Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata

Randy Smith   Cristian Estan   Somesh Jha   Shijin Kong
Computer Sciences Department, University of Wisconsin-Madison
{smithr,estan,jha,krobin}@cs.wisc.edu

ABSTRACT

Deep packet inspection is playing an increasingly important role in the design of novel network services. Regular expressions are the language of choice for writing signatures, but standard DFA or NFA representations are unsuitable for high-speed environments, requiring too much memory, too much time, or too much per-flow state. DFAs are fast and can be readily combined, but doing so often leads to state-space explosion. NFAs, while small, require large per-flow state and are slow.

We propose a solution that simultaneously addresses all these problems. We start with a first-principles characterization of state-space explosion and give conditions that eliminate it when satisfied. We show how auxiliary variables can be used to transform automata so that they satisfy these conditions, which we codify in a formal model that augments DFAs with auxiliary variables and simple instructions for manipulating them. Building on this model, we present techniques, inspired by principles used in compiler optimization, that systematically reduce runtime and per-flow state. In our experiments, signature sets from Snort and Cisco Systems achieve state-space reductions of over four orders of magnitude, per-flow state reductions of up to a factor of six, and runtimes that approach DFAs.

Categories and Subject Descriptors: C.2.0 [Computer Communication Networks]: General - Security and protection (e.g., firewalls)

General Terms: Algorithms, Performance, Security

Keywords: regular expressions, signature matching, deep packet inspection, XFA

1. INTRODUCTION

Network devices are increasingly using packet content for processing incoming or outgoing packets. *Deep packet inspection*, as the process is called, arises as networks incorporate increasingly sophisticated services into their infrastructure. Such services use application-specific data found in packet payloads, for example, to make routing decisions [17], to block or rate-limit unwanted traffic [11,18], to perform intrusion detection, and to provide quality of service [25].

To keep up with line speeds, signatures must be combined and matched simultaneously in a single pass over the input. String-based signatures, initially popular, have a fast multi-pattern matching algorithm [1, 35] but limited expressivity. Currently, vulnerability [6, 36], session [32], and intrusion detection [26, 28] signatures commonly use the full capabilities of regular expressions, which are highly expressive and compact. Regular expressions are typically implemented as either deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). Like strings, DFAs are fast and can be readily combined. However, for many common signatures their combination exhibits an explosion in the state space. On the other hand, NFAs are very succinct but have a slow matching procedure. Thus, 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.

Auxiliary variables can be used to reduce the memory requirements of an automaton. This approach, common to software verification [2] and model checking [8], associates one or more variables with an automaton and uses them to track matching state more compactly than explicit states alone can. But, prior techniques for including these variables are ad-hoc, and typical models that incorporate them are not designed for high speed packet inspection. For example, using variables to influence transitions via guards can be expensive at runtime, and their automata difficult to combine. Also, large signature sets need many variables (one of our test sets uses almost 200) in order to reduce the state space to a manageable size. Maintaining and manipulating all these variables can affect performance significantly. Incorporating auxiliary state variables is a step in the right direction, but to the best of our knowledge there is no general model that allows for the systematic construction, combination, and analysis that is suitable and necessary for high-speed packet inspection.

The high-level goal of our work is to make deep packet inspection practical at high speeds. We begin with a preliminary first-principles characterization of state-space explosion. We describe, formally, why it occurs and give ideal conditions that eliminate it when satisfied. We then illustrate how auxiliary state variables can be used to "factor out" the components of automata that violate these conditions. When these conditions are met, automata can be freely combined without any state explosion. We employ 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. Variables cannot affect state transitions, but they can influence acceptance. The model is fully deterministic and yields combination and matching algorithms that are straightforward extensions to those for DFAs. This characterization of state space explosion and the resulting model constitute the first main contribution of this work.

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 leads to large per-flow state and processing times. Taking inspiration from common principles used in optimizing compilers, 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. Altogether, these optimizations increase performance by up to an order of magnitude and decrease per-flow state by up to a factor of six. These systematic optimizations are the second main contribution of this work.

Our evaluation uses regular expression signature sets obtained from the Snort [28] and Cisco intrusion prevention systems. We also compare against two other recently proposed techniques, Multiple DFAs (mDFAs) [37] and D^{2}FAs [20]. Compared to standard DFAs, XFAs yield state space reductions in excess of four orders of magnitude in some cases. When optimizations are employed, performance approaches that of DFA matching. In all cases, XFAs are both smaller and faster than other evaluated techniques.

Finally, increasing line speeds put ever-increasing pressure on designers to move data-plane functionality into hardware. XFAs require no hardware support or assistance to operate, but there are no restrictions precluding them from hardware deployment either. As a third contribution, then, we present a hardware architecture for efficient execution of XFAs.

