(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
30 July 2009 (30.07.2009)

PCT

(10) International Publication Number
WO 2009/094544 A1

(51) International Patent Classification:
G10L 15/06 (2006.01)

(21) International Application Number:
PCT/US2009/031842

(22) International Filing Date: 23 January 2009 (23.01.2009)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/022,993 23 January 2008 (23.01.2008) US
12/032,380 15 February 2008 (15.02.2008) US

(71) Applicant *(for all designated States except US)*: **WISCONSIN ALUMNI RESEARCH FOUNDATION** [US/US]; P.O. Box 7365, Madison, WI 53707-7365 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **ESTAN, Cristian** [US/US]; 505 Isle Royal Drive, Madison, WI 53705 (US). **SMITH, Randy, D.** [US/US]; 5114 Holiday Drive, Madison, WI 53711 (US). **JHA, Somesh** [US/US]; 14 N. Woodmont Circle, Madison, WI 53717 (US).

(74) Agent: **CAMPBELL, Benjamin, A.**; Lathrop & Clark LLP, 740 Regent Street, Suite 400, P.O. Box 1507, Madison, WI 53701-1507 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

(54) **Title:** EXTENDED FINITE STATE AUTOMATA AND SYSTEMS AND METHODS FOR RECOGNIZING PATTERNS

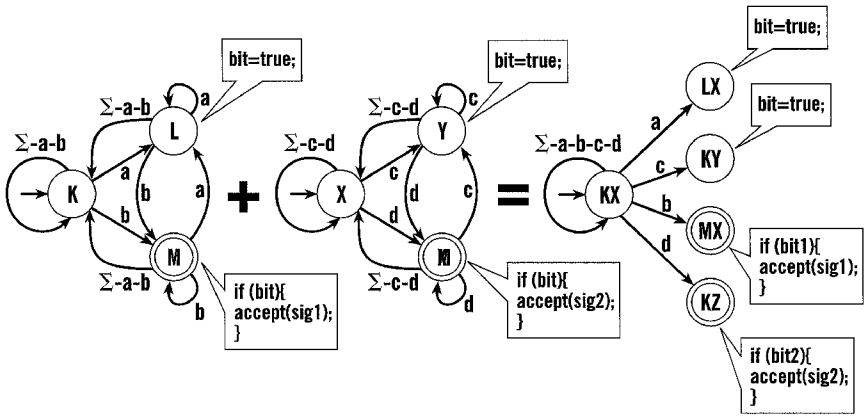

FIG. 3

(57) **Abstract:** Deterministic finite automata (DFAs) are popular solutions to deep packet inspection because they are fast and DFAs corresponding to multiple signatures are combinable into a single DFA. Combining such DFAs causes an explosive increase in memory usage. Extended finite automata (XFAs) are an alternative to DFAs that avoids state-space explosion problems. XFAs extend DFAs with a few bytes of 'scratch memory' used to store bits and other data structures that record progress. Simple programs associated with automaton states and/or transitions manipulate this scratch memory. XFAs are deterministic in their operation, are equivalent to DFAs in expressiveness, and require no custom hardware support. Fully functional prototype XFA implementations show that, for most signature sets, XFAs are at least 10,000 times smaller than the DFA matching all signatures. XFAs are 10 times smaller and 5 times faster or 5 times smaller and 20 times faster than systems using multiple DFAs.

WO 2009/094544 A1

EXTENDED FINITE STATE AUTOMATA AND SYSTEMS AND METHODS FOR RECOGNIZING PATTERNS

[0001] This application claims priority to U.S. Provisional Patent Application 61/022,993, filed January 23, 2008 and U.S. Patent Application 12/032,380, filed February 15, 2008, each of which is incorporated herein by reference in its entirety. The subject matter of this application was made with U.S. Government support, awarded by the following agencies: National Science Foundation, grant numbers 0716538 and 0546585. The United States has certain rights to this application.

BACKGROUND

1. Field of the Invention

[0002] This invention is directed to finite state automata and using such finite state automata to recognize patterns in data.

2. Related Art

[0003] Deep packet inspection is becoming prevalent for modern networking devices as they inspect packet payloads for a variety of reasons. Payloads are matched against signatures of vulnerabilities to detect or block intrusions. Application recognition, used for accounting and traffic policing, also increasingly relies on inspecting packet contents. Often, load balancing is done based on application layer protocol fields. Implementing such deep packet inspection at the high speeds of today's networks is one of the hardest challenges facing the manufacturers of network equipment. One type of deep packet inspection operation which is critical for intrusion detection and/or prevention systems (IPSes) is signature matching.

[0004] The move from simple string based signatures to more complex regular expressions has turned signature matching into one of the most daunting deep packet inspection challenges. To achieve good matching throughput, IPSes typically represent the signatures as deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). Both types of automata allow multiple signatures to be matched in a single pass. However, DFAs and NFAs present different memory vs. performance tradeoffs. DFAs are fast and large, whereas NFAs are smaller but slower. Measurements by the inventors confirm that DFAs recognizing signature sets

currently in use exceed feasible memory sizes, while NFAs are typically hundreds of times slower than DFAs.

[0005] Combining the DFAs for each of the individual signatures into a single DFA usable to recognize a signature set often results in a state space explosion. For example, the size of the minimal DFA that recognizes multiple signatures of the type ". *a_i. *b_i", where "a_i" and "b_i" are distinct strings, is exponential in the number of signatures. Less memory suffices when using multiple DFAs because the extent of the state space explosion is smaller. NFAs avoid state space explosion entirely. However, they are less efficient because, for every input byte, the NFA has to track a set of automaton states.

[0006] String matching was important for early network intrusion detection systems, as the signatures consisted of simple strings. The Aho-Corasick algorithm, which is disclosed in Alfred V. Aho et. al., "Efficient string matching: An aid to bibliographic search", Communications of the ACM, June 1975, builds a concise automaton that recognizes multiple different ones of such signatures in a single pass. This automaton is linear in the total size of the strings. Other conventional software and hardware solutions to the string matching problem have also been proposed. Some of these solutions also support wildcard characters. However, evasion techniques made it necessary to use signatures that cover large classes of attacks but still make fine enough distinctions to eliminate false positives. Consequently, the last few years have seen signature languages evolve from exploit-based signatures that are expressed as strings to richer session signatures and vulnerability-based signatures. These complex modern signatures can no longer be expressed as strings or strings with wildcards. As a result, regular expressions are used instead.

[0007] NFAs represent regular expressions compactly 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 renders the NFA vulnerable to algorithmic complexity attacks, or by maintaining and updating a "frontier" of states. Unfortunately, both of these solutions can be computationally expensive. While hardware-implemented NFAs can parallelize the processing required and thus achieve high speeds, software implementations of NFAs have

significant processing costs. One known NFA hardware architecture efficiently updates the set of states during matching.

[0008] DFAs can be efficiently implemented in software. However, DFAs have a state space explosion problem that makes it infeasible to build a DFA that matches all signatures of a complex signature set. On-the-fly determinization has been proposed for matching multiple signatures. This approach maintains a cache of recently visited states and computes transitions to new states as necessary during inspection. This approach has good performance in the average case, but can be subverted by an adversary who can repeatedly invoke the expensive determinization operation.

[0009] Fang Yu, et al., "Fast and memory-efficient regular expression matching for deep packet inspection", Technical Report EECS-2005-8, U.C. Berkeley, 2005 (hereafter "Yu"), discloses a solution that offers intermediary tradeoffs by using multiple DFAs to match a signature set. Each of these DFAs recognizes only a subset of the signatures and all DFAs need to be matched against the input. Yu proposes combining signatures into multiple DFAs instead of one DFA, using simple heuristics to determine which signatures should be grouped together. This approach reduces the total memory footprint, but for complex signature sets, the number of resulting DFAs can itself be large. Additionally, this approach results in increased inspection time, since payloads must now be scanned by multiple DFAs.

[0010] The D^2FA technique disclosed in S. Kumar, et al., "Algorithms to accelerate multiple regular expressions matching for deep packet inspection", Proceedings of the ACM SIGCOMM, September 2006, addresses memory consumption by reducing the memory footprint of individual states. It stores only the difference between transitions from similar states, and relies on some hardware support for fast matching. The technique does not address the state space explosion problem.

[0011] Other extensions to automata have been proposed. In the context of XML parsing, Michael Spergberg-McQueen, "Notes on finite state automata with counter", www.w3.org/XML/2004/05/msm-cfa.html, (available at least as early as May 2004), discloses counter-extended finite automata (CFAs). CFAs are nondeterministic and have the same performance problems as NFAs. Extended Finite State Automata (EFSA) that extend a traditional finite state automaton with the ability

to assign and examine values of a finite set of variables have been used in the context of information security. EFSAs have been used to monitor a sequence of system calls, and to detect deviations from expected protocol state for VoIP applications. Extensions, such as EFSA, fundamentally broaden the language recognized by the finite-state automata, i.e., EFSAs correspond to regular expression for events (REEs).

SUMMARY OF THE DISCLOSED EMBODIMENTS

[0012] A new type of Finite State Automata (FSAs), eXtended Finite Automata (XFAs), according to this invention, provide an alternate representation for signatures with memory requirements similar to NFAs and with a matching speed that approaches DFAs. XFAs extend DFAs by introducing a small "scratch memory" holding bits, counters and/or other appropriate data structures that are manipulated by simple programs attached to the transitions and/or the states of the XFA and executed whenever the annotated transitions are traversed and/or whenever the annotated states are reached. By using bits, counters and/or other appropriate data structures to independently record the progress in matching various signatures, XFAs largely avoid the state space explosion problem inherent in combining DFAs. A fully functional XFA prototype has been built and has been evaluated using signature sets from the Snort Network Intrusion Detection System, disclosed in M. Roesch, "Snort – lightweight intrusion detection for networks", Proceedings of the 13th Systems Administration Conference, USENIX, 1999, and from the Cisco Intrusion Prevention System (IPS).

[0013] Systems and methods according to this invention provide extended finite automata (XFAs).

[0014] Systems and methods according to this invention separately provide XFAs that are usable during deep packet inspection.

[0015] Systems and methods according to this invention separately provide XFAs that provide signature representation usable by an IPS.

[0016] Systems and methods according to this invention separately provide fully functional XFAs, including algorithms for matching, combining, and verifying XFAs.

[0017] Systems and methods according to this invention separately provide algorithms based on optimization techniques from compilers that reduce the memory usage and run time of XFAs recognizing multiple signatures.

[0018] An XFA operates similarly to a DFA. However, an XFA keeps a small amount of "scratch memory" that persists as the XFA moves from state to state. This scratch memory holds one or more data structures, such as, for example, one or more counters, one or more independently-set bits, and/or any other appropriate data structures that can be used to track progress in recognizing signatures or other types of data patterns. In various exemplary embodiments, various transitions in an XFA are provided with, i.e., annotated with, small programs that update at least some of the variables in scratch memory. In various other exemplary embodiments, various states in an XFA are provided or annotated with small programs that update at least some of the variables in scratch memory. An XFA recognizes its corresponding signature when it reaches an accepting state, but only if the values in the scratch memory also match an "acceptance condition".

[0019] XFAs can represent combined signatures more compactly than DFAs because they separate information about progress in matching signatures into distinct bits, counters and/or other appropriate data structures that can be updated independently, whereas DFAs need to keep explicit states for many possible interleavings of the subpatterns of various signatures.

[0020] Deep packet inspection required by network devices, such as intrusion detection systems, is becoming a performance bottleneck. Deterministic finite automata (DFAs) are a popular solution to this problem because they are fast and because it is possible to combine the DFAs corresponding to multiple signatures into a single DFA. However, combining DFAs recognizing complex signatures leads to an explosive increase in memory usage. To counter this effect, current systems use multiple DFAs each recognizing a subset of the signatures and each is matched separately against the traffic. In contrast, extended finite automata (XFAs) are an alternative to DFAs that avoid state space explosion problems. XFAs extend DFAs with a few bytes of "scratch memory" used to store bits, counters and/or other data structures that record progress. Simple programs associated with automaton transitions and/or states manipulate this scratch memory. XFAs are deterministic in their operation, are equivalent to DFAs in expressiveness, and require no custom

hardware support. Measurements from using fully functional prototype XFA implementations show that, for most signature sets, XFAs are at least tens of thousands of times smaller than the DFA matching all signatures. Compared to various configurations using multiple DFAs, the prototype XFAs are 10 times smaller and 5 times faster or 5 times smaller and 20 times faster.

[0021] These and other features and advantages of various exemplary embodiments of systems and methods according to this invention are described in, or are apparent from, the following detailed descriptions of various exemplary embodiments of various devices, structures and/or methods according to this invention.

BRIEF DESCRIPTION OF DRAWINGS

[0022] Various exemplary embodiments of the systems and methods according to this invention will be described in detail, with reference to the following figures, wherein:

[0023] Fig. 1 shows one exemplary embodiment of the combined DFA for expressions ".*a.*b" and ".*c.*d", which uses states to "remember" which subpatterns have occurred;

[0024] Fig. 2 shows one exemplary embodiment of a combined DFA for expressions ".*\na[^\n]{200}" and ".*bc", which replicates the second expression 200 times, once for each repetition;

[0025] Fig. 3 shows one exemplary embodiment of a state-based XFA according to this invention that, by using bits associated with states to remember subpattern occurrences, has required fewer states than the corresponding DFA shown in Fig. 1.

[0026] Fig. 4 shows one exemplary embodiment of an XFA according to this invention that, by substituting a counter associated with states for explicit counting states, avoids replicating ".*bc" as in the DFA shown in Fig. 2;

[0027] Fig. 5 shows one exemplary embodiment of the combined DFA for expressions ".*ab.*cd" and "*ef.*gh", which has state space blowup;

[0028] Fig. 6 shows one exemplary embodiment of a transition-based XFA according to this invention that, by using bits associated with transitions to remember

subpattern occurrences, requires fewer states than the corresponding DFA shown in Fig. 5;

[0029] Fig. 7 shows one exemplary embodiment of a DFA that recognizes the string ".{n}";

[0030] Fig. 8 shows one exemplary embodiment of a transition-based XFA according to this invention that, by substituting a counter associated with the transitions between the single state and an acceptance condition associated with that single state, requires fewer states than the corresponding DFA shown in Fig. 7;

[0031] Fig. 9 is a block diagram illustrating one exemplary method for converting NIDS signatures into an XFA and for using that XFA to monitor network traffic;

[0032] Fig. 10 shows one exemplary embodiment of a method according to this invention for combining two XFAs;

[0033] Fig. 11 shows one exemplary embodiment of a method according to this invention for matching an XFA against a packet payload or buffer of bytes;

[0034] Fig. 12 shows one exemplary embodiment of a method according to this invention for matching an XFA against a string S;

[0035] Fig. 13 shows one exemplary embodiment of a method according to this invention for counter minimizing applied to XFAs for signatures .*\na[^\n]{200} and .*\nb[^\n]{150};

[0036] Fig. 14 shows one exemplary embodiment of a method according to this invention for determining counter compatibility;

[0037] Fig. 15 is a transition-based XFA that recognizes .*a.{n}b, where k = n + 2 and illustrates the use of bitmaps in scratch memory;

[0038] Fig. 16 shows one exemplary embodiment of a distribution of instructions in states of the Snort SMTP-combined XFA before and after optimization;

[0039] Fig. 17 shows one exemplary embodiment of a distribution of instructions in states of the Snort FTP-combined XFA before and after optimization;

[0040] Fig. 18 shows one exemplary embodiment of the memory versus runtime tradeoffs for DFAs, mDFAs, NFAs, and XFAs for the Snort signature sets; and

[0041] Fig. 19 shows one exemplary embodiment of the memory versus runtime tradeoffs for DFAs, mDFAs, NFAs, and XFAs for the Cisco signature sets.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

[0042] The magnitude of DFA state space explosion depends strongly on the types of patterns being tracked. For example, in the case of strings, the number of states required to recognize n signatures is bounded by the total size of the strings, or O(n) if the size of the largest string is bound. Other types of patterns, such as those in Figs. 1 and 2, can cause the number of states to increase exponentially and quadratically, respectively, in the number of signatures. For example, Fig. 1 shows the combined DFA for expressions ".*a.*b" and ".*c.*d". The DFA of Fig. 1 uses states to "remember" which subpatterns have occurred. It should be appreciated that for simplicity, some transitions or edges have been omitted. Similarly, in Fig. 2, the combined DFA for expressions ".*\na[^\n]{200}" and ".*bc" replicates the second expression 200 times, once for each repetition of the first expression. Again, it should be appreciated that, for simplicity, some transitions or edges have been omitted.

