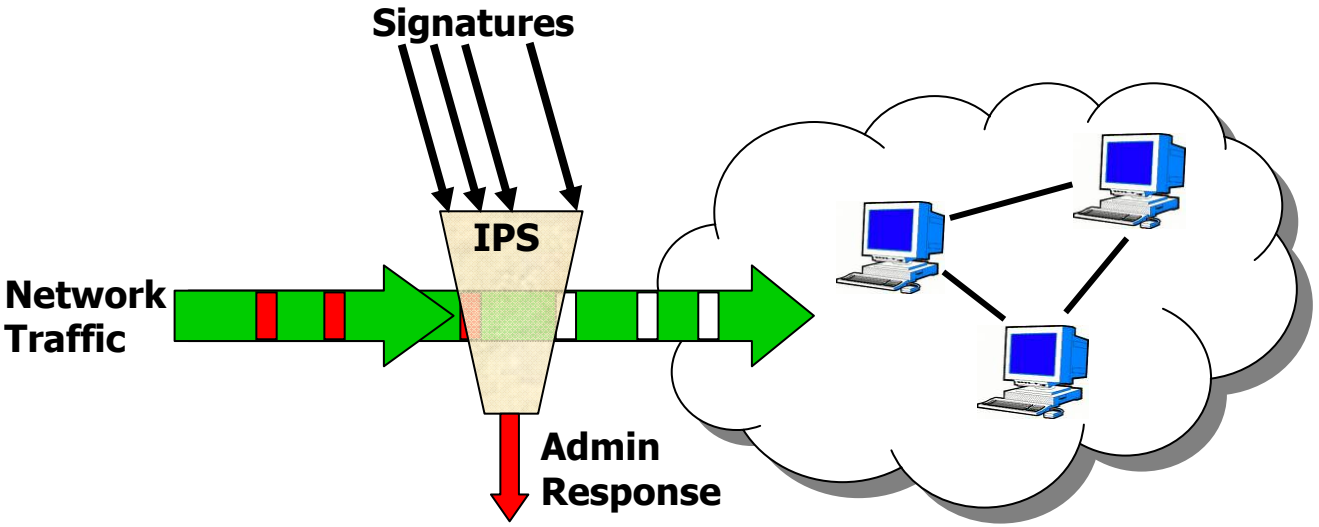
XFA: Faster Signature Matching with Extended Automata

Randy Smith
Cristian Estan
Somesh Jha
University of Wisconsin—Madison

IPS Operation

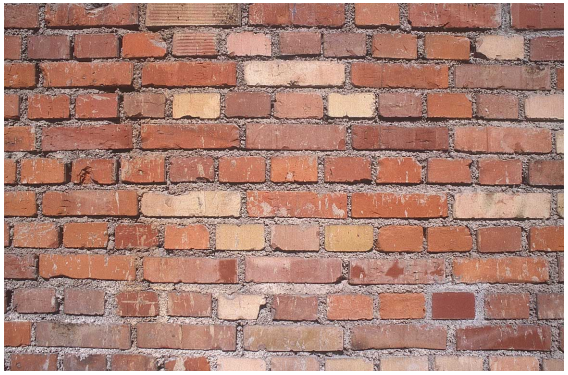
- Intrusion Prevention Systems (IPS) a critical component of modern network infrastructure



Between a Rock and a Hard Place

- Increase in network speeds
  - 1-10 Gbps common, 40 Gbps coming
- Increase in signature counts
  - April 2005 - 3166 Sourcefire rules
  - April 2007 - 8868 Sourcefire rules
- Increase in signature complexity
  - Strings easily evaded, imprecise
  - Regular expressions now standard

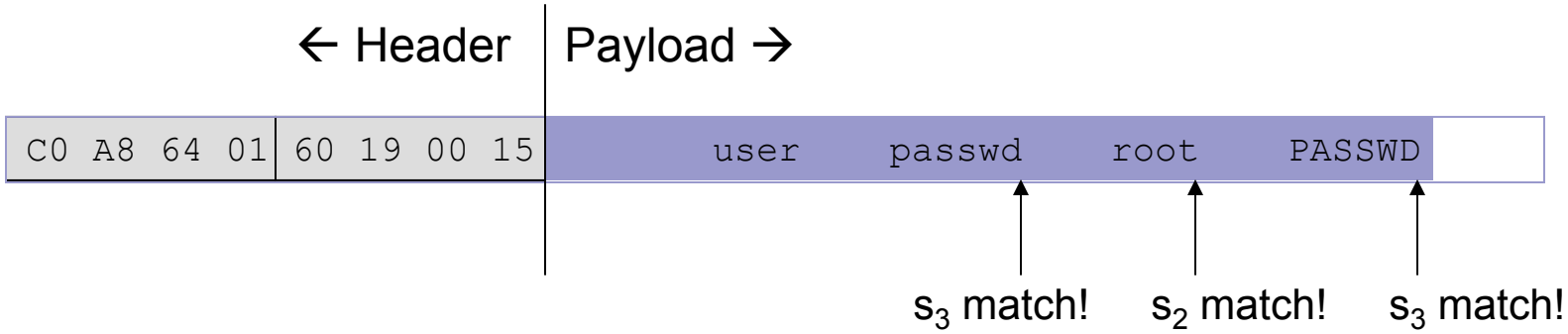
- Presence of active adversary
- Must perform at wirespeed
- Computational resources constant



Signature Matching

- Problem: find all signature occurrences that match the payload up to the currently scanned byte
- Example:

Signatures = { sig_1: /(.*)shadow/ ,
sig_2: /(.*)user(.*)root/ ,
sig_3: /(.*)[Pp][Aa][Ss][Ss][Ww][Dd]/ }



- Ideal: match signatures simultaneously in a single pass

Signature Trends

- Early IPS systems detected exploits with strings
  - Fast matching, bounded memory, but too weak

- Currently, regular expressions used to *express* signatures
  - Capture vulnerabilities rather than specific exploits
    - Buffer overflow: `/^RETR\s[^\n]{100}/`
    - Format string: `/^SITE\s+EXEC\s[^\n]*%[^\n]*%/`

- Finite automata used to *match* signatures
  - Simple, well-understood model of computation
  - Combine using standard cross-product operation

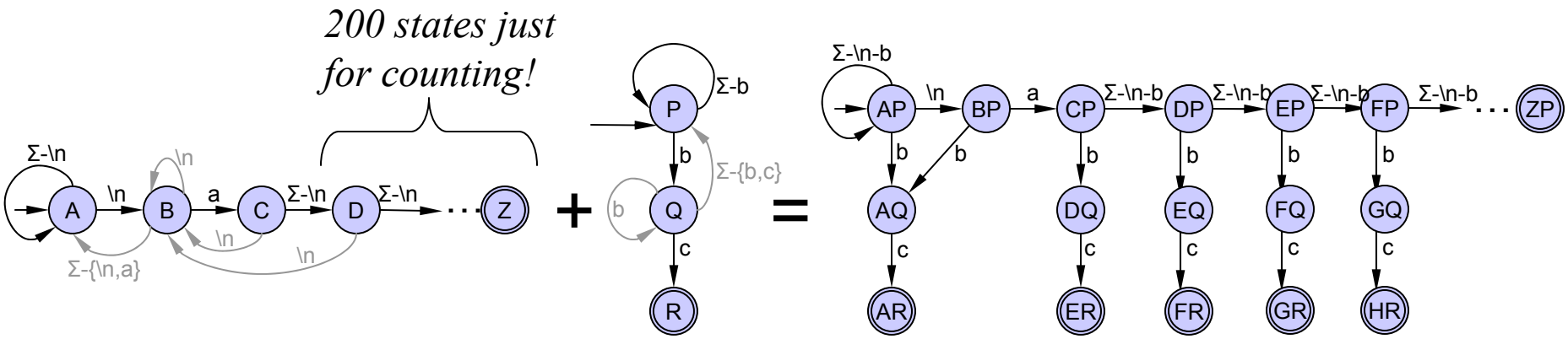
State-space explosion

- Matching progress encoded only as states
- Combined automaton tracks progress of individual automata simultaneously
  - Combined states = tuples of individual automata states
  - Distinct state for each reachable combination

$$S_c = (S_1, S_2, S_3, \ldots, S_n)$$

- For IPS signatures, REs overlap or subsume each other
  - Matching progress *interleaved*
  - Many distinct combinations of reachable states