This paper is organized as follows: after the related work, we discuss state-space explosion in Section 3 followed by a description of our model in Section 4. Section 5 discusses optimization and Section 6 gives experimental results. In Section 7 we present a possible hardware implementation and Section 8 concludes.

2. RELATED WORK

String-based signatures were initially popular for packet inspection and still find some use today. Classic multi-pattern algorithms such as Aho-Corasick [1] perform matching in $O(1)$ time yet grow linearly in the number of signatures, thereby avoiding state explosion. Many alternatives and enhancements have since been proposed [10, 22, 33–35] for use in both software and hardware. In adversarial settings, string-based signatures are not sufficient to withstand attack techniques such as evasion [12, 27, 29] and mutation [16], and modern systems have migrated towards signatures that use the full power of regular expressions [6, 28, 32, 36].

Unfortunately, regular expressions often do not scale when combined. DFAs often exhibit polynomial or exponential growth in the state space, and NFAs are unacceptably slow. Using DFAs (for their speed) as a starting point, many techniques have recently been proposed to reduce their memory footprint. Yu *et al.* [37] propose combining signatures into a group of DFAs using greedy heuristics to determine which signatures should be combined together so that a supplied upper memory bound is not exceeded. The technique does reduce the total memory footprint, but for complex signature sets the number of resulting DFAs can be large.

D^{2}FAs [20, 21] reduce memory by compressing transitions at the cost of longer matching times. This approach identifies states with similar transition tables, replacing them with small tables containing only the transitions that are distinct at each state. During matching, default transitions are followed from state to state until a compressed table entry is found that corresponds to the current input symbol. In [4], Becchi and Crowley propose improvements that bound the number of default transitions followed. These techniques are orthogonal to ours; we can incorporate them into our own work to achieve further memory reduction.

Becchi and Cadambi [3] propose state merging, which moves partial state information from states themselves into labeled transitions allowing states to be combined. The authors report memory savings of up to one order of magnitude. Kumar *et al.* [19] present a set of 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 blowup, whereas we begin with a formal characterization of blowup and then show how auxiliary variables can eliminate it. In addition, the interaction between states, variables, and transitions is not formalized; we provide an extensible formal model.

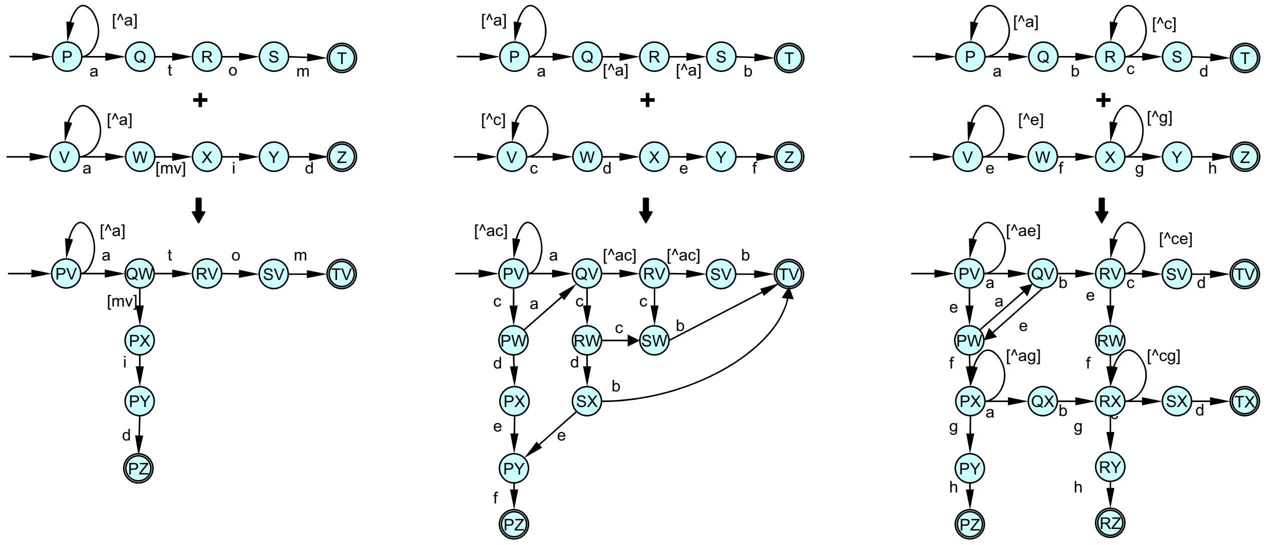
The Bro NIDS [26] uses regular expressions for signatures specified in the context of parsed protocol fields. One advantage of this approach is that signatures are simpler, which in principle leads to smaller automata. Nevertheless, Bro still experiences state explosion and uses on-the-fly determinization [32] of regular expressions at some runtime cost. In contrast, the Snort NIDS [28] uses NFA matching guarded by a string-based multi-pattern prefilter [1]. Matching is fast in the common case since most payloads never pass the prefilter. However, malicious traffic can be used to invoke NFA matching and induce severe slowdowns [9, 30].

