XFA: Faster Signature Matching With Extended Automata

Randy Smith Cristian Estan Somesh Jha
University of Wisconsin–Madison
{smithr,estan,jha}@cs.wisc.edu

Abstract

Automata-based representations and related algorithms have been applied to address several problems in information security, and often the automata had to be augmented with additional information. For example, extended finite-state automata (EFSA) augment finite-state automata (FSA) with variables to track dependencies between arguments of system calls. In this paper, we introduce extended finite automata (XFAs) which augment FSAs with finite scratch memory and instructions to manipulate this memory. Our primary motivation for introducing XFAs is signature matching in Network Intrusion Detection Systems (NIDS). Representing NIDS signatures as deterministic finite-state automata (DFAs) results in very fast signature matching but for several classes of signatures DFAs can blowup in space. Using nondeterministic finite-state automata (NFA) to represent NIDS signatures results in a succinct representation but at the expense of higher time complexity for signature matching. In other words, DFAs are time-efficient but space-inefficient, and NFAs are space-efficient but time-inefficient. In our experiments we have noticed that for a large class of NIDS signatures XFAs have time complexity similar to DFAs and space complexity similar to NFAs. For our test set, XFAs use 10 times less memory than a DFA-based solution, yet achieve 20 times higher matching speeds.

1. Introduction

Automata-based representations have found several applications in information security. In some of these applications automata are augmented with additional information. For example, extended finite state automata (EFSA) augment finite-state automata (FSA) with uninterpreted variables and are very useful for capturing dependencies between system calls [23]. A similar representation is used in STATL [8] to track dependencies between events. In this paper our primary goal is to improve the time and space efficiency of signature matching in network intrusion detection systems (NIDS).[1] To achieve our goal we introduce *extended finite automata (XFAs)* which augment traditional FSAs with a finite scratch memory used to remember various types of information relevant to the progress of signature matching. Since an XFA is an FSA augmented with finite scratch memory, it still recognizes a regular language, albeit more efficiently than an FSA. We demonstrate that representing signatures in NIDS as XFAs significantly improves time and space efficiency of signature matching. We also present algorithms for manipulating XFAs, such as constructing XFAs from regular expressions and combining XFAs.

In the past signatures in NIDS were simply keywords, which resulted in extremely efficient signature-matching algorithms. The Aho-Corasick algorithm [1], for example, finds all keywords in an input in time linear in the input size. Because of the increasing complexity of attacks and evasion techniques [19], NIDS signatures have also become complex. Therefore, current techniques for generating different types of signatures, such as vulnerability [4, 31] or session [21, 26] signatures, generate signatures that use the full power of regular expressions. Representing NIDS signatures as deterministic finite-state automata (DFAs) results in a time-efficient signature-matching algorithm (each byte of the input can be processed in $O(1)$ time), but for certain regular expressions DFAs blow up in space. Nondeterministic finite-state automata (NFAs) are succinct representations for regular expressions, but the time complexity of the signature-matching algorithm increases, *i.e.*, each byte of the input can take $O(m)$ time to process, where m is the number of states in the NFA. Therefore, *DFAs are time-efficient but space-inefficient, and NFAs are space-efficient but time-inefficient*. If signatures are rep-

[1] A NIDS that uses misuse detection matches incoming network traffic against a set of signatures. This functionality of a NIDS is called signature matching.

resented as XFAs, the scratch memory has to be updated while processing some input bytes. However, since the scratch memory is very small it can be updated very efficiently (especially if it is cached). Moreover, for many signatures XFAs are also a very succinct representation. For a large class of NIDS signatures *XFAs have time complexity similar to DFAs and space complexity similar to or better than NFAs*. The larger the scratch memory we can use, the smaller the space complexity of the required automaton (but the time complexity of the operations for updating the scratch memory may increase).

Recall that XFAs augment traditional FSAs with a small scratch memory which is used to remember various types of auxiliary information. We will explain the intuition behind XFAs with a short example. Consider n signatures s_i ($1 \leq i \leq n$) where $s_i = .*k_i.*k_i'$ (k_i and k_i' are keywords or strings). Note that s_i matches an input if and only if it contains a keyword k_i followed by k_i'. DFA D_i for signature s_i is linear in the size of the regular expression $.*k_i.*k_i'$. However, if the keywords are distinct, the DFA for the combination of the signatures $\{s_1, \cdots, s_n\}$ is exponential in n. The reason for this state-space blowup is that for each i ($1 \leq i \leq n$) the DFA has to "remember" if it has detected the keyword k_i in the input processed so far. The XFA for the set of signatures $\{s_1, \cdots, s_n\}$ maintains a scratch memory of n bits $(b_1, \cdots, b_n)$, where bit b_i remembers whether it has seen the keyword k_i or not. The space complexity of the XFA is linear in n and the time complexity is $O(n)$ because the bits have to be potentially updated after processing each input symbol, but this worst case happens only if n of the keywords overlap in specific ways. For the actual signatures we evaluated, the time complexity for XFAs is much closer to DFAs. Further, the XFA for an individual signature s_i is not much smaller than the corresponding DFA, but the combined XFA for the entire signature set is much smaller than the combined DFA. The reason is not that we use a special combination procedure, but that the "shape" of the automata the XFAs are built on does not lead to blowup. We discuss this example in detail in Section 3.1.

This paper makes the following contributions:

- We introduce XFAs, which augment an FSA with a small scratch memory to alleviate the state-space explosion problem characteristic to DFAs recognizing NIDS signature sets (see Section 3).
- We provide a general procedure for building XFAs from regular expressions that handles complex expressions used in modern NIDS (see Section 4).
- We perform a case study that builds XFAs for a real signature set, and we demonstrate that the matching performance and memory usage of XFAs is better than that of solutions based on DFAs which must resort to multiple automata to fit into memory (see Section 5). Even with a memory budget 10× larger than that used for XFAs, DFA-based solutions require 67 automata and have throughput 20× lower.

2. Related work

String matching was important for early network intrusion detection systems as their signatures consisted of simple strings. The Aho-Corasick [1] algorithm builds a concise automaton (linear in the total size of the strings) that recognizes multiple such signatures in a single pass. Other software [3, 6, 9] and hardware solutions [15, 27, 29] to the string matching problem have also been proposed. However, evasion [11, 19, 24], mutation [13], and other attack techniques [22] require signatures that cover large classes of attacks but still make fine enough distinctions to eliminate false matches. Signature languages have thus evolved from simple exploit-based signatures to richer session [21, 26, 32] and vulnerability-based [4, 31] signatures. These complex signatures can no longer be expressed as strings, and regular expressions are used instead.

