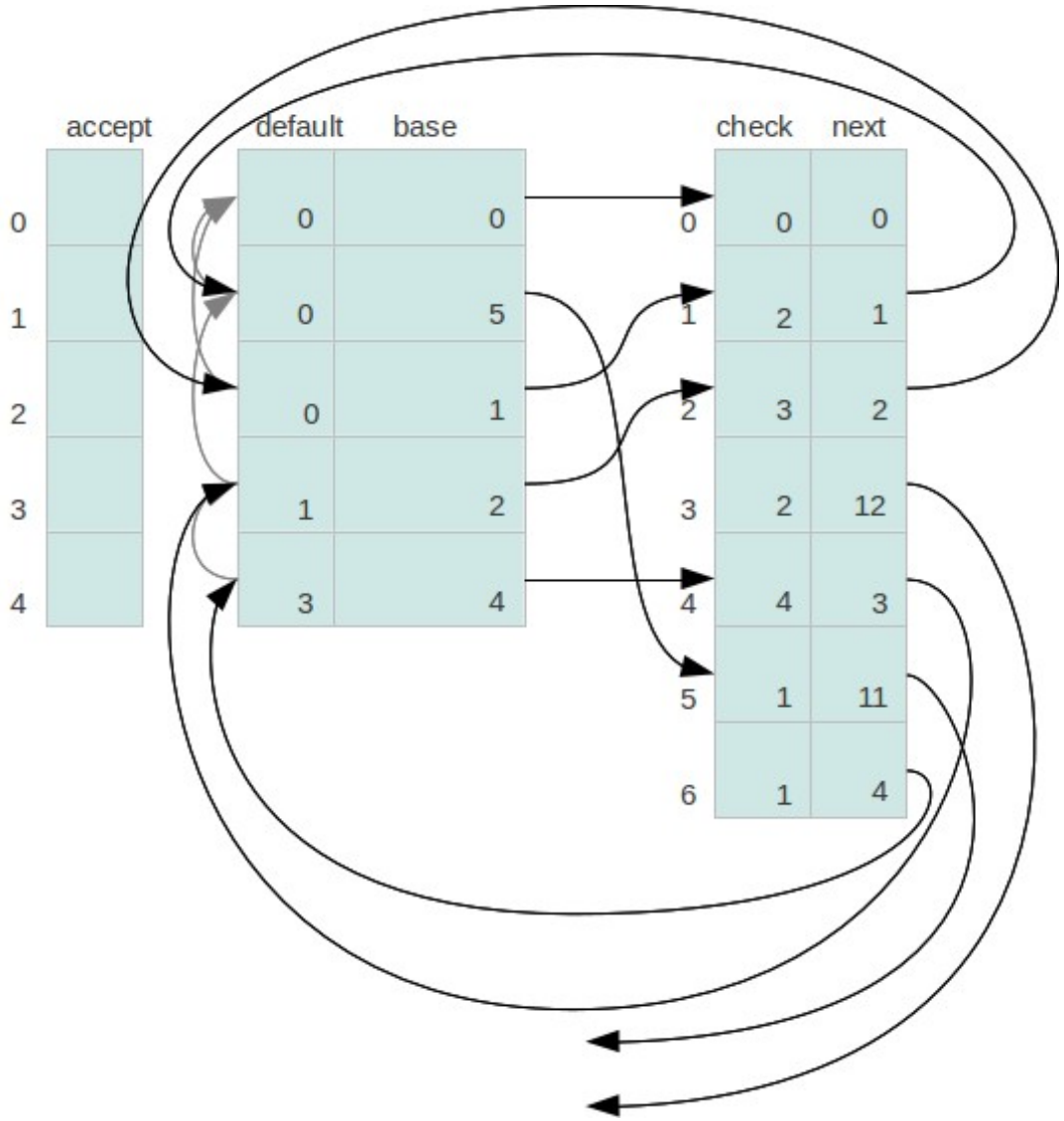
Format of the DFA/HFA

The AppArmor compressed DFA/HFA format is based off of the flex DFA table format. The base format has remand the same since its introduction but it has been tweaked and extend.

Base Layout DFA

The base layout consists of a header (at the start of the stream) containing DFA generic information and then a set of tables, each with their own headers with data following immediately after.



The base dfa layout is common to all versions of AppArmor, it has five tables: default, base, next, check, accept. The default, base, and accept tables have the same length and are indexed by the state number. The check, and accept tables are the same size and are index by a states base value + input character.
The default table stores the default transition for a given state, it is used when, when the check value

at base + input does not equal the current state. The base value is the start position in the next/check tables, for the states transitions. The next table provides the next state to go to for the base + input if the corresponding check entry matches the current state.

As shown above in states 2 transitions, they can be interleaved with over states transitions. This comb compression is done to save space, instead of requiring 256 transition entries for a given state only the transitions not covered by the default transition need to stored in the next check table. It is possible that some entries of the next/check tables may be empty but comb compression will usually fill the majority of entries.

The above diagram can not show the differential state compression, because default state transition are used to represent both differential state compression and simple default state transitions, the only difference being a flag that is stored in the high bit of the base table.

The accept table while being part of the DFA is not used by the DFA, its values are defined externally to the DFA.

DFA16

In the DFA16 format, the default, next and check tables are 16 bits wide allowing for at most 65,536 states. It is used when possible to reduce the amount of memory needed by a DFA at run time. The base table is 32 bits wide with the upper byte reserved for flags. The accept table size is defined by the application (see Encoding permissions for apparmor's use of it).

DFA32

In the DFA32 format, the default, next, and check tables are 32 bits wide allowing for the DFA to be much larger at the cost of a larger memory foot print. The base table remains 32 bits wide but a new table is added to store that states flags, thus giving it a full 32bit index. The accept table size remains defined by the application just like in the DFA16 format.

AppArmor 2.1

AppArmor 2.3 - 2.7

A second accept table was added

AppArmor 3.0

eHFA/eDFA

flags

At this time the upper 8 bits are reserved for flags. Would like to only use 4 or less for flags, and use the rest to extend the range

There are several flags used to track which states have special conditions.

- differential state compression
- variable conditional accept
- back ref - can this be tied to variable conds
- counting constraing - can this be tied to variable conds
- border state - could just do side lookup?