Clark and Schimmel [7] and Brodie *et al.* [5] have proposed hardware-based techniques that use multibyte symbols for transitions along with other optimizations. Our techniques are more general and can be applied equally to hardware, software, or FPGA environments.

Finally, we first introduced the XFA paradigm in previous work [31]. That work gave an informal characterization of state-space explosion and focused on algorithms for constructing XFAs from regular expressions. Experimental results were promising but preliminary. In contrast, as described above, this work formalizes state-space explosion, refines the XFA model, and focuses on optimizations necessary for high-speed inspection.

3. UNDERSTANDING STATE 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



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

Figure 1: 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).

that they satisfy these conditions and eliminate such explosion directly. This characterization provides the underlying foundation that motivates our work.

3.1 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:

- 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 $y \in \Sigma^\star$;
- 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:

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

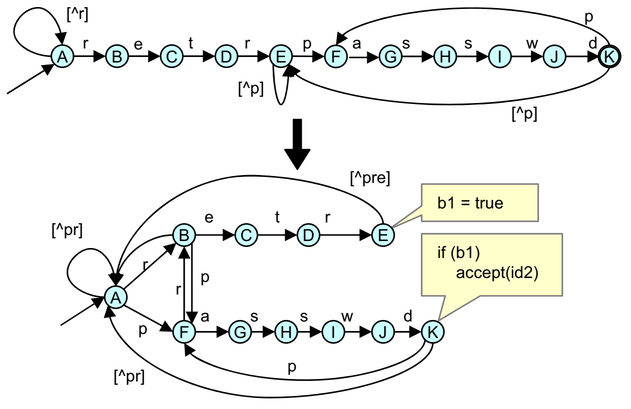
For an unambiguous automaton D, the first three conditions ensure that all strings in the language accepted by D are of the form $.* x_f$ where $f \in F$. The fourth condition ensures that there are no superfluous states that do not advance matching progress toward acceptance. Note that ambiguity is 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.

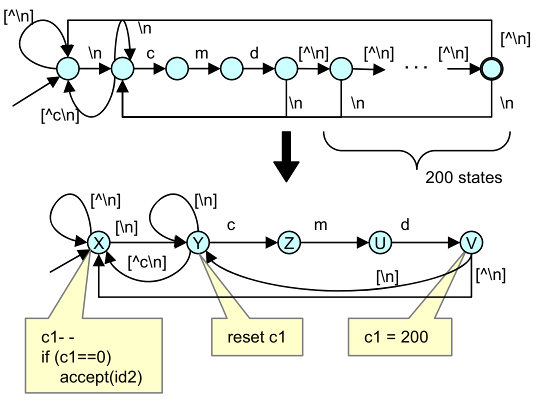
3.2 Combination and State Explosion

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 1. In this figure and in most others, we show all states but for clarity eliminate many transitions. In Figure 1a, 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 1b 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





(a) Adding a bit to `/retr.*passwd/` **(b)** Adding a counter to `/\ncmd[^\n]{200}/`

Figure 2: Auxiliary variables can transform automata so that they do not blow up when combined.

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

Figure 1c depicts the case in which both regular expressions contain a Kleene closure (`.*`). 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.

3.3 Eliminating Ambiguity Through 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 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 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 3.1, 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.

We define state-space explosion formally as a pairwise phenomenon occurring 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 can be combined without any appreciable blowup (Figure 1a, 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 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 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 altogether.

As an example, consider `/retr.*passwd/`, whose DFA 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 2a. 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.

As another example, consider `/\ncmd[^\n]{200}/`, which is characteristic of vulnerability signatures guarding against buffer overflows. The DFA for this expression contains 200 ambiguous states whose sole purpose is to count the distance in the input from the sequence `\ncmd` in which a newline is not observed. 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 2b. The state variable replaces the 200 counting states, leading to a sharp reduction in the size of the automaton. Most importantly, the 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 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. In general, the amount of auxiliary state we introduce and its effect on the underlying automaton is very fluid. At one extreme, standard DFAs require no auxiliary state but very commonly suffer from state explosion when combined. 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.