NFAs can compactly represent multiple signatures but may require large amounts of matching time, since the matching operation needs to explore multiple paths in the automaton to determine whether the input matches any signatures. In software, this is usually performed via backtracking (which opens the NFA up to serious algorithmic complexity attacks [7]) or by maintaining and updating a "frontier" of states, both of which can be computationally expensive. However, hardware solutions can parallelize the processing required and achieve high speeds. Sidhu and Prasanna [25] provide an NFA architecture that updates the set of states during matching efficiently in hardware. Further work [5, 28] has improved on their proposal, but for software implementations the processing cost remains significant.

DFAs can be efficiently implemented in software, although the resulting state-space explosion often exceeds available memory. Sommer and Paxson [26] propose on-the-fly determinization for matching multiple signatures, which keeps a cache of recently visited states and computes transitions to new states as necessary during inspection. This approach can be subverted by an adversary who can repeatedly invoke the expensive determinization operations. Yu *et al.* [33] propose combining signatures into multiple DFAs instead of one DFA, using simple heuristics to determine which signatures should be grouped together. The procedure does reduce the total memory footprint, but for complex sig-

nature sets the number of resulting DFAs can be large. The cost of this approach is increased inspection time, since payloads must now be scanned by multiple DFAs. The D^2FA technique [14] performs edge compression to reduce the memory footprint of individual states. It stores only the difference between transitions in similar states, and in some sense, extends the string-based Aho-Corasick algorithm to DFAs. The technique does not address state space explosion and thus is orthogonal to our technique which focuses on reducing the number of states required. The two techniques could be combined to obtain further reductions in memory usage.

Prefilter-based solutions such as those used by Snort [20] can achieve good average-case performance. The pre-filter performs string matching on subparts of a signature, invoking the matching procedure for the full regular expression only when a subpart has been matched. Our preliminary results show that this approach is vulnerable to algorithmic complexity attacks. By sending traffic crafted to defeat Snort's pre-filter and to cause expensive regular expression processing, an attacker can slow it down by as much as a factor of 5000.

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 [23] 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.* [8] 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 [2] and hybrid automata [12], 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.

Space-time or time-memory tradeoff is a technique where the memory use can be reduced at the cost of slower program execution, or vice versa, 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 [10]. Time-space tradeoffs have also been explored in the context of attacks [16, 17]. We are not aware of existing work on time-space tradeoffs in the context of signature matching for NIDS.

3. Technical overview

We begin with a discussion of simple signatures illustrating how XFAs need much fewer states than DFAs, followed by an overview of the steps for compiling realistic signatures to XFAs suitable for NIDS use.

3.1. Reducing state space with XFAs

Recognizing a signature set with n signatures of the form $.*S_i.*S_i'$, where all S_i and S_i' are distinct strings, leads to state space blowup with DFAs. Figure 1 shows an example for the case where $n = 2$, $S_1 =$`ab`, $S_1' =$`cd`, $S_2 =$`ef`, and $S_2' =$`gh`. In the general case, for each of the n signatures, the combined DFA must "remember" whether it already found the first string in the input so that it "knows" whether to accept if it sees the second string. For example, in Figure 1 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 `ab` and `ef` have been seen. In general, to remember n independent bits of information, the DFA needs at least 2^n distinct states. 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 $O(nl2^n)$.

Figure 2 shows the same signatures as in Figure 1 when DFAs are replaced with XFAs. In this figure, the XFAs for `.*ab.*cd` and `*ef.*gh` each use a single bit of scratch memory that is manipulated by instructions attached to specific edges (depicted in the figure as callout boxes). During matching, these instructions are "executed" each time the corresponding transition is followed. For each signature of the form $.*S_i.*S_i'$, as long as S_i does not overlap with S_i', we can build XFAs like those in Figure 2 that uses a single bit of scratch memory.[2] This bit explicitly encodes whether S_i has appeared in the input so far, and the shape of the underlying automaton is very similar to that of the combined DFA recognizing $.*S_i$ and $.*S_i'$ independently.

[2] If S_i and S_i' overlap it is still possible to build an XFA recognizing the signature, but it will use more than one bit.

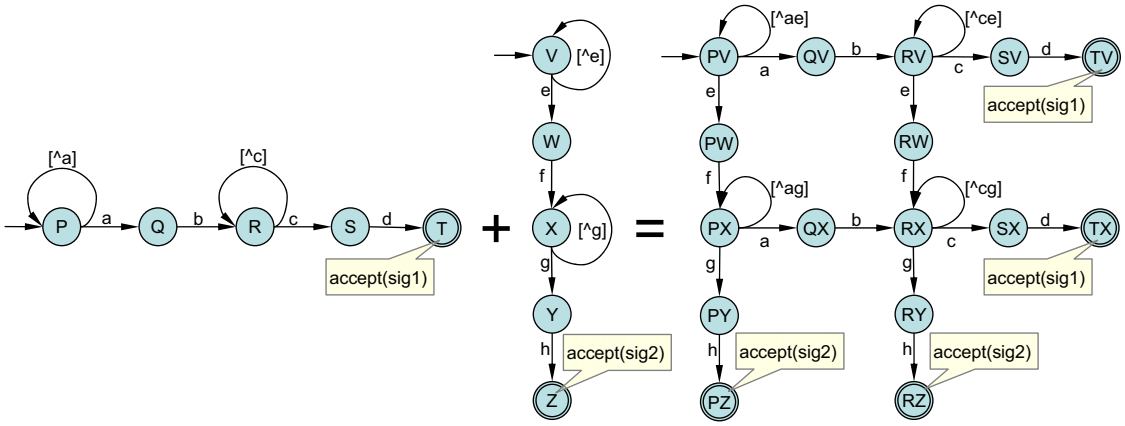

Figure 1. The DFA recognizing both `.*ab.*cd` and `.*ef.*gh` has state space blowup. For simplicity, we do not show some less important transitions.