[0043] When XFAs recognizing individual signatures are combined, no state space explosion takes place like in the case when DFAs recognizing the same signatures are combined. The reason for this is not the procedure for combining the XFAs, which is an extension of the procedure combining DFAs. In fact, the scratch memory of the XFAs does not influence in any way the shape of the underlying automaton for the combined XFA. The reason why the combined XFA is smaller than the combined DFA is that the shape of the underlying automata for the XFAs is different. These different shapes do not lead to state space explosion because when the XFAs are combined, they have benign interactions just as DFAs that perform string matching (i.e. recognize .*S). The scratch memory for the XFAs is important because it allows the XFAs to correctly implement complex signatures. That is, even though the underlying automata are very similar to those performing string matching, because of the use of scratch memory, they can recognize complex signatures such as those shown in the examples set forth herein.

[0044] Figs. 1-4 and the following detailed description of those figures, including Definitions 1.1 and 1.2, are directed to a first exemplary embodiment or type of an XFA according to this invention, where the states are extended according

to this invention. In contrast, Figs. 5-8 and the following detailed description of those figures, including Definitions 2.1 and 2.2, are directed to a second exemplary embodiment or type of an XFA according to this invention, where the transitions (or edges), as well as some of the states, are extended according to this invention.

[0045] In such a state-based XFA, the XFA instructions associated with (or attached to) a given state are executed whenever that given state is entered, independent of the transition traversed to reach that given state. In such a transition-based XFA, the XFA instructions associated with (or attached to) a given transition are executed whenever that given transitions is traversed, without regard to the states located at the beginning or end of that given transition. Similarly, instructions associated with or attached to a state are executed when that state is reached. The instructions act to either trigger alerts signaling that the input matches the signatures tracked, and/or to update the data structures in scratch memory. For state-based XFAs, both types of instructions are associated with some states of the automaton. For transition-based XFAs the first type of instructions are associated with some states and the second type with some transitions. It should be appreciated that transition-based and state based instructions of the first type can be incorporated in a single XFA.

[0046] Thus, in the state-based XFA, if a given state has a plurality of incoming transitions, the instructions attached to that state will be executed and the memory structures update when that state is reached, regardless of the incoming transition that was traversed to reach that state. In contrast, in the transition-based XFA, if a given state has a plurality of outgoing transitions, the instructions that will be executed upon leaving that state will depend on the particular transition that is traversed as that state is exited. That is, whenever a state is entered, if it has instructions associated with it, those instructions are executed. Whenever a transition is used to move from one state to another, if the transition has instructions associated with it, those instructions are executed. It should be appreciated that the number of transitions entering or exiting given states does not necessarily affect the selection of instructions to be executed.

[0047] 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. Figs. 1 and 5 show how signatures of the form ". $*s_1$. $*s_2$" can cause state space explosion.

Signatures such as ".*s_1. *s_2" should be read as "a first subpattern, followed by an arbitrary number of characters, followed by a second subpattern." For each signature of this type, the combined DFA needs to remember whether it has already seen the first subpattern, so that the combined DFA knows whether or not to accept the second subpattern. If there are n distinct such subpatterns, the DFA will need at least 2n states to remember what subset of these distinct subpatterns have appeared in the input.

[0048] Fig. 2 shows two sources of state space explosion in signatures commonly used to detect overflow attacks in text based protocols: ".*\n$s_1$[^\n]{k}", which should be read as "newline followed by a subpattern followed by k non-newline characters". The first problem is that the DFA must count up to k non-newline characters using a string of k states. The second problem occurs when this signature is combined with simple string-based signatures. In such cases, the combined DFA needs to concurrently track two independent things: how far along it is in matching this signature and its progress in recognizing strings, which can occur at any offset. This requires O(nk) states to track a single such signature and n strings and $O(n^2k)$ states to track n such signatures and n strings.

[0049] Fig. 3 shows how state-based XFAs can avoid the state space explosion for signatures of the type shown in Fig. 1. In this example, each state-based XFA uses one bit of scratch memory that is set whenever the first subpattern is recognized. In the accepting states, each state-based XFA recognizes its signature only if its bit is set. When many such state-based XFAs are combined, the state-based XFA uses one separate bit for each distinct subpattern to remember whether various subpatterns have been recognized. While this leads to larger scratch memory requirements, the state space grows linearly rather than exponentially. As shown in Fig. 3, by using bits to remember subpattern occurrences, the combined state-based XFA requires fewer states than the combined DFA shown in Fig. 1. It should be appreciated that, in the DFA shown in Fig. 1, the back transitions or edges to state KX and other transitions or edges have been removed for clarity.

[0050] Fig. 4 presents an example of a combined state-based XFA that avoids the state space explosion for signatures of the type shown in Fig. 2. It should be appreciated that only the automaton for the first signature is different. Here, the sequence of 200 states is replaced by a counter. When the combined state-based XFA

shown in Fig. 4 recognizes the starting subpattern "\na" in the input string at a given state M, the counter is initialized to 0. The counter is then incremented for every input character until it reaches 200, at which point the combined state-based XFA shown in Fig. 4 accepts the starting subpattern "\na".

[0051] If the combined state-based XFA shown in Fig. 4 sees a newline character after the counter was initialized but before accepting the input, this combined state-based XFA transitions to state L, where the program invalidates the counter. Alternatively, as outlined below with respect to Figs. 5-8, for a program associated with a transition instead of a state, the counter is invalidated as the XFA transitions to state L (i.e., as it traverses the edge between K and L or M and L). Combining such state-based XFAs with those recognizing strings leads to no explosion in state space. Similarly to Fig. 3, substituting a counter in the combined state-based XFA shown in Fig. 4 for explicit counting states in the DFA shown in Fig. 2 allows the combined state-based XFA shown in Fig. 4 to avoid replicating ".*bc" when the component state-based XFAs are combined. It should be appreciated that some transitions or edges have been omitted from the rightmost automaton shown in Fig. 4.

[0052] **Definition 1.1** An extended state-based nondeterministic finite automaton (XNFA), i.e., an XNFA of the first exemplary type, is denoted by a 9-tuple $(Q, \Sigma, \delta, Q_0, F, D, d_0, R, c)$, where:

[0053] Q is the set of states, Σ is the set of inputs, $Q_0 \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, δ is a function from $Q \times (\Sigma \cup \varepsilon)$ to 2^Q.

[0054] D is a finite data domain,

[0055] $d_0 \in D$ is the initial value,

[0056] R is the update relation and is a map from S to 2^{DxD}, and

[0057] c is the acceptance condition and is a map from D to {0,1}.

[0058] It should be appreciated that a state-based XNFA has four more components than a NFA. An XNFA should be viewed as a NFA with a global variable v whose value is drawn from the finite domain D. Initially, the state-based XNFA is in a state $q_0 \in Q_0$, with the value of the variable v as d_0. For each state $q \in Q$, R(q) denotes the allowable changes to the global variable in state q, i.e., if the state-based XNFA is in state q, the value of global variable v is d, and $(d, d') \in R(q)$, then the value of v can be updated to d' in the next state. The last component c of an

XNFA is called the acceptance condition, i.e., a string $\sigma \in \Sigma^*$ is accepted by the state-based XNFA if there exists a path from a initial state $q_0 \in Q_0$ to a final state $q_i \in F$, which is labeled with σ and the global variable v has value d after an update in the final state q_i such that $c(d) = 1$.

[0059] A state-based XNFA is equivalent to a NFA because the value of the global variable can always be incorporated into the state. Let $XF = (Q, \Sigma, \delta, Q_0, F, D, d_0, R, c)$ be a state-based XNFA. The corresponding NFA $M_{XF} = (Q \times D, \Sigma, \delta', Q'_0, F')$ is defined as:

[0060] δ is a function from $(Q \times D) \times (\Sigma \cup \varepsilon)$ to $2^{Q \times D}$, where given a state $(q, d) \in Q \times D$ and a $\alpha \in \Sigma \cup \varepsilon$, $(q', d') \in \delta'((q, d), \alpha)$ iff $q' \in \delta(q, \alpha)$ and $(d, d') \in R(q)$.

[0061] The set Q'_0 is equal to $Q_0 \times \{d_0\}$.

[0062] A state $(q, d) \in F'$ iff $q \in F$ and $c(d) = 1$.

[0063] It is easy to see that a string σ is accepted by a state based XNFA XF if and only if it is also accepted by the corresponding NFA M_{XF}. Therefore, the language L(XF) accepted by a state-based XNFA XF is equal to the language $L(M_{XF})$ accepted by the corresponding NFA M_{XF}.

[0064] **Definition 1.2** A deterministic state-based extended finite automaton (XFA), i.e., an XFA of the first exemplary type, is denoted by a 9-tuple $(Q, \Sigma, \delta, q_0, F, D, d_0, R, c)$, where:

Q is the set of states, Σ is the set of inputs, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, δ is a function from $Q \times \Sigma$ to Q.

[0065] D is a finite data domain,

[0066] $d_0 \in D$ is the initial value,

[0067] R is the update function and is a map from S to D^D, and

[0068] c is the acceptance condition and is a map from D to $\{0,1\}$.

[0069] The data domain D is the set of all possible values of the scratch memory and d_0 is the value that the scratch memory is initialized to. The update function U represents the programs associated with the automaton states. For each possible combination of an automaton state and a value, U defines the new value to be stored in the scratch memory. The acceptance condition c specifies for which combinations of final states and data values that the XFA will accept. The method or algorithm for applying the XFA to an input is presented below in Fig. 12.

[0070] Fig. 5 shows an exemplary DFA similar to the DFA shown in Fig. 1, where n = 2, s_1 = ab, s'_1 = cd,s_2 = ef, and s'_2 = gh. As indicated above with respect to Fig. 1, 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. To remember n independent bits of information, the DFA needs at least 2^n distinct states. In Fig. 5, when the DFA is in state PV, it "knows" that the input processed so far does not contain ab or ef. In contrast, when the DFA shown in Fig. 5 is in state RV, the input processed so far contains ab but not ef, when the DFA is in state PX, the input processed so far contains ef but not ab, and when the DFA is in state RX, the input processed so far contains both ab and ef. A closer analysis of the general example shows that, if the strings are of length l, the actual number of states used by the combined DFA is $O(nl2^n)$.

[0071] Fig. 6 shows the same signatures as in Fig. 5, but with the DFAs being replaced with transition-based XFAs, i.e., second exemplary type XFAs. In Fig. 6, the transition-based XFAs for ."*ab.*cd" and "*ef.*gh" each use a single bit of scratch memory that is manipulated by instructions attached to specific transitions or edges. In the transition-based XFAs shown in Fig. 6, these instructions are illustrated as callout boxes. During matching, these instructions are executed each time the transition or edge to which they are attached is traversed. For each signature of the form .*s_i.*s'_i, as long as s_i does not overlap with s'_i, a transition-based XFA similar to those in Fig. 6 can be built that uses a single bit of scratch memory. This bit explicitly encodes whether s_i has appeared in the input so far. Moreover, the shape of the underlying automaton is very similar to that of the combined DFA recognizing .*s_i, and .*s'_i independently. The combined XFA for the entire signature set uses n bits and O(nl) states. Of course, if s_i and s'_i overlap, it is still possible to build a transition-based XFA recognizing the signature, but that transition-based XFA will have to use more than one bit of scratch memory.

[0072] Thus, as shown in Fig. 6, by adding n bits of scratch memory, a combined transition-based XFA that is approximately 2^n times smaller than the combined DFA shown in Fig. 5 can be obtained. However, it should be appreciated that, while the processing time is reduced, the initialization time goes up from O(1) to O(n). Similarly, 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.

[0073] Transition-based XFAs can provide large reductions in the number of states even when recognizing individual signatures. Figs 7 and 8 show another exemplary DFA and a corresponding transition-based XFA, respectively, that each recognize the language defined by ".{n}", which consists of all strings of length n. While no NIDS signatures have this exact form, signatures detecting buffer overflows use sequences of states similar to those in Fig. 7 to count the number of characters that follow a given keyword. The minimal DFA for .{n} needs n + 2 states, while the corresponding transition-based XFA uses a single state and a counter. This counter is initialized to 0 and is incremented on every transition. The transition-based XFA shown in Fig. 8 accepts the input string only if the value of the counter is n. The counter only needs to count from 0 to n + 1. When the counter has value n + 1 and it is incremented, it stays n + 1. Thus, the counter needs to take only n + 2 values, so it needs $k = [\log_2(n + 2)]$ bits of scratch memory. By adding k bits of scratch memory, the number of states is reduced by a factor of close to 2^k. If run time is measured in bit operations, the initialization cost and the per byte processing increase from O(1) to O(k). In contrast, if run time is measured in instructions, a small constant is added to both the initialization cost and the per byte processing.

[0074] It should be appreciated that the scratch memory used by transition-based XFAs is represented as a finite data domain D. Any configuration of the scratch memory that is possible during matching is represented as a data value $d \in D$. Each transition is associated with an update function $U : D \rightarrow D$ (or, for non-deterministic XFAs, with an update relation $U \subseteq D \times D$), which specifies how d is to be updated. For the common special case where the data domain not updated by a transition, the identity function is associated with such transitions. Since the automaton is extended 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 a "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 $F \subseteq Q \times D$. It should be appreciated that this definitions of transition-based XFAs set forth above generalizes the standard DFA definition.

[0075] In various exemplary embodiments according to this invention, a transition-based extended finite automaton (XFA) is defined by a 7-tuple $(Q,D,\Sigma,\delta,U_\delta,(q_0,d_0),F)$.

[0076] **Definition 2.1** An extended transition-based deterministic finite automaton (XFA), i.e., an XFA of the second exemplary type, is denoted by a 7-tuple $(Q,D,\Sigma,\delta,U_\delta,(q_0,d_0),F)$, where:

[0077] Q is the set of states, Σ is the input alphabet, and δ: $Q \times \Sigma \rightarrow Q$ is the transition function;

[0078] D is the finite set of values in the data domain,

[0079] $U_\delta : Q\times\Sigma\times D \rightarrow D$ is the per transition *update function* which defines how the data value is updated on every transition,

[0080] (q0, d0) is the initial configuration which consists of an initial state q0 and an initial data value d0, and

[0081] $F \subseteq Q\times D$ is the set of accepting configurations.

[0082] Nondeterministic transition-based XFAs (XNFAs) differ from deterministic transition-based XFAs in a number of important ways. For example, XNFAs replace deterministic transitions with non-deterministic transitions, they define ϵ transitions, and instead of update functions, each transition has an update relation that can take a data domain value to multiple values. Furthermore, instead of an initial configuration (q0, d0), XNFAs have a set of initial configurations QD_0.

[0083] **Definition 2.2** An extended transition-based nondeterministic finite automaton (XNFA), i.e., an XNFA of the second exemplary type, is denoted by a 7-tuple $(Q,D,\Sigma,\delta,U_\delta,QD_0,F)$, where:

[0084] Q is the set of states, Σ is the input alphabet, and $\delta \subseteq : Q \times (\Sigma \cup \{\epsilon\}$ $\times Q$ is the nondeterministic relation describing the allowed transitions;

[0085] D is the finite set of values in the data domain,

[0086] $U_\delta : \delta \rightarrow 2^{D\times D}$ is the *nondeterministic update function* (or update relation) that defines how the data value is updated on every transition,

[0087] $QD_0 \subseteq Q\times D$ is the set of initial configurations of the NXFA, and

[0088] $F \subseteq Q\times D$ is the set of accepting configurations.