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.

4. A FORMAL MODEL

We formally define a (state-based) XFA as follows.

Definition 1. *A (state-based) extended finite automaton is a 7-tuple $(Q, V, \Sigma, \delta, U, (q_0, v_0), F)$, where*

- *Q is the set of states, Σ is the set of inputs (input alphabet), $\delta : Q \times \Sigma \to Q$ is the transition function,*
- *V is a finite set of variable values,*
- *$U : Q \times V \to V$ is the per-state update function which defines how the data value is updated on states,*
- *(q_0, v_0) is the initial configuration which consists of an initial state q_0 and an initial variable value v_0,*
- *$F \subseteq Q \times V$ is the set of accepting configurations.*

Informally, XFAs generalize standard DFAs to include a finite set of possible variable values along with programs attached to states that manipulate the variable during matching. Start states, accepting states, and transient "current" states each generalize to include a variable value along with a state. In principle, all auxiliary state is maintained using a single (possibly composite) variable, although in practice we can have many distinct variables without any loss of generality. Note also that according to the definition, a standard DFA is simply a XFA whose sets of variable values and update functions are each empty.

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. This distinct separation 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. On the other hand, the use of explicit instructions provides fertile ground for systematically applying optimizations and analysis techniques common to compiler construction.

In our earlier work [31], we used an XFA definition that associated instructions with edges rather than states. Edge-based XFAs are equivalent to state-based XFAs semantically, although state-based XFAs require more states.[1] On the other hand, combination, matching, and optimization algorithms are more efficient for state-based XFAs.

4.1 XFA Construction

We briefly summarize XFA construction from regular expressions and refer the interested reader to other work [31] for a full discussion. The steps for compiling a regular expression to an XFA are similar to those for transforming to a DFA [15]: parsing and constructing a non-deterministic automaton, determinization, and reduction. The key difference is the inclusion of an 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.

Conversion to state-based XFAs is straightforward. 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. We then move these instructions to the corresponding copies of S and retarget the incoming transitions appropriately.

4.2 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 Section 3, not in the combination process itself.

Combination of XFAs, given in Algorithm 1, is a straightforward extension to DFA combination. Each state in the combined automaton corresponds to a pair of states from the first and second input automata. In line 5, the combined start state initializes a worklist which is added to by each newly created state (line 15). 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.

[1]This is analogous to the differences between the classic Mealy and Moore automata [15].

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

Algorithm 1. XFA combination. Instructions are copied from source states to "paired" states.

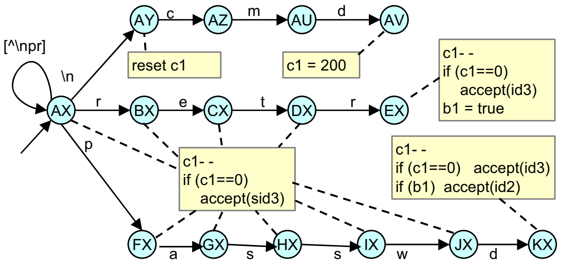


Figure 3: Automata produced by combining the XFAs in Figures 2a and 2b.

Lines 13 and 14 add instructions to combined states from their original counterparts. For each state $q = \langle s, t \rangle$ in a combined automaton c, we simply copy the instructions from s and t into q. The correctness of this follows from the fact that entering combined state q when matching is equivalent to entering states s and t simultaneously, implying that the instructions in both s and t need to be executed. Figure 3 shows the results of combining the XFAs in Figures 2a and 2b. For convenience, 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 requires 2194 states.

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.

4.3 Matching to Input

Deep packet inspection performs continuous matching, in which acceptance is indicated whenever an accepting state is reached, not just at the end of the input. XFA matching, given in Algorithm 2, simply extends this model by executing programs attached to states when they are reached. Note that since acceptance conditions are implemented as instructions, no special acceptance tests are needed. A raised alert is processed identically to any other instruction.

```
    MATCH(XFA M, uchar* buf, int len):
1   state curState = M.start
2   execInstrs ( curState.instrs)
3   for i ← 0 to len do
4       curState = curState.nextState(buf [i])
5       execInstrs ( curState.instrs)
```

Algorithm 2. Algorithm to match an XFA against an input buffer.

5. OPTIMIZATION

The conditions and model in Sections 3 and 4 allow XFAs to be independently constructed and easily combined without blowup, but this flexibility comes at a cost: in combined 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 [24], we present three distinct optimizations: exploiting runtime information 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.

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 2b is combined with other automata, the decrement and test instructions get replicated to most of the states, as shown in Figure 3, 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.

