to D^{2}FAs, D^{2}FAs + ACTs, and uncompressed DFAs. Combining all regular expressions into a single DFA exceeds feasible memory limits, so we used set splitting [118] to produce

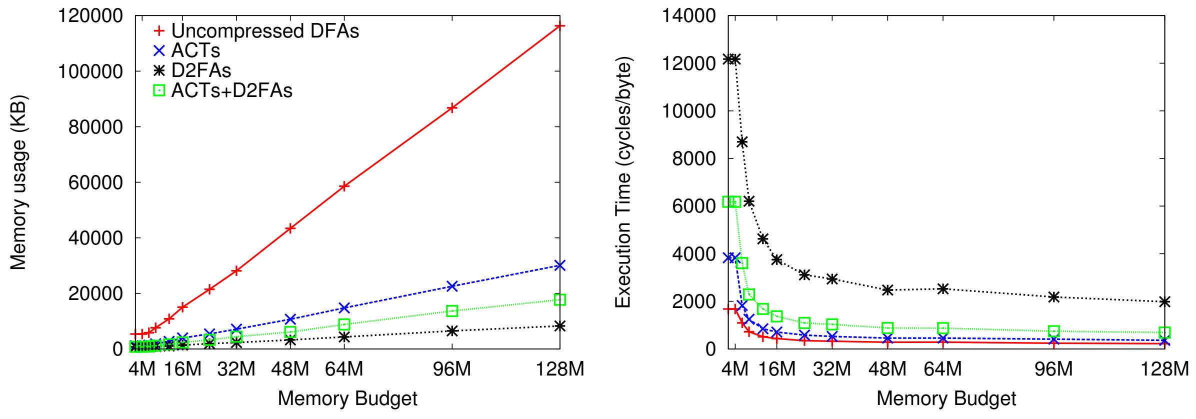
Signature set	Memory budget (MB)	Num. of DFAs	Uncompressed		multiple ACT		D^2FA		mult. ACT+D^2FA	
			Runtime (cyc/byte)	Memory (KB)	Runtime increase	Memory decrease	Runtime increase	Memory decrease	Runtime increase	Memory decreas
Snort SMTP	16	15	266	9,667	1.84×	65.96×	12.70×	1.10×	4.24×	75.43×
	48	13	236	30,058	1.82×	70.66×	12.25×	1.04×	4.51×	73.66×
	128	11	209	98,236	1.79×	17.26×	12.31×	2.20×	3.94×	28.52×
Snort HTTP	16	45	1,103	14,065	2.92×	6.74×	10.58×	2.83×	5.90×	15.17×
	48	28	651	23,693	1.62×	6.71×	9.98×	4.00×	5.33×	14.44×
	128	23	543	73,988	1.59×	9.11×	9.60×	6.81×	3.78×	16.88×
Snort FTP	16	18	434	11,127	1.56×	50.50×	9.47×	1.32×	3.36×	62.10×
	48	14	374	37,920	1.45×	33.28×	9.23×	1.67×	3.02×	40.99×
	128	4	131	94,288	1.35×	19.13×	8.76×	7.92×	2.97×	23.71×
Cisco SMTP	16	4	72	15,316	1.78×	3.92×	13.98×	15.04×	3.85×	6.03×
	48	3	57	40,367	1.72×	3.79×	14.22×	16.38×	3.98×	6.05×
	128	3	57	110,063	1.72×	3.78×	14.34×	17.34×	3.86×	5.92×
Cisco HTTP	16	19	432	15,015	1.64×	3.81×	8.66×	11.03×	3.16×	6.42×
	48	12	282	43,389	1.62×	4.06×	8.76×	13.37×	3.12×	7.12×
	128	9	220	116,352	1.64×	3.87×	8.98×	14.08×	3.11×	6.57×
Cisco FTP	16	3	83	13,308	1.34×	16.41×	9.38×	15.43×	2.61×	31.56×
	48	2	66	22,254	1.19×	16.97×	8.76×	16.92×	2.44×	33.95×
	128	2	70	83,162	1.14×	16.09×	8.26×	19.29×	2.23×	42.86×

Table 7.2: Comparison of run times and memory usage for uncompressed DFAs, DFAs using multiple ACTs, D^2FAs, and D^2FAs using multiple ACTs.