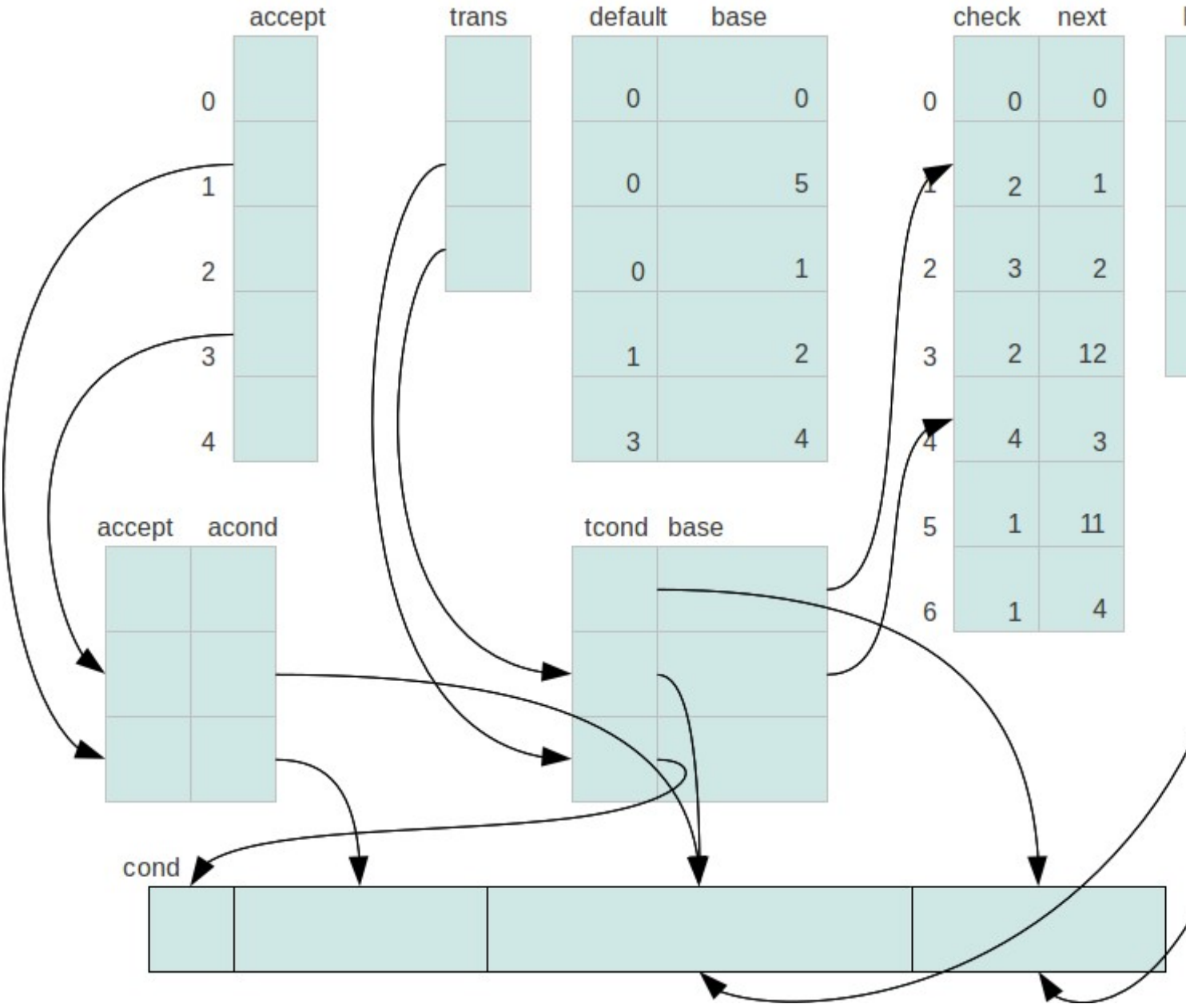
•forward lookahead, and negative forward lookahead for match at this state? Or should this be encoded in a perm

•need to know what state(s) perms to hit on the look ahead so probably in perms

2-5 bits would like to get 28 bits for base value

eDFA - variable extension

In addition to the tables and transitions outlined for the base dfa the variable extension add several new tables and transition relationships.



The variable eDFA extends the base DFA with six new tables: caccept, trans, inst, tcond, tbase, and cond. The caccept, and trans tables are the same size and maybe less than or equal to the size of the accept, default and base tables. The inst table maybe less than or equal to the size of next check and the tcond, tbase and cond tables size is independent of all other tables.

The cond table stores, logical variable combinations, like A&B, A&B&C, etc.

The trans table stores an index into the tcond and tbase tables that are used to find the transitions for a particular variable combination that is possible for the state. The tcond table is an index into cond for a particular variable condition, and tbase references into next/check for the actual transition.

The inst table stores an index into the cond table for the set of variables that a transition into next will instantiate.

If the state is flagged as conditionally accepting the accept table entry takes on a new meaning, it becomes an index into the acond, and aaccept tables. These provide the accept value that matches the current variable match conditions.

The tcond/tbase and acond/accept tables, are stored in an ordered set of perms that belong to the referencing state, that is the state can have more than one entry. ?? does the state have another table to list how many entries, or is a null entry used. Another table for size is smaller. So add an extra table?

Note that states that have conditional accept, or conditional transitions are sorted to be in the lower DFA state entries.

eDFA - back reference extension

Back references are handled as an extension of the variables extension with additional with additional tables that are used to set up variable values, so that they can be matched against.

eDFA - counting constraints

Counting constraints while similar to the variable extension are separate and see their own set of tables added (that are similar to the variable tables) to set up the counts, and handle conditional matching based off of the counts.

HFA - NFA border states

The addition of border states to the DFA changes it into an HFA by adding introducing an NFA state. A border state marks the transition from one sub DFA to a set of sub DFAs. It does this by storing a set of transitions, one for each sub DFA it transitions too, which can then be match by breadth or depth first match just has an NFA would do.

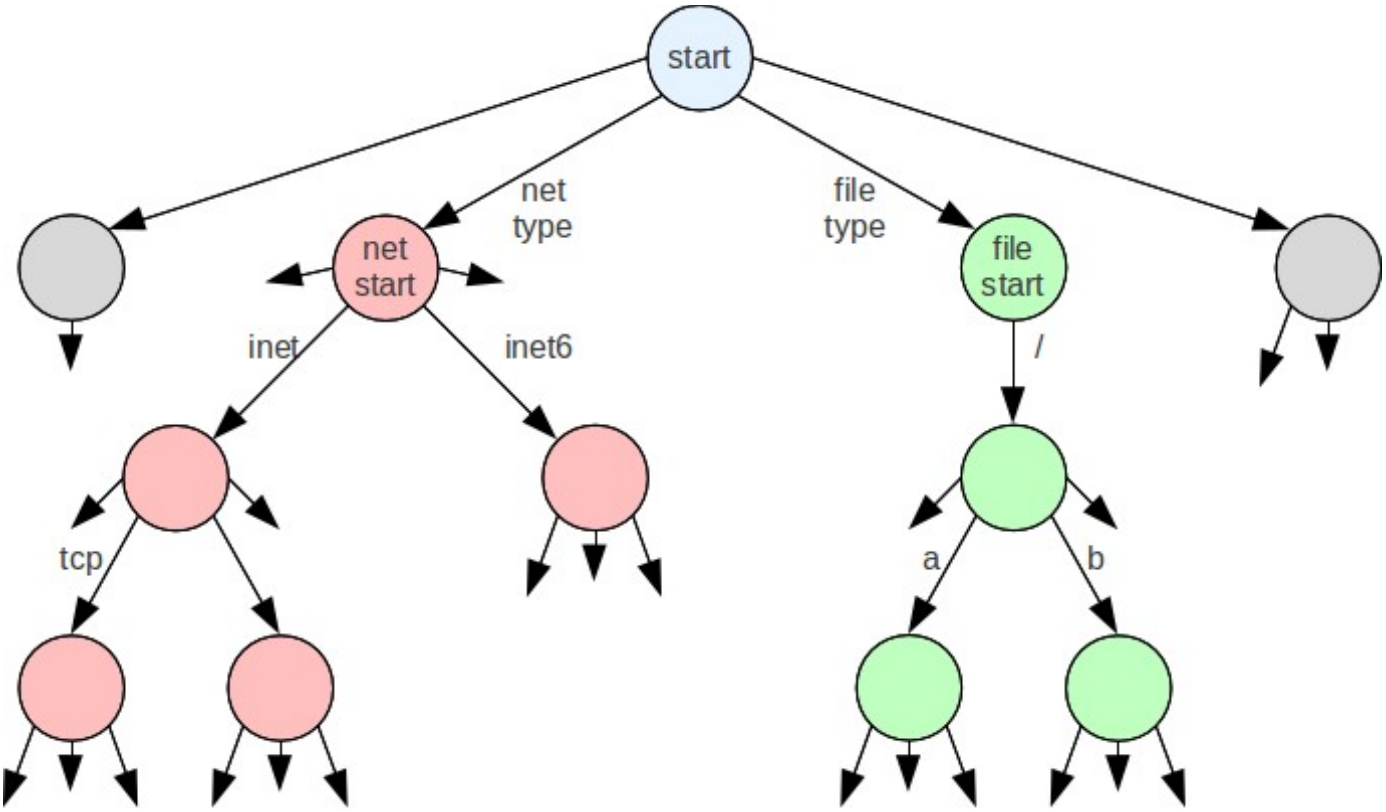
PolicyDB – encoding of policy rules into the HFA

