The AppArmor

Extended Hybrid Finite Automata

Table of Contents

Introduction

This documentation covers the implementation of the eHFA used in AppArmor, for a detailed discussion of the underlying algorithms and theory please refer to

AppArmor uses a byte based extended hybrid finite automata (eHFA or just HFA) for pattern matching. The eHFA is generic and independent from the rest of AppArmor so it can be used by other projects.

This document describes the details of eHFA, and is split into five sections. Section 1 covers the construction of the HFA, detailing the various stages, algorithms, and options. Section 2 covers the compression of the HFA, Section 3 covers using the uncompressed HFA, section 4 covers the using the compressed HFA and the final section provides a quick summary of how the various features of the HFA affect its creation, size and matching time.

What is an eHFA?

The term eHFA may not be standardized, with in this document is used to refer to a finite automata that provides both DFA and NFA characteristics with extensions providing features (variables, and backreferences) that are not possible to support in a pure DFA, or NFA based on regular languages.

Why an eHFA?

The eHFA is used because it is the best choice to provide advanced pattern matching. A pure DFA provides fast linear matching, with a fixed runtime memory bounds but can grow exponentially large. While an NFA provides for a small pattern matching engine that may not provide a linear match and may not have fixed runtime memory bounds.

The AppArmor HFA combines DFAs and NFAs in such a way as to use DFAs for pattern matching with NFA nodes that act as choke points controlling the expansion of the DFA size. The end result is a FA that has known bounds, and performance like a DFA, and smaller sizes closer to that of an NFA.

The AppArmor HFA is biased towards a DFA and can and does produce pure dfas for matching. It will never produce a pure NFA.

The extended part of the name means the HFA has been augmented with scratch memory and extra notations that provide abilities that are either handled poorly or can not be handled by pure NFAs and DFAs (such as variables, back references, counting constraints, and recognition of limited Dyck languages).

What is the eHFAs performance?

The exact performance characteristics of the eHFA are hard to quantify, but in general it tends to be close to that of a DFA for any given pattern. The amount of deviation will depend on how many simultaneous NFA alternations need to be traversed for a match and how many of the extended features are used.

It is even possible that the eHFA will be faster than the equivalent DFA because it is smaller and thus interacts with a cpus cache better.

Constructing the DFA

Accept State Qualifiers

The AppArmor HFA allows for multiple distinct accept states. The HFA engines handles accept states in a generic way, so that the meaning is determined by external routines. The HFA does this by requiring the set up of accept qualifiers that are the referenced referenced by the expression tree, hfa creation and compression routines.

Accept qualifiers ecode the meaning accept states, and have the information to resolve conflicts.

Setting up qualifiers
Functions to handle conflicts

- dominance
- OR

The meaning and encoding of the accept states is no

Rule Sets

Rule sets are a convenience wrapper that provides groupings of rules that will be combined into a single HFA. If rule sets are not convenient for a particular use the should be skipped and the expr tree and dfa routines should be used directly.

The rule set stores the common accept permission information and multiple expr trees that are to be combined together into a single dfa. This allow a given set of rules to be processed into one expression tree and then another set of rules into another expression tree.

Each expression tree with in the rule set is then converted to an intermediate dfa, minimized and then combined using set operations to create the a single large dfa which then in turn is minimized and has unreachable states removed.

????? what more do we want to say about these??? Do we actually store each expr tree and dfa or just the last one and the current set, and combine them together when a new set is opened.

Hrmm rulesets should be pulled out of the lib

Expression (expr) Trees

The AppArmor eHFA uses a modified Aho?/???*ref* algorithm to directly convert from a parsed expression tree to an HFA, there are no intermediate steps of creating an NFA and then doing the subset construction on it.

Parsing
The AppArmor eHFA library provides front end parsers for pcre style regular expressions and the apparmor globbing syntax. Both front end parser build up an expression tree from the regular expression string.

The returned expression tree can then be further manipulated and turned into

- parsing regular expressions into an expr tree
- Permission Tables
  - split each unique permission
  - encode rule/profile (include) on each permission
- Rule sets
  - combining rules to chain multi-stage accept (variables, link rules)
  -
- expr tree optimization
  - dag
  - character class merging
  - factoring, and duplicate pattern elimination

- dfa construction
  - permissions
  - reducing computation time (hash to reduce set comparison)
  - minimizing memory use
- Calculating Rule Dominance (after minimization) – external permission and dominance handling????
  - rule order dominance – add unique accept perm per rule, lowest accept number wins
  - matching dominance – add unique accept perm per rule (that needs it). If accept perm is in all sets +1 of conflicting perm then it dominates it, if in exactly same sets then they are equivalent, if in all same sets but less it is dominated, if partial overlap then overlap conflict.
- minimizing the dfa
  - hashing (setup of initial partitions)
- set operations, combining dfas
  - two kinds merging (partial compile), combining
- unreachable state removal

