Reservoir Labs
632 Broadway, #803
New York, NY 10012
(212) 780-0527

High-Speed Parallel Processing of Protocol-Aware Signatures

Jordi Ros-Giralt, James Ezick, Peter Szilagyi, Richard Lethin

SBIR funded by
Department of
Energy

The Problem

- A protocol aware signature can be understood in terms of a binary function
- Each signature can be expressed as a BDD

The Opportunity

- Offload the processing of large number of signatures from the core processor onto DFA Hardware engines potentially removing large number of cycles from the cores

The Approach

Assume we have N DFA engines, how can we map the complete set of signatures onto it?

- Generate one big OR signature S
- Obtain the CNF clauses of S using **SALT**^{TM}
- Given N DFA engines, find N cuts of the set of CNF clauses that yield BDDs that can fit in the DFA engines. This problem can be expressed as a min-max problem: find N cuts so that the size of the biggest cut is minimed.

Example

Optimal signature for Atphttpd attack:

f = bit1 & bit2 & bit3 & bit4 & ((bit5 & bit6) | (!bit5 & bit7) | (bit8 & bit9))

```
bit1 : strcmp(METHOD,"GET") == 0;
bit2 : URI[0] == '/';
bit3 : URI[1] != '/';
bit4 : strstr(URI_sub[1], "/../") == 0;
bit5 : isnotdir(URI_sub[1]);
bit6 : stat(URI_sub[1],ptr) < 0;
bit7 : stat(URI_sub[1]+"index.html",ptr) < 0;
bit8 : URI_sub[1] == 0;
bit9 : stat("index.html",ptr) < 0;
```

Optimal signature for GhttpdLog attack:

f = (!bit1 & bit10) | (bit1 & !bit11 & bit12) || bit1 & bit11 & bit13)

```
bit1  : strcmp(METHOD,"GET") == 0;
bit10 : strlen(METHOD) > 165;
bit11 : strstr(URI,"/..")) == 0;
bit12 : strlen(URI) > 170;
bit13  : strlen(URI) + strlen(ClientAddr) > 166;
```

BSC

```
;;;
;;; Copyright (C) 2009 Reservoir Labs
;;;
;;; Author: giralt
;;;

#comment signature : Atphttpd (optimal version)
#comment type : Protocol Aware
#comment logic : bit1 && bit2 && bit3 && bit4 && ((bit5 && bit6) ||
(!bit5 && bit7) || (bit8 && bit9))

;;;

$bit1 expr =? ; declare bits (could also be a 5-bit vector)
$bit2 expr =?
$bit3 expr =?
$bit4 expr =?
$bit5 expr =?
$bit6 expr =?
$bit7 expr =?
$bit8 expr =?
$bit9 expr =?

$conjuntion1 and $bit5 $bit6 =
$conjuntion2 and ~$bit5 $bit7 =
$conjuntion3 and $bit8 $bit9 =
$disjunction1 or $conjuntion1 $conjuntion2 $conjuntion3 =
$conjuntion and $bit1 $bit2 $bit3 $bit4 $disjunction1 =
$_ eval $conjuntion + ; assert return value true

#done
```

BSC

```
;;;
;;; Copyright (C) 2009 Reservoir Labs
;;;
;;; Author: James Ezick (modified by Jordi Ros)
;;;

#comment signature : Atphttpd (optimal version)
#comment type : Protocol Aware
#comment logic : (!bit1 && bit2) || (bit1 && !bit3 && bit4) || bit1 && bit3 && bit5)

;;;

$bit1 expr =? ; declare bits (could also be a 5-bit vector)
$bit2 expr =?
$bit3 expr =?
$bit4 expr =?
$bit5 expr =?
$conjunction1 and ~$bit1 $bit2 =
$conjunction2 and $bit1 ~$bit3 $bit4 =
$conjunction3 and $bit1 $bit3 $bit5 =
$disjunction or $conjunction1 $conjunction2 $conjunction3 =
$_ eval $disjunction + ; assert return value true

#done
```

U

OR
Signature

```
;;;
;;; Copyright (C) 2009 Reservoir Labs
;;;
;;; Author: giralt
;;;

#comment signature : OR(Atphttpd, ghttpdlog)
#comment type : Protocol Aware
#comment

;;;

$bit1 expr =? ; declare bits (could also be a 5-bit vector)
$bit2 expr =?
$bit3 expr =?
$bit4 expr =?
$bit5 expr =?
$bit6 expr =?
$bit7 expr =?
$bit8 expr =?
$bit9 expr =?

$conjuntion1 and $bit5 $bit6 =
$conjuntion2 and ~$bit5 $bit7 =
$conjuntion3 and $bit8 $bit9 =
$disjunction1 or $conjuntion1 $conjuntion2 $conjuntion3 =
$sigatphttpd and $bit1 $bit2 $bit3 $bit4 $disjunction1 =

$bit10 expr =?
$bit11 expr =?
$bit12 expr =?
$bit13 expr =?
$conjunction1 and ~$bit1 $bit10 =
$conjunction2 and $bit1 ~$bit11 $bit12 =
$conjunction3 and $bit1 $bit11 $bit13 =
$sigghttpdlog or $conjunction1 $conjunction2 $conjunction3 =

$SIGORdisjunction or $sigatphttpd $sigghttpdlog =
$_ eval $SIGORdisjunction + ; assert return value true

#done
```

