Hrmmm, split out apparmor specific stuff, except where used as an example. Another document

The AppArmor Extended Hybrid Finite Automata

AppArmor uses a byte based extended hybrid finite automata (eHFA or just HFA) for pattern matching. ??? TODO ???

This document describes the details of eHFA, and is split into two sections. Section 1 covers the construction of the HFA, detailing the various stages, algorithms, and options. Section 2 covers the compressed form of the HFA that is generally used for storage and run time pattern matching.

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?

AppArmor uses and eHFA because it is the best choice to provide advanced pattern matching. A pure DFA provides fast linear matching, with tight known 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 a bounded memory.

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 HFA augments the ability for NFAs and DFAs by providing a scratch memory that provides counting constraints, variables, and back references.

Counting constraints can be provided by DFAs, and NFAs however they are handled very inefficiently and are one of the causes of DFA size explosion.

Variables and back references provide an extension of what regular languages

can express. Variables and back references share an implementation, where variables are just a special case of back references.

What is the eHFAs performace?

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.

Compiler options (move to a different document)

Overview of how various stages and features affect dfa creation

- Duplicate include elimination: allows detecting which files have been included already and stopping them from being reincluded.
  - Creation time: reduces precompile time by reducing the number of rules that need to be evaluated
  - hfa size: none, unless duplicate rule elimination, expr tree simplification and minimization are disabled
  - match time: none
- Duplicate rule elimination & permission merge:
  - creation time: generally reduces creation time by eliminating duplicate rules, resulting in less work in tree simplification, and smaller non-minimized dfas (expr tree simplication doesn't get everything).
  - hfa size: none, unless expr tree simplification and dfa minimization are dissabled
  - match time: none
  - required by: none
- 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

    - 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: none
    - required by: none
- 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
    - 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
    - permission 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: none
- DFA combining – This form of combining merges 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 afer combination.
- 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 change
- 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):
  - 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)
  - 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.
- 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
- nfa states affect:
    - 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
- variable and backreference matching
    - creation time: ???
    - dfa size: ???
    - match time: increases book keeping and potentially nfa state travesals
- counting constraints
    - 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
    - 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
    - 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

Constructing the DFA

- rule set
- 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
- Compression
  - Relative State compression
    - maximal spanning tree
    -
  - 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
-

Compressed DFA

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

Header
- Alignment
- Tables
- extensible

Runtime

- mapping tables to structs (faster access cache align)
-