Selecting algorithm implementations by properties

Properties are associated with algorithms and are used to select between different implementations dynamically.

This implementation is based on a number of assumptions:

- Property definition is uncommon. I.e. providers will be loaded and unloaded relatively infrequently, if at all.
- The number of distinct property names will be small.
- Providers will often give the same implementation properties to most or all of their implemented algorithms. E.g. the FIPS property would be set across an entire provider. Likewise for, hardware, accelerated, software, HSM and, perhaps, constant_time.
- There are a lot of algorithm implementations, therefore property definitions should be space efficient. However...
- ... property queries are very common. These must be fast.
- Property queries come from a small set and are reused many times typically. I.e. an application tends to use the same set of queries over and over, rather than spanning a wide variety of queries.
- Property queries can never add new property definitions.

Some consequences of these assumptions are:

- That definition is uncommon and queries are very common, we can treat the property definitions as almost immutable. Specifically, a query can never change the state of the definitions.
- That definition is uncommon and needs to be space efficient, it will be feasible to use a hash table to contain the names (and possibly also values) of all properties and to reference these instead of duplicating strings. Moreover, such a data structure need not be garbage collected. By converting strings to integers using a structure such as this, string comparison degenerates to integer comparison. Additionally, lists of properties can be sorted by the string index which makes comparisons linear time rather than quadratic time - the O(n log n) sort cost being amortised.
- A cache for property definitions is also viable, if only implementation properties are used and not algorithm properties, or at least these are maintained separately. This cache would be a hash table, indexed by the property definition string, and algorithms with the same properties would share their definition structure. Again, reducing space use.
- A query cache is desirable. This would be a hash table keyed by the algorithm identifier and the entire query string and it would map to the

chosen algorithm. When a provider is loaded or unloaded, this cache must be invalidated. The cache will also be invalidated when the global properties are changed as doing so removes the need to index on both the global and requested property strings.

The implementation:

- property_lock.c contains some wrapper functions to handle the global lock more easily. The global lock is held for short periods of time with per algorithm locking being used for longer intervals.
- property_string.c contains the string cache which converts property names and values to small integer indices. Names and values are stored in separate hash tables. The two Boolean values, the strings "yes" and "no", are populated as the first two members of the value table. All property names reserved by OpenSSL are also populated here. No functions are provided to convert from an index back to the original string (this can be done by maintaining parallel stacks of strings if required).
- property_parse.c contains the property definition and query parsers. These convert ASCII strings into lists of properties. The resulting lists are sorted by the name index. Some additional utility functions for dealing with property lists are also included: comparison of a query against a definition and merging two queries into a single larger query.
- property.c contains the main APIs for defining and using properties. Algorithms are discovered from their NID and a query string. The results are cached.

  The caching of query results has to be efficient but it must also be robust against a denial of service attack. The cache cannot be permitted to grow without bounds and must garbage collect under-used entries. The garbage collection does not have to be exact.
- defn_cache.c contains a cache that maps property definition strings to parsed properties. It is used by property.c to improve performance when the same definition appears multiple times.