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

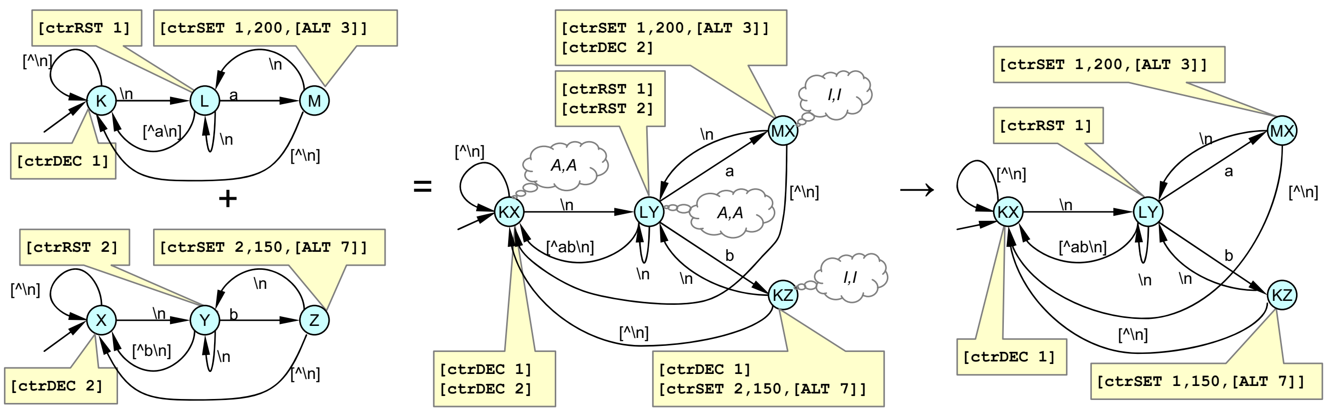


Figure 4: The counter minimization process applied to automata for signatures /\na[^\n]{200}/ and /\nb[^\n]{150}/. The optimization results in the elimination of one of the original counters.

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 their actual format, 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.

We illustrate with the running example in Figure 4. 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 4 (the "clouds" belong to a later stage). In the end, optimization finds that the two counters in the combined automaton are independent and reduces them to one counter.

5.2.1 Dataflow Analysis

As informally described in Section 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 2. *Let Q be the set of states containing a* `set` *operation for counter C. Then, C is active at state S if there is at least one sequence of input symbols forming a path of states from a state in Q to S in which no state in the path contains a* `reset` *operation for C. Otherwise C is inactive.*

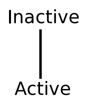


Figure 5: The value lattice that orders abstract counter facts. *Inactive* is the initial value.

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 [24]. 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 [24].

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 Figure 5. 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:

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

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 $\mathcal{C}$

		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

Figure 6: Counter compatibility matrix, specifying which counter operations are compatible at a state and the surviving operation.

```
    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; // fall to outer do...while loop
12  while compatible = true
```

Algorithm 3. Counter compatibility. Two counters are equivalent and can be reduced to one if they are compatible at each state.

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 4, 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.

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 contains 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

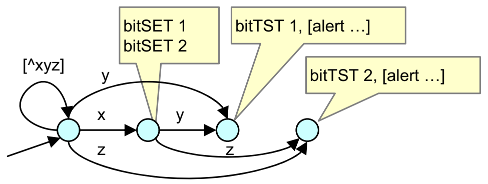


Figure 7: Combined automata for `/x.*y/` and `/x.*z/`. A stronger dataflow analysis can eliminate a bit.

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 3 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 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.

This optimization completes quickly, despite the $O(n^3)$ runtime of the dataflow and compatibility analysis. With one exception (which contained 172 bits) the procedure completes in less than one minute per test set. Stronger dataflow analyses can be used to identify further reduction opportunities that this analysis misses. For example, Figure 7 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.

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 [14]: 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]`,

	Num	# States		Variables		Instrs per state		Aux memory (bytes)	
Rule set	Sigs	DFA	XFA	# bits	# ctrs	max	avg	variables	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 1: Combined automata for several protocols, before and after optimization.