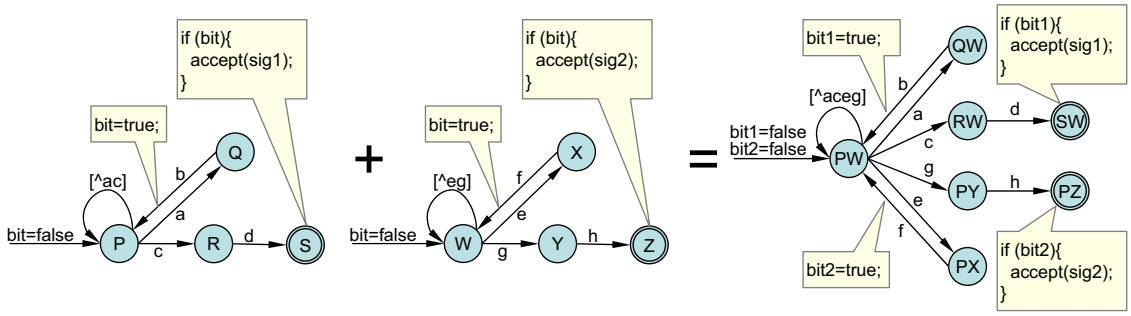

Figure 2. The XFA recognizing both `.*ab.*cd` and `.*ef.*gh` without state space blowup. For simplicity, we do not show some less important transitions.

The combined XFA for the entire signature set uses n bits and $O(nl)$ states. Thus by adding n bits of scratch memory we obtain a combined XFA approximately 2^n times smaller than the combined DFA. The initialization time goes up from $O(1)$ to $O(n)$ and, assuming that the strings in the signatures are not suffixes of each other, only a small constant is added to the worst-case per byte processing cost as at most one bit is updated for any given byte from the input.

Note that the presence of scratch memory has no influence on the shape of the underlying automaton for the combined XFA: the same process for combining DFAs is used for combining the underlying automata of XFAs. In reality, the combined XFA (Figure 2) is smaller than the combined DFA (Figure 1) because the automata structure in the source XFAs is different than for DFAs. When combined, these XFAs have benign interactions, just as with DFAs limited to string matching.

XFAs can provide large reductions in the number of states even when recognizing individual signatures. Figures 3 and 4 show the DFA and XFA, respectively, recognizing the language defined by $.\{n\}$ which consists of all strings of length n. Although no NIDS signatures have this exact form, signatures detecting buffer overflows use sequences of states similar to those in Figure 3 to count the number of characters that follow a given keyword. The minimal DFA for $.\{n\}$ needs $n+2$ states, whereas the XFA uses a single state and a counter. This counter is initialized to 0 and is incremented on every transition, signaling acceptance only when the value is n. Increment is defined so that once the counter reaches $n+1$ it remains at $n+1$. Thus the counter needs to take only $n+2$ values, requiring only

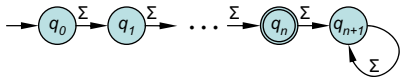

Figure 3. DFA recognizing $.\{n\}$.

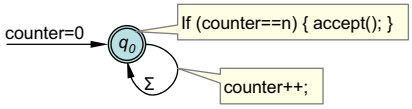

Figure 4. XFA recognizing $.\{n\}$.

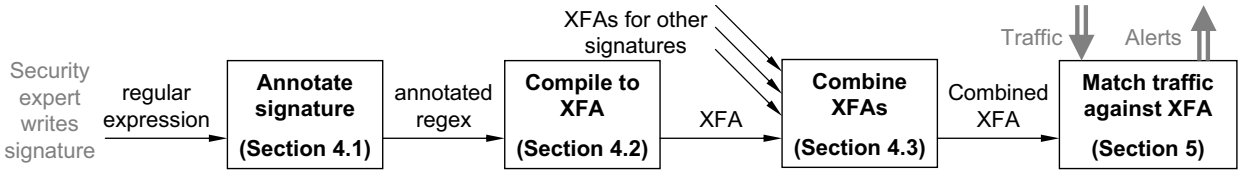

Figure 5. Lifecycle of signatures in a NIDS using XFAs.

$k = \lceil \log_2(n+2) \rceil$ bits of scratch memory. By adding these k bits we reduce the number of states by a factor of close to 2^k. Measuring run time in bit operations, the initialization cost and per-byte processing increase from $O(1)$ to $O(k)$. If we count instructions, a small constant is added to both initialization and per byte processing.

3.2. Using XFAs in a NIDS

Figure 5 depicts the steps involved in constructing XFAs and using them in a NIDS. Crafting NIDS signatures themselves is outside the scope of this paper since our proposal changes only the representation of signatures for matching, not the underlying semantics. Section 4.1 discusses how we extend regular expressions to indicate to the compiler when to use scratch memory operations. Section 4.2 outlines the compiler steps that convert an annotated regular expression into an XFA that recognizes the language defined by the regular expression. Section 4.3 briefly describes how individual XFAs are combined into a single XFA that recognizes all signatures simultaneously. In our feasibility study (Section 5) we use a signature set from the open-source Snort NIDS [20] to compare the performance of matching with an XFA against the performance of matching with DFAs.

4. Building XFAs from regular expressions

4.1. Annotating regular expressions

Transforming a regular expression into an XFA requires striking a balance between using states and transitions on one hand and executing instructions over scratch memory on the other. At one extreme we can produce a (possibly large) DFA which uses no scratch memory 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 XFA we want to build lies at one of these extremes. For expressions such as $.*S$, where S is a string, a simple DFA with no scratch memory is ideal. At the other extreme, the example from Figure 4 which recognizes $.\{n\}$ gives an XFA that is effectively just a program: there is a single state which does not influence at all how the scratch memory is updated or when acceptance happens. During construction, we use annotations to control where the resulting XFA lies along this spectrum.

Two types of constructs cause our compiler to add scratch memory objects: parallel concatenation, denoted with the symbol '#', adds a bit to the nondeterministic form of the scratch memory, and integer ranges (*e.g.* '$\{m,n\}$') add a counter. Parallel concatenation introduces a bit and changes the shape of the automaton, but it has the same semantics with respect to the language recognized as standard concatenation. Fortunately, integer ranges, a form of syntactic sugar, are already present in the signatures wherever appropriate (although we do re-interpret them to introduce a counter). Thus we only need to decide where to use the parallel concatenation operator '#'. In our current prototype, this is a partly manual step.

We use the parallel concatenation operator 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 $.*S_1.*S_2$, where S_1 and S_2 are strings, as $.*S_1\#.*S_2$. Put another way, we add the '#' operator right before subexpressions such as '`.*`' and `[^\n]{300}` that repeat characters from either the whole input alphabet or a large subset thereof. Table 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. This signature will be compiled to an XFA without any scratch memory (so it is actually a DFA). 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 subexpressions we just apply the rules for inserting '#' to the sub-expressions of the union separately. We do not insert a parallel concatenation operator in front of the `.*` at the beginning of each of these sub-expressions (it would actually be syntactically invalid).