The PolicyDB is the state machine that matches various permission requests to the permission permissions granted by a profile.

AppArmor 3.0 introduced the PolicyDB, which is extends the use of the HFA beyond file rules into other mediation types. The PolicyDB allows for generic queries to be made against AppArmor policy using just the HFA. For backwards compatibility reasons masks and some other structures are retained and used but all information is also recorded in the PolicyDB.

The layout of the PolicyDB can be thought of as a tree, that begins with the HFA start state. From here a single byte transition based on the type of permission request, finds the rules governing that

type. Further transitions within a type lead to more specific sub-types and eventually a match that can be used to determine permissions.



Eg. Doing

```
file_rules_start_state = next_state(PolicyDB, start_state, AA_FILE_TYPE)
```

will find the file rules within the PolicyDB. Note that file rules are stored in a backwards compatible manner so that, direct access is possible by specifying an alternate start state for file rules.

Each kind of permission request has a defined types, with unknown types reserved for future expansion. The Layout and ordering of matching within a given type, is tailored to the the input of the types permission request, so each type has its own layout and high level match routine.

The currently define classes for the policydb are

- AA_CLASS_CAP
- AA_CLASS_FILE
- AA_CLASS_ATTACH
- AA_CLASS_ENV
- AA_CLASS_ARGV
- AA_CLASS_MOUNT
- AA_CLASS_NET
- AA_CLASS_PIPE
- AA_CLASS_SYSV_MSGQ
- AA_CLASS_SYSV_SEM
- AA_CLASS_SYSV_SHMEM
- AA_CLASS_SIGNAL
- AA_RLIMIT_CLASS

- AA_MEMORY_CLASS
- AA_CPU_CLASS
- AA_CLASS_AUDIT
- AA_CLASS_KEY
- AA_CLASS_DBUS
- AA_CLASS_X

File type

Link and snapshot rules

change_profile

exec rules and environment filtering

Capability type

Mount type

Network type

Encoding permissions

The encoding of permissions in the DFA/HFA has evolved over time.

AppArmor 2.1 AppArmor 2.3 - 2.7

AppArmor 3.0

Reasons for the major permission rework

There are four basic reasons a permission layout was changed after AppArmor 2.7

Efficiency The layout of AppArmor 2.1 - 2.7 permissions was inefficient wasting space, even for the limited set of permissions covered by AppArmor 2.7. Even though the full set of bits per entry where in use, if a state shared the same permission with another state the information was duplicated, 64bits per state. With a DFA/HFA with a size of 10,000 states typically having between 20 and 30 unique accept permissions, the result was less than efficient, and would be even less efficient if the accept information was widened.

```
 2.7 accept table size = 8*(states)
 3.0 accept table size = 2*(states) + (# unique perms) * (size of expanded
permission block)
```

So assuming the accept block size was not expanded and encoded the same 64bit permissions, and there where 30 unique permissions to encode.

```
 2.7 format = 8*10,000 = 80,000 bytes
 3.0 format = 2*10,000 + 30*8 = 20,240 bytes
```

And as the accept block is expanded the saving increase.

Future expansion As documented in the AppArmor 2.7 layout all 64bits of the permission structure where in use, an expansion was needed to be able to add new abilities. The expansion could have just expanded the width of the accept table, but that would not have been as efficient or flexible.

Flexibility The older accept format was to ridged to be able to efficiently represent, all the new abilities that where desired.

Speed Reworking the layout provided for better cache efficiency, and no need to unpack and transform the entries to determine the final permissions.

The permission rework also affected the the internals of the DFA/HFA compilation engine. - rework to track information

```
- lost at tree to dfa state
```

- provide for full minimization - provide for per rule negation - provide for conditional tracking

AppArmor 2.1

AppArmor 2.1 used a single 32 bit accept table in DFA to store permission.

Layout

Compiler internals

AppArmor 2.3 - 2.7

File Permission Layout

AppArmor 2.3 introduced a second accept 32 bit accept table to the DFA making each accept state a total of 64 bits wide, and reworked the bit layout, breaking compatibility with AppArmor 2.1.

The permissions where packed into the accept states as follows

Bit layout of AppArmor 2.3 permissions

Bit	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**
	Global											**Other**																	**Owner**
Table 1 - perms mask	cp	on x			x3	x2	x1	x0	i	u	pu x	m	k/ ls	l	a	r	w	x	x3	x2	x1	x0	i	u	pu x	m	k/ ls	l	a
Table 2 - audit mask	cp	on x			q m	q k	ql	q a	qr	q w	qx	m	k/ ls	l	a	r	w	x	q m	q k	ql	q a	q r	q w	qx	m	k/ ls	l	a

The permissions stored in the **owner** mask are used if the objects uid == to the tasks fsuid, other wise the other mask is used.

Execute permissions is determined by x, with the transition type determined by a combination of

pux, u, i, and x0-x3. i - fall back to inherit current profile u - unsafe exec, have glibc scrub environment variables pux - unconfined flag used in combination with x0-x3 to create pux. Was added after original layout to extend pux support explaining duplication of unconfined flag. x0-x3 - index into transition table. 0 - not defined (use i, u to determine transition) 1 - unconfined (precludes px) 2 - use executable name 3 - use executable name, lookup child profile 4-15 - index into named transition table, to find transition name. Note: named transition to children profiles where stored using compound name (ie parent//child), so they could share in a single global table entry.

cp - is the change_profile permission onx - change_profile on exec permission

the link pair permission overlaid the on k permission on the second pair.

ls - link subset permission, overlaps k permission, only happens in second match of chain (see link below) where k is not possible.

The audit and quiet table (Table2). The permissions stored in table 2 are used to determine if a permission is forced audited, or if for denied requests where the auditing is quieted. The upper 7 bits of the Owner and Other sets store the quiet masks, which get shifted by 7 bits when applied to mask the x through m permissions.