	No-Opt		Opt 1	
		Inst/	ctrs	Inst/
Rule set	ctrs	state	gen/imp	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		Opt 2	
Rule set	bits	ctrs	bits	ctrs
Snort FTP	8	8/38	8	2/2
Snort SMTP	11	10/21	6	4/6
Snort HTTP	172	0/15	171	0/6
Cisco FTP	11	0/12	10	0/3
Cisco SMTP	8	0/3	8	0/2
Cisco HTTP	13	0/10	12	0/2

(b) Opt 2: Coalesce independ. vars

	Opt 2		Opt 3	
	Inst/State		Inst/State	
Rule set	max	avg	max	avg
Snort FTP	7	0.81	5	0.66
Snort SMTP	21	0.73	21	0.69
Snort HTTP	16	1.09	11	1.03
Cisco FTP	7	0.46	4	0.33
Cisco SMTP	9	0.33	7	0.28
Cisco HTTP	8	0.55	7	0.42

(c) Opt 3: Instruction merging

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

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. EXPERIMENTAL EVALUATION

6.1 Test sets and Optimizations

We evaluated XFAs on FTP, SMTP, and HTTP signatures from Snort [28] and Cisco Systems. We first produced individual state-based XFAs from regular expressions using the techniques described earlier. We then combined the signatures in each set using Algorithm 1. Combination is fast; the Snort HTTP set took the most time to combine at just over seven minutes whereas all other sets required less than a minute to combine. For comparison purposes, we built standard DFAs for each of the regular expressions and combined these per protocol as well.

Table 1 summarizes properties of the combined XFAs. 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 5 in consecutive order and show relevant results in Tables 2a, 2b, and 2c. In Table 2a 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 2b shows the effect of 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 2c 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 1 summarizes the cumulative effect of the optimizations in the bottom row of each set. Figure 8 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.

6.2 Memory Usage and Performance

In the second set of experiments 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

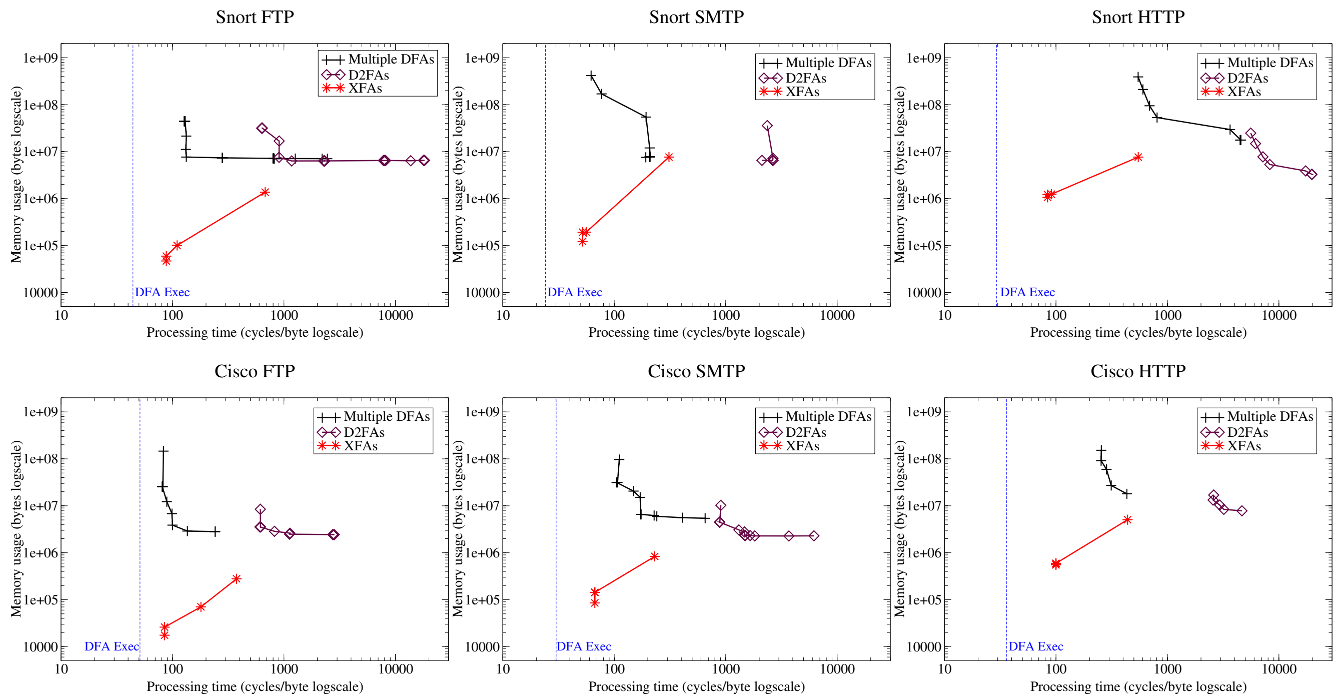


Figure 9: Memory versus run-time trade-offs for mDFAs, D2FAs, and XFAs.