4.2. Compiling to an XFA

Our XFA compiler takes annotated regular expressions and transforms them into deterministic XFAs. The

Num.	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 1. Snort signatures for web traffic annotated with the parallel concatenation operator '#'.

stages are the same as for traditional DFA compilers using the standard Thompson construction [30]: parsing the regular expression, building a non-deterministic automaton through a bottom-up traversal of the parse tree, ε-elimination, determinization, and minimization. We modify each of these steps to handle the scratch memory and implement new cases that handle the annotations added to regular expressions.

4.2.1. Definitions. Formally, we represent the scratch memory used by XFAs as a finite *data domain D*. Any configuration of the scratch memory that is possible during matching is represented as a *data value* $d \in D$. With each transition 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 special case of the data domain not being updated on a transition, we just associate the identity function with the transition. Since we extend the automaton with the data value, the current state of the computation is no longer fully described by the current state of the automaton $q \in Q$, but by what we call the current *configuration* of the automaton, $(q,d) \in Q \times D$. Similarly, the acceptance condition is not defined as a subset of states, but as a subset of configurations $F \subseteq Q \times D$. Note that our definition of XFAs below generalizes the standard DFA definition.

Definition 1 A (deterministic) *extended finite automaton (XFA)* is a 7-tuple $(Q, D, \Sigma, \delta, U_\delta, (q_0, d_0), F)$, 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,
- D is the finite set of values in the *data domain*,
- $U_\delta : Q \times \Sigma \times D \to D$ is the per transition *update function* which defines how the data value is updated on every transition,
- (q_0, d_0) is the *initial configuration* which consists of an initial state q_0 and an *initial data value* d_0,
- and $F \subseteq Q \times D$ is the set of *accepting configurations*.

Nondeterministic XFAs differ from deterministic XFAs in a number of important ways. Transitions can be nondeterministic, epsilon (ε) transitions are allowed, and per-transition update functions are generalized to update relations which can take a single data domain value to multiple values. Also, a set of initial configurations QD_0 replaces the single initial configuration (q_0, d_0). We define nondeterministic XFAs as follows.

Definition 2 A *nondeterministic extended finite automaton (NXFA)* is a 7-tuple $(Q, D, \Sigma, \delta, U_\delta, QD_0, F)$, where

- Q is the set of states, Σ is the set of inputs (input alphabet), $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the nondeterministic relation describing the allowed transitions,
- D is the finite set of values in the data domain,
- $U_\delta : \delta \to 2^{D \times D}$ is the *nondeterministic update function* (or 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 of the NXFA,
- and $F \subseteq Q \times D$ is the set of accepting configurations.

During the construction procedure, we represent D explicitly as a set of integers, the per transition update functions as unstructured sets of pairs (d_i, d_f), and F as a set of configurations. These are intermediate representations. The final XFA that performs the signature matching uses a much more compact representation, where D is not represented explicitly and small programs are associated with states and transitions. Thus, in the end, the amount of memory required is not much larger than that for a DFA based on Q and δ. We refer to these data domains used by the final XFAs during matching as *efficiently implementable data domains (EIDDs)*. We define them formally in Section 4.2.4.

4.2.2. From parse trees to NXFAs. Our procedure for constructing an NXFA from the parse tree extends the traditional method with provisions for manipulating the data domains and the data-dependent components of the NXFAs. We add two new constructs which extend the data domain: the parallel concatenation construct adds a bit and integer range constructs add a counter. For brevity, we only present simplified versions of these constructs that build on NFAs corresponding to the subexpressions they apply to.

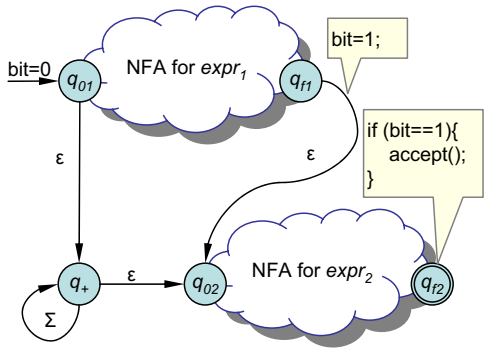

Figure 6. Simplified NXFA construction step for parallel concatenation $expr_1 \# expr_2$.

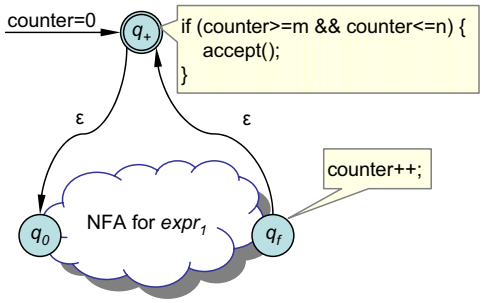

Figure 7. Simplified NXFA construction step for $(expr_1)\{m,n\}$.

Figure 6 shows a simplified version of the construction step triggered by the parallel concatenation operator $expr_1 \# expr_2$, where q_{01} and q_{f1} are the start and final states for the NFA that results from processing $expr_1$ and q_{02} and q_{f2} are the same for the NFA recognizing $expr_2$. Acceptance in q_{f2} is conditional on the bit being set to 1, but it is initialized to 0 and no transition changes it except the ε-transition from q_{f1} to q_{02}. Since there are no transitions from the states of the second automaton to the states of the first, every accepting input string must map to a path from q_{01} to q_{f1} through the first NFA, followed by the transition from q_{f1} to q_{02}, followed by a path from q_{02} to q_{f2} through the second NFA. Thus the NXFA in Figure 6 recognizes the language $expr_1 expr_2$. Note that structurally it is similar to the NFA recognizing $expr_1 \mid (.*expr_2)$.

Figure 7 shows a simplified version of the construction step triggered by an integer range of the form $(expr_1)\{m,n\}$. q_0 and q_f are the start and accepting states of the NFA that results from processing $expr_1$. Since acceptance in q_+ is conditional on the counter being between m and n, and the counter is incremented by the ε-transition from q_f to q_+, the non-deterministic XFA in Figure 7 recognizes the correct language. Note that a single copy of the automaton for $expr_1$ is used, unlike the traditional construction step which repeats the automaton n times. Structurally, the NXFAs recognizing $(expr_1)\{m,n\}$ and $(expr_1)*$ are identical. For example, for $.\{n\}$ the compiler produces the (deterministic) XFA shown in Figure 4 which has the same underlying structure as the DFA recognizing $.*$. On the other hand, standard DFA construction produces an automaton with $n+2$ states shown in Figure 3.