SALT

```
c Satisfiability Application Logic Translator (salt) v1.5
c     written by James Ezick (ezick@reservoir.com)
c Copyright (C) 2005-2007 Reservoir Labs
c
c Thu Mar 26 11:06:34 2009 EDT
c
c input file:         http-sig-atphttpd.salt
c echo:           no
c forced captures:    yes
c optimizations: yes
c capture threshold:  1410065407
c rounding mode: truncate
c ignore overflow:    no
c ignore conflicts:   no
c
p cnf 9 10
c signature : Atphttpd
c type : Protocol aware
c logic : bit1 && bit2 && bit3 && bit4 && ((bit5 && bit6) ||
(!bit5 && bit7) || (bit8 && bit9))

1 0
2 0
3 0
4 0
6 7 9 0
6 7 8 0
-5 6 9 0
-5 6 8 0
5 7 9 0
5 7 8 0
```

SALT

```
c Satisfiability Application Logic Translator (salt) v1.5
c     written by James Ezick (ezick@reservoir.com)
c Copyright (C) 2005-2007 Reservoir Labs
c
c Thu Mar 26 10:01:57 2009 EDT
c
c input file:         http-sig-ghttpdlog.salt
c echo:           no
c forced captures:    yes
c optimizations: yes
c capture threshold:  1410065407
c rounding mode: truncate
c ignore overflow:    no
c ignore conflicts:   no
c
p cnf 7 13
c signature : Atphttpd
c type : Protocol aware
c logic : (!bit1 && bit2) || (bit1 && !bit3 && bit4) || bit1
&& bit3 && bit5)

2 4 5 0
2 1 4 0
2 3 4 0
2 -3 5 0
2 1 -3 0
2 1 5 0
2 1 0
2 1 3 0
-1 4 5 0
-1 3 4 0
-1 -3 5 0
6 0
-7 0
```

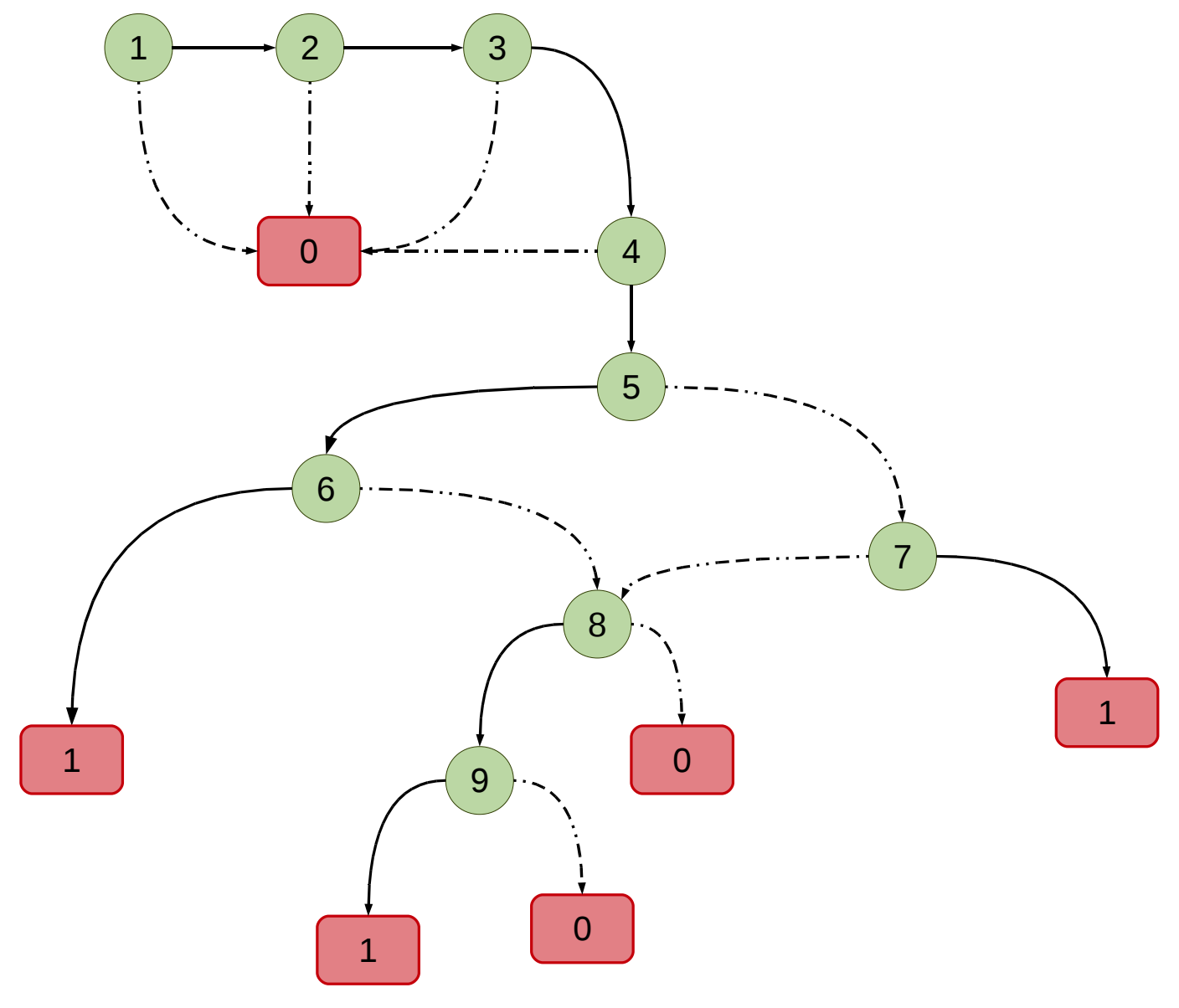
SALT

```
c Satisfiability Application Logic Translator (salt) v1.5
c     written by James Ezick (ezick@reservoir.com)
c Copyright (C) 2005-2007 Reservoir Labs
c
c Thu Mar 26 11:34:58 2009 EDT
c
c input file:         http-sig-or-atphttpd-ghttpdlog.salt
c echo:           no
c forced captures:    yes
c optimizations: yes
c capture threshold:  3
c rounding mode: truncate
c ignore overflow:    no
c ignore conflicts:   no
c
p cnf 20 30
c signature : OR(Atphttpd, ghttpdlog)
c type : Protocol aware

14 15 0
16 -14 0
9 17 -14 0
8 17 -14 0
-17 -16 14 0
-8 -9 -16 14 0
13 18 -15 0
11 18 -15 0
1 18 -15 0
-18 15 0
-1 -11 -13 15 0
4 -16 0
3 -16 0
1 -16 0
2 -16 0
-2 -1 -3 -4 16 0
6 7 -17 0
-5 6 -17 0
5 7 -17 0
-5 -6 17 0
5 -7 17 0
-1 12 -18 0
-1 -11 -18 0
10 12 -18 0
10 -11 -18 0
1 10 -18 0
1 -10 18 0
-1 11 -12 18 0
19 0
-20 0
```