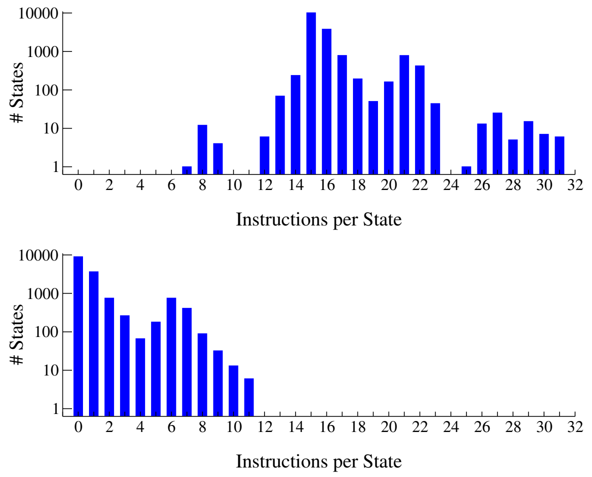


Figure 8: Instructions per state for Snort HTTP, before (top) and after (bottom) optimization.

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 also evaluate two other recently proposed techniques, multiple DFAs (mDFAs) [37] and D^{2}FAs [20], which we briefly described in Section 2. We implemented the mDFA algorithm and supplied memory ceilings ranging from 4K total states to 512K total states, which produced groups of combined automata for each setting. During runtime we matched mDFAs by modifying our matching code to maintain multiple state pointers. For the D^{2}FA evaluation, we applied the D^{2}FA edge compression algorithm to each combined DFA in each mDFA group. The D^{2}FA 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.

Execution time tests were performed on 10 GB traces captured on the link between a university campus and a departmental network at varying times. We performed all experiments on a 3.0 GHz Pentium 4 Linux workstation. 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 9 gives space-time comparisons for each test set. 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. mDFAs trace out a curve showing the trade-offs between memory usage and processing time. D^{2}FAs build on mDFAs and follow a similar curve with a reduced memory footprint. For XFAs, we plot the combined automata along with the cumulative effects of each optimization, leading toward the lower left corner. 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.

6.3 Per-flow State

Table 3 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

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

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 1. Reductions in per-flow state for XFAs are a direct result of optimization 2. As Table 1 indicates, per-flow state can be reduced by up to a factor of six. In Table 3, 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.

7. MIGRATING TO HARDWARE

We present here a preliminary chip design that can perform signature matching at 10 Gbps using XFAs with up to 24,576 states. The chip does not perform reassembly or packet classification, and it also uses techniques for compressing the transition tables. It has 8 packet processing pipelines each consisting of three loosely coupled stages: a DFA engine, a program lookup engine, and a processing element. We expect a clock speed close to 500MHz.

DFA engines consume one byte of input every two cycles and use a shared multiport SRAM to store transition tables. Each of the 2^{20} entries of this SRAM stores a 15-bit state identifier. The DFA engines implement a table compression algorithm based on ideas from the existing literature [4, 20, 21]. For brevity we omit the exact description of the algorithm, but we note that for each payload byte, the DFA engine performs a single access to the shared transition table, and up to two accesses to a private SRAM with 24,576 60-bit words.

Program lookup engines receive from their DFA engine a sequence of states visited and produce a sequence of programs to be executed. Since most states have no programs associated with them, the output of this engine includes only the addresses of the non-empty programs, and it pairs with each program the offset in the input to which it corresponds. Each engine uses a private SRAM with 24,576 15-bit words representing the addresses of the program associated with each state. At the output of each engine is a large queue of up to 32 addresses which provides decoupling between the DFA engines and the PEs.

Processing elements execute the programs associated with the traversed states. These programs are stored in a local 64KB instruction memory. For states whose programs are identical, a single copy needs to be stored. While the instruction memory of one processing element may not be large enough to store the programs for XFAs for all protocols, it can easily store the programs for the largest one. Hence each processing element can handle a subset of the protocols we have signatures for. Each PE uses 32 16-bit registers holding the variables. The instruction set does not have memory access operations or branches. We use simple predication to make instructions conditional. All instructions are 2 bytes. The PE uses a simple 3-stage pipeline with one instruction issue per cycle. The only unusual part is the logic for determining the address of the instruction to fetch. It can be the next instruction, the first instruction of the next program in the queue, or the next program in the offset list. The offset list is a chain of 8 elements that maintain a sorted list with the offsets in the input at which various implicit counters fire, together with the addresses of the programs to be executed in response.