4.2.3. From NXFAs to XFAs. Epsilon transition elimination, given in Algorithm 1, is the first step after the NXFA is initially constructed. It 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. After running Algorithm 1, we reduce the size of the NXFA by removing states that are not accepting and have no paths leading to accepting states.

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** $q_f \in \{q \mid (q_i, \varepsilon, q) \in \delta\}$ **do**
15 $Result \leftarrow Result \cup \{q_f\} \times \{d_f \mid (d_i, d_f) \in U_\delta(q_i, \varepsilon, q_f)\}$;
16 **return** $Result$;

Algorithm 1. ε-elimination for NXFAs.

Determinization is divided into two algorithms, which determinize transitions first (Algorithm 2) and update relations second (Algorithm 3). Note that the output of Algorithm 2 is still an NXFA because nondeterminism still exists in the data domain. After consuming an input string, the matching algorithm will know the exact automaton state it is in, but multiple data domain values may be possible. Algorithm 2 is similar to the algorithm for determinizing NFAs, but to preserve the semantics of the input NXFA, the data domain D is replaced by $D' = Q \times D$ in the output NXFA. Fig-

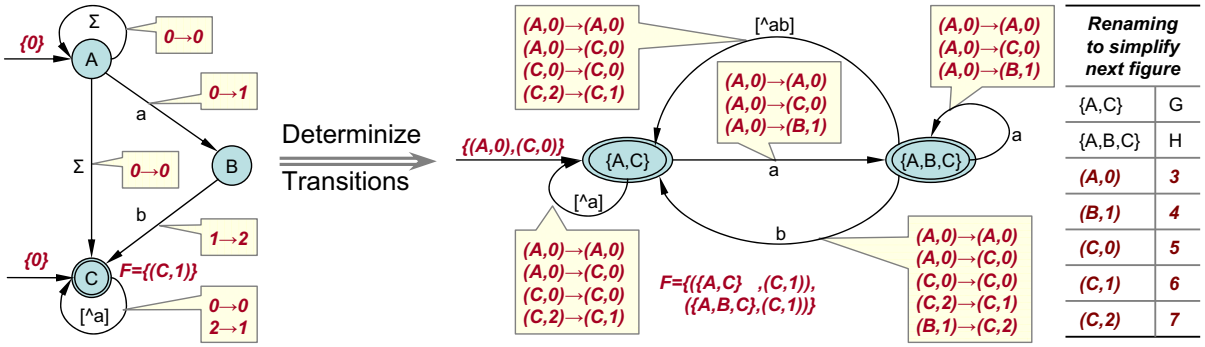

Figure 8. Applying algorithm 2 (determinizing transitions) to a simple NXFA.

DeterminizeTransitions$(Q, D, \Sigma, \delta, U_\delta, QD_0, F)$:

$D' \leftarrow Q \times D$;
// Data value in new NXFA = configuration in old
$D'_0 \leftarrow QD_0$;
// New states are sets of old states
$q'_0 \leftarrow \{q_0 | \exists d_0 \in D.(q_0, d_0) \in QD_0\}$;
$Q' \leftarrow \{q'_0\}$;
$\delta' \leftarrow \emptyset$;
$U'_\delta \leftarrow \emptyset$;
foreach $q'_i \in Q'$ **do**
  foreach $s \in \Sigma$ **do**
    $q'_f \leftarrow \{q_f | \exists q_i \in q'_i.(q_i, s, q_f) \in \delta\}$;
    $Q' \leftarrow Q' \cup \{q'_f\}$; *// Accum. reachable sets of old states*
    // New states have 1 trans. per symbol
    $\delta' \leftarrow \delta' \cup \{(q'_i, s, q'_f)\}$;
    $U \leftarrow \{((q_i, d_i), (q_f, d_f)) | q_i \in q'_i \wedge q_f \in q'_f \wedge (d_i, d_f) \in U_\delta(q_i, s, q_f)\}$;
    // Update relations preserve semantics
    $U'_\delta \leftarrow U'_\delta \cup \{(q'_i, s, q'_f)\} \times U$;
$F' \leftarrow \{(q', (q, d)) | q' \in Q' \wedge q \in q' \wedge (q, d) \in F\}$;
return $(Q', D', \Sigma, \delta', U'_\delta, \{q'_0\} \times D'_0, F')$;

Algorithm 2. Determinizing transitions.

ure 8 illustrates Algorithm 2 applied to an NXFA corresponding to `.*ab[^a]{1}` and demonstrates why this transformation is necessary. By not moving to a new domain D', a naive determinization algorithm would produce an automaton that moves to state $\{A,B,C\}$ on input 'a', with possible data values being $\{0,1\}$. Since in the input NXFA C accepts on a 1, this incorrectly determinized automaton will accept the string `a`. But in the input NXFA, the only way to get into the configuration $(C,1)$ is by going from A to B to C and looping in C once, thus the language it recognizes is `.*ab[^a]` and it does not include the string `a` . The NXFA produced by Algorithm 2 is in state $\{A,B,C\}$ after reading an 'a', but the possible values of the data domain are $\{(A,0),(B,1),(C,0)\}$. Since it only accepts on the data domain value $(C,1)$, this NXFA preserves the original semantics.

DeterminizeData$(Q, D, \Sigma, \delta, U_\delta, \{q_0\} \times D_0, F)$:

$d'_0 \leftarrow D_0$; *// New data values = sets of old data values*
$D' \leftarrow \{d'_0\}$;
// QD accumulates all reachable configurations
$QD \leftarrow \{(q_0, d'_0)\}$;
$U'_\delta \leftarrow \emptyset$;
foreach $(q_i, d'_i) \in QD$ **do**
  foreach $s \in \Sigma$ **do**
    $q_f \leftarrow \delta(q_i, s)$;
    $d'_f \leftarrow \{d_f | \exists d_i \in d'_i.(d_i, d_f) \in U_\delta(q_i, s, q_f)\}$;
    $D' \leftarrow D' \cup \{d'_f\}$; *// Accum. reachable sets of old values*
    $QD \leftarrow QD \cup \{(q_f, d'_f)\}$;
    // Build deterministic update functions
    $U'_\delta \leftarrow U'_\delta \cup \{((q_i, s), (d'_i, d'_f))\}$;
$F' \leftarrow \{(q, d') | (q, d') \in QD \wedge \exists d \in d'.(q, d) \in F\}$;
return $((Q, D', \Sigma, \delta, U'_\delta, (q_0, d'_0), F'), QD)$;

Algorithm 3. Determinizing NXFA data domains.