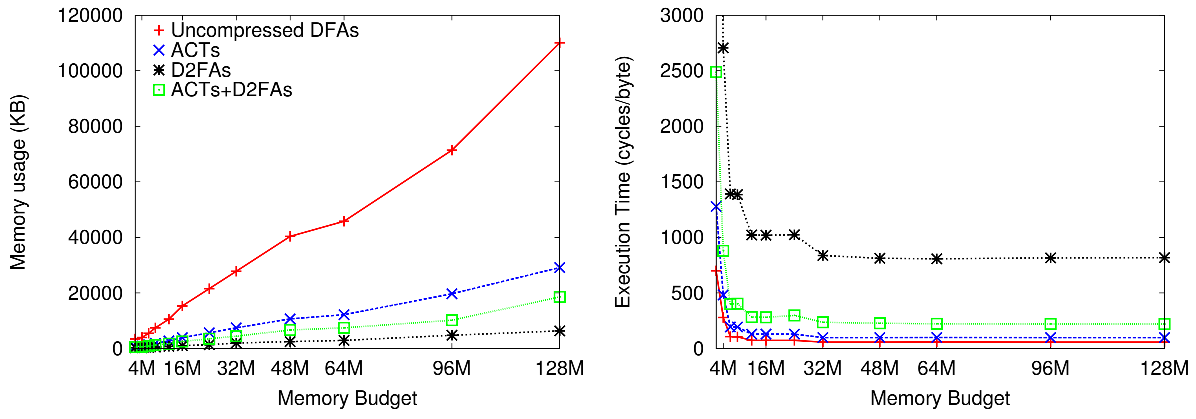
sets of combined DFAs that cover all the rules. For the construction, we supplied memory budgets ranging from 4 MB to 128 MB.[2] As shown in columns 2 and 3 of Table 7.2, smaller memory budgets result in large numbers of DFAs to match. We use the term *protocol set* to refer to the set of DFAs produced by the algorithm for a given protocol and a given total memory setting. We then built a distinct set of eight alphabet compression tables for each protocol set. Thus, for example, a rule set such as Snort SMTP combined into six DFAs would contain eight ACTs that are shared among the six DFAs. Finally, we repeated the construction process to produce D^2FAs for each of the DFAs in the protocol sets.

We performed signature matching using protocol sets with uncompressed DFAs, DFAs with ACTs, D^2FAs, and D^2FAs with ACTs, recording execution time and memory usage. Table 7.2 shows the results for three memory settings: 16 MB, 48 MB, and 128 MB. Execution times are higher in these results principally because we must repeat the matching procedure for each DFA in a protocol set. Note also that in some cases (Cisco SMTP), increasing the amount of available memory does not decrease execution time. This behavior is an artifact of the greedy algorithm [118] for building the protocol sets. In general, the table shows that increasing total available memory reduces the number of DFAs in the protocol set, decreasing execution time.

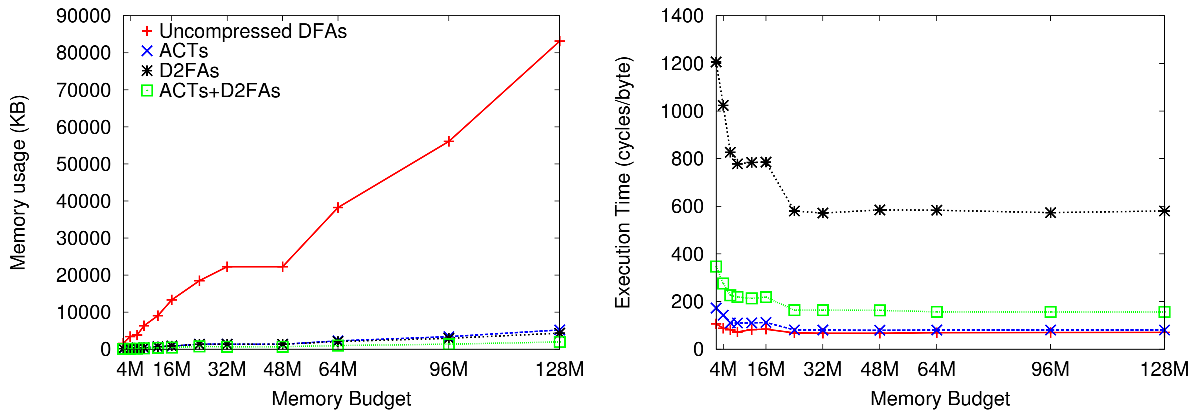
[2]Although 128 MB may seem rather small in relation to modern memory capacities, our tests are performed using a single protocol. In reality, DFAs for many protocols must reside in memory simultaneously.