[0089] During the construction procedure for transition-based XFAs, the data domain D is represented explicitly as a set of integers, the per transition update functions U_δ are represented as unstructured sets of pairs (d_i, d_f), and F is represented

as a set of configurations. These are intermediate representations. The final transition-based XFA that performs the signature matching uses a much more compact representation. In this more compact representation, D is not represented explicitly, and small programs are associated with states and transitions. Thus, the amount of memory required is not much larger than that for a DFA based on Q and δ. These data domains used by the final XFAs during matching are referred to herein as "efficiently implementable data domains" (EIDDs).

[0090] Fig. 9 is a block diagram illustrating the steps involved in constructing transition-based or state-based XFAs and using such XFAs according to this invention in a NIDS. The first step of crafting the NIDS signatures is outside the scope of this disclosure, as no changes to the semantics of the signatures are required to implement the XFAs according to this invention. Rather, the XFAs according to this invention change how such NIDS signatures are represented during matching. As shown in Fig. 9, a set of regular expressions for complex NIDS signatures are first extended by adding annotations to some or all of the regular expressions that indicate when to use scratch memory operations. Then, each regular expression is converted or compiled into a state-based and/or transition-based XFA that recognizes the language defined by that regular expression. In this case, each annotated regular expression will include state-based and/or transition based instructions that indicate how a specified bit, counter or other memory structure is to be initialized or updated.

[0091] Next, the resulting individual state-based and/or transition-based XFAs are combined into a single combined state-based and/or transition-based XFA that recognizes all signatures of the set simultaneously. This combined state-based and/or transition-based XFA is then supplied with an input string of network traffic so that any portions of the network traffic matching one or more of the signatures can be identified by that XFA.

[0092] Transforming a regular expression into an XFA requires striking a balance between using scratch memory to achieve the goal and using states and transitions. At one extreme, a (possibly large) DFA that uses no scratch memory can be produced. At the other extreme, a (possibly slow) program that does not rely on state information at all can be produced. There are regular expressions for which the XFA according to this invention lies at one of these extremes. For expressions such as .*s, where s is a string, it is best to use a DFA with no scratch memory at all. At

the other extreme, the exemplary XFA shown in Fig. 8, which recognizes .{n}, illustrates how an XFA according to this invention can turn into a program. That is, in the XFA shown in Fig. 8, there is a single state that does not influence at all how the scratch memory is updated or when acceptance happens. When building an XFA, annotations are used to control where the resulting XFA lies along this spectrum.

[0093] There are two types of constructs that result in scratch memory objects being added to the XFA. The first, a parallel concatenation operator, which can be designated, for example, by a "#", adds a bit to the nondeterministic form of the scratch memory. In contrast, the second, an integer range, which can be designated, for example, by "{m,n}", adds a counter. The parallel concatenation operator '#' has the same semantics with respect to the language recognized as the concatenation of regular expressions. Fortunately, integer ranges, a form of syntactic sugar to the basic regular expression syntax, are already present in the signatures wherever appropriate. Thus, annotating the regular expression typically only requires deciding where to use the parallel concatenation operator "#". In an implementation developed by the inventors, this decision is a partly manual step.

[0094] The use of the parallel concatenation operator "#" can be describe as "breaking" the regular expression into sub-expressions that resemble regular expressions that define string matching: .*s, where s is a string. Another way to describe this strategy for adding the parallel concatenation operator "#" is to add the parallel concatenation operator "#" right before sub-expressions such as ".*" and "[^\n]{300}" that repeat characters from either the whole input alphabet or a large subset of the input alphabet. Table 1 shows examples of regular expressions representing actual NIDS signatures from a test set that have been annotated with the parallel concatenation operator "#".

Signature Number	Signature text
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

[0095] It should be appreciated that, when annotating the SNORT signature number 2267, no parallel concatenations are used, as the expression is sufficiently string-like. This signature will be compiled to an XFA without any scratch memory, and thus is identical to the corresponding DFA. When annotating the SNORT signature number 3466, a parallel concatenation is not inserted in front of "\s*", as the character class \s contains few characters (the white spaces). For signatures such as the SNORT signature number 1735, which is a union of subexpressions, the rules for inserting a parallel concatenation to the sub-expressions of the union are simply applied to each sub-expression separately. A parallel concatenation operator is not inserted in front of the ".*"s at the beginning of these sub-expressions, as it would actually be syntactically invalid.

[0096] In various exemplary embodiments, an XFA compiler according to this invention takes the annotated regular expressions and transforms them into deterministic XFAs. The stages are the same as for traditional DFA compilers using the Thompson construction, as disclosed in "Programming techniques: Regular expression search algorithm", K. Thompson, *Commun.ACM*, 11(6):419–422, 1968. The Thompson construction comprises parsing the regular expression, building a non-deterministic automaton through a bottom-up traversal of the parse tree, ϵ-elimination, determinization and minimization. Each of these steps is modified to handle the scratch memory and to implement new cases that handle the annotations added to the regular expressions.

[0097] One exemplary embodiment of a procedure for constructing an NXFA from the parse tree according to this invention extends the traditional steps with provisions for manipulating the data domains and the data-dependent components of the NXFAs. In various exemplary embodiments, two new steps that extend the data domain are added. In particular, in such exemplary embodiments, the step for parallel concatenation adds a bit and the step for integer ranges adds a counter. For brevity, simplified versions of these steps, which build on NFAs corresponding to the subexpressions these constructs apply to, are presented herein.

[0098] The cost that XFAs pay for these large reductions in state space is a slight increase in per-byte processing. That is, when certain states of an automaton are reached or certain transitions are traversed during matching, a corresponding small program that updates data in the scratch memory needs to be run. In

experiments, the inventors have discovered that many states and/or transitions have either no programs associated with them or have programs that comprise just one or two instructions.

[0099] From a complexity theoretic point of view, XFAs are equivalent to DFAs since XFAs, like DFAs, use a finite amount of state. In various exemplary embodiments, this finite amount of state includes the pointer to the current state of the automaton and the scratch memory. One way to explain the large reductions in state space compared to DFAs is to view the XFA's state space as a compressed version of the DFA's state space that exploits certain regularities or redundancies in the transitions of the DFA. That is, instead of redundant states, the XFAs use the values stored in the scratch memory to differentiate between states with otherwise similar transitions. Another way to explain this large reduction in state space is by considering how the two solutions handle multiple "independent patterns" that can be interleaved arbitrarily in the input. DFAs need separate automaton states for each distinct interleaving. For example, with the two subpattern signatures presented in Figs. 1 and 3, the order in which the subpatterns appear in the input is relevant, but the distance between them, in terms of intervening characters, is not. However, with the signatures presented in Figs. 2 and 4, the number of characters between significant subpatterns also matters. In contrast, XFAs use separate bits and counters in scratch memory that can be updated independently. Since the state of the XFA is determined by both the automaton's state pointer and the scratch memory, working together, the XFA will still be in different states for the different interleavings it must distinguish between. However, unlike DFAs, in XFAs the differences are contained in scratch memory values rather than by using distinct automata states. Table 2 summarizes the scaling of the state space for XFAs and DFAs for the various types of signatures discussed earlier. As shown in Table 2, for many popular signature types, XFAs scale much better than DFAs as the number of signatures n grows.

Type of signatures tracked	DFA size number of states	XFA size	
		number of states	scratch memory
.*abcd	$O(n)$	$O(n)$	0
.*a.*b	$O(2^n)$	$O(n)$	$O(n)$
.*a.*b and .*abcd	$O(n2^{n/2})$	$O(n)$	$O(n)$
.*\na[^\n]{k}	$O(nk)$	$O(n)$	$O(\log(k) + \log(n))$
.*\na[^\n]{k} and .*abcd	$O(n^2k)$	$O(n)$	$O(\log(k) + \log(n))$

Table 2

[0100] Because XFAs extend DFAs, methods (i.e., algorithms) for combining and matching DFAs must be adapted to properly process XFAs. By definition, XFAs have all of the components of DFAs, such as, for example, states, transition functions, a start state, and accepting states. XFAs then extend a scratch memory that can be updated via simple instructions attached to the states and/or transitions. In various exemplary embodiments, the instructions are restricted to manipulation of scratch memory and raising alerts although this can also be permitted. In the following exemplary embodiments, the instructions consume no input and induce no state changes.

[0101] In the following exemplary embodiments, the instructions include instructions to manipulate three kinds of structures that can be provided in the scratch memory: bits, counters, and offset counters, which are a specialized form of counters. An additional instruction type, called an "alert", is used to signal matched rules and does not manipulate the memory. It should be appreciated that, in various exemplary embodiments, instructions include an instruction type, an id, and an operation, which is represented textually as a three element list. Each distinct bit, counter, and offset counter is assigned a unique location in scratch memory that is referenced by the id value contained in the associated instructions. For example, the instruction [bit,3,set()] has a bit type and sets bit 3 in the scratch memory when executed. It should be appreciated that the set of instructions outlined herein is not fixed or set in stone. Any kinds of instructions that may be desired or useful can be defined for a given implementation of an XFA according to this invention. The instructions defined above and in the following paragraphs are exemplary only, and thus a subset of those that could be used, but are not intended to be complete or definitive.

[0102] Instances of bits, counters, and offset counters become active in scratch memory when they are set and go inactive when they are reset, as described below. Manipulating an active instance changes its data in scratch memory, whereas manipulating an inactive instance is a no-operation (no-op) CPU operation. In various exemplary embodiments, all data structures in scratch memory are initially inactive.

[0103] In various exemplary embodiments, instructions are fully composable, meaning that they can be arbitrarily nested, subject to proper syntax and

semantic rules. This composability is naturally seen in operations that test a counter or bit value: if the test evaluates to true, then a consequent, which is itself just another instruction, is executed. Table 3 provides a concise summary of each instruction.

Type & Operation	Description	Sample Usage
bit set	sets a bit to 1	[bit, *id*, set ()]
bit reset	resets a bit to 0	[bit, *id*, reset ()]
bit test	test & exec consequent	[bit, *id*, test (*consequent*)]
ctr set	sets init value, max value, and consequent	[ctr, *id*, set (max, *consequent*)]
ctr reset	invalidates a ctr	[ctr, *id*, reset ()]
ctr incr	increment, compare to max, and exec consequent	[ctr, *id*, incr ()]
offset ctr set	insert entry on offset List	[off, *id*, set (max, *consequent*)]
offset ctr reset	remove entry from List	[off, *id*, reset ()]
alert	raise an alert	[alert, *rule_id*]

Table 3

[0104] The "bit" type instruction provides operations for setting, clearing, and testing bit values stored in scratch memory. The "set()" instruction sets the designated bit to 1 and marks it as active. The "reset()" instruction sets the bit to 0 and marks it as inactive. The "test(consequent)" instruction tests the value of the designated bit.

[0105] In the exemplary XFA shown on Fig. 3, state L of Fig. 3 contains the instruction [bit,1,set()], while state M contains the instruction [bit,1,test([alert,sig1])]. The specified "consequent" instruction will be executed if and only if the bit has value 1. Counter instructions likewise manipulate numeric values that can be set, reset, and incremented.

[0106] The "set(max,consequent)" instruction initializes a counter to 0 and marks the counter as active. The maximum value "max" is stored in the counter's data structures along with the "consequent" instruction that will be executed if the counter reached is the "max" value.

[0107] The "reset()" instruction resets a counter, marking it as inactive.

[0108] The "incr()" instruction performs an increment-and-test operation. For an active counter, the "incr()" instruction increments a counter's value and tests the new value against the maximum value supplied in the "set()" operation. If the new value matches the specified "max" value, the consequent instruction (also

supplied in the set operation) is executed. For inactive counters, a "no operation" is performed. Table 3 shows that the id field contains a valid numeric instance id. consequents, which are simply instructions themselves.

[0109] Referring back to the state-based XFAs shown in Fig. 4, rewriting the instructions shown in Fig. 4, the state L contains [ctr,1,reset()], the state M contains the instruction [ctr,1,set(200,[alert,sig1])], and state K contains the instruction [ctr,1,incr()]. It should be appreciated that, even though the counter is incremented in the start state K, the counter is initially inactive. Thus, regardless of how long this XFA remains in state K, the increment instruction has no effect until the counter is activated in the state M.

[0110] It should be appreciated that, in various exemplary embodiments, there is a slight asymmetry between the counter operation and the bit operation. For example, the consequent for the bit type is provided in the test() operation. In contrast, the consequent is given in the set() operation for counters. This reflects a conscious design decision that leads to simpler programs and enhances the optimizations described below.

[0111] The "offset counter" type instruction is a variation of the counter type instruction described above. Whereas counters are incremented by an explicit invocation of the incr() operation, offset counters are incremented each time a byte is read from the input. Offset counters can be implemented as counters with incr() operations on all (or most of) the states and/or transitions, but this leads to a large numbers of instructions, for example, on subpatterns with repetition count specifiers (e.g., [^\n]{200}). Instead, in various exemplary embodiments, offset counters are treated as a distinct type and provide an alternate implementation that is more efficient without changing counter semantics.

[0112] The "set(max, consequent)" instruction marks the offset counter as active and specifies the maximum offset value "max" relative to the current matching offset at which point the consequent is executed.

[0113] The "reset()" instruction inactivates the offset counter.

[0114] When activated via the set() instruction, the offset counter values increment on every input symbol until the maximum value "max" is reached. At that point, the associated "consequent" instruction specified on the counter instruction associated with the current state is executed. Increased efficiency over standard

counters comes from the elimination of explicit "incr()" instructions and the use of an auxiliary list that maintains offset counter values and provides efficient access to them.

[0115] As shown in Fig. 4, counters can be replaced with offset counters to reduce program size. In this case, the state L contains the instruction "[off,1,reset()]", the state M contains the instruction "[off,1,set(200,[alert,sig1])]", and the start state K contains no instruction.

[0116] The "alert"-type instruction simply raises an alert. As shown in Table 3, unlike the other types of instructions that can be associated with a state, the "alert"-type instruction contains only a rule id and the alert identifier.

[0117] Data structures for each type of data, i.e., bits, counters and the like, require minimal amounts of scratch memory. The bit data structure requires only a simple bit of memory, while a counter needs 12 bytes of memory, to store the current and maximum values and a consequent pointer, and an offset counter needs 8 bytes, to store the maximum value and consequent pointer. This counter data structure can be optimized to reduce its memory requirements to 8 bytes, by initializing the counter to the max value and counting down to 0 instead of initializing the counter to zero (the default situation) and testing to determine if the specified max value has been reached.

[0118] Fig. 10 shows one exemplary embodiment of a method or algorithm for combining two XFAs. Fig. 11 shows one exemplary embodiment of a method or algorithm for matching an XFA against a packet payload or a buffer of bytes. In particular, Figs. 10 and 11 are pseudocode representations of these methods or algorithms. In Fig. 10, each state in the combined machine represents a pair of states, one from each original machine. Instructions are copied from old states to the new "paired" states.

[0119] The method or algorithm shown in Fig. 11 is similar to that for matching DFAs to packet payloads, with a few notable exceptions. First, as with the Aho-Corasick string matching method, the payload is applied to the XFA until it is exhausted, not just until a match occurs. Second, instructions attached to a given state S must be executed whenever that state S is reached, which is performed by the "execInstrs" function specified in the pseudocode shown in Fig. 11 at lines 2 and 5.

[0120] As specified in lines 6-8 of the pseudocode shown in Fig. 11, the offset counter values must be checked to determine if the max value has been reached each time a new byte is read. In various exemplary embodiments, an auxiliary list, the "offsetList", is used to hold the offset counter values in sorted order. If the value at the head of the "offsetList" matches the current offset (i), then the "consequent" associated with the current state is executed and the offset list entry is removed. In the implementation shown in Fig. 11, checking the offset list and the "reset()" operation complete in O(1) time, while the "associated with the current status set()" operation completes in O(m) time, where m is the number of active offset counters. Alternate implementations, such as timing wheels can be used to reduce the O(m) traversal costs.