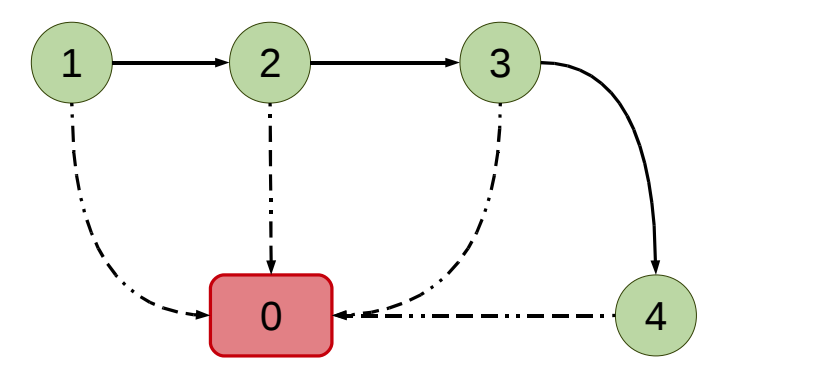
cnf2bdd

```
./cnf2bdd http-sig-atphttpd.cnf
-o http-sig-atphttpd.bdd -n 2 -h LINEAR
-r SINGLE http-sig-apthttpd.regex

CNF expression:
==================
+1 0 0 0 0 0 0 0 0
+2 0 0 0 0 0 0 0 0
+3 0 0 0 0 0 0 0 0
+4 0 0 0 0 0 0 0 0
+6 +7 +9 0 0 0 0 0 0
+6 +7 +8 0 0 0 0 0 0
-5 +6 +9 0 0 0 0 0 0
-5 +6 +8 0 0 0 0 0 0
+5 +7 +9 0 0 0 0 0 0
+5 +7 +8 0 0 0 0 0 0

BDD graph:
==================
ROOT: 73
[   18]   8 :   0   1
[   35]   7 :   0  18
[   36]   6 :  35   1
[   54]   5 :  35   1
[   69]   4 :  36  54
[   70]   3 :   0  69
[   71]   2 :   0  70
[   72]   1 :   0  71
[   73]   0 :   0  72

Reordering gain = 0
Number of nodes = 9
Number of variables = 9
Number of CNF clauses = 10

OPTIMIZED CUT:
=======================

BDD Cut 0 (# nodes = 4):
============================
ROOT: 45
[    8]   3 :   0   1
[   43]   2 :   0   8
[   44]   1 :   0  43
[   45]   0 :   0  44

CNF Cut 0 (# clauses = 4):
============================
+1 0 0 0 0 0 0 0 0
+2 0 0 0 0 0 0 0 0
+3 0 0 0 0 0 0 0 0
+4 0 0 0 0 0 0 0 0

BDD Cut 1 (# nodes = 4):
============================
ROOT: 29
[   18]   8 :   0   1
[   29]   5 :  36   1
[   35]   7 :   0  18
[   36]   6 :  35   1

CNF Cut 1 (# clauses = 2):
============================
+6 +7 +9 0 0 0 0 0 0
+6 +7 +8 0 0 0 0 0 0

BDD Cut 2 (# nodes = 5):
============================
ROOT: 69
[   18]   8 :   0   1
[   35]   7 :   0  18
[   36]   6 :  35   1
[   54]   5 :  35   1
[   69]   4 :  36  54

CNF Cut 2 (# clauses = 4):
============================
-5 +6 +9 0 0 0 0 0 0
-5 +6 +8 0 0 0 0 0 0
+5 +7 +9 0 0 0 0 0 0
+5 +7 +8 0 0 0 0 0 0

EQUIVALENT
REGULAR EXPRESSION:
============================
((111110.11|111111...)|11110(.011|.1..))
```