Algorithm 3 determinizes the data domain of the NXFA produced by Algorithm 2, yielding a deterministic XFA. Figure 9 illustrates its operation when applied to the resulting NXFA in Figure 8. Instead of replacing the data domain with a new domain $D' = 2^D$, the algorithm assigns to D' the (typically small) subset of 2^D that is reachable from the start configurations. Note that while the update functions associated with transitions U'_δ 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. Algorithm 3 also computes an auxiliary data structure QD, the set of configurations reachable from the initial configuration (q_0, d'_0). This data structure is used by subsequent stages of the compiler.

Minimization, the next step, is not possible for XFAs as there is no single canonical minimal form for an XFA. Instead of a minimization stage, we developed two separate algorithms: one for reducing the number of data domain values possible in each state, and one for reducing the number of states. Currently, our compiler only uses the data reduction algorithm which is

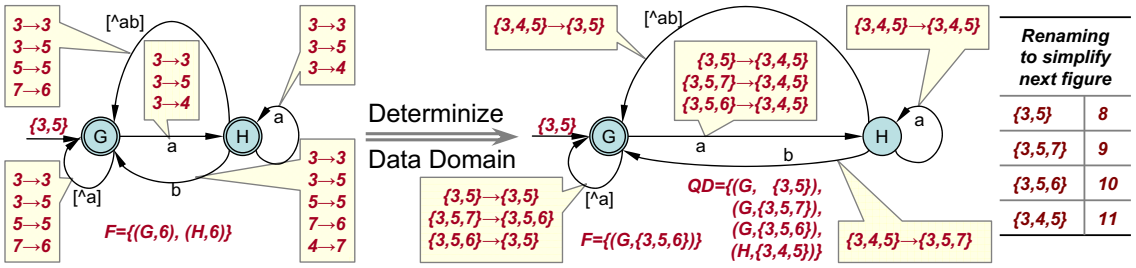

Figure 9. Applying algorithm 3 (data determinization) to the NXFA produced in Figure 8.

applied to the XFA produced by Algorithm 3. Due to space constraints, we eliminate a detailed description of these algorithms.

4.2.4. Finding efficient implementations. 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 determines the structure of the scratch memory. Figure 10 gives a simplified view of how this step works for the XFA produced in Figure 9. More formally, in this step the compiler finds a mapping from data domain operations in an XFA to an efficiently implementable data domain (EIDD), which we define as follows:

Definition 3 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,
- $U_E : E \to D^D$ 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 $A_C : C \to 2^D$ is a mapping from these names to acceptance conditions that can be associated with XFA states.

The XFA does not rely explicitly on U_δ and F during its operation. Instead, it uses E_δ, which maps each transition from δ to an update function from E, and C_Q, which maps each state from Q to an acceptance condition in C. Although definition 3 specifies that the update functions in E must be "efficient to compute" and the acceptance conditions from C "efficient to check", it is out of scope to give a definition for what it means to 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 4 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) $D'_{QD} : Q \times D \to D'$ maps all configurations from QD (produced by Algorithm 3) to values in the new data domain D', (2) $E_\delta : Q \times \Sigma \to E$ maps all transitions to efficient update functions, and (3) $C_Q : Q \to C$ maps all states to efficient acceptance conditions. Note that the mapping for data domain values is from QD to D', rather than from D to D'. Thus 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 both values from D can occur. In Figure 10, for example, the data domain size is reduced from 4 to 3. 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.

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

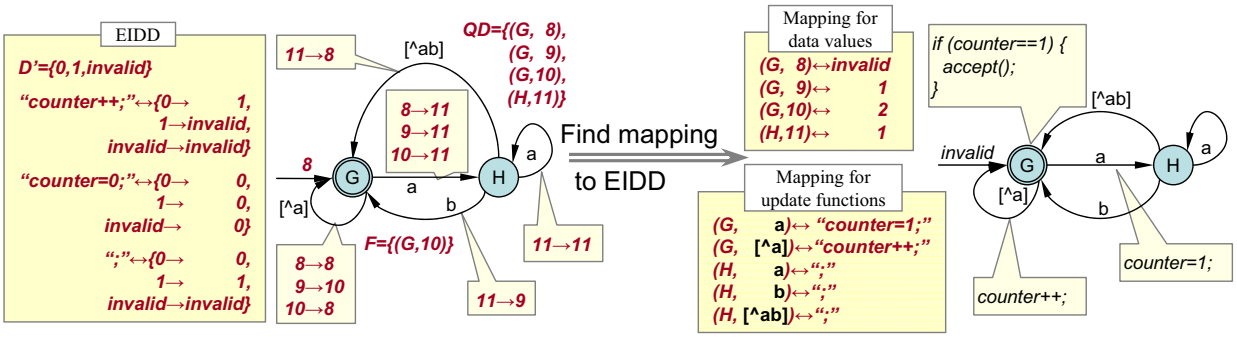

Figure 10. Finding a mapping to an EIDD for the XFA produced in Figure 9.

Algorithm 4 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 $(\emptyset, \emptyset)$.

This algorithm has $O(|E|^{|\delta|})$ worst-case complexity and thus is not practical, but it is the starting point for the more complex algorithm used in our implementation. 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.

4.3. Combining XFAs

For combining XFAs recognizing individual signatures into a single XFA that recognizes an entire signature set we extend the algorithm for combining DFAs with provisions for manipulating the scratch memories and the associated operations. The example from Figure 2 illustrates these extensions. The scratch memory of the combined XFA holds the scratch memories of the two input XFAs side by side. For each state, the acceptance condition combines the acceptance conditions of the two states it corresponds to in the two input XFAs. Similarly, for each transition, the update function combines the update functions of the corresponding transitions. Since the combined XFA contains separate copies of the scratch memories of the input XFAs, the update functions and update conditions from different automata never interfere.