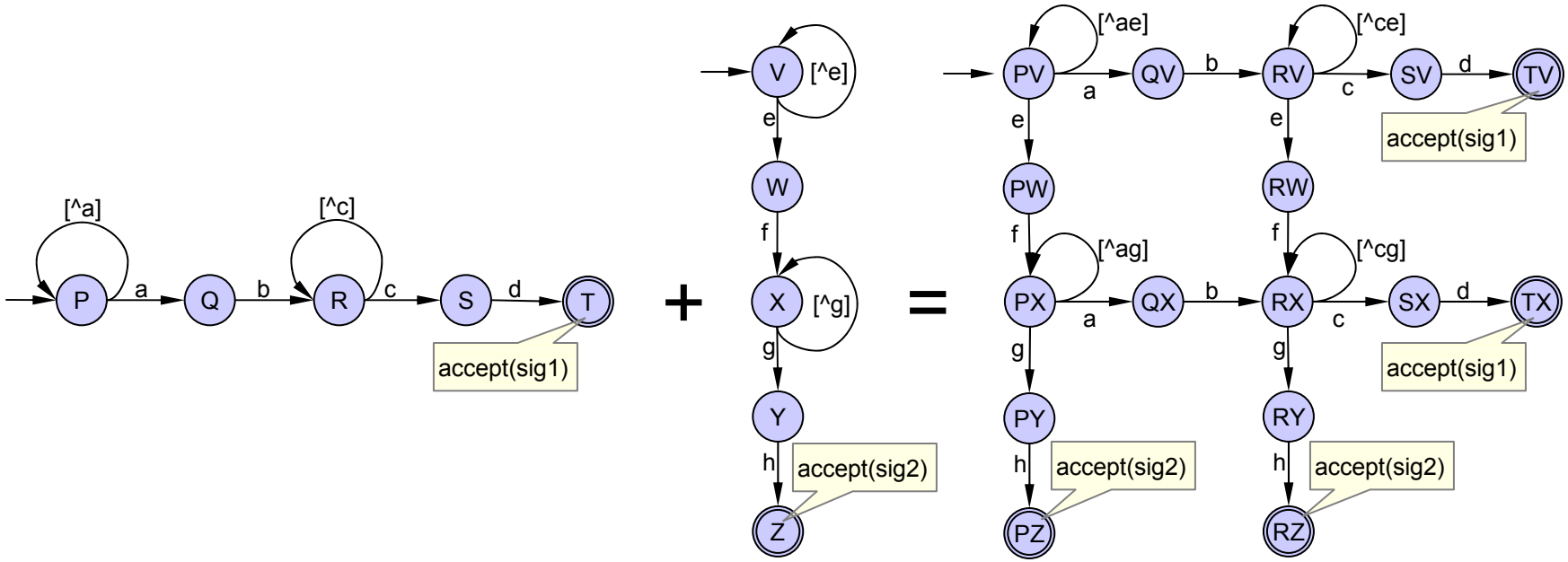
Polynomial Blowup



/.*\na[^\n]{200}/ /.*bc/

Cause: "counting states" subsume other automata

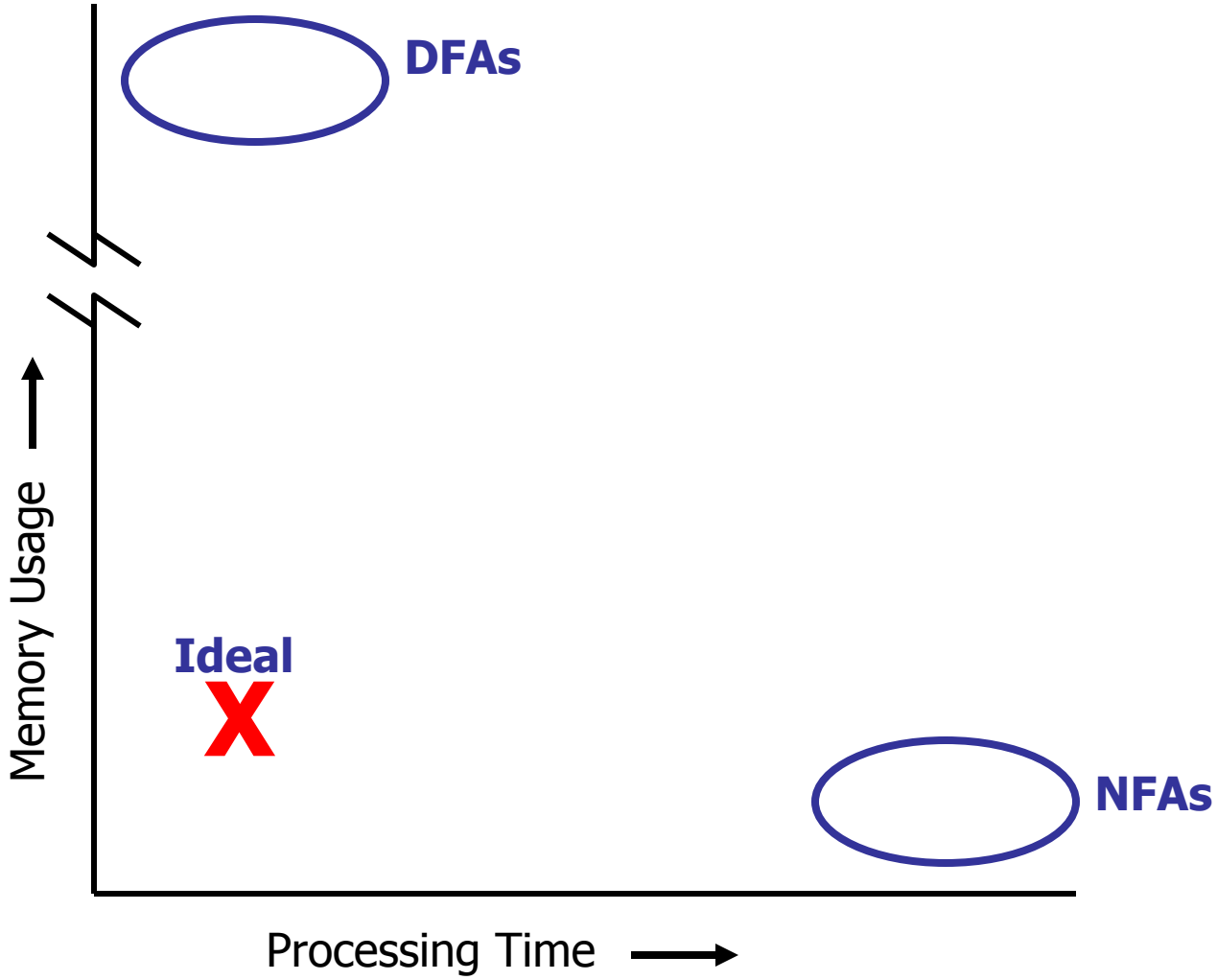
Exponential Blowup



/.*ab.*cd/ /.*ef.*gh/

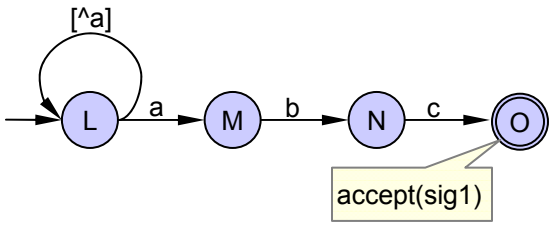
Cause: RE substrings can be interleaved

Time vs. Space



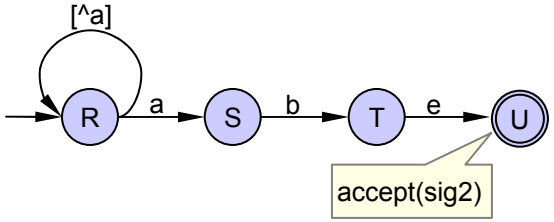
No State Explosion

/.*abc/

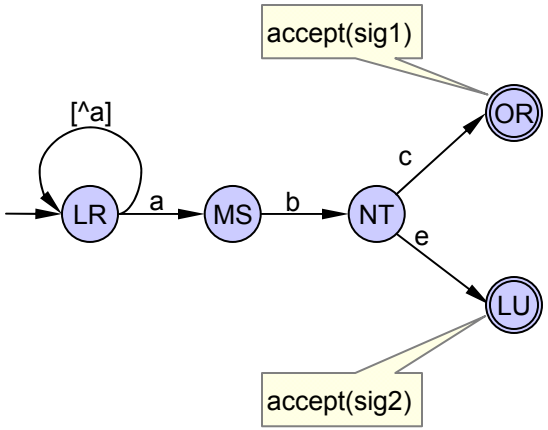


+

/.*abe/