(a) Cisco HTTP

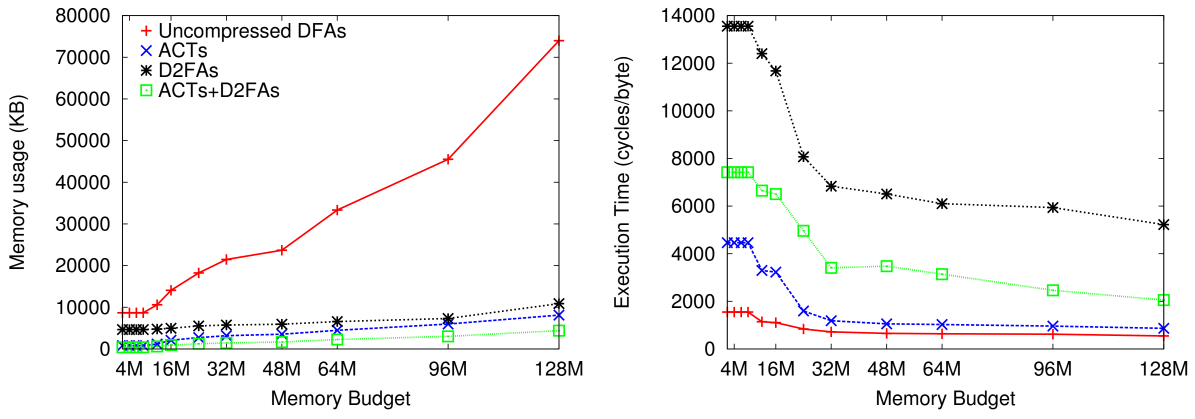


(b) Cisco SMTP

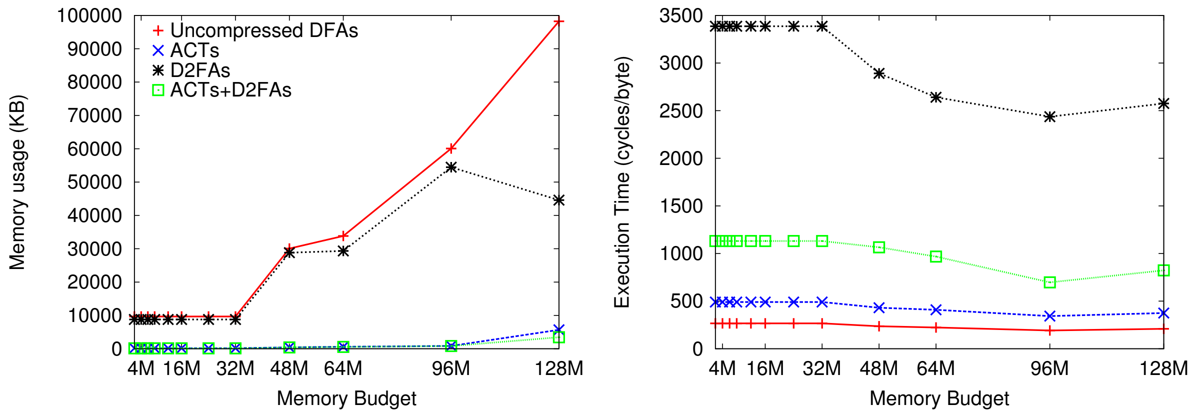


(c) Cisco FTP

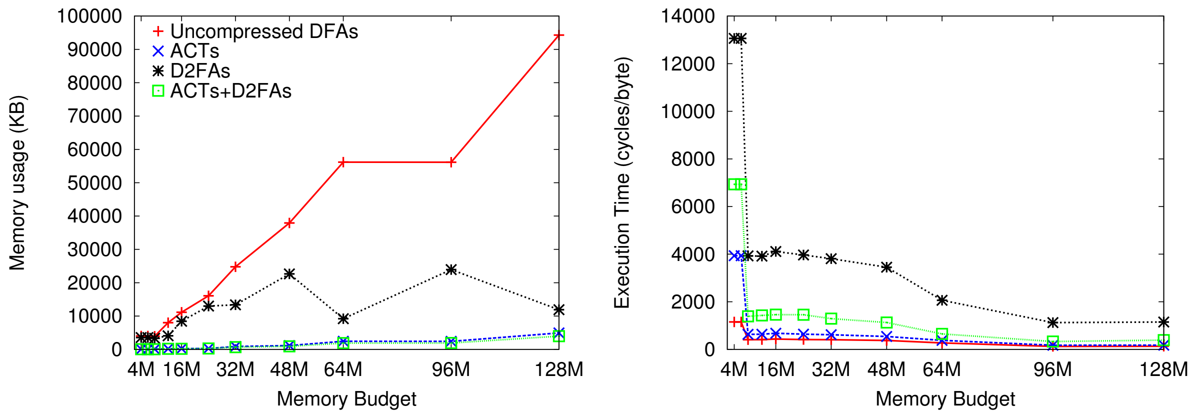
Figure 7.9: Comparing memory usage (left) and performance (right) of ACTs to mDFAs, D^{2}FAs, and their combination, using Cisco rule sets.