Link rules are encoded requiring sequential matching and permissions in two permission tables. First the link name is matched, and a permission lookup for the l permission bit is done against the matched state. A transition on the 0 byte from the current state is done and then the target name is matched, and a second permission look up for the l permission bit is done against the new match state. The link_subset permission (overlaps k in permission table) can also be looked up against the second match.

change profile rules use cp and onx permissions and are overlayed over top of file rule dfa entries. They are encoded in a multiphase lookup, with the change_profile permission bit being used to determine permission. First a check is done against the profile name for permission, if that doesn't succeed then a second lookup is done against the namespace name followed by a colon followed by a profile name.

Compiler internals

- AppArmor 2.6 saw flattening of permissions at state creation time as part of internal optimizations to speed up the compiler.

AppArmor 3.0

Permission Layout

AppArmor 3.0 extends and reworks the dfa and permissions structures used in the AppArmor 2.x series. Unlike the AppArmor 2.x series a single dfa can be shared by more than one profile, with each profile having its own accept permissions for a give state. To accommodate this the permission structure is split off from the dfa. The dfa accept table now contains a single 16 or 32 bit index value (size dependent on dfa16, or dfa32 format).

If the accept index value has its high bit clear (15 or 31) then it is an index into a profiles permission table. If the accept index value has its high bit set then it is an index into a profiles conditional table.

Each profile that references the dfa defines its own conditional and permission tables allow each profile to have a different set of permissions for any given state.

This split of the dfa and permissions was done to allow for dfa sharing and to provide greater performance as when a dfa is shared multiple profile matches can performed at once. It also allowed reducing the amount of memory dedicated to storing permission data, by allowing for duplicate elimination of permissions.

The split of the conditional information from the permissions structure was done to allow for a much broader range of conditions that could be easily extended in the future, and for better sharing of the extended permission structure.

The permission structure for AppArmor 3.0 is extended to support a much broader set of permissions, and information. Firstly as mention with the dfa accept table value, it has been split into a permission table and a conditional table. The conditional table contains entries that encode conditional expressions to obtain a permission (eg. the owner flag), the permission table contains the actual permissions that the profile will grant.

The permission table has been expanded to support a broader set of permissions, including file, network, ipc.

File Permission Entry

Network Permission Entry

Capability Permission Entry

????

Compiler internals

- rework of permission handling to find unique permissions during DFA/HFA creation and to carry full permission information through out the DFA/HFA creation pipe line. Carry the full information through the entire pipeline reduced performance slightly but was offset by improvements brought by reducing comparison time by separating out the accept states into unique sets.

The AppArmor DFA/HFA uses multiple unique accept states to encode permissions, and related control information for any given match.

Why multiple distinct accept states

could encode in stream but

```
 have to make multiple matches one for each permission
 still have to store extra information for control that happens for the matched
state (xtrans ..)
```

Encoding Conditional permissions

AppArmor encodes conditional information both in the DFA/HFA and in a separate conditional structure. The split is determined by how the permission request is done. Permission queries done with information external to the object (file path vs. inode) are done within the dfa while, permissions conditional on object properties are done in the conditional table.

Encoding within the DFA/HFA (parameter)

To encode conditionals within the DFA alternations and ordering are used.

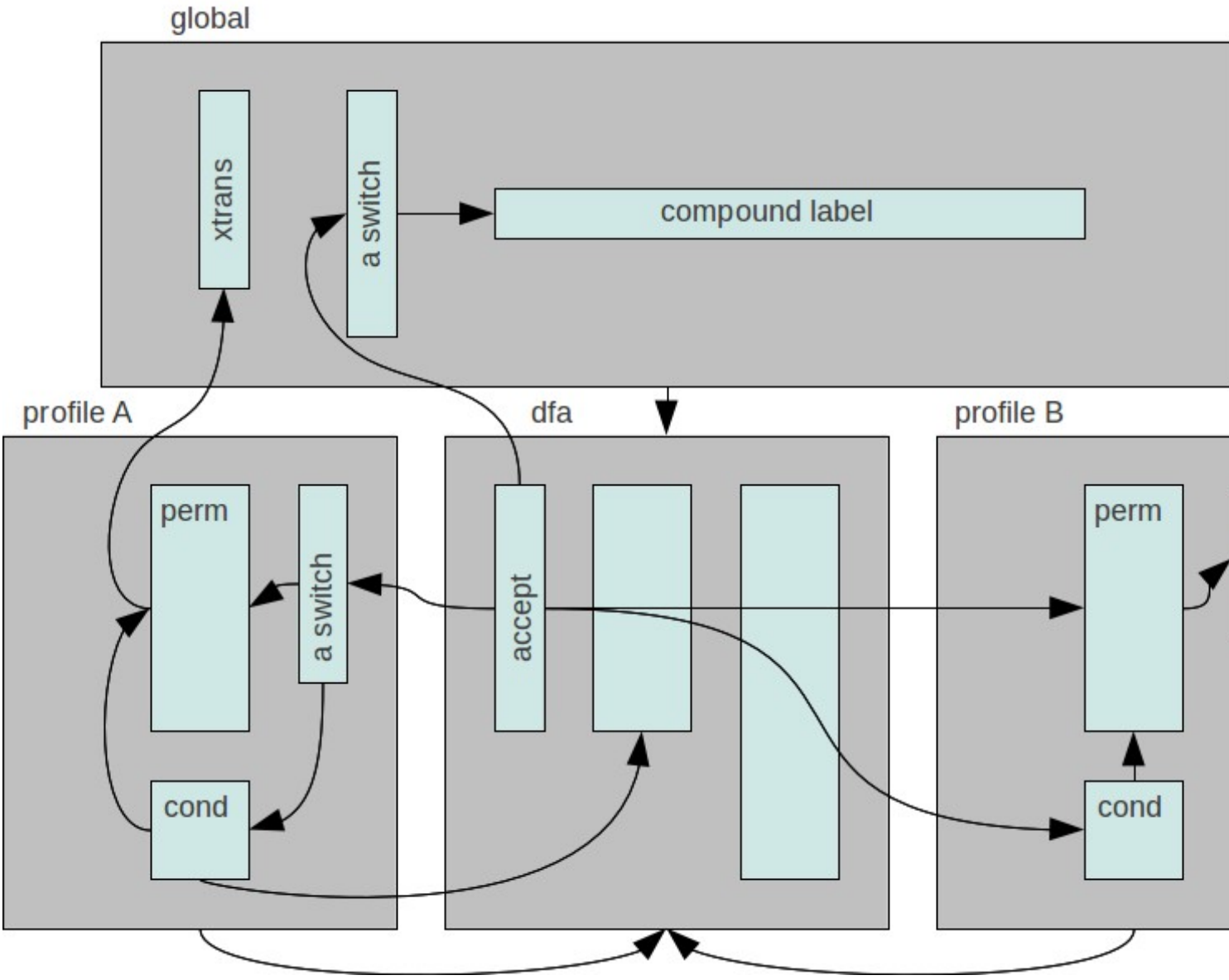
Encoding in the (object)