[0121] Raising an alert (e.g., recognizing a signature) is an instruction in XFAs, so no explicit acceptance check is necessary. If an alert needs to be raised, it is processed identically to any other instruction.

[0122] The method or algorithm for combining two XFAs is performed offline prior to the signature matching method shown in Fig. 11. In the method or algorithm shown in Fig. 10, each state in the combined machine corresponds to an ordered pair of states in the first and second input machines, respectively. In line 5 of the pseudocode shown in Fig. 10, the start state of the combined machine initializes a worklist. In line 15, each newly created state adds to this worklist. The method or algorithm shown in Fig. 10 iterates until the worklist is empty, when all combined states have been created and processed. Since the number of states in the two input machines is finite, the method or algorithm must terminate.

[0123] Lines 13 and 14 of the pseudocode shown in Fig. 10 add instructions to combined states from their original counterparts. For each state that is formed by combining the states s and t, i.e., $q = < s, t >$, in a combined machine c, the instructions from states s and t are simply copied into the state q. The correctness of this follows from the fact that entering state q when matching machine c is equivalent to entering states s and t of the two original machines simultaneously, implying that the instructions in both states s and t need to be executed. It should be appreciated that reduction in state space requirements comes from the XFA representation itself and not from the combination method or algorithm, which is a modification of the classic method or algorithm for combining DFAs. During combination, it is likely

that distinct XFAs will use the same id numbers in instructions. Although not shown, in the exemplary embodiment shown in Fig. 10, the method algorithm first scans through each XFA to be added and relabels all ids already in use.

[0124] Some of the XFAs that have been used were manually constructed. The automated methods used to build other XFAs rely on heuristics that are not formally proven to preserve the semantics of the regular expressions used to specify the signatures. Thus, a method or algorithm for determining whether XFAs correctly implement the semantics of signatures is desirable. A reference DFA, DFA_{ref}, is used in this process. This reference DFA is constructed from the signature using provably correct methods. The equality of the languages recognized by the XFA and DFA_{ref} are tested by converting the XFA into a DFA, DFA_{equiv}, which can be compared against DFA_{ref} using well known techniques.

[0125] It should be appreciated that, in the formal definition of an XFA set forth above, some aspects of pattern matching are simplified to keep the presentation short. For example, the data domain is treated as an opaque set and the fact that the scratch memory has more structure (e.g., semantically meaningful bits and counters) are ignored. Also, for XFAs designed for the IPS signature matching task described herein, the actual signature matching algorithm gives the number of the rule that was matched when a given state produces an alarm, not just a binary accept/reject decision. It should also be appreciated that the goal of signature matching in IPSes is not to determine whether a string representing a packet or a protocol field matches the known signature, but to determine whether any prefix of the string matches. This is achieved by moving lines 6 and 7 from Fig. 12 inside the loop.

[0126] Combining all signatures from a particular protocol into a single machine allows the signatures to be analyzed in parallel, but it can result in XFAs that have large numbers of instructions per state and/or per transition, affecting program size and runtime performance. In the worst case, if each individual signature contains a bit, counter, or offset counter and/or other data structure, then the combined XFA may have states and/or transitions with program sizes equal to the number of signatures.

[0127] Fortunately, in some scenarios, two logically distinct bits or counters can be reduced to a single bit or counter. For example, if one counter is active in some set of states and/or transitions and another counter is active in a disjoint set of

states and/or transitions, then the two counters can share the same scratch memory without interference. In general, this leads to reduced scratch memory requirements and, potentially, smaller programs. This scenario is analogous to the register assignment problem in compilers, where multiple variables can share the same register as long as they cannot be simultaneously "live".

[0128] The inventors have developed techniques to automatically identify such independent counters and combine them. These optimizations are actually stronger than in the example above; in some cases, two counters can be reduced to one even if they are active in an overlapping set of states and/or transitions. These techniques apply equally to bits, counters, offset counters, and other appropriate data structures, although for presentation purposes the focus of the following discussion is on counters.

[0129] At a high level, the goal of optimization is to identify pairs of counters whose operations are compatible at each state and/or transition in the XFA. This is achieved through a two-step process. The first step performs a dataflow analysis to determine the states and/or transitions at which a counter is active. The second step uses this information to iteratively find and combine independent counters. This process is illustrated using the example shown in Fig. 13. The leftmost automata in Fig. 13 depict two distinct state-based XFAs corresponding to the regular expressions ".*\na[^\n]{200}" and ".*\nb[^\n]{150}", adapted from Fig. 4, that are combined to give the state-based XFA in the middle of Fig. 13. It should be appreciated that the "cloud" annotations refer to a later stage. In the end, optimization determines that the two counters in the combined state-based XFA are independent and reduces them to one counter.

[0130] Determining whether two counters are reducible to one requires that, for each counter C and for each state and/or transition S, a determination be made whether the counter C is active in a given state and/or transition S or not. The analysis requires a precise definition of active and inactive counters:

[0131] **Definition 3.** Let Q be the set of states and/or transitions containing a set operation for a counter C. Then, the counter C is active at the state and/or transition S if there is at least one sequence of input symbols forming a path of states and transitions from a state in the "setting" set of states and/or transitions Q to the

state and/or transitions S in which no state and/or transition in the path contains a reset operation for the counter C. Otherwise, the counter C is inactive.

[0132] In other words, the counter C is active at the state and/or transition S if and only if there exists at least one input sequence ending at the state and/or transition S containing a set but no subsequent reset for the counter C. The term "activity" refers to the active or inactive status of a counter.

[0133] Counter activity is a dynamic (runtime) property, but optimization is performed statically, so the analysis must return correct results for all possible runtime inputs. Thus, the definition above refers to counters that "may be active", depending on the input provided.

[0134] The counter C's activity can be determined at each state and/or transition by applying a dataflow analysis to the XFA, which results in an activity assignment of the counter C that is consistent with the definition above. In a dataflow analysis, a counter's actual value is abstracted away from and conclusions are drawn based only on the counter's activity, not the actual values it may contain. Let the counter C' be the abstract counter obtained from the counter C. The counter C' takes on values from the domain {active,inactive} and its value is referred to as a dataflow fact. Dataflow analyses produce sound and conservative facts for each counter at each state and/or transition that represent the best information about runtime behavior that can be obtained statically. Facts are obtained by iterating over the states and/or transitions using a fixed-point algorithm until a solution is reached.

[0135] In the exemplary embodiment shown in Fig. 13, the counter minimization process has been applied to the state-based XFAs for the signatures ".*\na[^\n]{200}" and ".*\nb[^\n]{150}". As shown in Fig. 13, the optimization results in eliminating one of the original counters.

[0136] The dataflow analysis is a forward-flow "may" analysis that determines whether counters are definitely inactive or "may" be active at a state and/or transition. The initial value for a fact is inactive. The value lattice orders counter facts. Inactive is the initial value for facts. The top ($\top$) and bottom ($\bot$) nodes are implicit.

[0137] In a dataflow analysis, flow functions define the effects that instructions have on the facts for each counter. Flow functions for this analysis are defined as follows for the abstracted counter C':

[0138] $f_{set}(C') \rightarrow$ Active $f_{incr}(C') \rightarrow C'$

[0139] $f_{reset}(C') \rightarrow$ Inactive $f_{test}(C') \rightarrow C'$

[0140] For the "init" and "reset" operations, C' becomes active and inactive, respectively. The "incr" and "test" operations do not change the value of C'. The "test" operation is used for bit optimization.

[0141] In Fig. 13, the annotations in the middle state-based XFA show the activity of each counter at each state after the dataflow analysis has completed. Both counters are inactive when the state MX is reached, because all paths to the state MX pass through the state LY, which resets both counters. Similarly, the counters are active in the state KX because there is a path from the state MX that sets counter 1, making it active, and a path from the state KZ that sets counter 2 making it active.

[0142] It should be appreciated that two counters or other appropriate data structures can be reduced to one if they are compatible at all states and/or transitions in the automaton. Two counters or other appropriate data structures are compatible at a single state and/or transition S if their operations and activity status can be combined without changing the semantics of either counter or other appropriate data structures associated with that state and/or transition.

		Inactive		Active			
		r,i,p	set	reset	set	incr	pres
Inactive	r,i,p	r,i,p	set	reset	set	incr	pres
	set	set	NC	set	NC	NC	NC
Active	reset	reset	set	reset	set	NC	NC
	set	set	NC	set	NC	NC	NC
	incr	incr	NC	NC	NC	incr	NC
	pres	pres	NC	NC	NC	NC	pres

Table 4

		Inactive		Active			
		r,t,p	set	reset	set	test	pres
Inactive	r,t,p	r,i,p	set	reset	set	test	pres
	set	set	set	set	NC	test	NC
Active	reset	reset	set	reset	set	NC	NC
	set	set	set	set	set	NC	NC
	test	test	NC	NC	NC	NC	NC
	pres	pres	NC	NC	NC	NC	pres

Table 5

[0143] Table 4 shows one exemplary embodiment of a counter compatibility matrix that summarizes state-wise compatibility for all possible counter and activity combinations. Similarly, Table 5 shows one exemplary embodiment of a bit compatibility matrix that specifies which bit operations are compatible at a state and what the surviving operation is.

[0144] In the counter compatibility matrix, the "preserve" column handles the cases in which a counter has no associated instruction in the state in question. The "r,i,p set" inactive column coalesces the entries for the "reset", "increment", and "preserve" operations, which have identical behavior for inactive counters. If two operations are compatible, the corresponding entry holds the operation that could survive when the counters are combined. "NC" indicates that operations are not compatible.

[0145] Active counter operations are incompatible with most other operations, whereas inactive operations are mostly compatible. One exception is inactive set operation, which transitions a counter to the active state and is therefore incompatible with most active operations.

[0146] Entries in the bottom right quadrant of Tables 4 and 5 highlight the importance of the dataflow analysis in determining compatibility. The active portion of the rightmost column handles the cases in which a state has instructions for only one counter, but the dataflow analysis determines that a second counter is also active. Combining the two counters and using the operation of the counter present at the state could change semantics of the second counter, and so the counters are deemed to be not compatible.

[0147] Fig. 14 shows one exemplary embodiment of a pseudocode program that implements, for state-based XFAs, an algorithm for identifying and reducing equivalent counters according to this invention. As shown in Fig. 14, in line 5 of the pseudocode, for each pair of counters, the method algorithm cycles through all states and compares the pair using the "areCompat function". The "areCompat" function extracts counter activity values and operations for a pair of counters c_1 and c_2 at a given state s and invokes the counter compatibility matrix. In lines 7 and 8, the pair of counters c_1 and c_2 are marked as reducible and eligible for reduction if they are found to be compatible at all states.

[0148] Lines 9-13 perform the actual reduction for each compatible pair of counters. When a reduction results in the elimination of one or more instructions at a given state, the operation that remains is returned from the compatibility matrix via a call to the "getReduced" function.

[0149] In the exemplary embodiment shown in Fig. 13, the rightmost state-based XFA shows the combined machine after compatibility analysis has determined that counters 1 and 2 are compatible. As a result, in the combined machine, all references to counter 2 are replaced by a reference to counter 1, and irrelevant "reset" and "incr" operations are removed.

[0150] Counter optimization completes quickly, despite the seeming complexity of the methods. In the implementation illustrated in Figs. 13 and 14, the entire optimization procedure for a single signature set typically completes in less than 10 seconds on a standard uniprocessor workstation.

[0151] The techniques described above apply directly to bits, offset counters and/or other appropriate data structure with only slight modification. Offset counters do not have explicit increment operations, so a "fake" increment instruction is just inserted in each state and/or transition for each offset counter. The procedure above is then performed, after which the increment instructions are removed. For bits, the compatibility matrix varies slightly from the counter compatibility matrix, reflecting the fact that the "consequent" for a bit is supplied in the test operation, rather than in the "set" operation, as occurs for counters.

[0152] Fig. 15 is a transition-based XFA that recognizes .*a.{n}b, where k = n + 2 and illustrates the use of bitmaps in scratch memory. The XFA shown in Fig. 15 recognizes strings matching this regular expression using 2 states and k = n+2 bits

of scratch memory. In contrast, DFAs must use at least 2^(n+1) states to recognize such regular expressions. If the regular expression is annotated as ".*a#.{n}b", and if the appropriate EIDD is built, this XFA can be compiled for small values of n. However, for large values of n, such an XFA is difficult to compile because the XFA represents the data domain as an explicit set and thus a typical compiler will run out of memory when determinizing the update functions. It should be appreciated that small changes to the regular expression turn it into something for which XFAs can built efficiently. For example, it is possible to recognize ".*a[^a]{n}b" with an XFA that has two states and has a data domain of size n + 2 and which is used essentially as a counter.

[0153] The class of regular expression that exhibits this type of behavior also includes all expressions of the type ".*s.{m, n}", where s is a string and expressions where the character class repeating after the string includes all characters in the string. Surprisingly, dozens of such regular expressions exist among Snort's web rules, such as Rule 3519, which recognizes the regular expression ".*wqPassword=[^\r\n&]{294}". It should be appreciated that the actual regular expression defined in Rule 3519 performs case insensitive matching.

[0154] Some simple regular expressions require a state space exponential in the size of the expression. For example, any deterministic automaton recognizing ".*a.{n}b" needs to remember which of the previous n + 1 bytes in the input have been "a", so that the automaton knows whether to accept the input string or not if the automaton sees a "b" in the next n + 1 input characters. DFAs must use at least 2^{n+1} states to be able to distinguish between the relevant inputs. An XFA according to this invention also needs at least 2^{n+1} distinct configurations. However, in contrast to a DFA, the distinctions between these configurations in the XFA are represented by the values stored in scratch memory, not only in the current state of the automaton.

[0155] The inventors conducted experiments to examine the memory usage and runtime performance of XFAs for real signatures. These experiments used intrusion detection signatures from both Cisco Systems and Sourcefire (makers of the popular "Snort" IPS). For each IPS source, the signatures were subdivided into their respective protocols, consistent with the behavior of many commercial IPSes. The results presented below evaluate state-based XFAs using FTP, SMTP, and subdivided

HTTP signatures from both IPSes, for a total of 752 signatures spread over eight signature sets.

[0156] This evaluation compares state-based XFA performance to DFAs, NFAs, and "Multiple DFAs" (mDFAs) constructed using the algorithm given in Yu (see above). The NFAs were implemented by maintaining a frontier of states, thereby avoiding any backtracking. The mDFA implementation computes a group of combined DFAs, whose sum of sizes is guaranteed to fit into a total amount of memory supplied as a parameter. For these experiments, mDFA groups were computed for nine different memory budgets (represented as total numbers of states): 128K, 64K, 48K, 32K, 16K, 8K, 4K, 2K, and 1K total states.

	Num	Machines				Instrs per state		Aux memory (bytes)	
Ruleset	Sigs	# states	#bits	#ctrs	#offs	max	avg	scratch	program
Snort FTP	74	370	9	10	34	45	1.01	285	3840
Snort FTP OPT			8	4	2	6	0.48	45	1680
Snort SMTP	56	1,412	20	10	28	49	1.42	250	19274
Snort SMTP OPT			11	6	9	18	1.05	104	13612
Snort HTTP	271	3,068	90	0	12	30	0.32	83	8276
Snort HTTP OPT			88	0	12	30	0.32	83	8220
Snort HTTP-URI	135	1,701	8	1	3	9	1.05	27	21176
Snort HTTP-URI O			8	1	3	9	1.05	27	21176
Cisco FTP	31	323	17	0	1	15	1.50	8	1754
Cisco FTP OPT			17	0	1	15	1.50	8	1754
Cisco SMTP	96	2,673	16	0	0	11	0.46	2	9928
Cisco SMTP OPT			15	0	0	11	0.27	2	5704
Cisco HTTP-HDR	52	1,759	3	0	0	3	0.01	1	200
Cisco HTTP-HDR O			3	0	0	3	0.01	1	200
Cisco HTTP	37	850	9	0	0	12	2.08	2	14168
Cisco HTTP OPT			8	0	0	11	2.04	1	13864

Table 6

[0157] Table 6 shows the statistics for combined state-based XFAs for several protocols. In each group, the second row gives the number bits, counters, and offset counters after optimization has completed.