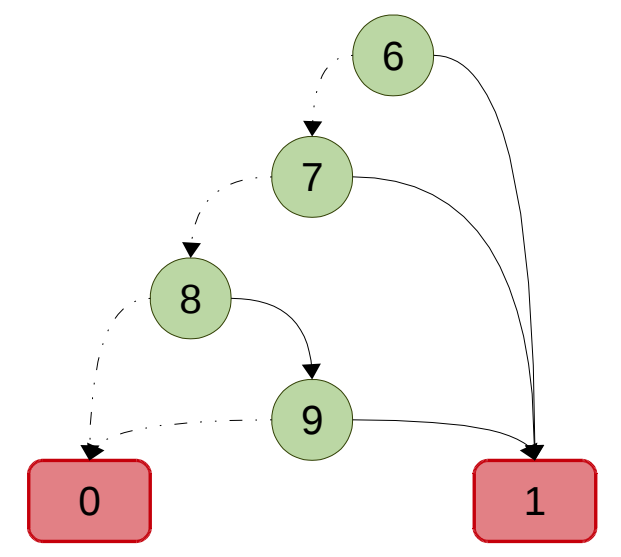
SINGLE BDD



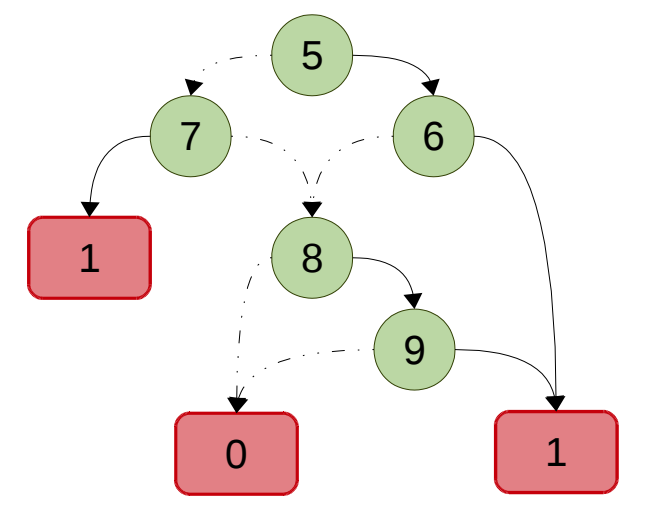
OPTIMIZED 3-CUT



bit1 & bit2 & bit3 & bit4

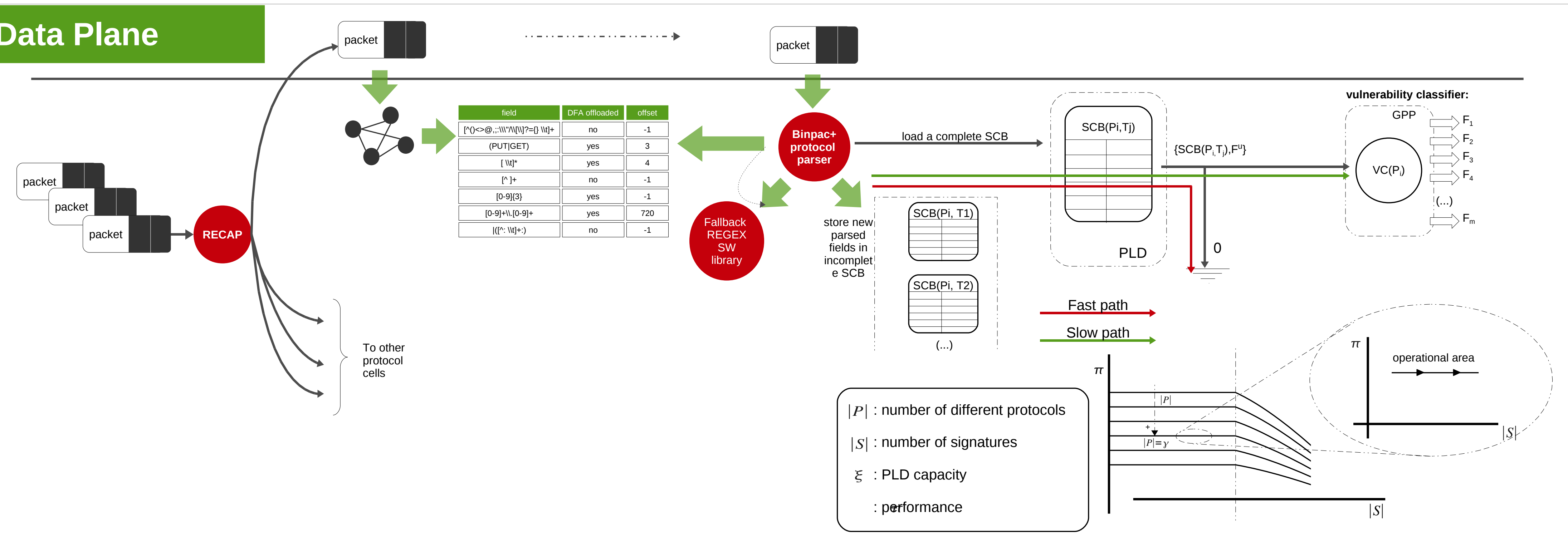


(bit6 | bit7 | bit9) & (bit6 | bit7 | bit8)