Compressing the HFA

- Relative State compression
  - maximal spanning tree
  -
- Packing compression (NOT implemented ever) – alternative to com compression, that packs all transitions together in a tight list of some form. In general it can get slightly better compression than comb compression for a significant impact on match performance
  - Creation time: increases dfa creation time by O(n)
  - hfa size: greatly reduces size of the dfa, except in the case where the vast majority of the states are dense with transitions.
  - Match time: significantly increased
-
- Comb compression
  - Finding holes efficiently
    - bit search
    - Searching the entire
    - Sub region searching (don't search entire set of dfa)
      - n partitions
      - divide and conquer
      - selection criteria
        - fixed
        - random
        - density
- Default field
  - Inverting Default and [^]
- State Relative
  - maximal spanning tree
- check field vs bitmap
- Non matching state
-
- Start States
-
- Equivalence Classes
-
- Non deterministic states
-
- Variables
- Counting constraints
-
- Alphabet Reduction
- why alphabet reduction? Better compression, stride doubling

–
–
– Stride Doubling
– allows processing two states at once
–
–
– Mapping Accept states to permissions
– Accept states can be used to directly encode permissions
– v2 mappings
– v3 mappings
–

Using the uncompressed HFA

It is possible to use the uncompressed HFA to do matching, and this is recommended for cases where the a generated HFA needs to be available as soon as possible or if it is not going to be used for very many matches, as the process of compressing the dfa can take longer than creating the dfa.

???? how to use HFA in this case.

Using the compressed HFA

This section covers the formats of the exported compressed hfa and its use.

The compressed hfa for historical reasons is based around the flex dfa container. The container has a fixed sized header with the tables of the compressed hfa stored separate from each other linearly with in the file. The headers an the tables have alignment constraints making it possible to use the containers data directly.

This format is flexible allowing the hfa to be expanded with new tables or to replace old tables as required. If a variable number of tables are expected a verification or unpack routine should be used to ensure that the required data is present.

Flags – conditional accept perm (table selector, or meaning of accept perms is completely external to the hfa)? Export flags?

Header

- Alignment
- Tables
- extensible

table sizes – 16 and 32 bit dfas, etc.

any state can be used as a start state
matches can stop and start at any point
0 is the non-accepting state
1 is the default start state.

Mapping of other states to start states?

Matching

The AppArmor eHFA comes with a reference matching engine that has been tailored towards AppArmor's matching needs. It requires an unpack phase to verify and convert the compressed dfa into a memory resident version, with tables entries logically grouped so that they can be cache aligned.

This increases an initial start time for use and reduces the flexibility of the dfa as the set of tables needed is fixed. However it provides for a dfa that never has a bad reference and is as fast as can be for multiple matches.

If a generated dfa was

- mapping tables to structs (faster access cache align)

Overview of how various features affect dfa creation

This is a quick summary of the affects of various features on the eHFA, the details for each can be found below.

Rule sets:

- allows for partial hfa compiles so that precomputed parts can be reused
- creation time: reduces computation when rule set can be reused.
- hfa size: none, unless dfa minimization is turned off
- required by: special inverted match rules {^* }, {^**}

Expr tree normalization

- creation time: slows it down by doing linear walk and rotation (if necessary) of all tree nodes
- hfa size: dependent on expr tree simpilfication
- match time: no affect
- required by: expr tree simplification. Separate phase in v2.3-v2.5, integrated into alternation merging part of expr tree simplification.

Expr tree simplification

- creation time: generally reduces creation time by allowing creation of a non-minimized dfa that is closer to the minimal. If the dfa created is smaller it also will reduce the amount of memory needed to build the dfa. If the dfa created is small and the not very different than the dfa created by not simplifying the tree then it can slow down creation time.
- hfa size: none unless dfa minimization is turned off, in which case it will result in a dfa less than or equal to the non-simplied expr tree.
- match time: no affect
- required by: no affect

DFA minimization

- creation time: adds an extra O(n log n) phase to creation. If it removes states it can speed up dfa creation by reducing the number of states comb compression (O n^2) has to evaluate.
- hfa size: reduces the size of the dfa by removing redundant states.
- match time: none

DFA minimization – transition hashing

- creation time: reduces minimization time by doing an initial O(n) pass to provide a better initial set of partitions.
- hfa size: none
- match time: none

DFA minimization – permissions hashing

- creation time: reduces minimization time by creating a larger set of initial partitions
- hfa size: can increase it as it can prevent normalization from eliminating all redundant states.
- match time: none

DFA merge – combine multiple dfas into a single dfa with shared start state (set operations)

- Creation time: extra merge phase during compile. Can reduce compile time by allowing for precomputed components so compilation can be incremental and a give component is never recomputed. For best results a merge must be done after merge, adding another phase to the compile.
- hfa size: depending on the set operation performed the resultant dfa may be smaller or larger.
- Match time: no affect

DFA combining – combine two dfas together into one but keeps the start states separate.