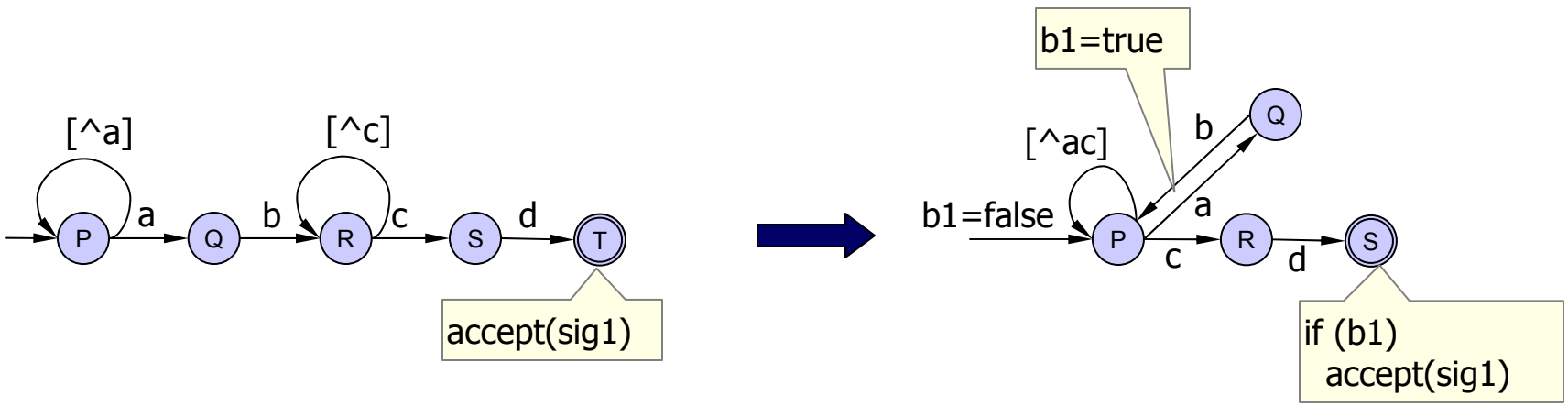
=



XFA: Extended Finite Automata

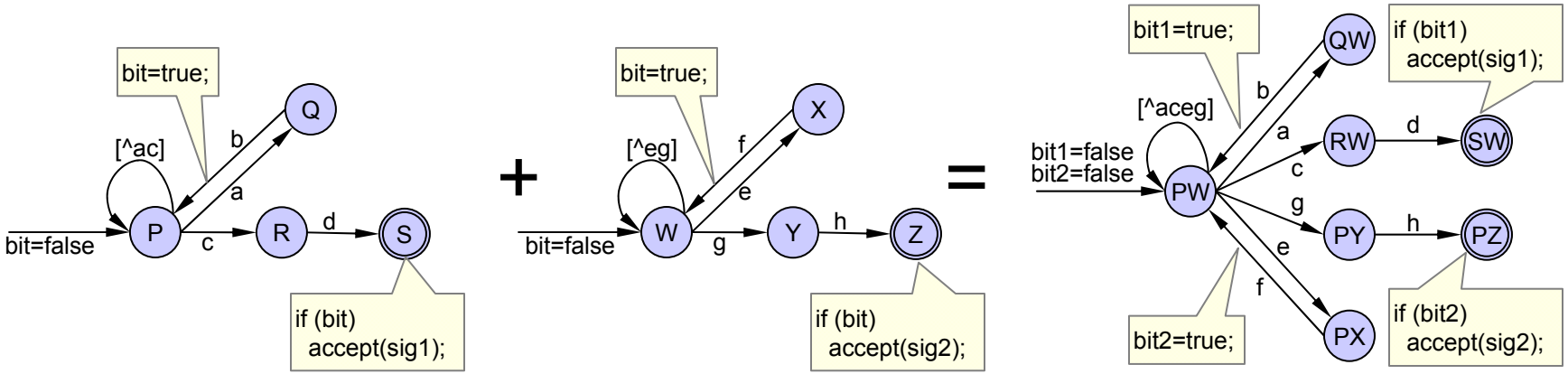
- No distinction between DFA state and computation state
- Idea: extend DFAs with variables that more efficiently track computation state
  - Variables reside in a small auxiliary memory
  - Small programs update data during inspection
- Intuition: including variables change shape of automata, make them look like strings