(!bit5 | bit6 | bit9) & (!bit5 | bit6 | bit8) & (bit5 | bit7 | bit9) & (bit5 | bit7 | bit8)

Data Plane



field	DFA offloaded	offset
[^()<>@,;:\\\"/\\[\\]?={} \\t]+	no	-1
(PUT|GET)	yes	3
[\\t]*	yes	4
[^]+	no	-1
[0-9]{3}	yes	-1
[0-9]+\\.[0-9]+	yes	720
|([^: \\t]+:)	no	-1

$|P|$: number of different protocols

$|S|$: number of signatures

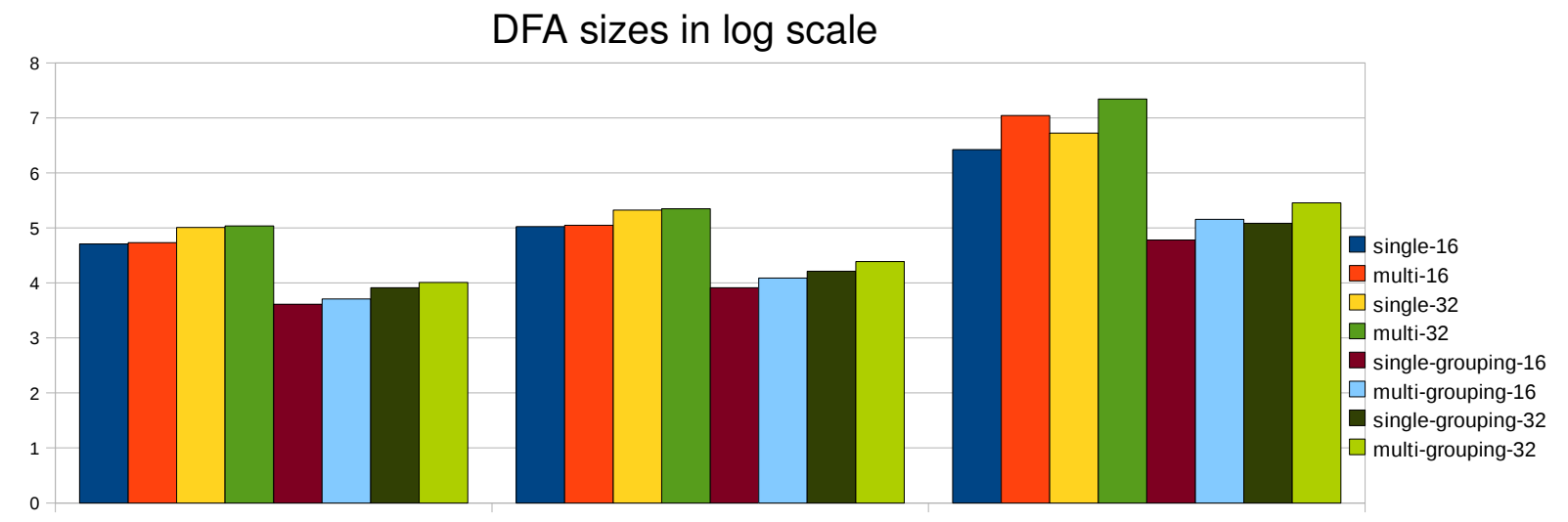
ξ : PLD capacity

π : performance

DFA Generation Tool

```
usage: cnf2bdd input_cnf_file_name -o output_bdd_file_name
                [-n number_of_dfa_engines]
                [-h HEURISTIC METHOD]
                [-p node_probability_file_name]
                [-r (MULTI|SINGLE) output_regex_file_name]
                [-g GROUPING_METHOD [length]

    HEURISTIC := (LINEAR | EXHAUST | BINARY)
    GROUPING_METHOD := (VARIABLE | FIXED)
```

DFA sizes in log scale



log scale	atphttpd	ghttpdlog	atphttpd OR ghttpdlog
single-16	4.71	5.02	6.42
multi-16	4.73	5.05	7.04
single-32	5.01	5.32	6.72
multi-32	5.04	5.35	7.35
single-grouping-16	3.61	3.91	4.78
multi-grouping-16	3.71	4.09	5.16
single-grouping-32	3.91	4.21	5.08
multi-grouping-32	4.01	4.39	5.46