[0158] These experiments demonstrated that XFAs are both space and time efficient, with memory footprints close to NFAs and processing times approaching DFAs. These experiments demonstrated that for the most complex signature sets, XFAs are both smaller and faster than mDFAs. In simpler sets, mDFAs are slightly faster but require larger memory footprints. These experiments demonstrated that for complex signature sets, optimization cuts the number of counters, offset counters, and instructions roughly in half. These experiments demonstrated that relative XFA

performance improves with increasing signature set complexity, suggesting that XFAs may scale well to increasingly complex signature sets.

[0159] The XFAs created from the signatures were combined into a single XFA per signature set using the XFA COMBINE method outlined in Fig. 10. The optimization methods described above were then applied to the XFAs. Table 6 also summarizes the combined XFAs and the effects of optimization. Each pair of rows in Table 6 corresponds to a specific signature set and describes the combined XFA first and the optimized combined XFA second. The Machines column group of Table 6 depicts the number of bits, counters, and offset counters in the combined machine. The "Instrs per state column" group summarizes the distribution of instructions among the states, and the "Aux memory" group gives the amount of auxiliary memory needed to provide the scratch memory and to store the states' programs.

[0160] Not all signatures are created equal. Consistent with other observations, Snort rules are more complex than Cisco rules. As Table 6 shows, the combined automata for Snort manipulate many bits, counters and offset counters. In contrast, the Cisco signatures are simpler, and there is correspondingly less opportunity for XFA-specific features to reduce state space explosion. The reason for this discrepancy is protocol parsing: Cisco signatures are applied to parsed protocol fields, whereas Snort does very limited protocol parsing, which is compensated for by more complex signatures.

[0161] For the most complex signature sets, optimization reduces the number of instructions by a factor of two. The maximum number of instructions in a single state is seven times smaller, and the scratch memory size is cut by a factor of six. Fig. 16 shows the distribution of instructions among states before and after optimization for the Snort SMTP signature set. Optimization reduces the number of instructions in a state from 8 or less in most cases to 4 or less. Some states have close to 50 instructions in them. For these states, optimization cuts the number of instructions in half. Fig. 17 shows the distribution of instructions for the Snort FTP signature set.

[0162] The principal metrics of interest in evaluating XFAs are memory usage and execution time. The above-outlined experiments were performed on a 3.0 GHz Pentium 4 Linux workstation that was otherwise idle. Runtime measurements

were collected using cycle-accurate performance counters and are scaled to units of seconds/GB.

[0163] Execution time tests were performed on a 7 minute, 10 GB trace captured on the link between a university campus and a departmental network. During XFA matching, packets in the trace are filtered so that combined XFAs apply only to packets of the same protocol (e.g., combined FTP XFAs are applied only to FTP packets). XFA signature matching was performed using the method or algorithm shown in Fig. 11. NFAs and mDFAs were matched using standard state matching algorithms modified to maintain multiple state pointers.

[0164] Figs. 18 and 19 provide space-time comparisons for Snort and Cisco signature sets, respectively. In each plot, the x-axis (processing time) and the y-axis (memory usage) are both presented on a log scale. Entries toward the bottom left require reduced resources (either in space or in time) and are thus preferred. DFAs and NFAs represent the extreme points at the upper left and lower right corners, respectively. It should be appreciated that, in all but one plot, the DFA point is an underestimate of the actual memory cost. The plus marks ('+') in the plot show the points for each of the nine mDFA instances. In some cases, introducing additional memory did not reduce the number of combined DFA groups, resulting in plot points that end up on top of each other. The points hint at the tradeoffs obtained through pure DFA approaches and suggest lower bounds given specific time or memory requirements.

[0165] XFAs, represented by a star, are in most cases below and to the left of the line drawn out by the DFA-based approaches, indicating that XFAs require fewer resources overall. For the Snort FTP, SMTP, and HTTP signature sets, which are among the most complex, XFAs are many times smaller and faster than mDFAs. In a less complex set, Cisco FTP, the XFA is not always the fastest, but it requires less than 50KB of memory, which is between 10 and 100 times less memory than is required for the mDFAs. Cisco HTTP is the only case in which XFAs are less desirable over mDFAs.

Signature set	DFA	mDFA States			NFA	XFA
	# States	128K	32K	8K	# States	# States
Snort FTP	>21,800,000	94,288(4)	24,777(16)	3,935(17)	4,077	370
Snort SMTP	>21,800,000	98,236(11)	9,667(15)	9,667(15)	9,834	1,412
Snort HTTP	>21,800,000	73,988(23)	21,468(31)	8,672(50)	9,076	3,068
Snort HTTP-URI	>21,800,000	17,193(4)	15,256(4)	4,828(6)	2,436	1,701
Cisco FTP	3,431,667	83,162(2)	22,254(2)	6,318(3)	344	323
Cisco SMTP	>21,800,000	100,474(3)	26,263(3)	6,659(4)	2,666	2,673
Cisco HTTP-HDR	8,242	8,242(1)	8,242(1)	4,947(2)	1,429	1,759
Cisco HTTP	69,765	69,765(1)	16,985(2)	5,423(2)	496	850

Table 7

[0166] Table 7 details XFA memory requirements compared to the memory requirements for DFAs, mDFAs, and NFAs, as measured in numbers of states in each automaton. With regard to memory, NFAs and DFAs show the extreme points for traditional approaches. Most entries for DFAs (the second column) are underestimates, since combining all DFAs require significantly more memory than is available. In some cases, the supplied size may be a gross underestimate: the Snort HTTP set exceeded 21.8 million states after only the first 88 signatures (out of 271) were combined.

[0167] The prototype XFAs used in these experiments stored transition tables as an array of state pointers. Actual memory requirements for DFA subsets, mDFAs, and NFAs are obtained by multiplying the number of states by 1024 (256 pointers × 4 bytes/pointer). For combing DFAs, a compressed state implementation was used that reduced state sizes by approximately 50%. The combination algorithm was executed on a 64-bit machine with 16GB of memory. XFA memory requirements are computed similarly, except that scratch and program memory sizes from Table 6, albeit small, must also be added. It should be appreciated each result represents a single protocol. An IPS would need to simultaneously fit machines in memory for each protocol it examines.

[0168] The mDFA entries in the table represent the memory requirements for three different memory budgets: 128K states, 32K states, and 8K states. In each column, the parenthesized number gives the number of DFA groups produced by the grouping heuristic presented in Yu. Because the heuristic does not do a global optimization, it is possible that the number of resulting groups does not always decrease even when the amount of available memory is increased, as is the case for the Cisco FTP and Cisco SMTP sets.

[0169] The number of mDFA groups echoes the differences in signature set complexity. At the maximum allowed memory setting, Snort signature sets still require many groups (4, 11, and 23, respectively), whereas the Cisco rules can be combined into considerably smaller numbers of groups (2, 3, and 1).

[0170] The NFA entries are the "sum of states" for each signature set and represent, in some sense, the minimal number of states required to represent all the signatures. Even so, XFAs have fewer states in many cases. There are two reasons for this. First, signatures with repetition count specifiers, such as, for example the "([^\n]{200})" signature must use explicit states to keep track of the number of repetitions when implemented as DFAs or NFAs, whereas, with XFAs, a counter or offset counter can be used instead that tracks repetition values inside the scratch memory. Second, when signatures with common prefixes are combined, only a single sequence of states is needed to match the prefix for either signature.

Signature set	DFA	mDFA Runtime			NFA	XFA
	Exec	128K	32K	8K	Exec	Exec
Snort FTP	15.0	71.1(4)	196.2(16)	164.1(17)	2,084.8	33.6
Snort SMTP	8.8	80.5(11)	101.0(15)	101.3(15)	1,442.4	26.7
Snort HTTP	10.1	214.6(23)	316.6(31)	1,169.3(50)	7,158.3	46.6
Snort HTTP-URI	11.9	42.8(4)	41.7(4)	49.3(6)	3,300.8	36.3
Cisco FTP	18.1	24.7(2)	24.3(2)	30.6(3)	240.5	45.5
Cisco SMTP	10.4	21.5(3)	35.9(3)	26.3(4)	2,549.8	23.0
Cisco HTTP-HDR	13.7	12.4(1)	12.4(1)	15.3(2)	1,164.3	15.7
Cisco HTTP	12.6	11.6(1)	14.8(2)	14.7(2)	254.5	41.5

Table 8:

[0171] Table 8 compares runtimes for DFAs, mDFAs, NFAs, and state-based XFAs, as measured in second/GB. In particular, Table 8 compares XFA runtime performance to the other mechanisms, showing quantitative results for the same mDFA data points provided in the memory comparison. As expected, the combined DFA subsets are fastest since they perform the least amount of work per byte. In these exemplary embodiments, this work includes just a transition table lookup and pointer dereference. In contrast, NFAs are the slowest, since the per-byte cost of a DFA is multiplied, roughly, by the number of machines in the protocol group. mDFA runtimes fall between the extremes, in which execution time increases follow the number of groups.

[0172] For the most complex Snort sets, XFAs are at least twice as fast as mDFAs, even when mDFAs have 250 times more memory. For the Cisco signature

sets, which are simpler, mDFAs are often faster but require more memory as shown in the plots in Fig. 18. For example, in the Cisco SMTP set the XFA outperforms the mDFA set with one-third less memory, and requires about one-tenth the memory of a faster mDFA. When both memory and CPU resources are constrained, as is the case for signature matching, XFAs have a distinct advantage.

[0173] While this invention has been described in conjunction with the exemplary embodiments outlined above, various alternatives, modifications, variations, improvements and/or substantial equivalents, whether known or that are or may be presently foreseen, may become apparent to those having at least ordinary skill in the art. Accordingly, the exemplary embodiments of the invention, as set forth above, are intended to be illustrative, not limiting. Various changes may be made without departing from the spirit or scope of the invention. Therefore, the invention is intended to embrace all known or earlier developed alternatives, modifications, variations, improvements and/or substantial equivalents.

IN THE CLAIMS:

1. A method, executing on a data processing system having a controller and a memory, for recognizing one of a plurality of predefined patterns in a data stream received by the data processing system using a recognition network, the recognition network comprising a plurality of nodes, a plurality of transitions connecting the nodes and an amount of memory usable to store at least one of at least one bit data element, at least one counter data element, at least one bytemap element and at least one other data structure, at least some of the transitions including at least one instruction for interacting with at least one of a bit data element, a counter data element, a bytemap element and another data structure, the method comprising:

inputting a stream of data comprising a plurality of ordered data elements;

transitioning from a current node of the recognition network to a next node of the recognition network, the next node being one of the current node and another node of the recognition network, based on a next one of the plurality of ordered data elements of the stream of data;

executing, upon transitioning from the current node, if the current transition includes at least one instruction for interacting with at least one of a bit data element, a counter data element, a bytemap image and another data element, that instruction;

determining, based upon executing the at least one instruction, if an acceptance condition has been met; and

outputting, if the acceptance condition has been met, an alert indicating that the acceptance condition has occurred.

2. A method, executing on a data processing system having a controller and a memory, for recognizing one of a plurality of predefined patterns in a data stream received by the data processing system using a recognition network, the recognition network comprising a plurality of nodes, a plurality of transitions connecting the nodes and an amount of memory usable to store information about the data stream, at least some of the transitions including at least one instruction for interacting with the stored information, the method comprising:

inputting a stream of data comprising a plurality of ordered data elements;

transitioning from a current node of the recognition network to a next node of the recognition network over a transition linking the current node to the next node, the next node being one of the current node and another node of the recognition network, based on a next one of the plurality of ordered data elements of the stream of data;

executing, upon traversing the transition, if the transition includes at least one instruction for interacting with at least one portion of the stored information, that at least one instruction;

entering the next node;

determining if an acceptance condition associated with the next node has been met; and

outputting, if the acceptance condition has been met, an alert indicating that the acceptance condition has occurred.

3. The method of claim 1 or 2, further comprising at least one node including at least one instruction for interacting with at least one portion of the stored memory.

4. The method of claim 3, further comprising executing, upon entering the next node, if the next node includes at least one instruction for interacting with at least one portion of the stored information, that at least one instruction.

5. The method according to any of the preceding claims, wherein executing, upon traversing the transition, if the transition includes at least one instruction for interacting with at least one portion of the stored information, that at least one instruction comprises:

executing at least one instruction for updating at least one of the at least one portion of the stored information.

6. The method according to any of the preceding claims, wherein the amount of memory stores at least one data structure, each data structure associated with at least one transition.

7. The method of claim 6, wherein the at least one data structure includes at least one single-bit data element.

8. The method of claim 6, wherein the at least one data structure includes at least one multi-bit data element.

9. The method of claim 6, wherein the at least one data structure includes at least one bytemap data element.

10. The method of claim 6, wherein the at least one data structure includes at least one counter data element.

11. A recognition network, executable on a data processing system having a controller and a memory, usable to recognize one of a plurality of predefined patterns in a stream of data, the stream of data comprising a plurality of ordered data elements, received by the data processing system, the recognition network comprising:

a plurality of nodes;

a plurality of transitions connecting the nodes; and

an amount of memory usable to store information about the stream of data, wherein at least some of the transitions include at least one instruction for interacting with the stored information.

12. The recognition network of claim 11, wherein, for each transition having at least one instruction, each such instruction comprises an instruction for updating at least some of the stored information.

13. The recognition network of claim 11 or 12, wherein at least one of the nodes includes at least one instruction for interacting with the stored information.

14. The recognition network according to any of claims 11-13, wherein, in response to receiving a next one of the ordered data elements, the recognition network:

transitions from a current node of the recognition network to a next node of the transition network over a transition linking the current node to the next node and associated with the next one of the ordered data elements, the next node being one of the current node and another node of the recognition network;

executes, upon traversing that transition, if that transition includes at least one instruction for interacting with at least one portion of the stored information, that at least one instruction;

enters the next node;

determines if an acceptance condition associated with the next node has been met; and

outputs, if the acceptance condition has been met, an alert indicating that the acceptance condition has occurred.

15. The recognition network according to any of claims 11-14, wherein the amount of memory stores at least one data structure, each data structure associated with at least one of at least one transition and at least one node.

16. The recognition network of claim 15, wherein the at least one data structure includes at least one single-bit data element.

17. The recognition network of claim 51, wherein the at least one data structure includes at least one multi-bit data element.

18. The recognition network of claim 15, wherein the at least one data structure includes at least one bytemap data element.

19. The recognition network of claim 15, wherein the at least one data structure includes at least one counter data element.

20. A computer-readable medium having computer-executable instructions for performing a method comprising:

analyzing a stream of data comprising a plurality of ordered data elements;

transitioning from a current node of the recognition network to a next node of the recognition network, the next node being one of the current node and another node of the recognition network, based on a next one of the plurality of ordered data elements of the stream of data;

executing, upon transitioning from the current node, if the current transition includes at least one instruction for interacting with at least one of a bit data element, a counter data element, a bytemap element and another data element, that instruction;

determining, based at least upon executing the at least one instruction, if an acceptance condition has been met; and

outputting, if the acceptance condition has been met, an alert indicating that the acceptance condition has occurred.