(a) Snort HTTP



(b) Snort SMTP



(c) Snort FTP

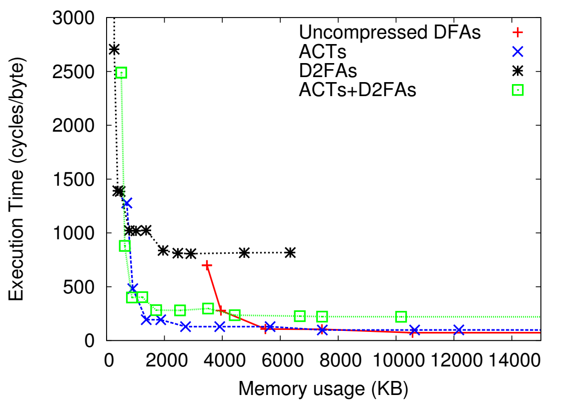
Figure 7.10: Comparing memory usage (left) and performance (right) of ACTs to mDFAs, D^{2}FAs, and their combination, using Snort rule sets.

Compared to uncompressed DFAs (column 4 in Table 7.2), the table shows a sharp reduction in memory costs when eight ACTs are employed (column 5). For 16 MB total memory, ACTs are between 66$\times$ smaller (Snort SMTP) and 4$\times$ smaller. At 128 MB, DFAs with ACTs are between 19$\times$ and 4$\times$ smaller. As expected, however, there is a slight increase in execution time: execution times with ACTs are typically between 35% to 85% slower, the largest slowdown approaches a factor of 3$\times$. Figure 7.9 (Cisco rules) and Figure 7.10 (Snort rules) show the memory usage (top graph) and execution time (bottom graph) for all supplied values of available memory for three signature sets.

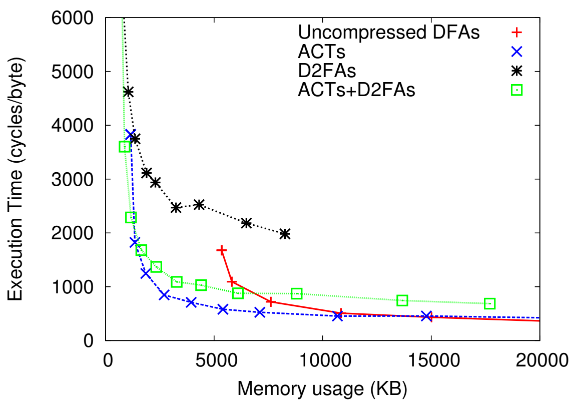
D^{2}FAs (column 6) exhibit wider variability in their performance and memory usage than ACTs. For Cisco rule sets, our tests give an 11$\times$ to 17$\times$ reduction in memory usage. These results are generally consistent with those reported in [64]. For the Snort signature sets, however, which were not included in the original D^{2}FA evaluation, the memory reduction is always less than a factor of 8 and often less than a factor of 2. This is consistent with our observation that D^{2}FAs are designed to optimize DFAs in which certain symbols in the alphabet (almost) always go to the same state. This is not characteristic of the Snort sets, and thus there is little opportunity for compression.

The hybrid algorithm that combines D^{2}FAs and ACTs (rightmost column in Table 7.2) always achieves low memory (often the lowest of all solutions), and run-times that are close to, but larger than those of ACTs. ACTs are faster because the matching algorithm does not need to follow default transitions. Interestingly, in one of the signature sets D^{2}FAs use less memory than the hybrid approach. The reason is that after applying ACTs to D^{2}FAs, for a given state there may be multiple entries in the actual transition table storing the “not handled here” symbol, resulting in higher memory usage than D^{2}FAs that do not store these entries.

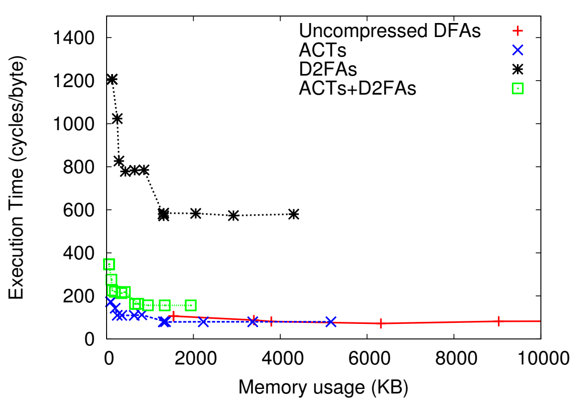
Both execution time and memory usage are critical resources in signature matching and induce a space-time trade-off. Figure 7.11 depicts a space-time comparison for all six of our test sets, directly showing the trade-offs that occur between memory usage (the x-axis) and execution time (the y-axis). We have truncated the axes in some sets to more clearly highlight the data in the lower left-hand quadrant; this does not influence the interpretation. Each point on the plot refers to an observed total available memory setting. Data points belonging to the same compression technique trace out a curve that shows the trade-offs between execution time and memory for that technique. In the limit, large memory yields fast execution, and small memory requires large execution times. Entries toward the origin (the bottom left corner) require reduced resources in space and time and are thus preferred.