There is one exception to this property of not sharing scratch memory objects. In our test set there are 338 signatures of the form `.*<OBJECT#[^>]*classid=11cf-9377` that differ only in the hexadecimal class identifier following the `classid` string (signature simplified for presentation purposes). The XFA corresponding to such a signature resembles those in Figure 2. A bit is set when `<OBJECT` is observed and reset whenever the symbol '>' is seen. The XFA accepts after seeing `classid=11cf-9377` if the bit is set. The bits for all 338 XFAs corresponding to such signatures encode the same information (whether the input processed so far contains a `<OBJECT` not followed by a '>') and can actually share a single bit without interference. We have defined a heuristic that determines when bits in XFAs such as these and others can safely be shared, and in such cases this allows us to reduce the amount of needed scratch memory. We also use this heuristic

MapXFAtoEIDD$((Q,D,\Sigma,\delta,U_\delta,(q_0,d_0),F),QD,EIDD)$:

$(D',d'_0,E,U_E,C,A_C) \leftarrow EIDD$;
$E_\delta \leftarrow \emptyset$;
$D'_{QD} \leftarrow \{((q_0,d_0),d'_0)\}$;
$(D'_{QD},E_\delta) \leftarrow$ `FindValidMapping` (D'_{QD},E_δ);
if $(D'_{QD},E_\delta) = (\emptyset,\emptyset)$ **then return** $(\emptyset,\emptyset,\emptyset)$;
$C_Q \leftarrow \emptyset$;
foreach $q \in Q$ **do**
  $c_{correct} \leftarrow \{c \in C | \forall((q,d),d') \in D'_{QD}.d' \in A_C(c) \iff (q,d) \in F\}[0]$;
  $C_Q \leftarrow C_Q \cup \{(q,c_{correct})\}$;
return (D'_{QD},E_δ,C_Q);

FindValidMapping(D'_{QD},E_δ) :

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**
  $D'_{QD} \leftarrow D'_{QD} \cup \{((\delta(q_i,s),U_\delta(q_i,s)(d_i)),U_E(E_\delta(q_i,s))(d'_i))\}$;
if `FindInconsistency` (D'_{QD},E_δ) **then return** $(\emptyset,\emptyset)$;
if $|E_\delta| = |Q| \cdot |\Sigma|$ **then return** (D'_{QD},E_δ) ;
$trans \leftarrow \{(q_i,s) | (q_i,s) \in Q \times \Sigma \wedge \nexists e \in E.((q_i,s),e) \in E_\delta\}[0]$;
foreach $e \in E$ **do**
  $Result \leftarrow$ `FindValidMapping` $(D'_{QD},E_\delta \cup \{(trans,e)\})$;
  if $Result \neq (\emptyset,\emptyset)$ **then return** $Result$;
return $(\emptyset,\emptyset)$;

FindInconsistency(D'_{QD},E_δ) :

foreach $(q_i,s,d_i) \in Q \times \Sigma \times D$ **do**
  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**
    if $\exists e \in E.((q_i,s),e) \in E_\delta \wedge (d'_i,d'_f) \notin U_E(e)$ **then**
      return true ;
foreach $q \in Q$ **do**
  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** ;
return false ;

Algorithm 4. Basic algorithm for finding a mapping of an XFA to a given EIDD.

for combining the 213 signatures from our test set that are the UNICODE equivalents of some of these 338 signatures. We then use the normal combination procedure to combine these XFAs with those for the rest of the signatures.

5. Feasibility study

We examined the feasibility of XFAs with a case study that applied them to HTTP signatures used by the Snort NIDS. We focus on two aspects of XFAs: feasibility of construction and memory usage and performance. We briefly summarize the results of this study:

Feasibility of construction. In Section 5.2 we describe the process in which 1450 Snort HTTP signatures are converted into efficient XFAs. Construction of this test set required one day of manual effort, but this is a one-time cost in general, and our experience suggests that new XFAs can often be constructed and incorporated within a matter of minutes.

Memory usage and performance. In Section 5.3, we compare the memory usage and performance of our test-set XFA to DFAs and multiple DFA-based approaches. Despite inefficiencies in our prototype, our results shows that the combined XFA was 20× faster and 10× smaller than the next-best result.

5.1. Experimental methodology

We have implemented a fully-functional evaluation prototype divided into two main applications: *re2xfa* and *trace_apply*. *re2xfa* implements all of the XFA construction algorithms described earlier and produces XFAs for annotated regular expressions supplied as input. *trace_apply* requires an XFA and a tcpdump-formatted trace and applies the XFA to HTTP payloads in the trace. Instructions on edges and states are executed using an interpreter we implemented and built into *trace_apply*. Since our primary goal is to study the feasibility of XFAs, standard NIDS operations such as defragmentation and normalization are beyond the scope of the experiments performed here.

We also compare against multiple-DFA (MDFA) techniques using the combination heuristics proposed by Yu, *et al.* [33]. MDFAs trade memory usage for time by enforcing an upper limit on the available memory and producing as many groups of combined DFAs as necessary to stay within that limit. To facilitate a fair comparison, all automata use the same format and evaluation environment (*trace_apply*), except that only XFAs have edge-based instructions. We model DFA accepting states with an instruction that unconditionally accepts when the state is reached.

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 our 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 per payload, scaled to seconds per gigabyte (s/GB). All experiments were performed on a standard Pentium 4 Linux workstation running at 3 GHz with 3 GB of memory.

5.2. 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 [18] software distribution) to do an initial parsing and conversion of Snort's HTTP signatures into Bro format, which we then passed through

Examples (some simplified)	# Sigs	EIDD name	Scratch mem.
`.*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 2. Description of common kinds of signatures and their mappings to XFAs.

# 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	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0
% States	78.9	20.0	0.9	0.03	0.00	0.00	0.00	0.00	0.05	0.05	0.00	0.00	0.00

Table 3. Distribution of instructions on edges and states. Entries marked 0.00 contribute less than .01%.

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. In total, we spent approximately one day on this phase, not including EIDD creation time.

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 4. In our test set, 85% of the signatures completed within 10 seconds each.

Run time (seconds)	Number of sig.
< 1	37.1%
1..10	48.1%
10..100	0.1%
100..1,000	1.2%
1,000..10,000	13.5%

Table 4. Distribution of XFA construction times.

Finally, Step 4 combines each of the XFAs produced in the previous step using the incremental combination algorithm outlined in Section 4.3. Combination of all individual XFAs into a single equivalent XFA required just over 10 minutes. Table 3 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 scratch 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,[3] this is also a one-time cost.

Signatures were removed from our 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 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. We discuss these difficulties and possible workarounds in more detail in Section 6.

In summary, these results demonstrate that XFAs can be readily constructed for large numbers of real-

[3] EIDDs are declarative and parsed by our prototype, so that they can be supplied at runtime and do not require a recompile.