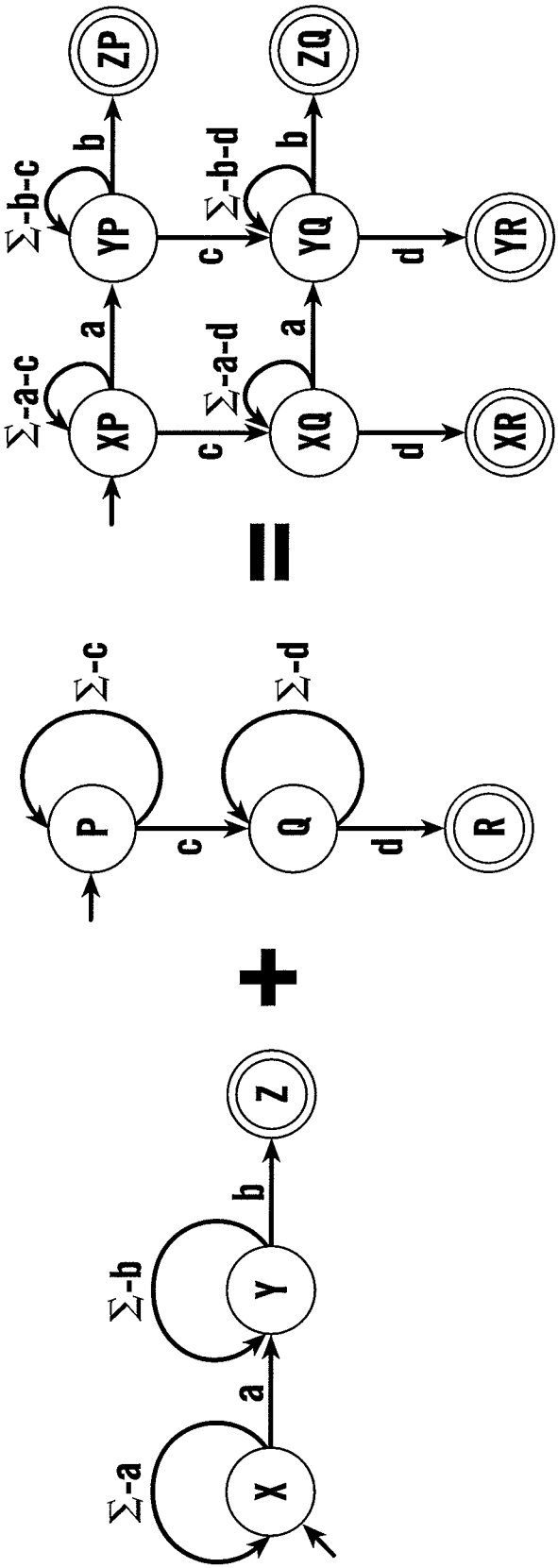

FIG. 1

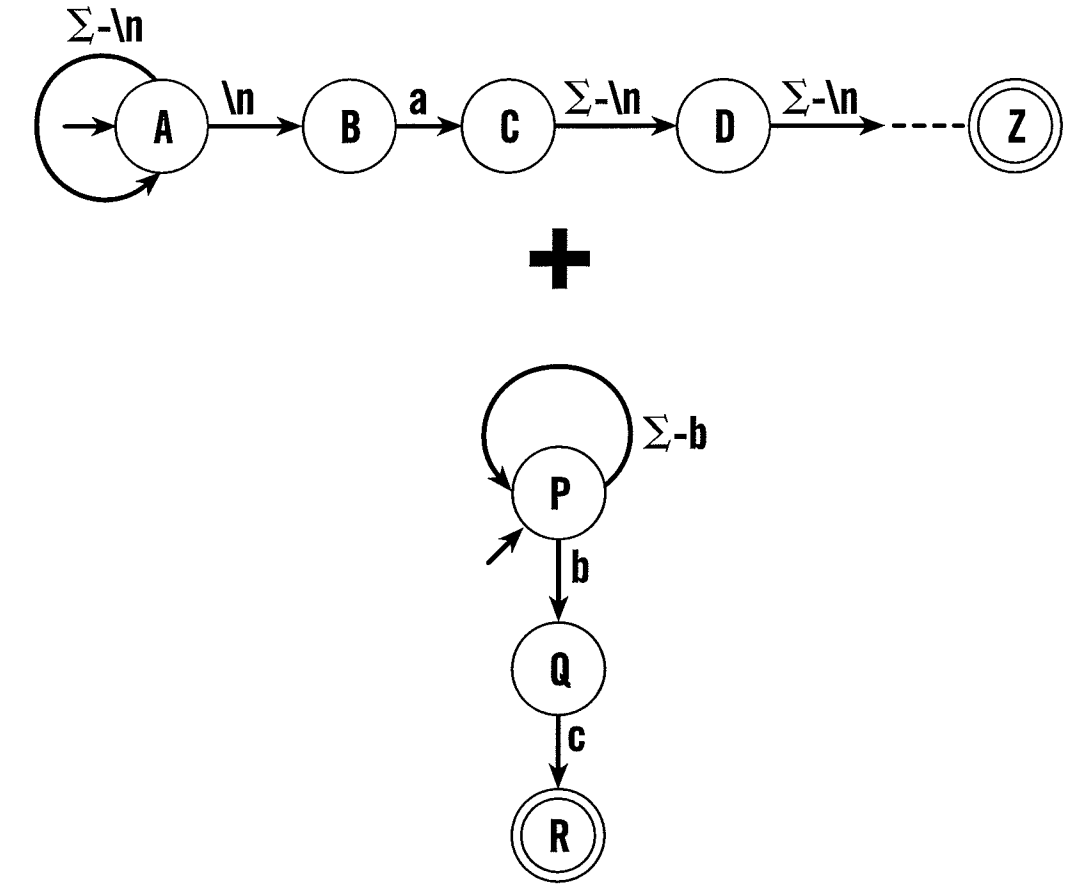
Σ-\n
A
\n
B
a
C
Σ-\n
D
Σ-\n
Z
Σ-b
P
b
Q
c
R
Σ-\n-b
AP
\n
BP
a
CP
Σ-\n-b
DP
Σ-\n-b
ZP
b
b
b
b
AQ
DQ
EQ
c
c
c
AR
ER
FR

FIG. 2

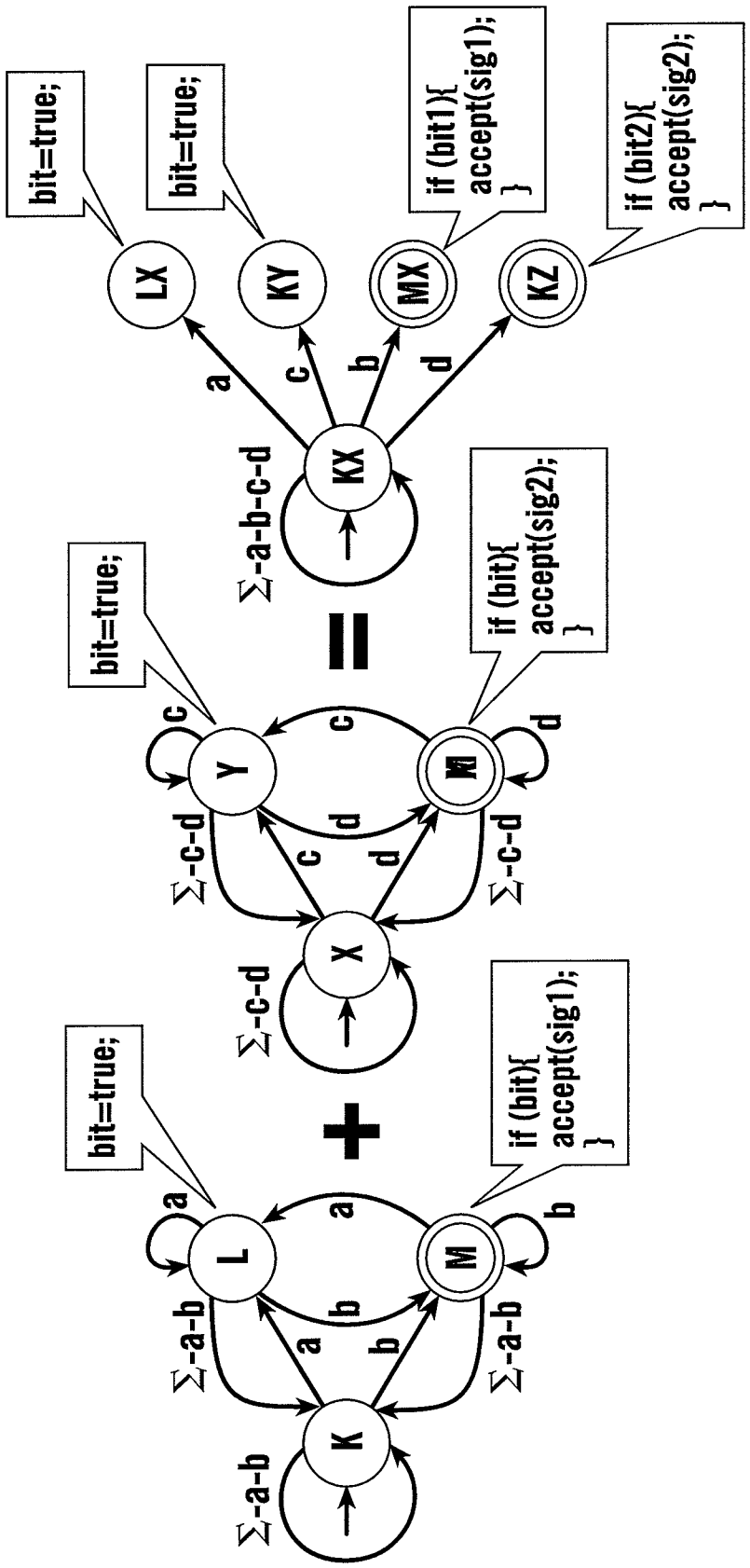
bit=true;
bit=true;
if (bit1){
accept(sig1);
}
if (bit2){
accept(sig2);
}
LX
KY
MX
KZ
a
c
b
d
Σ-a-b-c-d
KX
bit=true;
if (bit){
accept(sig2);
}
c
Y
M
d
Σ-c-d
c
d
X
bit=true;
if (bit){
accept(sig1);
}
a
L
M
b
Σ-a-b
a
b
K

FIG. 3

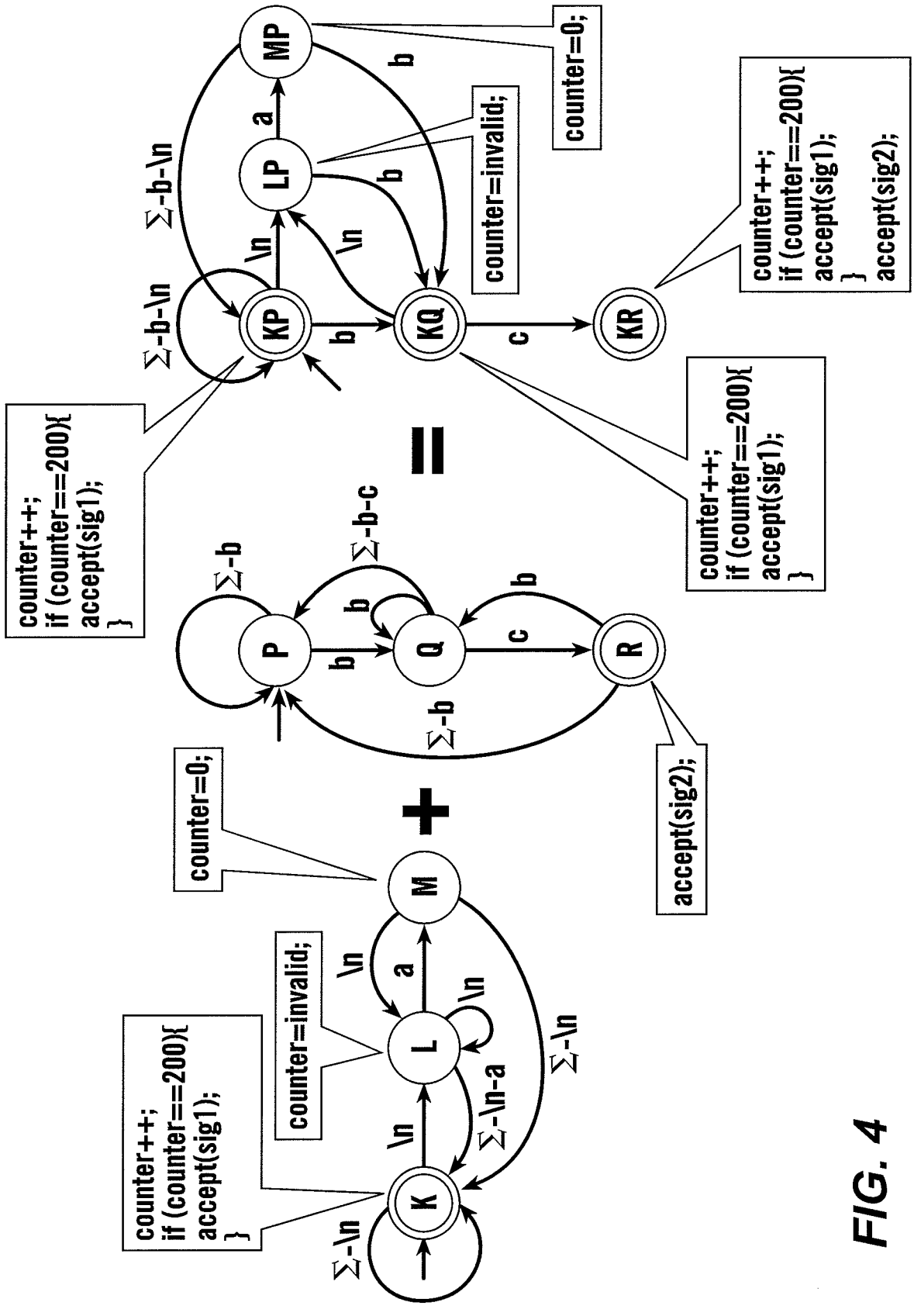
counter++;
if (counter==200){
accept(sig1);
}
counter=0;
counter=invalid;
counter++;
if (counter==200){
accept(sig1);
}
accept(sig2);
counter++;
if (counter==200){
accept(sig1);
}
accept(sig2);
counter=0;
counter=invalid;
MP
LP
KP
KQ
KR
P
Q
R
M
L
K
a
b
c
\n
Σ-b-\n
Σ-b
Σ-b-c
Σ-\n-a
Σ-\n