When conditionals are encoded in stream

When conditionals are encoded out of band

Putting it altogether – relationship of the PolicyDB and DFA/HFA

Overview



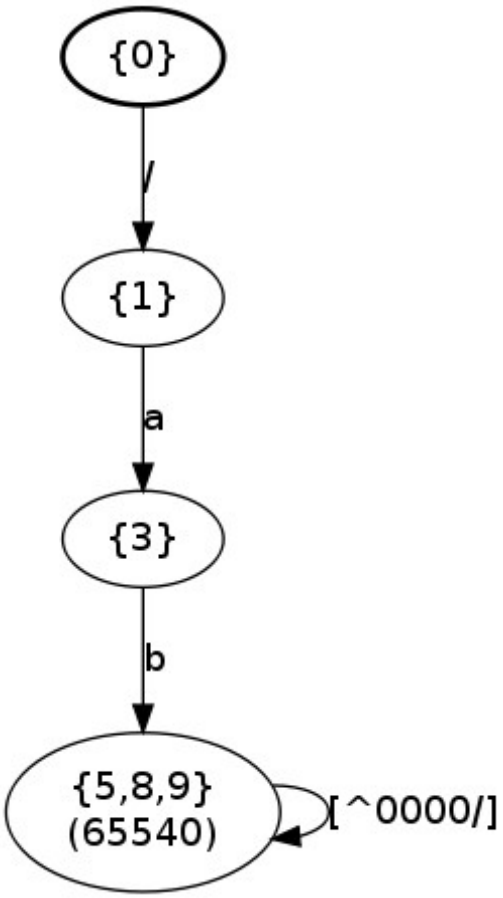
Steps in generating the DFA/HFA

A Brief explanation of eHFAs in AppArmor

AppArmor uses an extened Hybrid Finite Automata (eHFA) as its matching engine. An HFA is merely a finite automata (NFA) that has been specially constructed so that it is made up of finite automatas (dfas) that are joined by special transitions or border states. In its simpliest form an HFA can be reduced to a single DFA. The "extended" prefix indicates that the HFA has been extended beyond traditional nfas/dfas with additional functionality (currently limited to dynamic runtime variable matching).

AppArmor 2.1 was the first iteration of apparmor to use a dfa to perform its pattern matching as they provide a fast matching that is linear to the input stream. It has since been extended to be an eHFA.

AppArmor is capable of generating dfa graphs that can help understand what is happening in the apparmor_parser back end.



- •circles represent a state
- •{ } within a circle are the set of expression tree nodes that make the state
- •( ) within a circle are optional and indicate an accepting state. The number is the permission set for the state
- •characters along a given edge indicate valid transitions. If there is input that doesn't match a valid transition then the dfa transitions to a null non-accepting state that can not be exited (ie it will consume all input).

Issues with AppArmor's DFA

The dfa that apparmor uses, is a simple direct representation with the most basic of table compression schemes. It also does not allow for in line variable matching or other advances that are desirable.

Specific issues with the current implementation

- •it generates large tables that need to be loaded into kernel consuming memory that could be used else where
- •it takes a relatively long time to compute

- •the generation of the dfa without the factoring steps can be impractical even on modern hardware
- •the dfa is subject to exponential state explosion.
- •equivalence classes (if used) are applied post generation

•the table compression algorithm is a simple and straight forward with O(n^2) running time (slow). The compression achieved is not perfect but is fairly good given the limits of the compression format.

•the table compression format is simple and direct and does not leverage any state or transition redundancies, resulting in poor compression compared to what could be achieved

•The implementation needs to be cleaned up

Overall the dfa engine needs to be updated and replaced with a better atomata representation that keeps many of the dfas properties while being flexible, faster to compute (control state explosion), and leverages a better compression format. Better alternative exist so it is only a matter of examining the alternatives and implementing.

Rule to DFA conversion

AppArmor file rules are compiled into a dfa that is loaded into the kernel to do path matching. To do this AppArmor transforms the rules in the profiles, going through several steps to create the final dfa. The pipeline is as follows

```
profile rules
    |
    v
sort profile rules
    |
    v
merge duplicate rules
    |
    v
convert rules to expression tree
    |
    v
expression tree simplification
    |
    v
dfa creation
    |
    v
dfa minimization
    |
    v
dfa unreachable state removal
    |
    v
creation of equivalence
    |
    v
transition table compression
```

To illustrate the steps of the conversion pipeline the following simple profile is used through out the following discussion

```
/usr/bin/example {
  /etc/passwd r,
  /home/*/** rl,
  /home/*/bin/ ix,
  /home/likewise/*/*/** rwl,
  /{usr,}/bin/** px,
  /etc/passwd r,   # duplicate
  /home/*/** w,    # duplicate

}
```

rule sorting

The first step is sorting path based rules this allows for simple merging can take place, and also aids in regular expression factoring as it put similar expressions close to each other in the tree.

```
/usr/bin/example {
  /etc/passwd r,
  /etc/passwd r,   # duplicate
  /home/*/** rl,
  /home/*/** w,    # duplicate
  /home/*/bin/ ix,
  /home/likewise/*/*/** rwl,
  /{usr,}/bin/** px,
}
```

merge duplicates rules

Rules with the exact same path description are merged, merging permission as well. There are exceptions in rule merging x permission and rules that have a pair, like link rules may actually be separated. This step reduces the number of rules going through pcre conversion and also the amount of rules that expression simplifications needs to consider. The permissions merging done in this step results in smaller expression trees (tree simplification can not do permission merging as it does evaluate at the rule level), and hence smaller dfas being constructed.

```
/usr/bin/example {
  /etc/passwd r,
  /home/*/** rwl,
  /home/*/bin/ ix,
  /home/likewise/*/*/** rwl,
  /{usr,}/bin/** px,
}
```

rule to expression tree conversion

AppArmor globbing rules are converted into a single pcre expression tree, that represents the rules for entire profile. This is done by first converting an individual rule into and expression tree and then using an alternation to merge it into the main expression tree (this is equivalent to combining two rules with an alternation).

Early versions of AppArmor this step is split into two parts, a text conversion to pcre syntax, and then a pcre parse tree is generated from that.

```
 eg.  rule1 r, rule2 w,    can be combined as (rule1 r|rule2 w)
```

The handling of rule permissions in tree conversion ???

Note: each parse tree includes a fake node containing the permission of the rule.