Using Bits



/.*ab.*cd/ /.*ab.*cd/

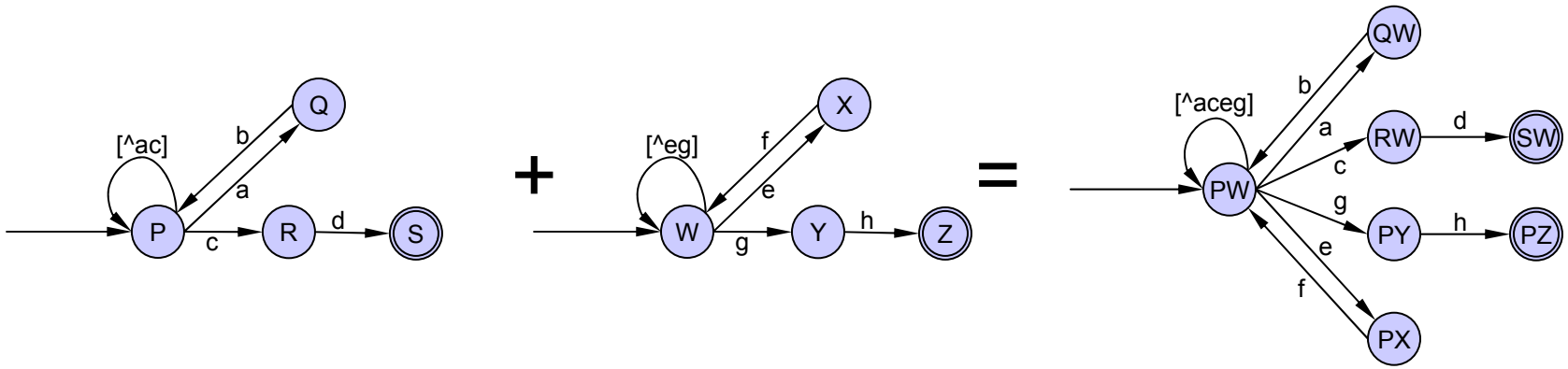
XFAs with Bits



/.*ab.*cd/ /.*ef.*gh/

n signatures – *O(n)* states

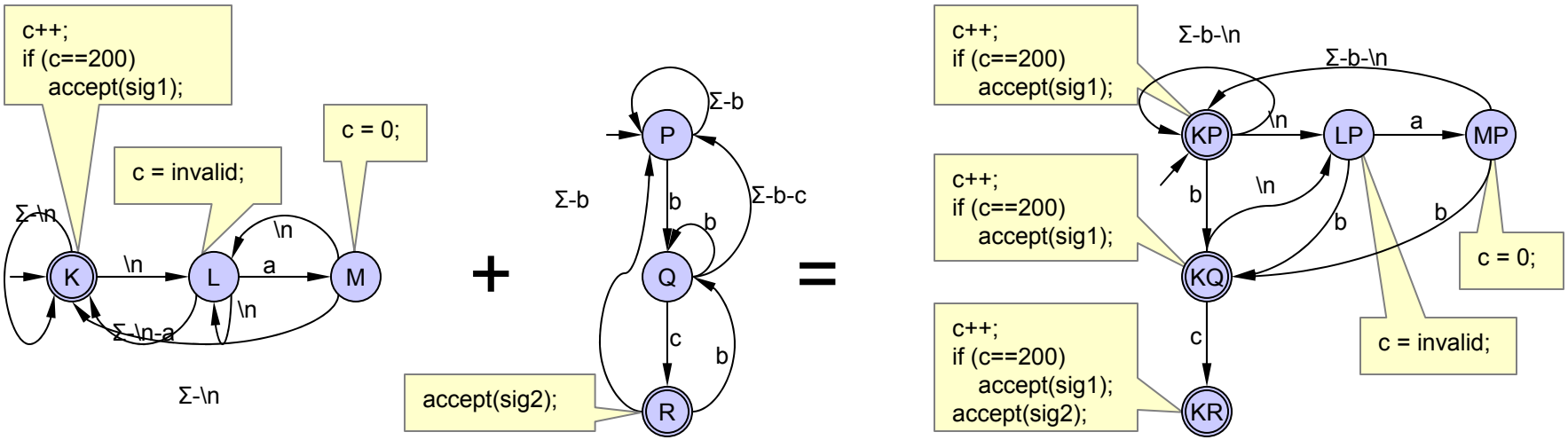
XFAs with Bits



/.*ab.*cd/ /.*ef.*gh/

n signatures – *O(n)* states

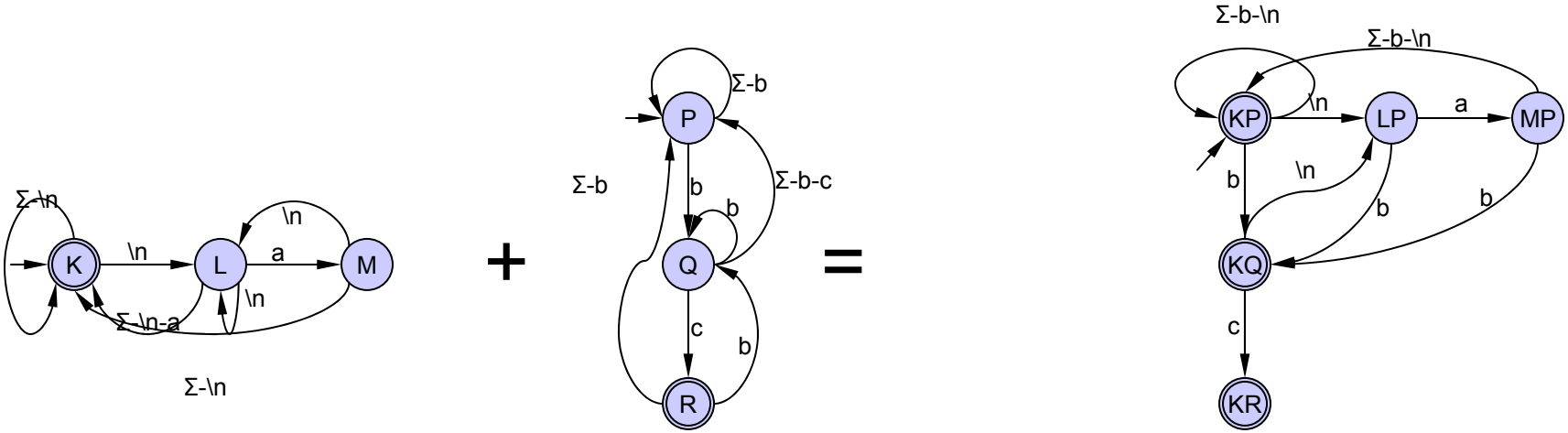
XFAs with Counters



/.*\na[^\n]{200}/ /.*bc/

n signatures – *O(n)* states

XFAs with Counters



/.*\na[^\n]{200}/ /.*bc/

n signatures – *O(n)* states

XFA Key Contribution

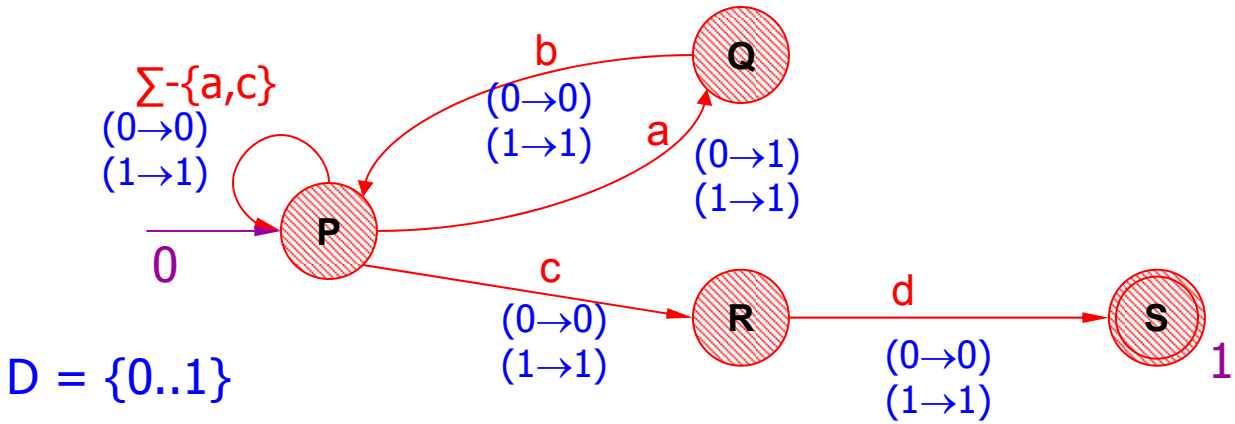
- XFA = DFAs + auxiliary variables
  - Changes shape of automata
  - Tames state space explosion