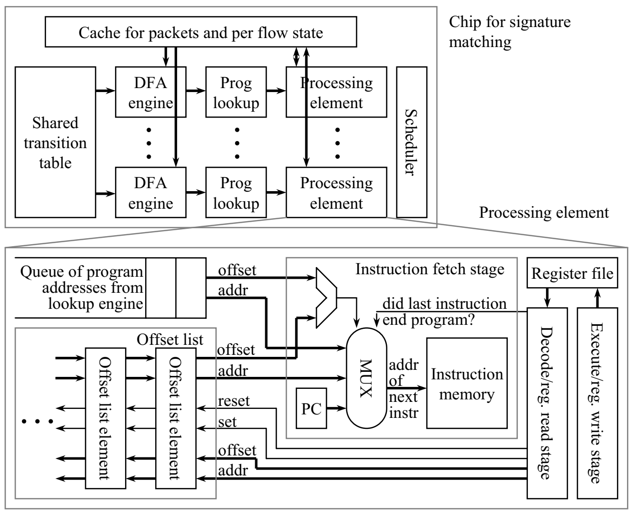


Figure 10: Chip that can do signature matching at 10Gbps using XFAs.

Switching between packets happens with the assistance of the scheduler. For the DFA engine, the switch only requires loading a new value for the current state. The processing element needs to write out the dirty registers and the offset list into the cache line holding the per flow state of the old packet, and read in the registers and offsets corresponding to the next one. To avoid slowing the PE down, we use two sets of registers and two offset lists so that while the PE is working with one packet, the variables for the previous one can be written out and those for the next one read in. A 256KB cache holds packets and their per flow state.

We estimate the chip size by comparing against the Niagara2 floorplan [23]. Our design uses 4.4MB total SRAM compared to Niagara2's 4MB of L2 cache. Our processing elements are much simpler than Niagara2's cores; it has many components not needed in our design and is more complex. Since Niagara2 uses $342mm^2$ in a 65nm technology, we estimate that our chip would use less than $200mm^2$.

8. CONCLUSION AND FUTURE WORK

The Big Bang Theory [13] 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. In this work, our running hypothesis is that the systematic use of auxiliary variables and optimizations provides a practical mechanism for deflating explosive DFAs.

In this paper 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.

Acknowledgments

This work is sponsored by NSF grants 0546585 and 0716538 and by a gift from the Cisco University Research Program Fund at Silicon Valley Community Foundation. We thank Karu Sankaralingam, George Varghese, and the anonymous reviewers for suggestions that improved this paper.

9. REFERENCES

[1] A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, June 1975.
[2] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. January 2002.
[3] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *IEEE Infocom 2007*.
[4] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS 2007*.
[5] B. Brodie, R., and D. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.
[6] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, May 2006.
[7] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE FCCM*, April 2004.
[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
[9] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, August 2003.
[10] S. Dharmapurikar and J. W. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Comm.*, 24(10):1781–1792, 2006.
[11] The Guardian. Trouble on the line. `http://technology.guardian.co.uk/weekly/story/0,,1747343,00.html`, 2006.
[12] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Usenix Security*, August 2001.
[13] S. W. Hawking. *A brief history of time. From the Big Bang to Black Holes*. Bantam Book, 1988.
[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann, 1996.
[15] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2006.
[16] Myles Jordan. Dealing with metamorphism. *Virus Bulletin Weekly*, 2002.
[17] C. Kachris and S. Vassiliadis. Design of a web switch in a reconfigurable platform. In *ANCS 2006*.
[18] P. Kapustka. Vonage complaining of VoIP blocking. `http://www.networkcomputing.com/channels/networkinfrastructure/60400413`, 2005.
[19] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS 2007*, pages 155–164.
[20] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM*, September 2006.
[21] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS 2006*, pages 81–92.
[22] R. Liu, N. Huang, C. Chen, and C. Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Trans. on Embedded Computing Sys.*, 3(3):614–633, 2004.
[23] H. McGhan. Niagara 2 opens the floodgates. In *Microprocessor Report*, November 2006.
[24] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
[25] M. Neider. Deep packet inspection: A service provider's solution for secure VoIP. *VoIP Magazine*, Oct 2005.
[26] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Computer Networks*, volume 31, pages 2435–2463, 1999.
[27] T. Ptacek and T. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. In *Secure Networks, Inc.*, January 1998.
[28] M. Roesch. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conference*. USENIX, 1999.
[29] U. Shankar and Vern Paxson. Active mapping: Resisting nids evasion without altering traffic. In *IEEE Symp. on Security and Privacy*, May 2003.
[30] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a NIDS. In *ACSAC 2006*, pages 89–98.
[31] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008.
[32] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, Oct. 2003.
[33] 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.
[34] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA*, June 2005.
[35] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004*, pages 333–340.
[36] H. J. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, August 2004.
[37] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS 2006*.