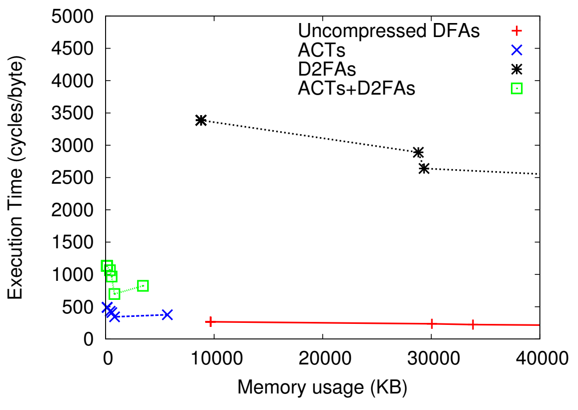
(a) Cisco SMTP



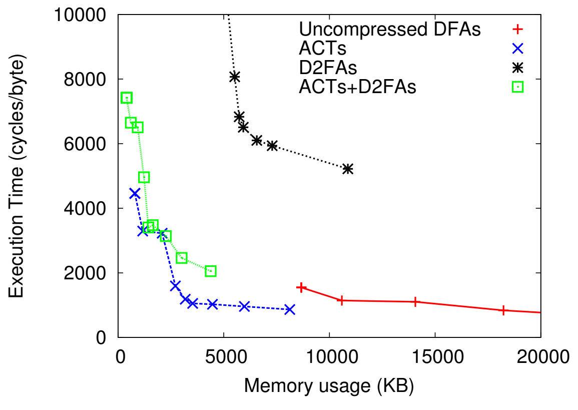
(b) Cisco HTTP



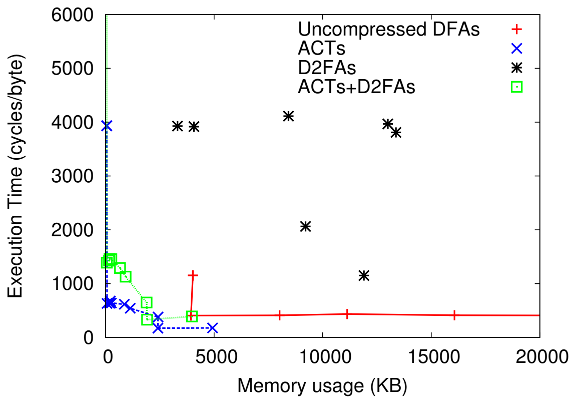
(c) Cisco FTP



(d) Snort SMTP



(e) Snort HTTP



(f) Snort FTP

Figure 7.11: Comparing memory usage vs. run-time performance.

Signature Set	Mem Budget	# DFAs	
		Sequential	Act-Aware
Cisco SMTP	16	4	3
	48	3	3
	128	3	2
Cisco HTTP	16	19	9
	48	12	7
	128	9	6
Cisco FTP	16	3	2
	48	2	2
	128	2	2
Snort SMTP	16	15	7
	48	13	7
	128	11	5
Snort HTTP	16	45	38
	48	28	31
	128	23	17
Snort FTP	16	18	3
	48	14	3
	128	4	2

Table 7.3: ACT-Aware Set Splitting. When DFA Set Splitting is extended to include alphabet compression, the number of resulting DFAs is reduced.

Most importantly, for all protocol sets ACTs provide the most favorable trade-offs between run time and memory usage. Admittedly, it may be surprising that ACTs can be faster than uncompressed DFAs despite the overhead of the compression table mapping. In reality, large available memory sizes (resulting in bigger but fewer DFAs) combined with excellent ACT memory reduction yields a memory footprint that is smaller than for uncompressed DFAs, and the time savings obtained from executing fewer DFAs more than compensates for the ACT overhead. Thus, a small number of highly compressed DFAs can be both smaller and faster than other alternatives.

7.3.3 ACT-aware DFA Set Splitting

We briefly quantify the effects of extending DFA Set Splitting heuristics with alphabet compression tables. We extended DFA Set Splitting to measure the memory usage of multiple ACT-compressed combined DFAs rather than uncompressed DFAs. By doing so, more signatures are grouped into fewer combined DFAs. Table 7.3 shows the effects ACT-aware Set Splitting for different memory thresholds. Column 3 shows the number of DFAs required when Set Splitting and ACTs are applied sequentially and independently. Here, the effective memory is significantly below the memory budget (ref. column 7 of Table 7.2).

Column 4 shows the number of DFAs required when Set Splitting is augmented with ACT-based memory usage). In some cases, the reduction is dramatic. For example, at the 128MB threshold, the Snort SMTP set requires just 5 DFAs, down from 11 using the unmodified algorithm.

The downside to this integrated approach is the construction time: alphabet compression must be performed at each intermediate stage as well as at the end, and for our experiments this required several hours of computation time per configuration. This cost may prove untenable in dynamic settings with frequently added or updated signatures. Lightweight techniques for estimating the memory savings may reduce construction time to more reasonable levels.