Outline

- Problem Definition
- XFA Introduction
- XFA Model and Operation
- Experimental Results
- Conclusion

XFAs, formally

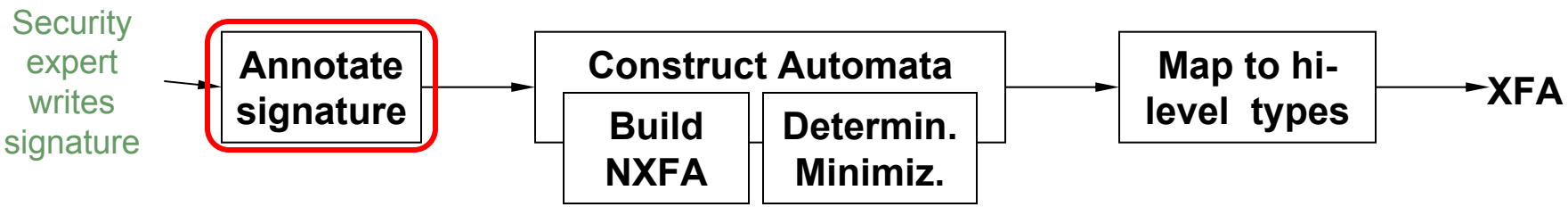
- Formally, XFAs are defined as follows:



  - States and transitions
  - Abstract domain (set of integers)
  - Update functions and acceptance conditions

- See paper for details

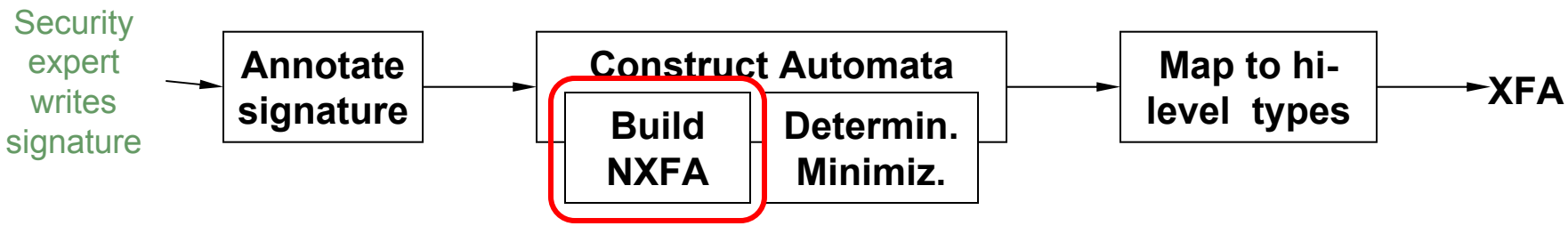
Construction



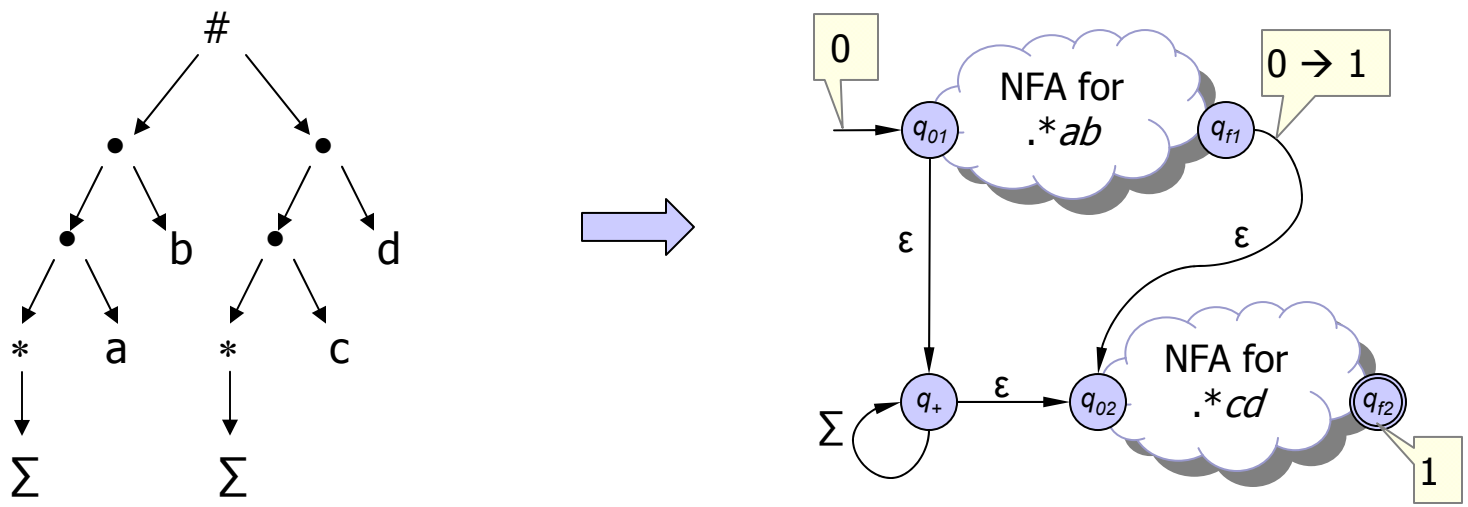
- New operators change parse tree and add domain values
  - Parallel concatenation (#) adds a bit
    - Breaks up RE into string-like components
    - Set a bit when the left operand accepts
    - Test the bit when the right operand accepts