FIG. 4

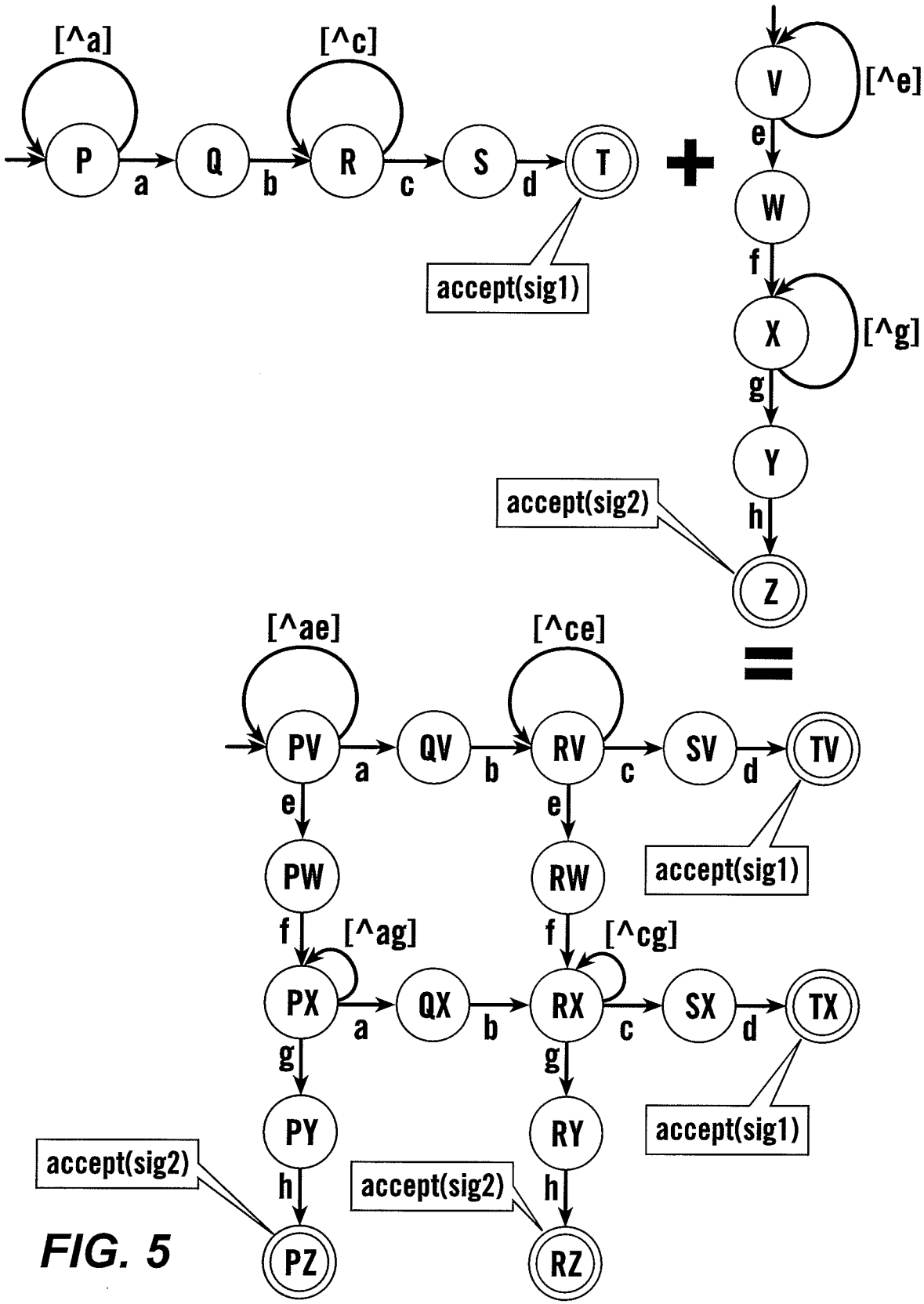
[^a]
[^c]
P
Q
R
S
T
a
b
c
d
accept(sig1)
V
[^e]
e
W
f
X
[^g]
g
Y
h
accept(sig2)
Z
[^ae]
[^ce]
PV
QV
RV
SV
TV
PW
RW
[^ag]
[^cg]
PX
QX
RX
SX
TX
PY
RY
PZ
RZ

FIG. 5

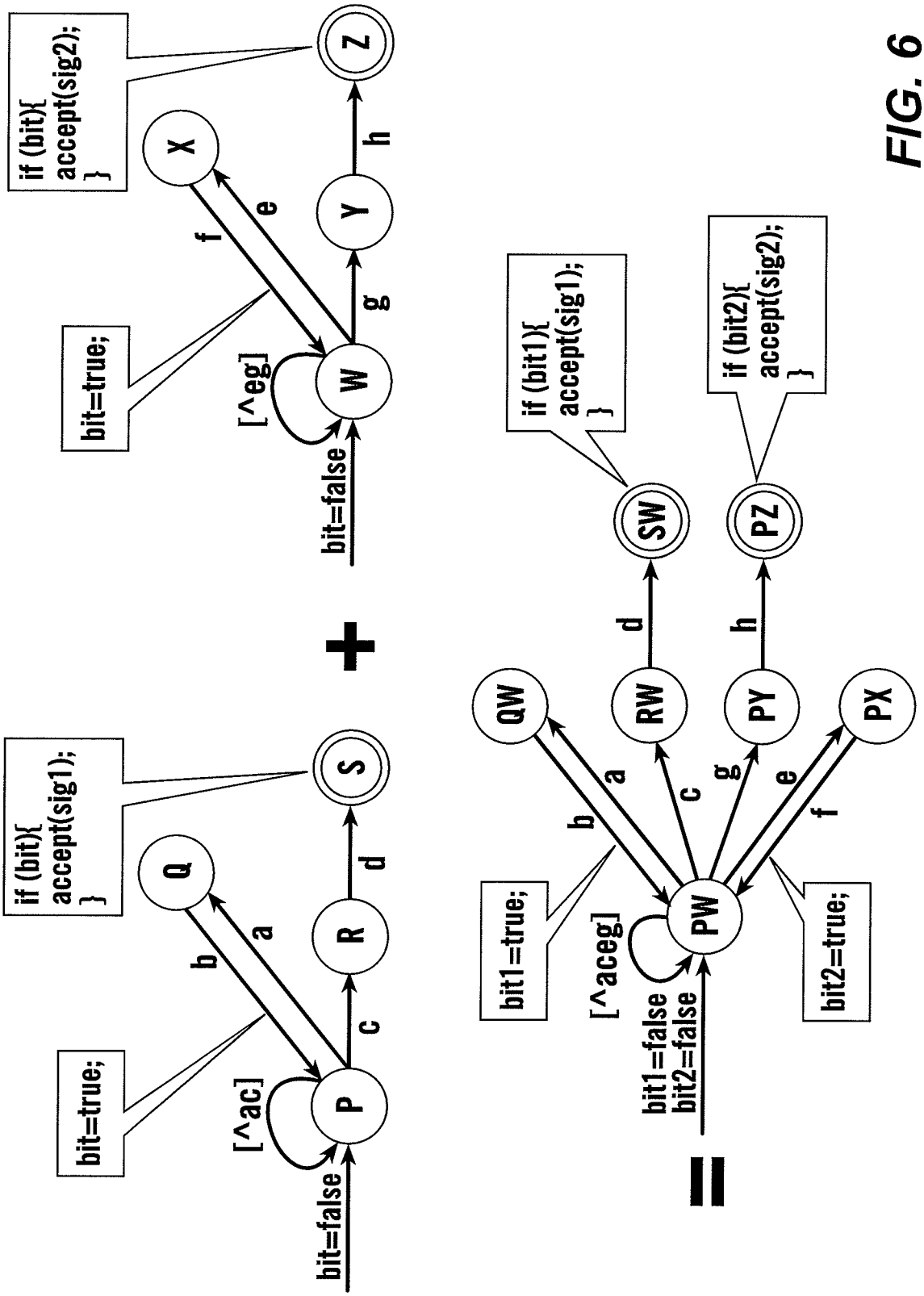
if (bit){
accept(sig2);
}
bit=true;
X
Z
f
e
Y
h
g
[^eg]
W
bit=false
if (bit){
accept(sig1);
}
bit=true;
Q
S
b
a
[^ac]
P
R
c
d
bit=false
+
if (bit1){
accept(sig1);
}
if (bit2){
accept(sig2);
}
SW
PZ
QW
RW
PY
PX
d
h
b
a
c
g
e
f
bit1=true;
[^aceg]
PW
bit2=true;
bit1=false
bit2=false
=

FIG. 6

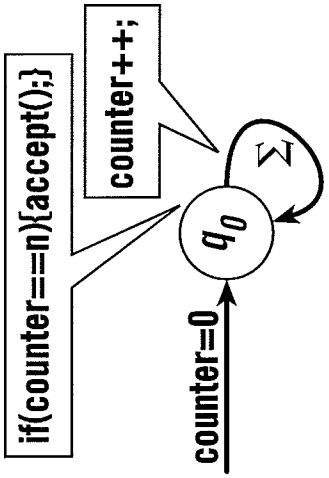

FIG. 8

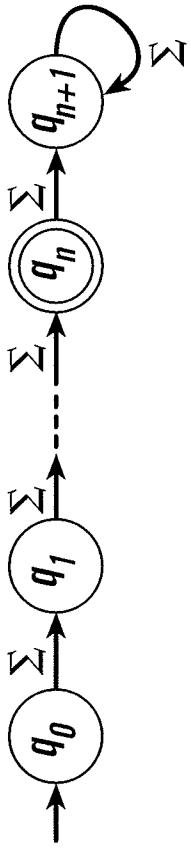

FIG. 7

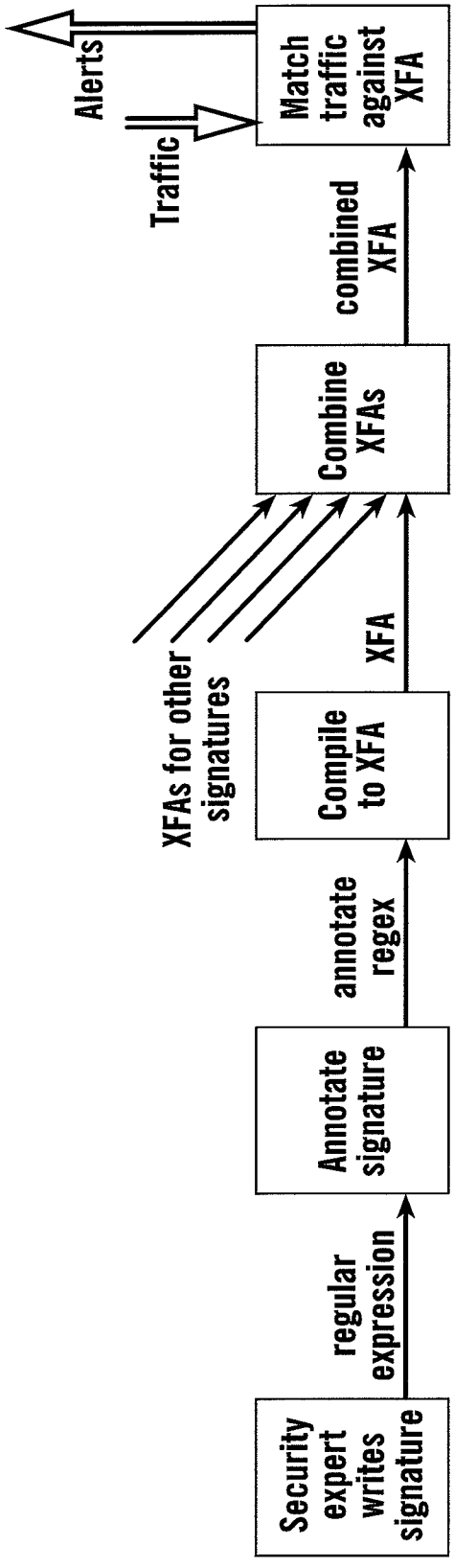

FIG. 9

```
XFA_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 (β∈Σ) do
 9     s' = first.getNextSt( s ,β )
10     t' = second.getNextSt( t ,β )
11     if < s',t'> ∉ 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' >, β )

17 return c
```

FIG. 10

```
XFA_APPLY(XFA M, first, unsigned char* 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 )
  |  // Check the offset list
6 |  while ( offsetList→head.offset==i ) do
7 |  |  execInstrs ( offsetList→head→instr )
8 |  |  offsetList→head = offsetList→head.next
  |  |
  |  |
```

FIG. 11

```
TEST (S ∈ Σ*):

1 q←q0;

2 d←d0;

3 foreach s ∈ S do
4 |  q←σ(q, s);
5 |  d←U(q, d);

6 if q ∈ F and c(q, d) then
7 |  return Accept

8 else
9 |  return Reject
```

FIG. 12

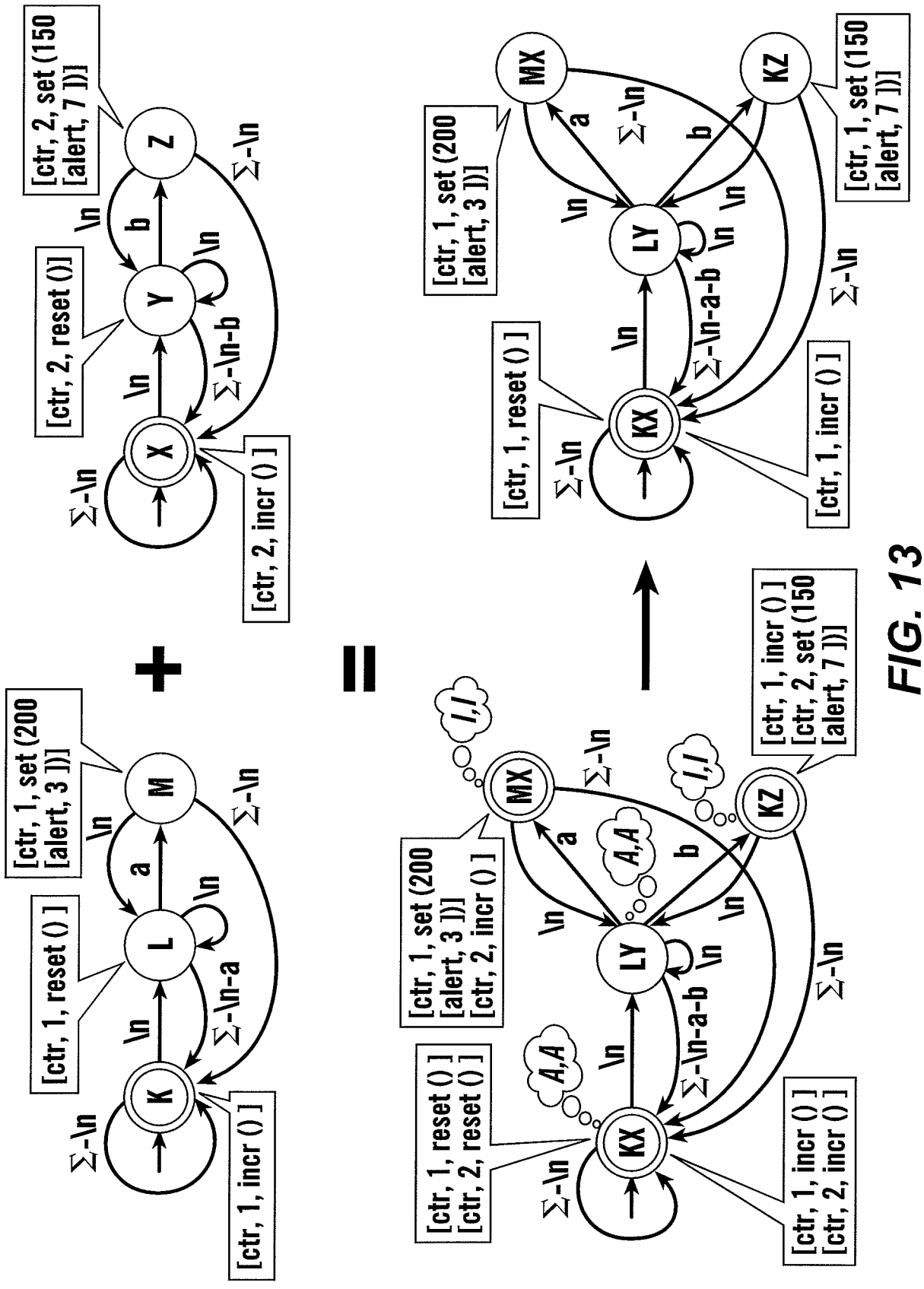
[ctr, 2, set (150
[alert, 7])]
[ctr, 2, reset ()]
[ctr, 2, incr ()]
Σ-\n
\n
b
Σ-\n-b
[ctr, 1, set (200
[alert, 3])]
[ctr, 1, reset ()]
[ctr, 1, incr ()]
a
Σ-\n-a
[ctr, 1, set (150
[alert, 7])]
[ctr, 1, set (200
[alert, 3])]
[ctr, 1, reset ()]
[ctr, 1, incr ()]
[ctr, 1, set (200
[alert, 3])]
[ctr, 2, incr ()]
[ctr, 1, reset ()]
[ctr, 2, reset ()]
[ctr, 1, incr ()]
[ctr, 2, set (150
[alert, 7])]
[ctr, 1, incr ()]
[ctr, 2, incr ()]
A,A
I,I

FIG. 13

```
FIND_EQUIVALENT (XFA M):

1  for c1←1 to numCounters do
2    for c2←c1 + 1 to numCounters do
3      compatible = true
4      foreach state s ∈ M.states do
5        if areCompat( s, c1, c2) == FALSE then
6          compatible = false
7      if compatible then
8        mark (c1, c2) as reducible
9  foreach reducible counter pair (ci , cj ) do
10   foreach state s ∈ M.states do
11     op = getReduced (s, ci , cj )
12     combine counters ci and cj , keeping operation op
```