This step can be examined in AppArmor 2.4 and later by doing

```
apparmor_parser -QT -D expr-tree <profile>
```

eg, for a single rule from a profile do

echo "profile example { /home/*/** rl,} " | ./apparmor_parser -QT -D expr-tree

DFA: Expression Tree

(/home/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*(((<4>|<16>)|<65536>)|<262144>)|/home/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*\0000/[^/](.)*((<16>|<32>)|<262144>))

eg. for the profile

./apparmor_parser -QT -D expr-tree <profile_name>

DFA: Expression Tree

((((((/etc/passwd(<4>|<65536>)|/home/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*(((((((<2>|<4>)|<8>)|<16>)|<32768>)|<65536>)|<131072>)|<262144>))|/home/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*\0000/[^/](.)*((<16>|<32>)|<262144>))|/home/[^\0000/]([^\0000/])*/bin/(((<513>|<64>)|<8404992>)|<1048576>))|/home/likewise/[^\0000/]([^\0000/])*/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*(((((((<2>|<4>)|<8>)|<16>)|<32768>)|<65536>)|<131072>)|<262144>))|/home/likewise/[^\0000/]([^\0000/])*/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*\0000/[^/](.)*((<16>|<32>)|<262144>))|/(usr|[])/bin/[^\0000/]([^\0000])*(<2305>|<37765120>))

understanding the tree output

The tree is output as as a regular expression where ( ) - groups and expression precedence
| - alternation that separates two alternatives eg. A|B

- * - means repeat the previous expression 0 or more times

+ - means repeat the previous expression 1 or more times
. - match any character
[] - character class which can contain multiple alternative characters
[^] - an inverted character class
\xxxx - a non-printable character

expression tree simplification (AppArmor 2.3 and later)

This stage does expression tree factoring which can remove common subexpressions that cancel out. To do this the tree is first normalized into a left or right normal form, and then a factoring pass is made finding common nodes. The normalization is then flipped and the process is repeated as necessary until the tree stabilizes and no more simplification is possible.

```
eg. (ab|ac) would be factored into a(b|c)
```

This step was introduced because the dfa conversion step can cause exponential state explosion. The factoring is biased towards minimizing the cases that will cause state explosion (right normalization

first). This stage is critical for large or complicated policies as it can make them possible to compute in reasonable amounts of time and memory.

Normalization

Normalization rewrites the expression tree so that alternations and concatenations always have the same consistent form, which simplifies the the Factoring stage.

```
 left normalization rules (right normalization is opposite)
 (E | a) -> a | E
 (a | b) | c  ->  a | (b | c)
 [b] | a -> a | [b]

 Ea -> aE
 (ab)c -> a(bc)



 todo add actual tree diagrams
```

Factoring Patterns

```
 aE -> a
(a | a) -> a
a | (a | b) -> (a | b)
a | (ab) -> a (E | b) -> a (b | E)
(ab) | (ac) -> a(b|c)
```

To see the tree after the apparmor_parser has done simplification do

./apparmor_parser -D expr-simple -QT <profile_name>

DFA: Simplified Expression Tree

/((usr|[])/bin/[^\0000/]([^\0000])*(<2305>|<37765120>)|(etc/passwd(<4>|<65536>)|home(/[^\0000/]([^\0000/])*/bin/(<513>|(<8404992>|(<1048576>|<64>)))|(/likewise/[^\0000/]([^\0000/])*|[])/[^\0000/]([^\0000/])*/[^\0000/]([^\0000])*(<4>|(<16>|(<32768>|(<65536>|(<131072>|(<262144>|(<2>|(<8>|\0000/[^/](.)*(<16>|(<32>|<262144>)))))))))))

dfa creation

AppArmor does direct single regex to dfa creation as described in Compilers - Principles, Techniques and Tools (aka the Dragon Book). This combines the traditional two step conversion of regex to NFA and then subset construction, and should general be faster and consume less memory.

At this stage of the dfa lots of information is available to do analysis on.

To see the dfa statistics and compare how previous optimization have do

```
 # tree stats without simplification
 apparmor_parser -O no-minimize -O no-remove-unreachable -O no-expr-simplify -D
dfa-stats -QT example.txt
 Created dfa: states 58 matching 69     nonmatching 56
```

```
 # tree with simplifications applied
 apparmor_parser -O no-minimize -O no-remove-unreachable -D dfa-stats -QT
example.txt
 Created dfa: states 54 matching 67     nonmatching 52
```

The important output from dumping dfa-stats is the number of states created. The matching entry indicates the number of expression node comparisons that were folded into existing states while nomatching is the number of expression nodes that resulted in the creation of new states.

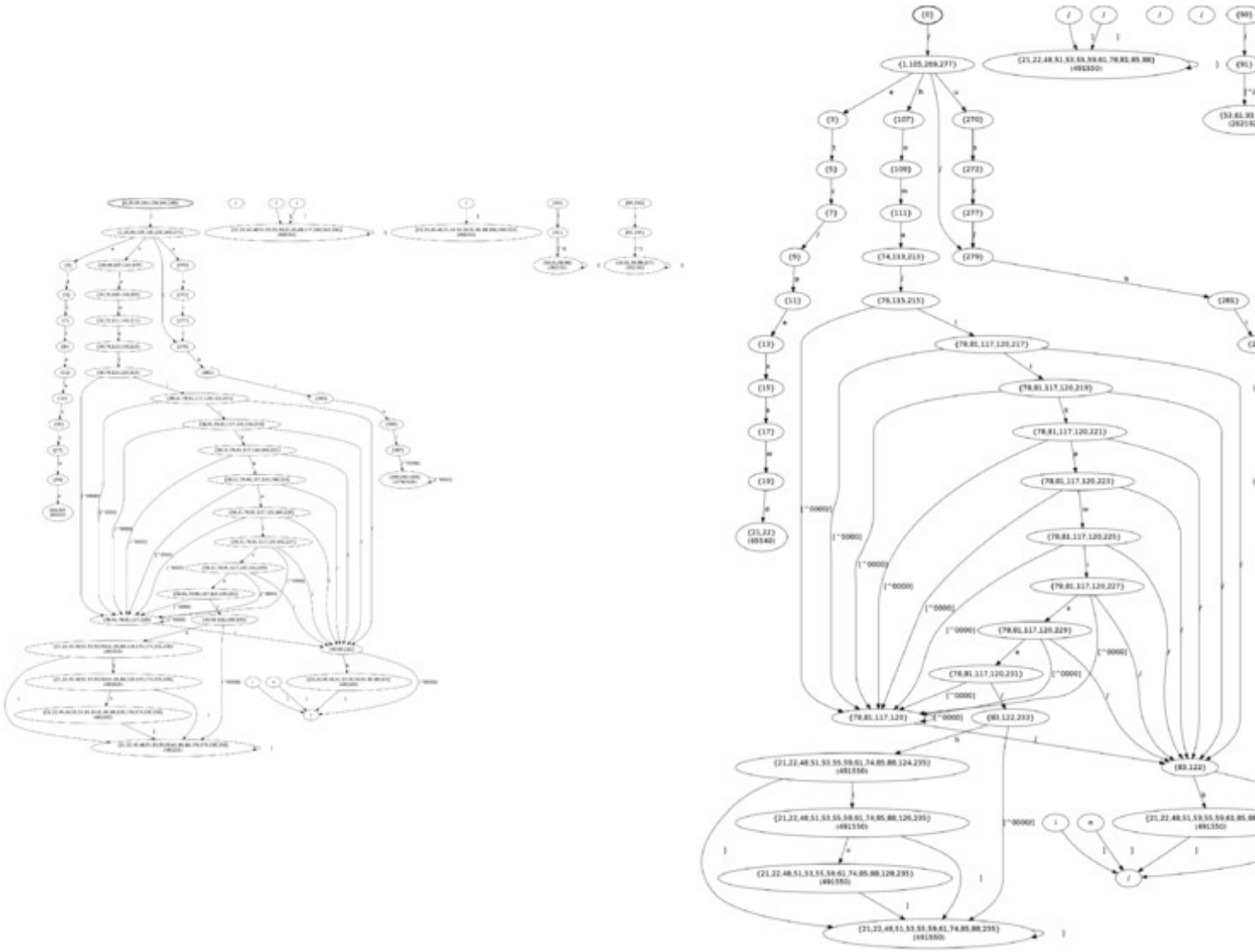
At this stage you can also obtain a visual graph of the generated dfa. This is done by dumping the dfa in graphviz format and using graphviz to generate a diagram. Warning it is easy to create graphs that are to big to be viewed.