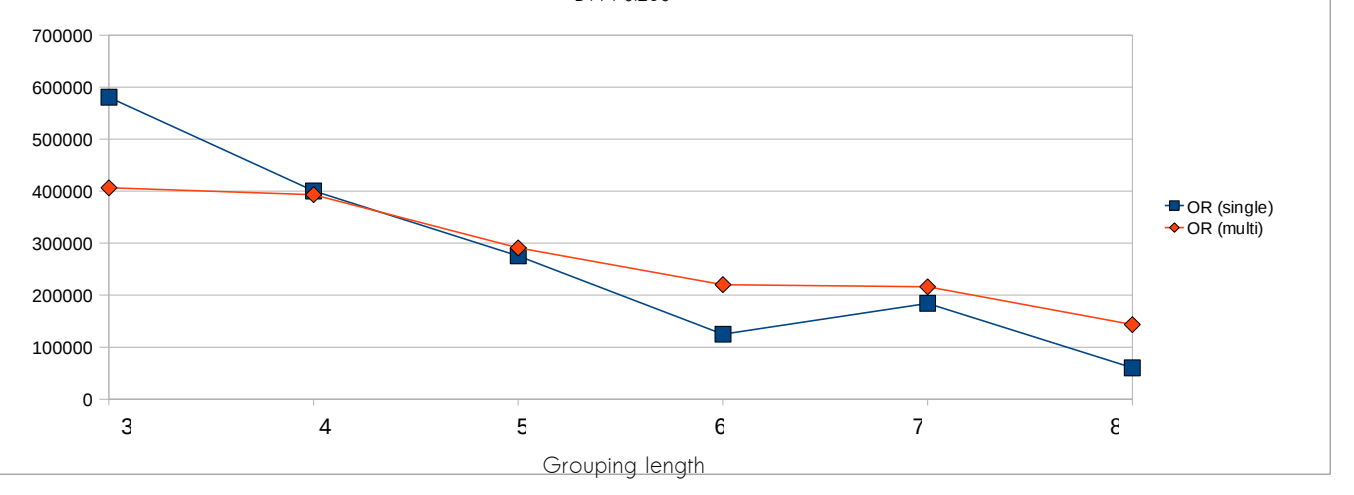
linear scale	atphttpd	ghttpdlog	atphttpd OR ghttpdlog
single-16	51200	105472	2654208
multi-16	54272	111616	11087872
single-32	102400	210944	5308416
multi-32	108544	223232	22175744
single-grouping-16	4096	8192	60416
multi-grouping-16	5120	12288	143360
single-grouping-32	8192	16384	120832
multi-grouping-32	10240	24576	286720

DFA sizes

Effect of grouping

grouping length (single)	3	4	5	6	7	8
size (16bit)	580608	400384	275456	124928	184320	60416

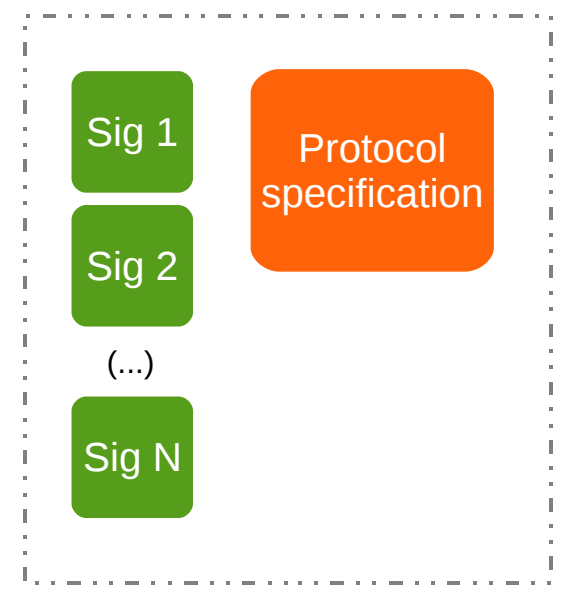
grouping length (multi)	3	4	5	6	7	8
size (16bit)	406528	393216	290816	220160	216064	143360

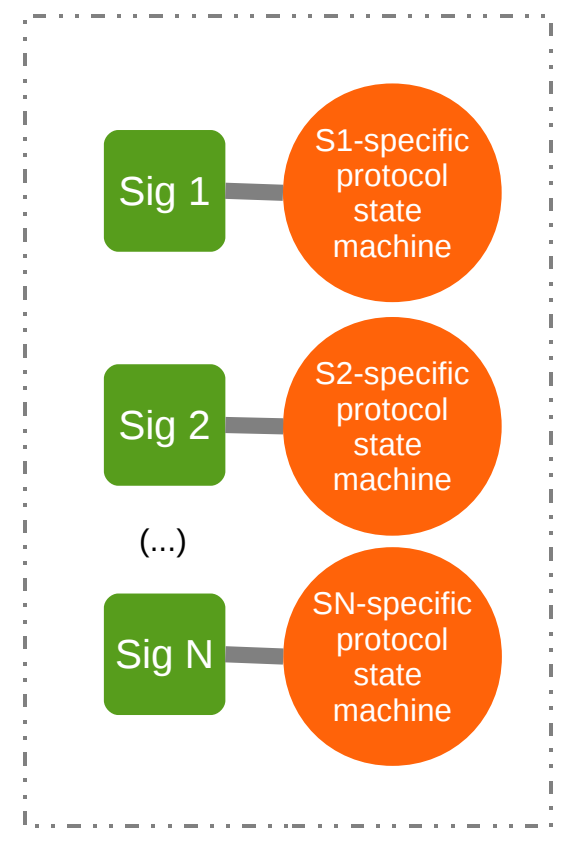


Related Work

N. Schear, D. R. Albrecht, N. Borisov, "High-speed Matching of Vulnerability Signatures," Symposium on Recent Advances in Intrusion Detection, 2008.