7.4 Conclusion

In this chapter we introduced multiple alphabet compression tables (ACTs) for reducing the memory footprint of DFA-based signature matching. This technique uses heuristics to partition the states of a DFA, computing a distinct ACT for each partition. Using Multiple ACTs achieves increased memory reduction over single ACTs with no additional runtime cost. We present algorithms for constructing multiple ACTs and demonstrate their effectiveness using signatures found in Cisco IPS and Snort. ACTs are applicable in software-only environments, although they may be easily included in hardware-based solutions.

Compared to uncompressed DFAs, multiple ACTs achieve memory savings of between a factor of 4 and a factor of 70 at the cost of an increase in run time that is typically between 35% and 85%. Compared to D^2FAs, multiple ACTs are between 2 and 3.5 times faster in software, and for some signature sets they use less than one tenth of the memory. Overall, for all signature sets and compression methods evaluated, ACTs offer the best memory versus run-time trade-offs.

Chapter 8

Conclusion

Network intrusion detection operates in a difficult environment. Restrictive memory sizes and limited processing capabilities coupled with increasing traffic loads, increasing numbers of signatures, and increasing signature complexity exert constant pressure to do more work with comparatively fewer available resources. Further, systems that fail to satisfy performance requirements are vulnerable to evasion.

Signature matching is at the heart of intrusion detection, with regular expressions the language of choice for writing signatures. However, standard matching techniques such as NFAs and DFAs induce a time-space tradeoff between the memory footprint and the execution time and are unsuitable for NIDS use. One alternative is to employ approximation techniques that quickly identify benign traffic, retaining potentially malicious traffic for more detailed analysis. But, this induces a type of tradeoff between accuracy and execution time, where higher accuracy comes at the cost of longer execution.

The goal of this work has been to discover and properly characterize the principles behind these tradeoffs, and to develop richer matching mechanisms that either eliminate the tradeoffs or make them more manageable. This leads to signature matching mechanisms that are more amenable to the NIDS environment, that require fewer resources to operate, and that are more resilient to attack.

For DFA-based matching, we introduced the notion of ambiguity and showed that it is at the root of the state-space explosion phenomenon and subsequent memory exhaustion that often occurs when DFAs are combined. We then proposed the use of auxiliary state variables as a way to “factor out” the ambiguity in DFAs. When auxiliary variables are properly employed, the state-space explosion still occurs, but the state-space is structured so that the number of explicit automaton states is bounded, and memory exhaustion is avoided. From a DFA perspective, a slight decrease in matching performance yields a significant reduction in memory usage.

Similarly, as we demonstrated in Chapter 3 for the Snort NIDS, the divide between worst-case and average-case signature matching performance can be bridged by using memoization and other techniques that track intermediate state and avoid unnecessary computation. At the expense of a slight increase in memory usage, we reduced the worst-case slowdown by four orders of magnitude, bringing the worst case to within one order of magnitude of the average case, based on our measurements.

As these and other examples from our work illustrate, by accepting slightly degraded behavior in one initially acceptable aspect of a time-space tradeoff (either time or space), we can obtain significantly improved behavior from the other, formerly unacceptable, aspect. These results demonstrate that the tradeoffs associated with intrusion detection are not immutable obstacles, but rather can be changed through appropriate research and engineering effort.

For this author, the results presented here have raised at least as many questions as they have answered. For example, with regard to DFAs, ambiguity provides a sufficient criterion for assessing the behavior of DFAs under combination. But, it is very strict, and more refined conditions may lead to a better characterization of state-space explosion, among other things. Regarding XFAs, auxiliary variables can be used to eliminate ambiguity, but the practical limits on their use as ambiguity-removing mechanisms is not known. Also, constructing XFAs from regular expressions requires some human intervention; techniques that minimize or eliminate human involvement altogether would be useful. Finally, our work has focused almost exclusively on regular expression-based signatures. It would be interesting to consider the use of more expressive languages, such as context free or context sensitive languages, and the implications of their matching models for network intrusion detection.

LIST OF REFERENCES

[1] Alfred. V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, June 1975.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[3] Mansoor Alicherry, M. Muthuprasannap, and Vijay Kumar. High speed pattern matching for network IDS/IPS. In *ICNP*, November 2006.

[4] Rajeev Alur. Timed automata. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 8–22, 1999.

[5] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable internet protocol (aip). In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 339–350, New York, NY, USA, 2008. ACM.

[6] James P. Anderson. Computer security threat monitoring and surveillance. Technical Report, James P. Anderson Co., April 1980.

[7] Brenda S. Baker. Parameterized pattern matching by Boyer-Moore type algorithms. In *Proc. Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, January 1995.

[8] Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, February 1996.

[9] Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. January 2002.

[10] Paul Baran. On distributed communioations networks. *IEEE Transactions on Communication Systems*, CS-12:1–9, March 1964.