- ex: `/.*ab.*cd/` ➔ `/.*ab#.*cd/`

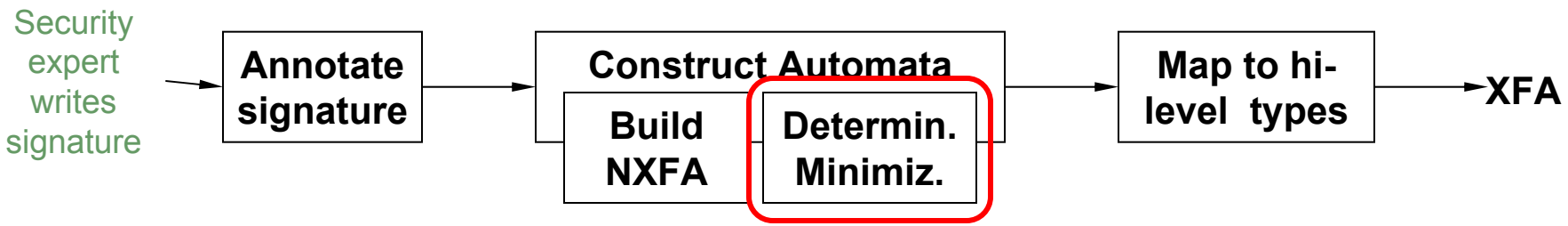
Construction



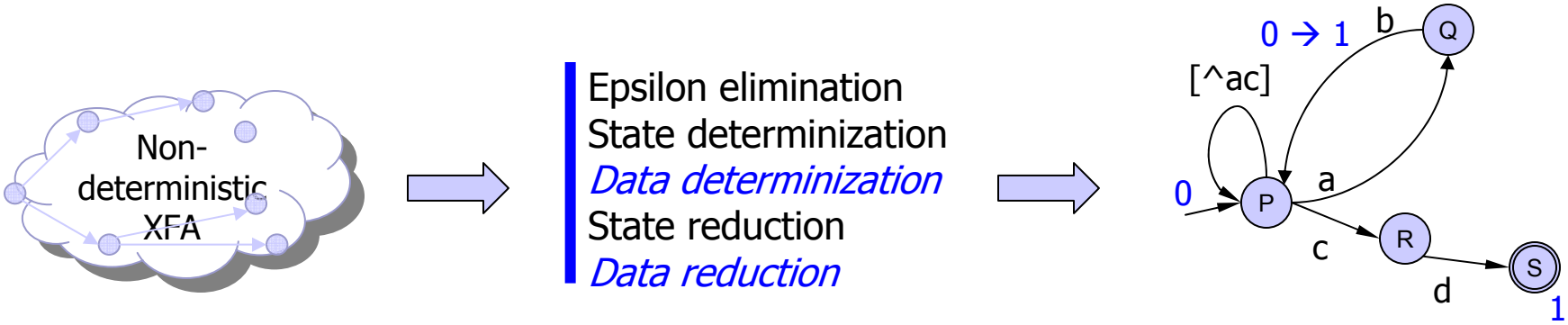
- ex: `/.*ab#.*cd/`



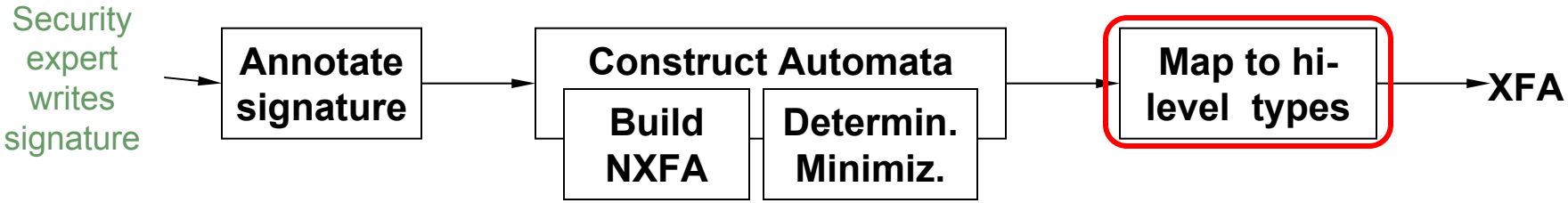
Construction



- ex: `/.*ab#.*cd/`



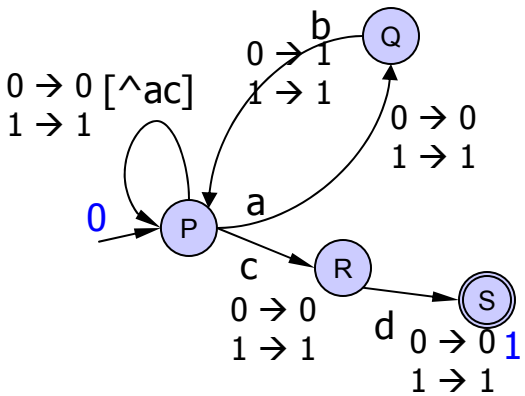
Construction



- Set-based update functions cumbersome
  - Too abstract, too much memory, difficult to combine

- Templates tie domain values and update functions to efficient data types

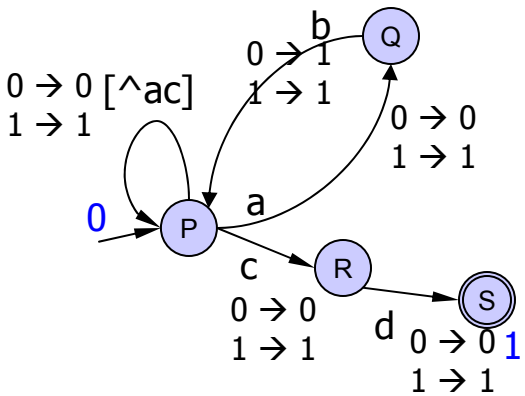
Mapping to Efficient Data Types




```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,       {0→0, 1→1}, "”),
                (bit set,     {0→1, 1→1}, "bitSET %d”),
                (bit clear,   {0→0, 1→0}, "bitCLR %d”),
                (bit flip,    {0→1, 1→0}, "bitFLP %d”) },

     Amap = {   (nonaccepting,      {}, "”),
                (accept_ifset,     {1}, "if (bit %d) alert”),
                (accept_ifclear,   {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

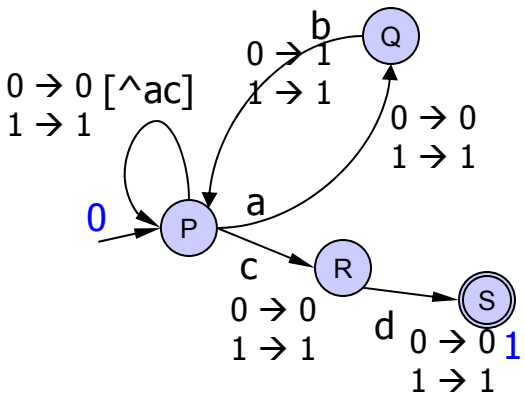
Mapping to Efficient Data Types

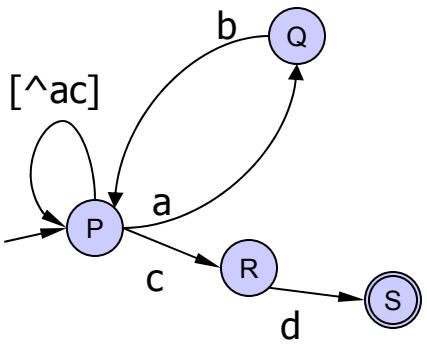



```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,       {0→0, 1→1}, "”),
                (bit set,     {0→1, 1→1}, "bitSET %d”),
                (bit clear,   {0→0, 1→0}, "bitCLR %d”),
                (bit flip,    {0→1, 1→0}, "bitFLP %d”) },

    Amap = {    (nonaccepting,      {}, "”),
                (accept_ifset,     {1}, "if (bit %d) alert”),
                (accept_ifclear,   {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