- Creation time: increases computation time because a combining pass is needed, along with a minimization pass to see any potential dfa size reductions. Can reduce or increase compression time depending on whether the combined dfa reduces the number of states enough to be faster than compressing the dfas separately, otherwise it will increase compression time.
- hfa size: reduced, at a minim 1 less dfa header, and shared non-accepting state. Smaller dfas by the number of redundant states removed by minimization after combination.
- Match time: no affect

Multiple start states

- creation time: only as a result of the combining phase that creates the multiple start states.
- hfa size: no change
- match time: no affect

Unreachable state removal

- creation time: extra O(n) pass to remove unreachable states
- hfa size: reduced if unreachable states found
- match time: none

DFA compression

- equivalence class:
  - creation time: increases time to compute equivalence classes, potentially reduce creation time by speeding up matching in dfa comb compression
  - dfa size: reduces dfa size by improving comb compression
  - match time: requires extra table lookup per character matched
- default entry:
  - creation time: time to compute default entry
  - hfa size: reduces size, if combined with another compression (comb, packing) to remove /reduce space used by sparse transition table. Can increase dfa size if all states transitions are completely distinct so no transition can be advantageously set as the default.
  - match time: time to fail initial test and fall through to default entry.
- Inverted default entry:
  - creation time: time to determine if the majority of state transitions point to a single state (often free as inverted char class entries are computed as if this is the case).
  - dfa size: reduces size by increasing compression
  - match time: time to fall through to default case on first match failure
- state relative default entry (early dfa compression – not implemented yet):
  - creation time: adds an extra compilation phase (O n log n?), however this can reduce comb compression time as there are fewer states to insert.
  - dfa size: reduces size by increasing compression
  - match time: can increase matching time by a variable amount

dependent on algorithm used to find states to build from. The maximal spanning tree to previous nodes by depth ensures that the maxim increase in state traversal is 2x.

- Transitions bitmask (not implemented – maybe never)
  - creation time: minimal increment over creating check table
  - dfa size: can reduce size of dfa when the average dfa state has multiple transitions by eliminating check table, and allowing dfas to share next entries directly (check entry sharing is also handled by state relative default entry). Can increase the size of the dfa if the average state has few entries as the overhead of the bitmask is greater than the check table.
- State relative next/check tables (not currently implemented)
  - creation time: can affect states that are compared against
  - dfa size: can decrease dfa size by allowing states that have the same transition pattern (to different states) to be shared. Can increase dfa size by reducing sharing of state transition patterns when states transition to the same state. State relative next/check is best used with state relative default entries and or Transition bitmasks
  - match time: none, base next transition from current location instead of from 0
- comb compression
  - creation time: increases dfa creation time by O(n^2)
  - hfa size: greatly reduces size of the dfa, except in the case where the vast majority of states are dense with transitions.
  - match time: small increase to test if check == current state.

NFA states (not implemented yet)

- creation time: alters creation of dfa, depending on how its done could result in faster creation as dfa state explosion is kept in check. Could also be slower if creation drops previously computed results to remove state explosion)
- dfa size: reduces state explosion
- match time: requires traversing multiple paths
- required by variables and references

Variables and backreferences (not implemented yet)

- creation time: ???
- dfa size: ???

- match time: increases book keeping and potentially nfa state travesals
- requires: NFA states

Counting constraints (not implemented yet)

- creation time: ???
- dfa size: reduces size of dfa that uses counting constraints
- match time: increases book keeping, increases locality using same states for each count iteration

Alphabet reduction – uber equivalence classes (not implemented yet)

- creation time: increased to compute classes and then alphabet tables per class. Can speed up compression as there are smaller vectors to compare
- dfa size: same as equivalence classes but requires more tables
- match time: requires an extra table lookup per character
- required by: stride doubling

Stride doubling (not implemented yet)

- creation time: increased due to having to compute double transition vectors and alphabet reductions
- hfa size: ???
- match time: faster as two states can be traversed at a time

References

Dragon book

blah blah blah