To generate a graph do

```
 apparmor_parser -O no-minimize -O no-remove-unreachable -D dfa-graph -QT
<profile_name> 2>profile.gv
 dot -Tpng -o profile.png profile.gv
```

Comparison of unsimplified and simplified dfa

no tree simplification **tree simplification**



dfa minimization (AppArmor 2.5 and later)

AppArmor uses Hopcroft's dfa minimization algorithm to reduce the number of states in the dfa to the smallest dfa possible. This step was added to help reduce kernel memory consumption by the dfa and compilation times.

insert dfa state example pic

hashing partition setup

Traditionally minimization starts with 2 partitions (1 accepting, 1 non-accepting) and the partitions are iteratively split until all states in a partition are indistinguishable. A single representative state is then chosen from each partition, removing the other states and a minimized dfa is created. Instead of splitting the dfa states into 2 partitions, and repartitioning, a linear pass is made through the states setting up an initial set of partitions using hashing to separate states that can never be equivalent.

transition based hashing

By default transition based hashing is used.

The hashing algorithm takes advantage of the fact that states are stored in a compressed form with a default transition and then a set of N non-default transitions.

The rules for the hash are

- •states can not be used as part of the hash as different states may be reduced to the same Partition. Only transitions and known unique partitions can be considered
- •states will only be equivalent if they have the same set of non-default transitions.
  - •number of none default transitions are the same
  - •the set of characters used for non-default transitions are the same

permission based hashing

The permissions of the accept state can be used as part of the initial partition setup (it is defaulted on in AppArmor 2.4 - 2.6). This can lead to even more initial partitions (speeding up minimization) but it can result in a non-minimum dfa as some states that would have been merged end up in separate partitions, which will increase the final size and can slow down total creation time as state minimization has more states to compress.

partition merging

The current algorithm for state comparison within a partition is not the most efficient form, but since the partition setup does a very good job, it is adequate for the number of states currently present in partitions.

dfa unreachable state removal (AppArmor 2.5 and later)

The creation and or minimization of a dfa can result in unreachable states. This phase walks the dfa from the start state finding all reachable states and deletes any that can not be reached.

insert dfa state example pic

Default state differential encoding (compression)

To reduce the number of transitions a state has to encode states can be made to encode their transitions as a differential to a "default" state. If a transition is not represented in the current state the default state is entered and the process is repeated until the transition is found or the state is not marked to be encoded relative to it default.

This type of state encoding can be quite effective in reducing the number of states but it can increase both encoding and matching time (as multiple states must be traversed for a single match). However if the states to encode against are chosen carefully, then both the encoding time and matching time can be bounded, and differential encoding can even result in a faster HFA as it can reduce the memory bandwidth required by the HFA.

The requirements AppArmor uses to choose the states to encode against are

- •The state must have been previously matched while walking the dfa (it will be hot in the cache then)
- •or the state must be at the same level in a DAG decomposition of the dfa, sharing a common ancestor (more on this below)

The first requirement was primarily for performance concerns but in practice works out well for compression too, as states that are close to each other often have similar transitions. The second allows expanding the reach of the compression to a few more likely options while keeping a

potentially common hot path, and without breaking other properties of only referencing previously matched states.

In practice requirement 1 can not be met as each match string takes a different path through the dfa. It can however be approximated by converting the dfa into a directed acyclic graph (DAG) with the start states as the root. The DAG provides a good approximation for requirement 1 and at the same time limits how many states have to be considered for compression (only backwards in the DAG). It also provides guarentees on how many states will be walked at run time (at most 2n).

Converting the HFA into DAG for compression does have a limitation in that it removes many of a states immediate neighbours from consideration. In a DAG a states neighbours can be broken into five classes, immediate predecessor, predecessor on another branch, sibling on another branch, immediate successor, successor on another branch. The immediate predecessor and immediate successor cases are covered by the predecessor differential compression scheme described above (successor as the current state is the successor state predecessor, and thus will be considered when the successor is differentially encoded). However the successor and predecessor on another branch and sibling cases are not covered, and they maybe the more optimal path for encoding, and may be the hot path the match came through.

To help account for this AppArmor also compares to the immediate successors of the state being consider if there are transitions between the states. Sibling states are also considered if there are transitions between the state and the sibling is differentially encoded against a predecessor (not another sibling), or not differentially encoded. This broadens the set of states considered but limits it to states that were potentially matched against and thus in the cache. It also has the property of looking backwards in the DAG thus keeping the maximum number of states that are required to be transitioned to in a match to a linear constant. If only branch predecessor where used then the limit could be kept at 2n but because immediate siblings can be used iff they transition to a predecessor the limit is bounded to a slightly higher value of 5/2n.

When considering which state to differentially encode against AppArmor computes a weighted value and chooses the best one. The value is computed as follows.

- For each defined transition in the state
    - +0 to candidate state weight - if the transition is undefined in the candidate state (the transition must be represented in the current state)
    - +1 to candidate state weight - if the candidate state has the same transition to the same state (the transition can be eliminated from the current state)
    - -1 to candidate state weight - if the transition is defined in the candidate state and it is not the same transition (current state must add a transition entry to override candidate transition)
- For each undefined transition in the state
    - +0 to candidate state weight - if the transition is undefined in the candidate state
    - -1 to candidate state weight - if the transition is defined in candidate state (current state must add an entry to override the candidate transition)

The current state will be differentially encoded against the candidate state with the largest weight > 0. If there is no weighting > 0 then no differential encoding for the state will be done as their is no benefit to doing so.

Note: differential encoding can in one special case reduce a state to 0 stored transitions. This can happen when two states have the exact same transitions but belong in different partitions when minimized. This would happen for example when one state was an accept state and the other a none

accepting state. Other wise if states have the same transitions they are redundant and removed during state minimization.

creation of equivalence classes

AppArmor can use (they are optional) equivalence classes to further compress a states transitions. AppArmor uses a single HFA table to encode equivalence classes at the byte level (ie 256 entries). The equivalence class can be used in two ways to improve compression.

- map multiple characters to a single transition
- remap characters to reduce the distance between characters allowing the better comb compression

The mapping of multiple characters to a single character transition is the traditional use of equivalence classes. Each equivalence class in the HFA consists of the set of characters that have the exact same transitions for every state in the HFA. That is 'a' and 'A' are in the same equivalence class if and only if for every state that 'a' has a defined transition, 'A' also has a defined transition and 'a' and 'A' transition to the same state. This allows 'a' and 'A' to represented by a single transition, and for an equivalence class table to be used to remap 'a' and 'A' to the equivalence class.

Even if there are no equivalence classes to be found within the HFA the equivalence class table can be used to remap characters, so that they can be better compressed by the following comb compression step. In this case the distance between common transitions is reduced (ideally to zero). For example assume that 'a' and 'A' appear as transition with in states frequently but are not equivalence classes, and that no characters between 'a' and 'A' are used in transitions. When the transitions for 'a' and 'A' are encoded in the transition table there is a gap between the two transitions as the transition table is index by character. Comb compression is a technique to try and fill the gap between those characters so that the transition table is fully packed, but this is a hard problem (np complete) and so comb compression often does a less than optimal job. The equivalence class can be used to map these two characters to be adjacent so that there is no gap to fill in the transition table. This can result in higher and faster compression as the comb compressor then has less combinations to consider.