Mapping to Efficient Data Types

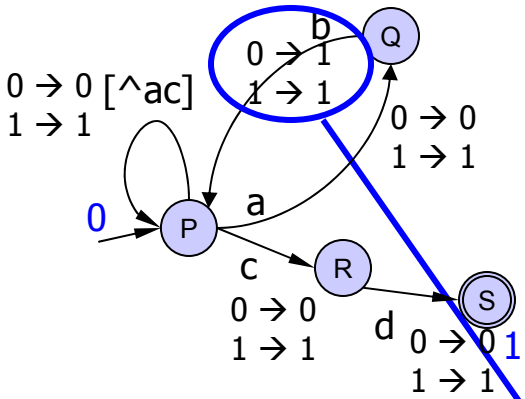


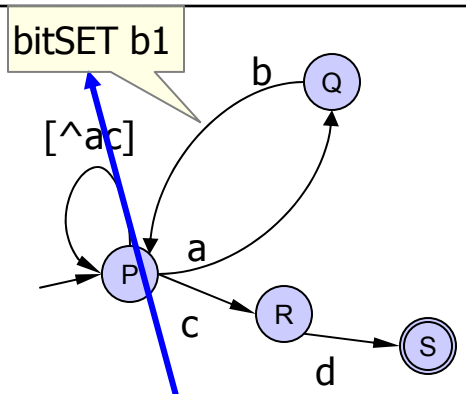



```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,       {0→0, 1→1}, "”),
                (bit set,     {0→1, 1→1}, "bitSET %d”),
                (bit clear,   {0→0, 1→0}, "bitCLR %d”),
                (bit flip,    {0→1, 1→0}, "bitFLP %d”) },

     Amap = {   (nonaccepting,      {}, "”),
                (accept_ifset,     {1}, "if (bit %d) alert”),
                (accept_ifclear,   {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

Mapping to Efficient Data Types

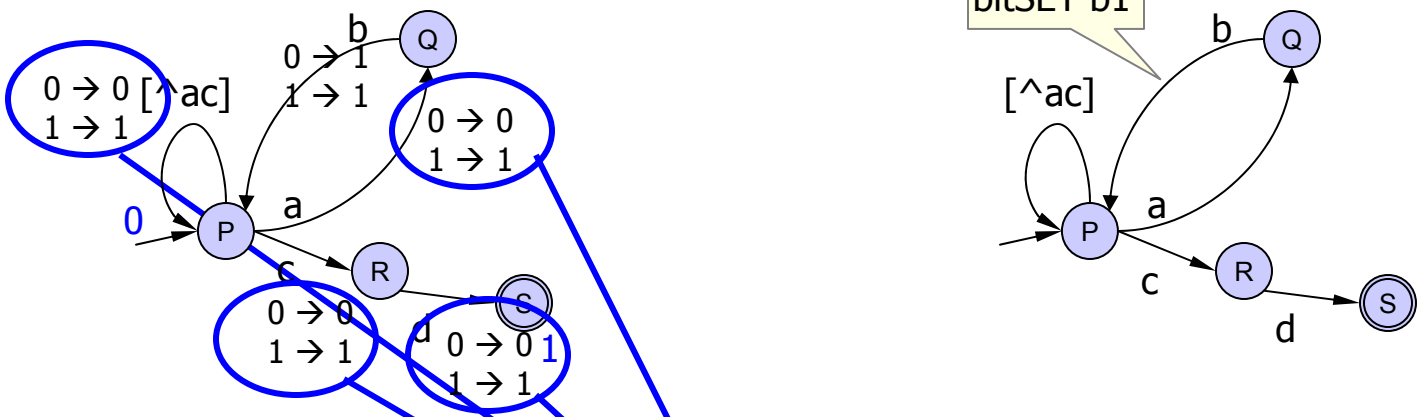





```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,       {0→0, 1→1}, "”),
                (bit set,     {0→1, 1→1}, "bitSET %d”),
                (bit clear,   {0→0, 1→0}, "bitCLR %d”),
                (bit flip,    {0→1, 1→0}, "bitFLP %d”) },

     Amap = {   (nonaccepting,      {}, "”),
                (accept_ifset,     {1}, "if (bit %d) alert”),
                (accept_ifclear,   {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

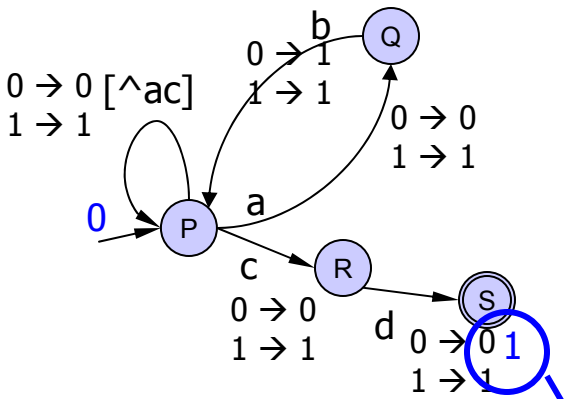
Mapping to Efficient Data Types

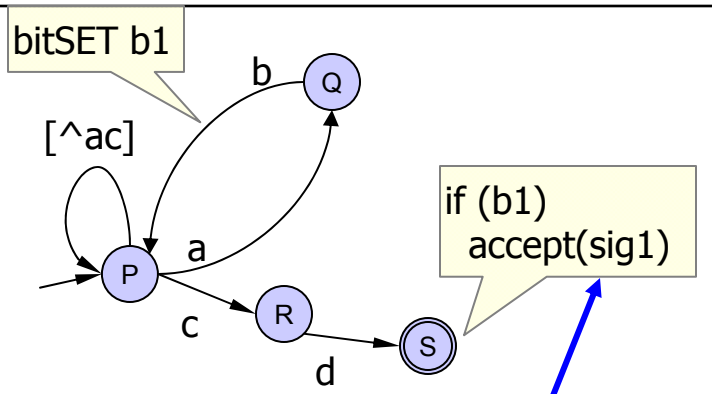



```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,        {0→0, 1→1}, "”),
                (bit set,      {0→1, 1→1}, "bitSET %d”),
                (bit clear,    {0→0, 1→0}, "bitCLR %d”),
                (bit flip,     {0→1, 1→0}, "bitFLP %d”) },

     Amap = {   (nonaccepting,       {}, "”),
                (accept_ifset,      {1}, "if (bit %d) alert”),
                (accept_ifclear,    {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

Mapping to Efficient Data Types

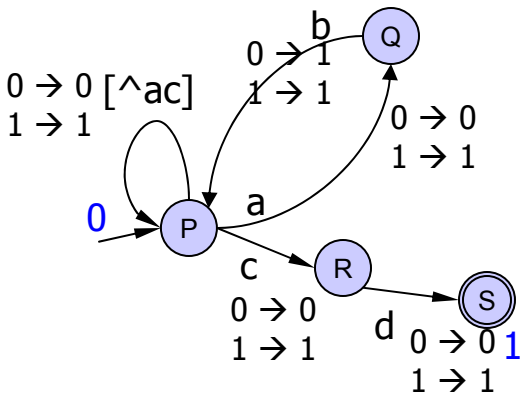


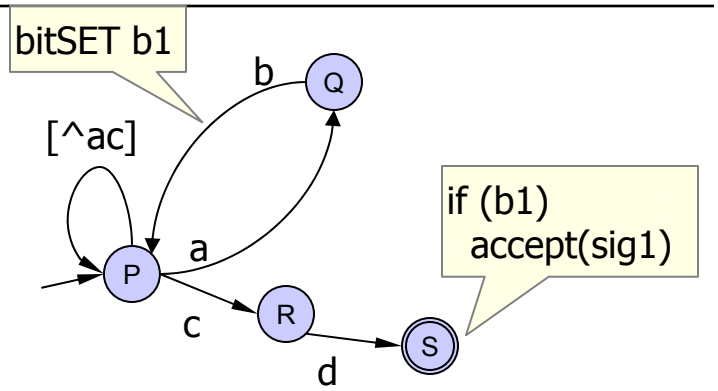