[11] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *Internet Measurement Workshop*, pages 71–82, 2002.

[12] Michela Becchi and Srihari Cadambi. Memory-efficient regular expression search using state merging. In *Proceedings of IEEE Infocom*, May 2007.

[13] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154, 2007.

[14] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. In *Communications of the ACM*, volume 20, October 1977.

[15] Benjamin Brodie, Ron Cytron, and David Taylor. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.

[16] Richard M. Brualdi. *Introductory Combinatorics, 2nd edition*. Prentice Hall, 1992.

[17] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, May 2006.

[18] João B.D. Cabrera, Jaykumar Gosar, Wenke Lee, and Raman K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *43rd IEEE Conference on Decision and Control*, December 2004.

[19] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker (2nd Edition)*. Addison-Wesley, 2003.

[20] Cisco intrusion prevention system. `http://www.cisco.com/en/US/products/ps6634/products_ios_protocol_group_home.html`.

[21] Christopher R. Clark and David E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE FCCM*, April 2004.

[22] David Clark. The design philosophy of the darpa internet protocols. In *ACM SIGCOMM 1988: Symposium proceedings on Communications architectures and protocols*, pages 106–114, 1988.

[23] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[24] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards faster pattern matching for intrusion detection, or exceeding the speed of snort. In *2nd DARPA Information Survivability Conference and Exposition*, June 2001.

[25] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proc. 6th International Cooloquium on Automata, Languages, and Programming*, pages 118–132, 1979.

[26] The Snort network intrusion detection system on the intel ixp2400 network processor. Consystant White Paper, 2003.

[27] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[28] Scott Crosby. Denial of service through regular expressions. In *Usenix Security work in progress report*, August 2003.

[29] Scott Crosby and Dan Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, August 2003.

[30] Peter Dencker, Karl Durre, and Johannes Heuft. Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst.*, 6(4):546–572, 1984.

[31] Dorothy E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, 1987.

[32] Theo Detristan, Tyll Ulenspiegel, Y. Malcom, and M.V. Underduk. Polymorphic shellcode engine using spectrum analsyis. `http://www.phrack.org/archives/61/p61-0x09_Polymorphic_Shellcode_Engine.txt`.

[33] Sarang Dharmapurikar and Vern Paxson. Robust tcp stream reassembly in the presence of adversaries. In *USENIX Security*, August 2005.

[34] Qunfeng Dong, Suman Banerjee, Jia Wang, and Dheeraj Agrawal. Wire speed packet classification without TCAMS: One more register (and a bit of logic) is enough. TR1549, Department of Computer Sciences, University of Wisconsin–Madison, 2006.

[35] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational experiences with high volume network intrusion detection. In *CCS*, October 2004.

[36] Steven T. Eckmann, Giovanni Vigna, and Richard Kemmerer. Statl: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[37] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved dfa for fast regular expression matching. *SIGCOMM Computer Communications Review*, 38(5):29–40, 2008.

[38] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[39] Mike Fisk and George Varghese. Fast content-based packet handling for intrusion detection. TR CS2001-0670, UC San Diego, May 2001.

[40] Lance Fortnow. Nondeterministic polynomial time versus nondeterministic logarthmic space: Time-space tradeoffs for satisfiability. In *Proceedings of Twelfth IEEE Conference on Computational Complexity*, 1997.

[41] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[42] Jonathan Giffin. *Model-based Intrusion Detection System Design and Evaluation*. PhD thesis, University of Wisconsin–Madison, 2006.

[43] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, 2002.

[44] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[45] Yu Gu, Andrew McCallum, and Donald F. Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *Internet Measurment Conference*, pages 345–350, 2005.

[46] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, March 2001.

[47] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Usenix Security*, August 2001.

[48] Stephen W. Hawking. *A brief history of time. From the Big Bang to Black Holes*. Bantam Book, 1988.

[49] L. Todd Heberlein, Gihas V. Dias, Karl N. Levitt, Biswanath Mukherjee, Jeff Wood, and David Wolber. A network security monitor. pages 296–304, May 1990.

[50] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition.* Morgan Kaufmann, 1996.

[51] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292, 1996.

[52] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[53] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[54] Paul Innella. The evolution of intrusion detection systems, November 2001.

[55] S.C. Johnson. Yacc – yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, 1975.

[56] Myles Jordan. Dealing with metamorphism. *Virus Bulletin Weekly*, 2002.

[57] Richard Kemmerer and Giovanni Vigna. Intrusion Detection: A Brief History and Overview. *IEEE Computer*, pages 27–30, April 2002. Special publication on Security and Privacy.

[58] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[59] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[60] Shijin Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. In *International Conference on Security and Privacy in Communication Networks (Securecomm)*, September 2008.

[61] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *Internet Measurement Comference*, pages 234–247, 2003.

[62] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–293, Oakland, CA, May 2002. IEEE Press.