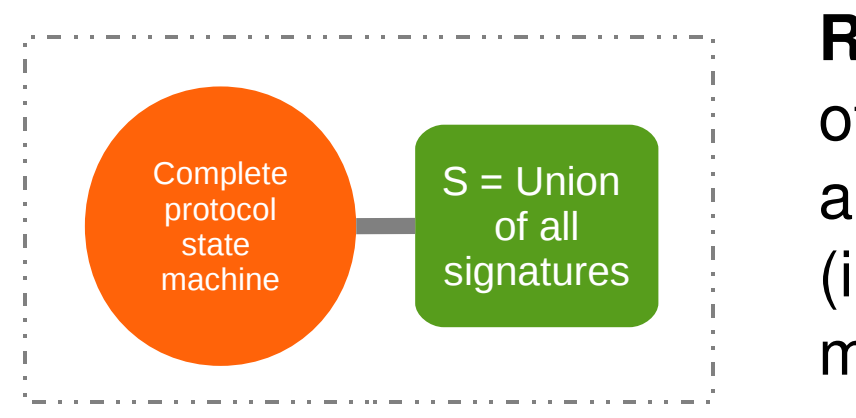
Input: set of protocol-aware signatures plus protocol specification associated with signatures




Sig 1
S1-specific protocol state machine
Sig 2
S2-specific protocol state machine
(...)
Sig N
SN-specific protocol state machine

Schear's approach:

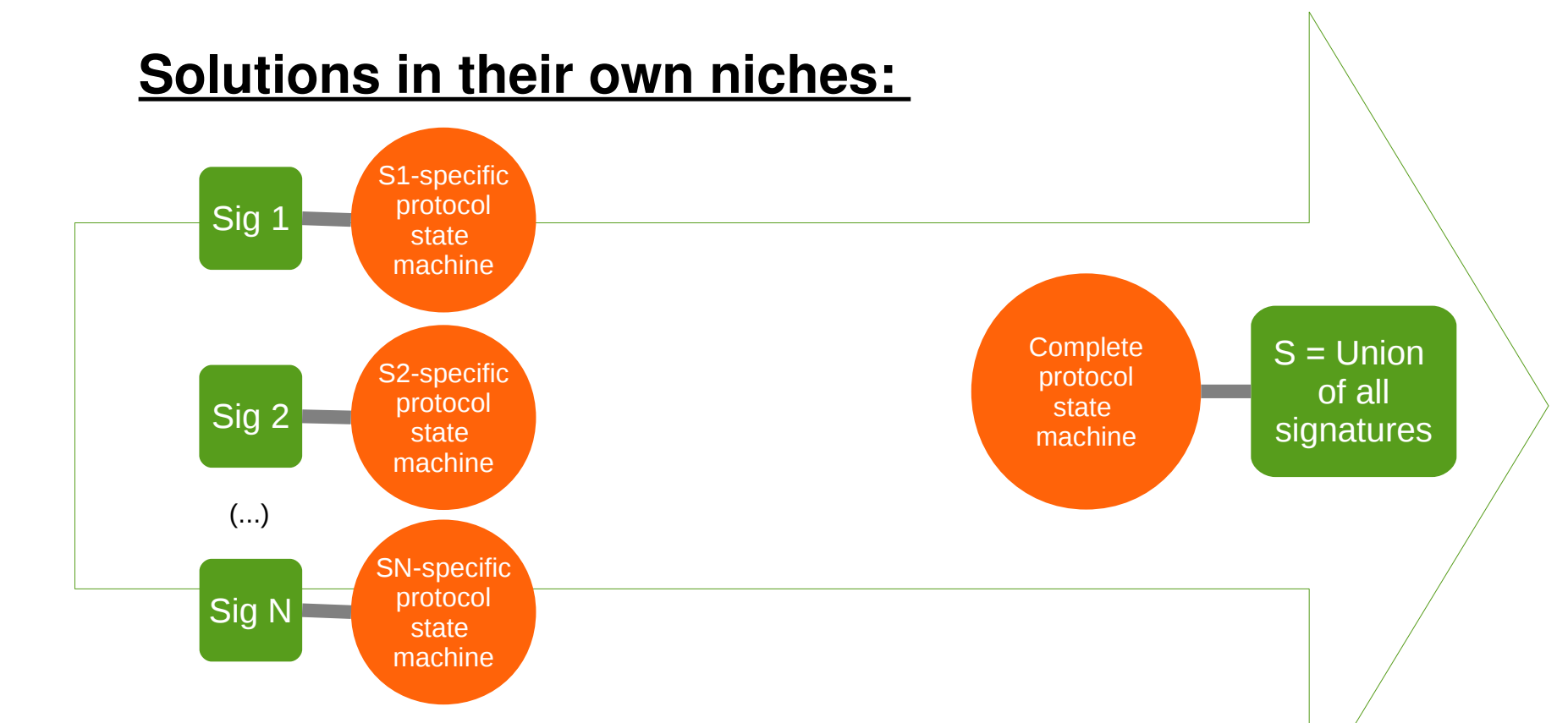
Rationale: a single signature only uses a small portion of the protocol state machine; by customizing the state machine to each signature (removing those elements in the state machine that are irrelevant to the signature), each signature can run much faster



Our approach:

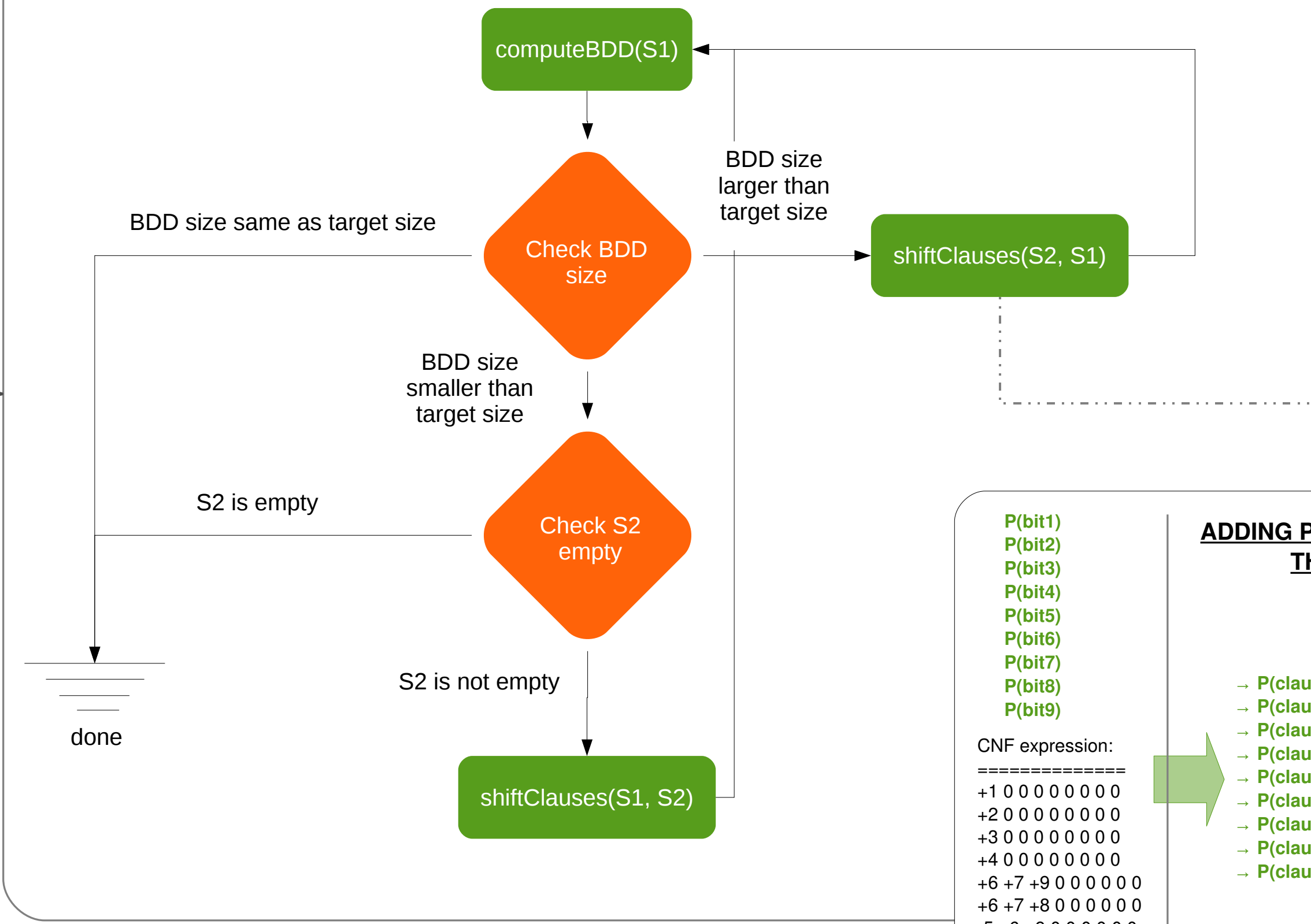
Rationale: if N is large, then it is likely that the union of the state machines implemented using Schear's approach will be close to the complete state machine (in the limit, it will match the complete state machine); in this case, it pays off to have one single complete state machine parsing the complete set of inputs needed to resolve all the signatures; the signatures can be computed in parallel by resolving their OR.

Solutions in their own niches:


Sig 1
S1-specific protocol state machine
Sig 2
S2-specific protocol state machine
(...)
Sig N
SN-specific protocol state machine
Complete protocol state machine
S = Union of all signatures

Small number of Signatures **Large number of Signatures**

Heuristic for simple case with 1 DFA engine and a Software regex fallback library (2 cuts):



ADDING PROBABILITIES TO THE MODEL

P(bit1)
P(bit2)
P(bit3)
P(bit4)
P(bit5)
P(bit6)
P(bit7)
P(bit8)
P(bit9)

```
CNF expression:
==============
+1 0 0 0 0 0 0 0 0
+2 0 0 0 0 0 0 0 0
+3 0 0 0 0 0 0 0 0
+4 0 0 0 0 0 0 0 0
+6 +7 +9 0 0 0 0 0 0
+6 +7 +8 0 0 0 0 0 0
-5 +6 +9 0 0 0 0 0 0
-5 +6 +8 0 0 0 0 0 0
+5 +7 +9 0 0 0 0 0 0
+5 +7 +8 0 0 0 0 0 0
```

→ P(clause1)
→ P(clause2)
→ P(clause3)
→ P(clause4)
→ P(clause5)
→ P(clause6)
→ P(clause7)
→ P(clause8)
→ P(clause9)

Use P(clause) to know how to shift clauses from one cut to another. The objective would be to throw on the DFAs those clauses that have the highest probability of being FALSE

- → S1 and S2 are initially an arbitray partition of the CNF expression
- → S1 refers to the set of partitions offloaded to HW DFA
- → S2 refers to the set of partitions that will run on SW (fallback)
- → obs: S2 only needs to be invoked if all the CNF clauses in S1 are true;
- → S1 defines a fast path, S2 defines a slow path

- → **computeBDD(X)** : given CNF expression X, it computes its BDD minimizing state explosion
- → **shiftClauses(X, Y):** shifts clauses from Y to X
