User-defined hash methods often work by combining the hash code of several values into one. This requires some logic to combine the values. In our experience, users are making a variety of choices for the algorithm for this, and in many cases are picking an option which may not be the best for security and performance in multiple ways. It's also a shame that users are having to think about how to do this basic operation themselves.

For example, this hash method creates a single hash code by combining the hash value of three values that make up the object. The user has combined the values using the xor operator and multiplication by prime numbers to distribute bits. This is an ok way to combine multiple values into a hash code.

```ruby
def hash
  x.hash ^ (y.hash * 3) ^ (z.hash * 5)
end
```

But users have to know to do this, and they sometimes get it wrong, writing things like this.

```ruby
def hash
  x.hash ^ y.hash ^ z.hash
end
```

This isn't distributing the bits very well. A bad hash code may harm performance if it cause more collisions in a hash table. Collisions may also cause excess cache eviction, which would further harm performance. If performance is reduced in this way there's a potential security risk due to denial-of-service. (We don't think this is an immediate practical security problem, which is why we're discussing in the open issue tracker, not the security mailing list.)

The `x.hash ^ (y.hash * 3) ^ (z.hash * 5)` pattern is still not ideal, as users have to manually write it, and it's a lot of logic to execute in the Ruby interpreter, when it could be possibly be done in native code instead. A better pattern we think is this:

```ruby
def hash
  [x, y, z].hash
end
```

This leaves the logic of creating a suitable and safe hash value to `...`.hash, which does it correctly.

Why doesn't everyone already use this pattern? Because it's not documented as the right thing to do. We want to present a couple of options for what could be done to encourage people to use this pattern or an equivalent, to help people write more concise and clear code that is also more performant and secure.

**Document `...`.hash as best practice and optimise it**

If we want people to use `...`.hash, we should say that in the documentation for Kernel#hash as the best practice. Wording along the lines of

```
If you're implementing a hash code for a compound set of values, best practice is to combine them with `...`.hash. For example....
```

This way people reading the documentation on Kernel#hash get pointed in the clear, concise, performant, secure direction.

We can combine this recommendation with an optimisation to `...`.hash to remove the array allocation in implementation of Ruby without escape analysis and scalar replacement, similar to what is done for Array#min and #max. This way the best practice is even faster.
Introduce a new similar method, but specifically for the purpose so it is discoverable

A second option is to introduce a new method, specifically for this task, hash_objects(...). This is inspired by Java's Objects.hash(...). The reason for the new method is that it should make it more discoverable - if you go looking for a tool to combine hash values you'd find one. We'd still link to it from Kernel#hash. This method would not require the array allocation removal optimisation, as it's just a simple call.

Examples of hash methods

Even the MRI codebase has some suboptimal hash methods we don't need to look very far for examples. For example lib/resolv.rb, these two hash methods don't distribute the bits they combine

- [https://github.com/ruby/ruby/blob/c445963575a8572f6b0baf7135093c128adab3b9/lib/resolv.rb#L1734](https://github.com/ruby/ruby/blob/c445963575a8572f6b0baf7135093c128adab3b9/lib/resolv.rb#L1734)
- [https://github.com/ruby/ruby/blob/c445963575a8572f6b0baf7135093c128adab3b9/lib/resolv.rb#L1307](https://github.com/ruby/ruby/blob/c445963575a8572f6b0baf7135093c128adab3b9/lib/resolv.rb#L1307)

Both these examples could be replaced with either of our proposals.

A good example of someone already using best practice is this.

- [https://github.com/ruby/ruby/blob/128972189284f4338722e8a910d0b4f6e7a02b31/lib/bundler/source/git.rb#L50](https://github.com/ruby/ruby/blob/128972189284f4338722e8a910d0b4f6e7a02b31/lib/bundler/source/git.rb#L50)

But this would still be faster with the optimisation we proposed, or using hash_objects(...), as that'd remove the array allocation and the hash call.

Other things we've already done

We've proposed a RuboCop cop to try to catch the pattern we think is suboptimal [https://github.com/rubocop/rubocop/pull/10441](https://github.com/rubocop/rubocop/pull/10441).

Co-authored with [sambostock (Sam Bostock)](https://github.com/rubocop/rubocop/pull/10441).
Including the class is often (maybe even always?) unnecessary, and does impact performance. Struct is special because it generates generic classes, so there it makes sense to include the class, but that's about the only case where it does IMHO.

I think [a, b, c].hash is the best by far, could you open a PR to document that?

Adding a new method would prevent adopting it now and just delay solving this problem, I don't think it's a productive way if an existing way already works fine ([a, b, c].hash).

(BTW, hash_objects would still likely allocate an array (internal or Ruby-level) for the var-args, so that's worse than a literal array actually.)