```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,        {0→0, 1→1}, "”),
                (bit set,      {0→1, 1→1}, "bitSET %d”),
                (bit clear,    {0→0, 1→0}, "bitCLR %d”),
                (bit flip,     {0→1, 1→0}, "bitFLP %d”) },

     Amap = {   (nonaccepting,      {}, "”),
                (accept_ifset,      {1}, "if (bit %d) alert”),
                (accept_ifclear,    {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

Mapping to Efficient Data Types






```
template  bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {   (no-op,       {0→0, 1→1}, "”),
                (bit set,     {0→1, 1→1}, "bitSET %d”),
                (bit clear,   {0→0, 1→0}, "bitCLR %d”),
                (bit flip,    {0→1, 1→0}, "bitFLP %d”) },

     Amap = {   (nonaccepting,      {}, "”),
                (accept_ifset,     {1}, "if (bit %d) alert”),
                (accept_ifclear,   {0}, "if !(bit %d) alert”),
                (unconditional, {0,1}, "alert”)
};
```

Combination and Matching

- Straightforward extensions to DFA counterparts

- Combination
  - Perform standard cross product combination
  - “Append” instructions to transitions and states
    - Data type instances are independent

- Matching
  - Execute attached instructions when transitions followed

Outline

- Problem Definition
- XFA Introduction
- XFA Model and Operation
- Experimental Results
- Conclusion

Experiment Highlights

- Combined XFA only slightly larger than sum of sizes of individual automata
- Combined XFA smaller *and* faster than other techniques

Test Environment – Signatures

- 1450 Regular expressions extracted from Snort HTTP
- Retain semantics of Snort rules:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-CGI dcboard.cgi invalid..."; flow:to_server,est;
uricontent:"/dcboard.cgi"; content:"command=register";
content="&admin"; ... sid:817; rev:10;)
```

→ /(.*command=register#(.*)&admin) | (.*&admin#(.*)command=register)/

  - Snort2Bro and custom scripts to compose expressions
  - Most annotations automatic

Test Environment – XFA

- Characteristics of combined XFA:
  - 41,994 total states → 42 MB (additive sum: 36,631 states)
  - 195 bits (~25 bytes) of aux memory
  - Instruction memory: 3.5 MB

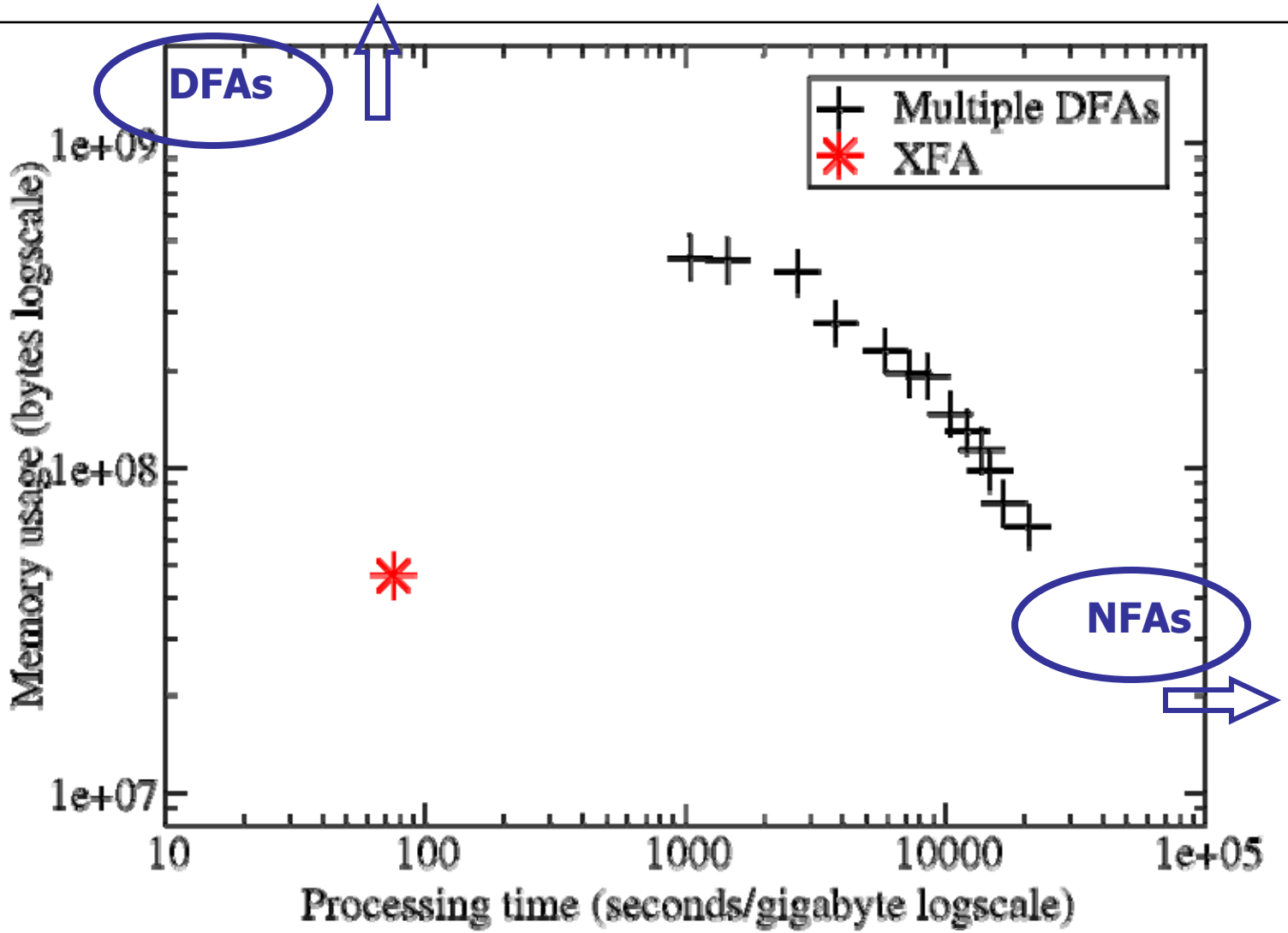
Table: Distribution of instructions on edges and states

#instrs	0	1	2	3	4	5	6	7	8
% edges	1.0	**94.8**	2.7	0.47	0.80	0.00	0.00	0.00	0.13
% states	**78.9**	20.0	0.9	0.03	0.00	0.00	0.00	0.00	0.05

DFA Set Splitting

- DFA set splitting ("Multiple DFAs") (Yu *et al.* ANCS'06):
  - Given memory bound M, partition REs so that:
    - The number of RE subsets is as small as possible
    - The set of combined automata is less than M
  - Use heuristics to group REs that behave well
  - Traces a curve in time vs. space between DFAs and NFAs
    - Increasing the memory ceiling decreases the number of combined automata

Memory vs. Execution Time



Limitations and Future Work

- Expression coverage
  - Complex data types require complex templates
- Hi-level mapping time
  - Backtracking match algorithm can be time-intensive
- Other issues
  - Further work coming in Sigcomm 2008

Conclusion

- DFAs for regular expressions often blow up when combined
- XFAs = DFAs + auxiliary variables
  - Changes shape of automata
  - Tames state space explosion
- Result: compared to other feasible approaches, reduce both time and space

XFA: Faster Signature Matching with Extended Automata

Thank you