[63] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS 2007*, pages 155–164.

[64] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM*, September 2006.

[65] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS 2006*, pages 81–92.

[66] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM*, pages 219–230, 2004.

[67] Wenke Lee, João B. D. Cabrera, Ashley Thomas, Niranjan Balwalli, Sunmeet Saluja, and Yi Zhang. Performance adaptation in real-time intrusion detection systems. In *RAID*, Zurich, Switzerland, October 2002.

[68] Harry Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[69] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Trans. on Embedded Computing Sys.*, 3(3):614–633, 2004.

[70] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *Recent Advances in Intrusion Detection RAID*, 2009.

[71] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. In *In Proceedings of INFOCOM 2000*, pages 1381–1390. IEEE, 2000.

[72] Evangelos P. Markatos, Spyros Antonatos, Michalis Polychronakis, and Kostas D. Anagnostakis. Exclusion-based signature matching for intrusion detection. In *IASTED International Conference on Communications and Computer Networks (CCN 2002)*, pages 146–151, October 2002.

[73] Ellen Messmer. Hacker alert. *Network World*, September 1999.

[74] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[75] Robert Muth and Udi Manber. Approximate multiple string search. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, number 1075, pages 75–86, Laguna Beach, CA, 1996. Springer-Verlag, Berlin.

[76] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[77] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[78] Gonzalo Navarro and Mathieu Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *J. Exp. Algorithmics*, 5:4, 2000.

[79] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.

[80] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[81] Nimda worm, cert advisory ca-2001-26, 2001. `http://www.cert.org /advisories /CA-2001-26.html`.

[82] Marc Norton. Optimizing pattern matching for intrusion detection. Available at `http://www.idsresearch.org/`, 2004.

[83] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *23rd Annual International Cryptology Conference (CRYPTO)*, 2003.

[84] Vern Paxson. The flex fast scanner generator, 1995. Available at `http://flex.sourceforge.net/`.

[85] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *Computer Networks*, volume 31, pages 2435–2463, 1999.

[86] Vern Paxson. Defending against network IDS evasion. In *Recent Advances in Intrusion Detection (RAID)*, 1999.

[87] PCRE: The perl compatible regular expression library. `http://www.pcre.org`.

[88] Thomas Ptacek and Timothy Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. In *Secure Networks, Inc.*, January 1998.

[89] Thomas Reps. "Maximal-munch" tokenization in linear time. *ACM Transactions on Programming Languages and Systems*, 20(2):259–273, 1998.

[90] Daniel J. Roelker. HTTP IDS evasions revisited. Available at `http://www.idsresearch.org/`, 2003.

[91] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference*. USENIX, 1999.

[92] Shai Rubin, Somesh Jha, and Barton Miller. Automatic generation and analysis of nids attacks. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, pages 28–38, December 2004.

[93] Shai Rubin, Somesh Jha, and Barton Miller. Language-based generation and evaluation of nids signatures. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2005.

[94] Shai Rubin, Somesh Jha, and Barton P. Miller. Protomatching network traffic for high throughputnetwork intrusion detection. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 47–58, New York, NY, USA, 2006. ACM Press.

[95] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Usenix Security Symposium*, August 1999.

[96] Umesh Shankar and Vern Paxson. Active mapping: Resisting nids evasion without altering traffic. In *IEEE Symp. on Security and Privacy*, May 2003.

[97] Reetinder Sidhu and Viktor Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[98] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.

[99] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 89–98, December 2006.

[100] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008.

[101] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, August 2008.

[102] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM Conference on Computer and Communications Security (CCS)*, Oct. 2003.

[103] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Internet Measurement Conference*, pages 68–81, 2004.

[104] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Int. Conf. on Field Programmable Logic and Applications*, sep. 2003.

[105] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *FCCM*, April 2004.

[106] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA*, June 2005.

[107] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005.

[108] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[109] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004*, pages 333–340.

[110] George Varghese. *Network Algorithmics, An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2005.

[111] Tim Vermeiren, Eric Borghs, and Bart Haagdorens. Evaluation of software techniques for parallel packet processing on multi-core processors. In *IEEE Consumer Communications and Networking Conference*, January 2004.

[112] David Wagner and Drew Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, May 2001.

[113] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, August 2004.

[114] Bruce W. Watson. The performance of single and multiple keyword pattern matching algorithms. In *Proceedings of the Third South American Workshop on String Processing*, Recife, Brazil, August 1996.

[115] Jeff Wilson. Network security market to pass $5 billion mark in 2007. `http://www.infonetics.com/`, March 2007.

[116] S. Wu and Udi Manber. A fast algorithm for multi-pattern searching. TR 94-17, Department of Computer Science, University of Arizona, 1994.

[117] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *14th USENIX Security Symposium*, Baltimore,Maryland, August 2005.

[118] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, 2006.