FIG. 14

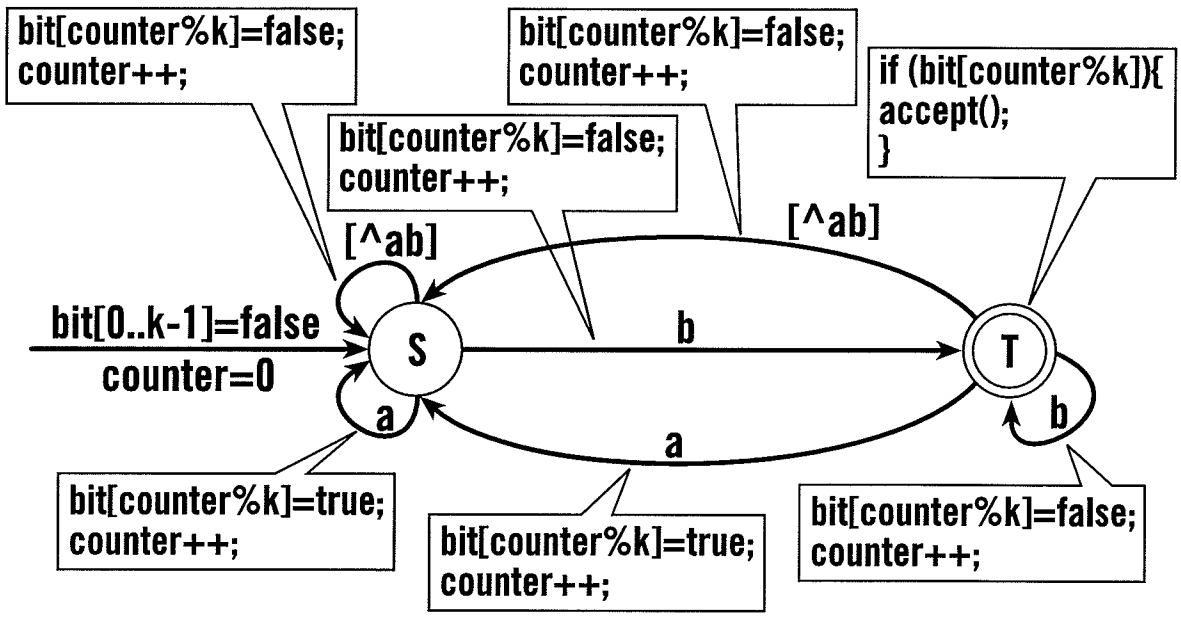

FIG. 15

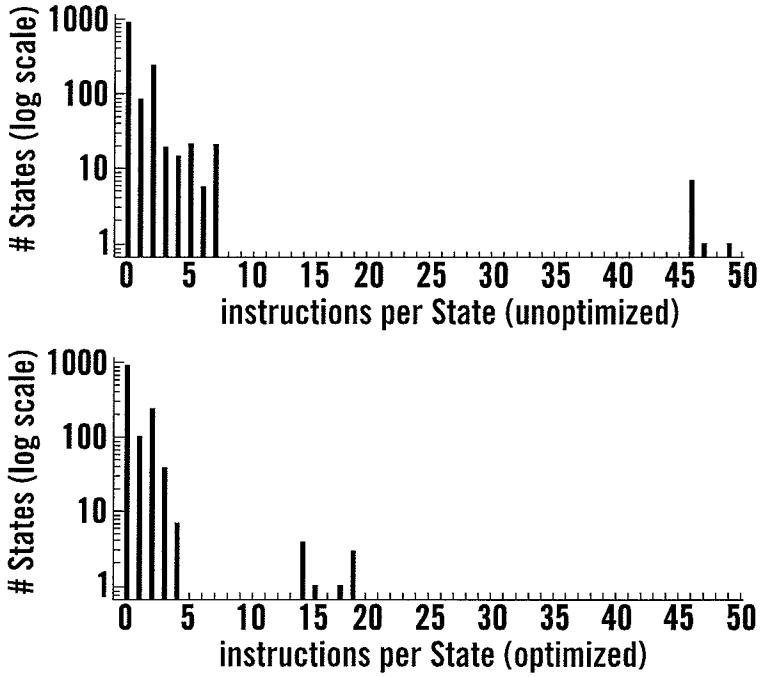
States (log scale)
1000
100
10
1
0 5 10 15 20 25 30 35 40 45 50
instructions per State (unoptimized)
States (log scale)
1000
100
10
1
0 5 10 15 20 25 30 35 40 45 50
instructions per State (optimized)

FIG. 16

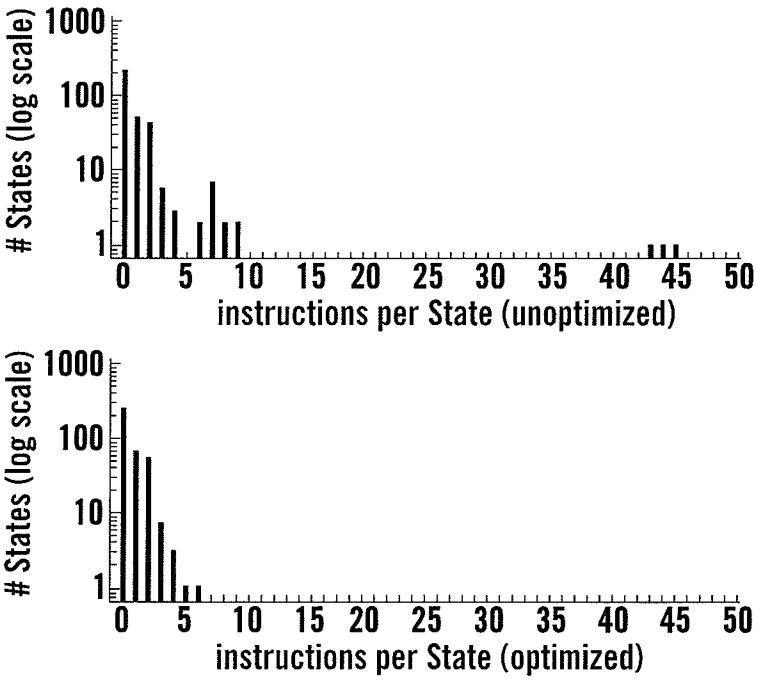
States (log scale)
1000
100
10
1
0 5 10 15 20 25 30 35 40 45 50
instructions per State (unoptimized)
States (log scale)
1000
100
10
1
0 5 10 15 20 25 30 35 40 45 50
instructions per State (optimized)

FIG. 17

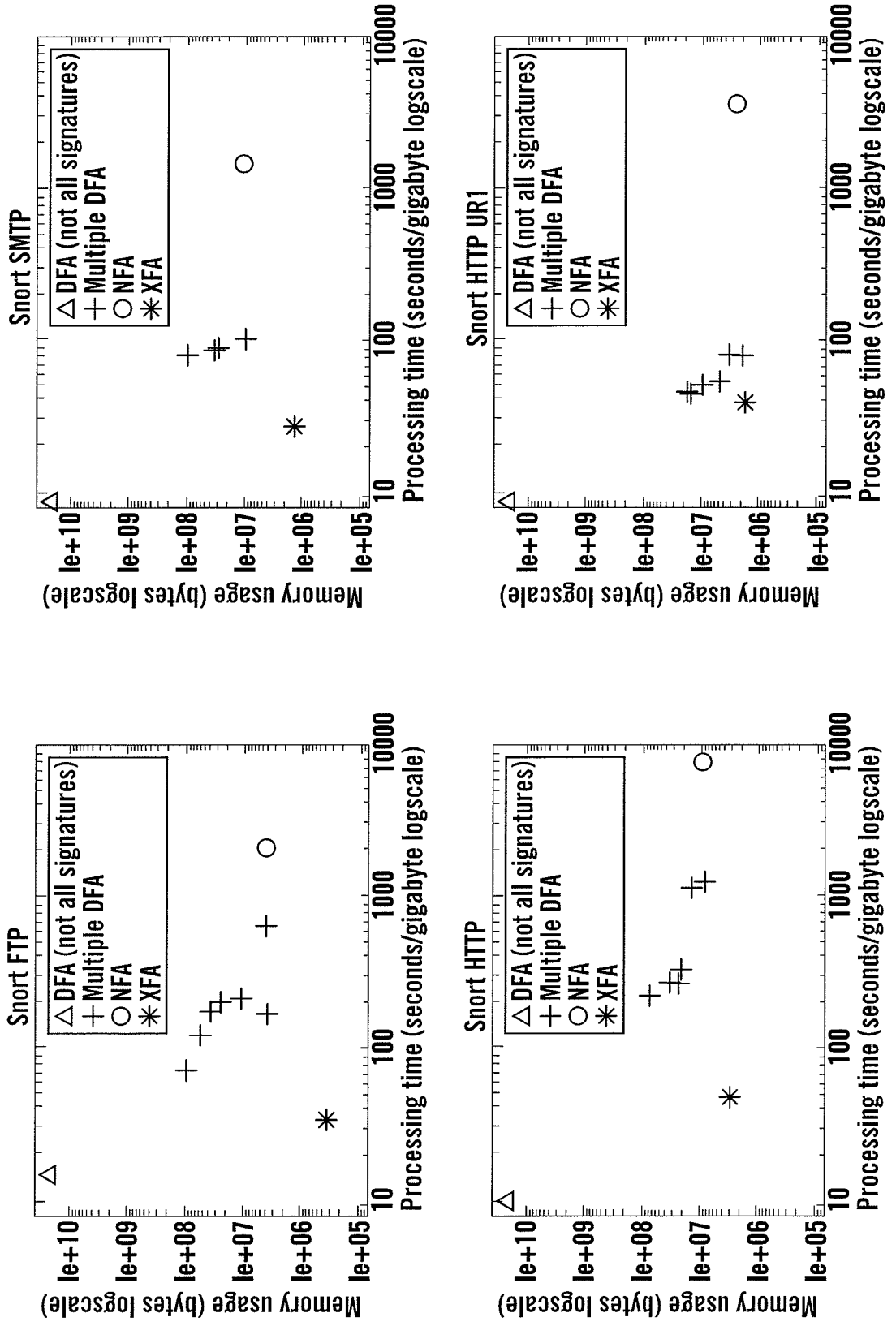
Snort SMTP
Snort HTTP UR1
Snort FTP
Snort HTTP
△ DFA (not all signatures)
+ Multiple DFA
○ NFA
✳ XFA
Memory usage (bytes logscale)
Processing time (seconds/gigabyte logscale)
1e+10
1e+09
1e+08
1e+07
1e+06
1e+05
10
100
1000
10000

FIG. 18

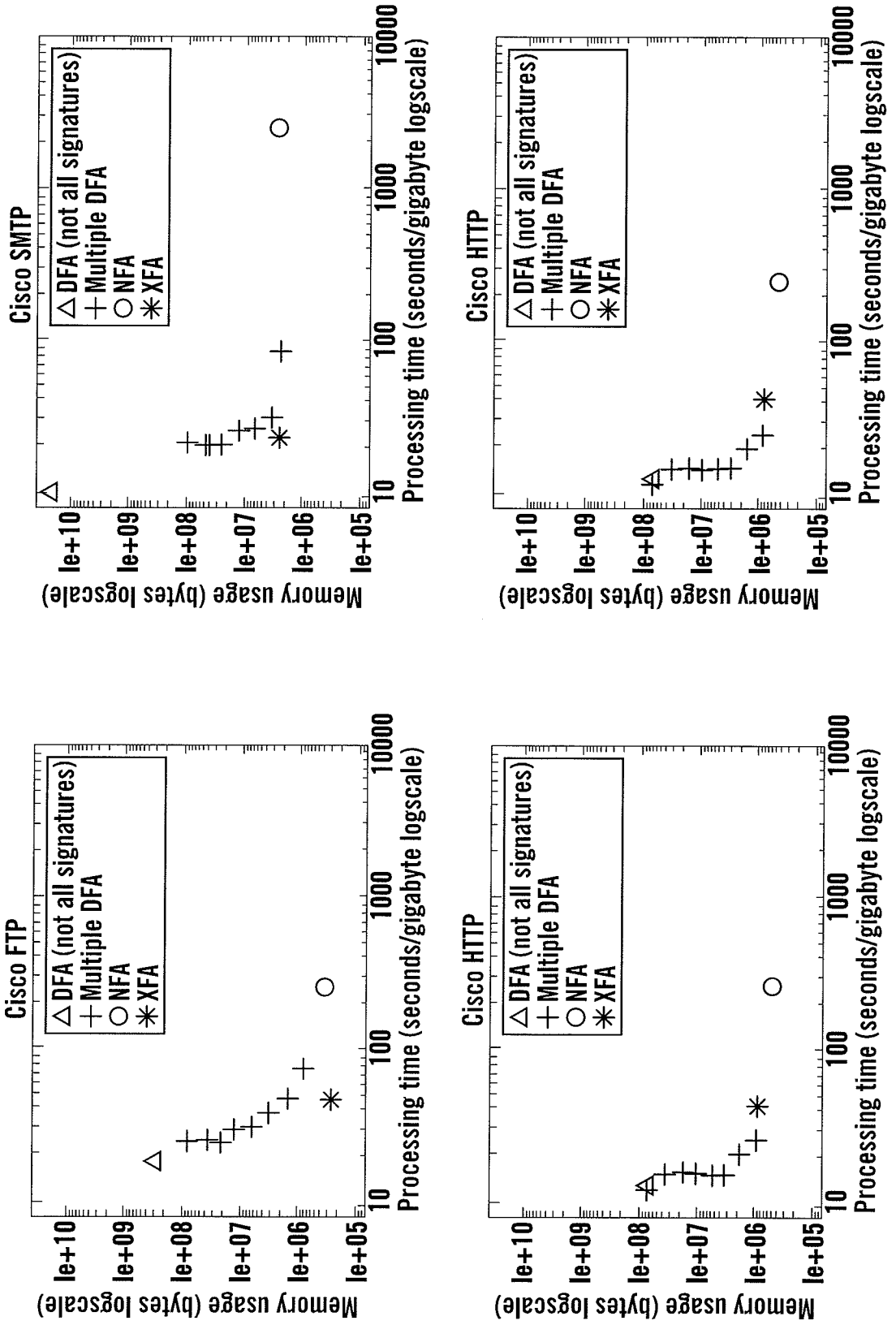
Cisco SMTP
Cisco HTTP
Cisco FTP
Cisco HTTP
△ DFA (not all signatures)
+ Multiple DFA
○ NFA
✳ XFA
Memory usage (bytes logscale)
Processing time (seconds/gigabyte logscale)
1e+10
1e+09
1e+08
1e+07
1e+06
1e+05
10
100
1000
10000

FIG. 19

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US 09/31842

A. CLASSIFICATION OF SUBJECT MATTER
IPC(8) - G10L 15/06 (2009.01)
USPC - 704/243
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
USPC: 704/243

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
USPC: 704/10, 221, 232, 243-244, 255-257, 276, E17.007; 706/17-18, 20-21, 48; 382/229 (keyword limited - see search terms)

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
PubWEST (USPT, PGPB, EPAB, JPAB); Google Patents; Google
Search Terms Used: extended finite automata; instruction; memory; acceptance; counter; bitmap; node

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2003/0204273 A1 (DINKER et al.) 30 October 2003 (30.10.2003) entire document, especially Astract; Fig.3; para [0013], [0015]-[0016], [0026]-[0031], [0033]-[0036], [0052], [0078]	1-4, 11-13, 20
A	US 7,305,383 B1 (KUBESH et al.) 04 December 2007 (04.12.2007) entire document.	1-4, 11-13, 20

☐ Further documents are listed in the continuation of Box C. ☐

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
19 February 2009 (19.02.2009)	03 MAR 2009
Name and mailing address of the ISA/US Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, Virginia 22313-1450 Facsimile No. 571-273-3201	Authorized officer: Lee W. Young PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

Form PCT/ISA/210 (second sheet) (April 2007)

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US 09/31842

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☒ Claims Nos.: 5-10 and 14-19
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

☐ No protest accompanied the payment of additional search fees.