Automata Type	Total Mem	Num Automata	Exec (s/GB)
XFA	43MB + 3.5MB	1	75.6
DFA	$>$ 15GB	n/a	$\sim$11.6
MDFA	432 MB	67	1,458
	397 MB	107	2,690
	277 MB	147	3,780
	191 MB	346	8,570
	98 MB	587	14,889
	66 MB	786	20,865

Table 5. Machine size and execution times for XFAs, DFAs, and Multiple DFAs for several memory settings. XFAs approach DFA performance at small memory sizes.

world signatures. We produced efficient XFAs for 93% of Snort's HTTP signatures. Construction of this set required a day of manual effort, but our experience suggests that new XFAs can be quickly constructed and incorporated in many cases.

5.3. Performance and memory usage

We compared XFAs to traditional DFAs and to multiple DFA-based solutions, 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. We produced MDFAs for several memory limits ranging from 66 MB (the smallest memory size that could hold all signatures) to 512 MB.

Table 5 summarizes the performance and memory usage individually for each of the techniques. DFAs, if realizable, would have the best performance with 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 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 3.

Figure 11 compares the MDFAs to XFAs graphically. In the plot, the y-axis reflects total memory usage and for XFAs includes both instruction and scratch memory (46.5 MB). Both axes are on a logarithmic scale. The plus marks ('+') in the plot show the points

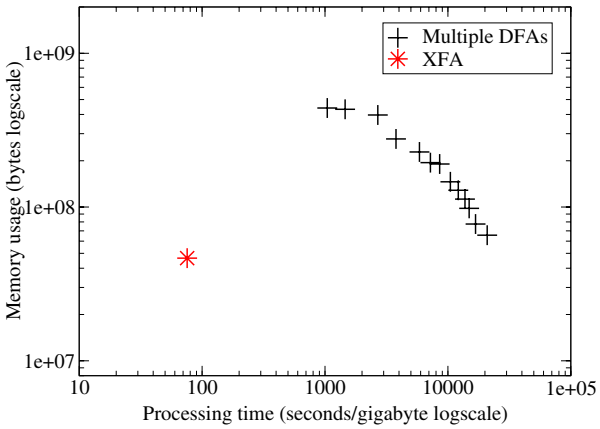

Figure 11. Memory vs. run-time for MDFAs and XFAs. XFAs are both smaller and faster than MDFAs for many memory ceilings.

for several MDFA 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. Even with the inefficiencies of our prototype system, the XFA yields superior results as compared to MDFAs both in memory usage and performance.

6. Limitations and discussion

6.1. Mapping to EIDDs

The basic procedure for mapping an XFA with abstract data domains to an appropriate EIDD, given in Algorithm 4, 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 scratch memory 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 Al-

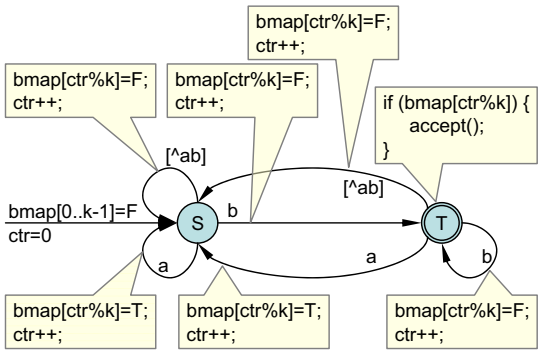

Figure 12. The XFA recognizing `.*a.`$\{n\}$`b` where $k = n+2$.

gorithm 4 selects only those that it needs. But in our prototype, the resulting mapping times would be infeasible. We are working to address this issue.

6.2. 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 scratch memory rather than only in explicit automaton states. For example, an XFA corresponding to this regular expression, given in Figure 12, needs only two states, a counter, and a bitmap with $k = n+2$ bits of scratch 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 12. 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}`.

We are working to develop techniques that address the difficulties described above and expand the class of signatures that can be readily mapped to compact, efficient XFAs. 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* [20, 22] that identify only subparts of signatures and invoke full signature evaluation using DFAs, NFAs, or other techniques when the subparts are matched. Alternatively, MDFAs [33] 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.

7. Conclusion and future work

In this paper we have introduced Extended Finite Automata (XFAs), which augment traditional finite state automata with a scratch memory that is manipulated by instructions attached to edges and states. We provide a formal definition for XFAs and present a technique for constructing them from regular expressions. We performed a feasibility study using a set of HTTP signatures from Snort and observed that XFAs have matching speeds approaching DFAs yet memory requirements similar to NFAs. Compared to multiple DFA-based techniques, our tests used 10× less memory *and* were 20× faster.

The techniques and results we have presented here are preliminary in many respects and we are actively working to refine them. Some aspects of our construction procedure require some manual input, and some signatures require inordinately long construction times. In addition, there is still some missing functionality and inefficiencies in our interpreter and execution environment. We are investigating techniques for addressing these and other issues. Notwithstanding these open problems, we are hopeful that in the end XFAs will lead to better solutions for high speed signature matching.

Acknowledgements

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.

References

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

[2] R. Alur. Timed automata. In *Proceedings of the Int. Conf. on Computer Aided Verification*, pages 8–22, 1999.

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

[4] 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*, Oakland, California, May 2006.

[5] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, April 2004.

[6] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster pattern matching for intrusion detection or exceeding the speed of Snort. In *2nd DARPA Information Survivability Conference and Exposition*, June 2001.

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

[8] S.T. Eckmann, G. Vigna, and R.A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

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

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

[11] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Usenix Security*, August 2001.

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

[13] M. Jordan. Dealing with metamorphism. *Virus Bulletin Weekly*, 2002.

[14] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of ACM SIGCOMM*, September 2006.

[15] R-T Liu, N-F Huang, C-H Chen, and C-N Kao. A fast string-matching algorithm for network processor-based intrusion detection system. *Transactions on Embedded Computing Sys.*, 3(3):614–633, 2004.

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

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

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

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

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

[21] S. Rubin, S. Jha, and B. Miller. Language-based generation and evaluation of NIDS signatures. In *IEEE Symposium on Security and Privacy*, May 2005.

[22] S. Rubin, S. Jha, and B. P. Miller. Protomatching network traffic for high throughput network intrusion detection. In *ACM Conference on Computer and Communications Security (CCS)*, pages 47–58, 2006.

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

[24] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *IEEE Symposium on Security and Privacy*, May 2003.

[25] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines (FCCM)*, April 2001.

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

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

[28] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2004.

[29] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *International Symposium on Computer Architecture (ISCA)*, June 2005.

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

[31] H. J. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, August 2004.

[32] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *14th USENIX Security Symposium*, August 2005.

[33] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of Architectures for Networking and Communications Systems (ANCS)*, pages 93–102, 2006.