Implementation wise the equivalence class is a 256 byte table that is index by the input character to find the equivalence class that is then used as the index for the state transition. The equivalence class transformation is one way as multiple characters can map to a single equivalence class (which is just a character). While equivalence classes require and extra lookup they can actually increase performances as they reduce the memory bandwidth required to traverse the HFA and the equivalence class table is small and cache hot.

The equivalence class computation is performed after state differential transition encoding as that reduces the number of transitions that must be considered. This in turn can improve the chance of finding equivalence classes, and even if it doesn't can result character transitions being mapped tighter together for comb compression.

dfa table compression

Currently there are is a single compression scheme with some options that can affect compression times, match times and size. The compression format is based on the split tables and comb (interleaved column) compression from the Dragon book. The current compression schemes are designed to be used at dfa runtime such that the the data is kept in an immediately accessible form

using naturally accessible word entries with no bit compression. A consequence is that the tables could be further compressed using a gzip style compression for on disk storage.

The simplest form of the compression is straight comb compression with non-differential entries. Other compression variants modify the base compression by making entries differential to other states as discussed above.

Compression Efficiency

The base compression format of split tables + comb compression works well as long as the average number of transitions per state is low. The use of differential encoding does not change the base compression format but serves to further reduce the average number of transitions that need to represented.

The effectiveness is determined by comparing the compressed format to an uncompressed dfa representation which is a table indexing the rows by state, the columns by input character, and the cells containing the next state and accept information. Ignoring any extended HFA information, this would be

```
 size = (sizeof(next state)) * (# of input characters) * (# of states) +
sizeof(accept index) * (# of states)
      = (2) * 256 * s + 2 * x
      = 512s + 2s
      = 514s
```

Therefore AppArmor's uncompressed encoding (2 bytes for next state, 2 bytes for accept index) takes up 514bytes of memory for each state. The compressed table size is represented by:

```
 size = ((sizeof(base index) + sizeof(default) + sizeof(accept index)) +
((sizeof(next)+sizeof(check)) * (Average transitions per state) * (Packing
factor))) * (# of states)
      = ((4 + 2 + 2) + ((2 + 2) * Ave * Pf)) * s
      = 8s + 4(Ave*Pf)s
```

The packing factor (Pf) expresses the effeciency of the packing done by comb compression. A perfect packing of 1000 transitions would be a table with 1000 entries. When the packing isn't perfect the comb compression leaves unused entries (gaps) in the table so a Pf of 1.1x for a 1000 transitions would result in a table with a size of 1100 entries.

The theoretical maximum compression of the split table + comb compression scheme is 64.25x (514s/8s) and only occurs when there is only a default transition for every state in the DFA. Since this makes the average number of transition per state 0, the second term of the table size equation can be eliminated.

In practice, 64.25x compression is not achievable but good compression is still achieved. For a given profile, the average number of transitions per state ranges from 3 up to approximately 16 (see apparmor_parser -D stats) and the packing factor ranging from 1.1 to 1.4.

Compression efficiency for different transition averages and packing factors:

Ave	1.0x	1.1x	1.2x	1.3x	1.4x	1.5x
1	42.83	41.45	40.16	38.94	37.79	36.71

2	32.13	30.60	29.20	27.93	26.77	25.70
3	25.70	24.25	22.95	21.78	20.73	19.77
4	21.42	20.08	18.90	17.85	16.91	16.06
6	16.06	14.94	13.97	13.11	12.36	11.68
8	12.85	11.90	11.08	10.36	9.73	9.18
12	9.18	8.45	7.84	7.30	6.84	6.43
16	7.14	6.56	6.06	5.64	5.27	4.94
32	3.78	3.45	3.18	2.95	2.75	2.57
64	1.95	1.77	1.63	1.51	1.40	1.31
128	0.99	0.90	0.83	0.76	0.71	0.66
240	0.53	0.48	0.44	0.41	0.38	0.35
250	0.51	0.46	0.43	0.39	0.37	0.34
256	0.50	0.45	0.42	0.38	0.36	0.33

When differential state compression is used, the average number of transitions stays closer to the low end (2 and 3) and the packing factor approaches 1. This is because as the number of transitions approach the upper and lower bounds (1 and 256), the vector are denser and tend to pack better.

Effect various options have on the size of the dfa

add no tree-expr, no min

old style accept

An example of the effectiveness of the compression scheme is the evince profile, which compiles into 3 profiles with DFAs.

Evince DFA size with different DFA options

Profile 1	options	DFA # of states	no compression	2 table (no comb)	2 table + comb	2 table + comb + differential	
1	no minimize, no sharing	11977	5.87 MB		563KB ave=7.54, Pf=1.33		
2	no minimize, no sharing	8172	4.01 MB		306KB ave=6.22, Pf=1.22		
3	no	5954	2.92 MB		230KB		

	minimize, no sharing				ave=6.40, Pf=1.23		
1	minimize, no sharing	8852	4.34 MB		425KB ave=7.74, Pf=1.33		
2	minimize, no sharing	6016	2.99 MB		237KB ave=6.61, Pf=1.22		
3	minimize, no sharing	4086	2.95 MB		168KB ave=6.93, Pf=1.23		
1,2,3	minimize + sharing	???			Ave=, Pf=		

The Basic format

The HFA is split into multiple tables, for a basic DFA (subset of the EHFA) there are 5 table types used, default, base, next, check, and accept. The default, base, and accept tables store the basic information about a state and are indexed by the state number. While the next and check tables store state transitions are index by the base value (obtained from base table) + input character.

The accept table provides the information to determine the permission associated with that state, whether directly stored in the table or an index into another structure.

The default table stores the default transition that should be taken for the state, if the check entry does not match.

The base table stores the start position value into the next check tables for the state.

The next check, tables store the non-default transitions for the dfa. Each state begins in the table with a location determined by its base value and then it is index from that position by the character (byte) being checked. The next table stores the state to transition to for the character being checked, and the check table stores whether the entry being checked belongs to the current state.

Each state with in the table is treated as a sparse matrix column only storing non default entries. The states are then interleaved so that one states entries are mapped into the holes of another state.

Comb compression affects the base value which determines the position of the states transitions in the next, check tables. State differential encoding affects the default state and number of transition that need to be stored for the state.

performance optimization, grouping tables together so they are the same cache line.

Basic default entry compression

Uncompressed dfa state table

State	**\ 000**	**\ 001**	**\ 002**	**..**	**\ 254**	**\ 255**
0	0	0	0	..	0	0

(dead)						
1 (start)				..		
2				..		
53				..		
54				..		

Compressed Dfa

State	Accept	Default	Base
0 (dead)	0	0	0
1 (start)			
2			
3			
4			

Base	Next	Check
0		
1		
2		
3		
4		

Recursive default entry compression

Computing State Permissions from Rules

accept nodes

- unique per permission

deny

x permission

- intersection test - dominance