If `opts.integrity` is passed in, it should be an `integrity` value understood by `parse` that the stream will check the data against. If verification succeeds, the integrity stream will emit a `verified` event whose value is a single `Hash` object that is the one that succeeded verification. If verification fails, the stream will error with an `EINTEGRITY` error code.

If `opts.size` is given, it will be matched against the stream size. An error with `err.code` `EBADSIZE` will be emitted by the stream if the expected size and actual size fail to match.

If `opts.pickAlgorithm` is provided, it will be passed two algorithms as arguments. ssri will prioritize whichever of the two algorithms is returned by this function. Note that the function may be called multiple times, and it **must** return one of the two algorithms provided. By default, ssri will make a best-effort to pick the strongest/most reliable of the given algorithms. It may intentionally deprioritize algorithms with known vulnerabilities.

Example

```
const integrity =
        ssri.fromData(fs.readFileSync('index.js'))
fs.createReadStream('index.js')
.pipe(ssri.integrityStream({integrity}))
```


ssri

`ssri`, short for Standard Subresource Integrity, is a Node.js utility for parsing, manipulating, serializing, generating, and verifying Subresource Integrity hashes.

Install

```
$ npm install --save ssri
```

Table of Contents

Example
Features
Contributing
API
Parsing & Serializing
`parse`
`stringify`
`Integrity#concat`
`Integrity#merge`

- Integrity#toString
- Integrity#toJSON
- Integrity#match
- Integrity#pickAlgorithm
- Integrity#hexDigest
  - Integrity Generation
- fromHex
- fromData
- fromStream
- create
  - Integrity Verification
- checkData
- checkStream
- integrityStream

Example

```
const ssri = require('ssri')

const integrity = 'sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEFyht3
        r/pAe1DM+JI/A+line3jUBgzQ7A==?foo'

// Parsing and serializing
const parsed = ssri.parse(integrity)
ssri.stringify(parsed) // === integrity
        (works on non-Integrity objects)
parsed.toString() // === integrity
```



```
//   digest: 'sha512-
        yzd8ELD1piyANiWnmdnpCL5F52f10UfUdEkHywV
// }>

ssri.checkStream(
  fs.createReadStream('index.js'),
  'sha256-
        l981iLWj8kurw4UbNy8Lpxqdzd7UOxS50Glhv8F
) // -> Promise<Hash>

ssri.checkStream(
  fs.createReadStream('index.js'),
  'sha1-BaDDigEST'
) // -> Promise<Error<{code:
        'EINTEGRITY'}>>
```

```
> integrityStream([opts]) ->
IntegrityStream
```

Returns a `Transform` stream that data can be piped through in order to generate and optionally check data integrity for piped data. When the stream completes successfully, it emits `size` and `integrity` events, containing the total number of bytes processed and a calculated `Integrity` instance based on stream data, respectively.

If `opts.algorithms` is passed in, the listed algorithms will be calculated when generating the final `Integrity` instance. The default is `['sha512']`.

If `opts.single` is passed in, a single `Hash` instance will be returned.

```
> ssri.checkStream(stream, sri, [opts])
-> Promise<Hash>
```

Verifies the contents of `stream` against an `sri` argument. `stream` will be consumed in its entirety by this process. `sri` can be any subresource integrity representation that `ssri.parse` can handle.

`checkStream` will return a Promise that either resolves to the `Hash` that succeeded verification, or, if the verification fails or an error happens with `stream`, the Promise will be rejected.

If the Promise is rejected because verification failed, the returned error will have `err.code` as `EINTEGRITY`.

If `opts.size` is given, it will be matched against the stream size. An error with `err.code` `EBADSIZE` will be returned by a rejection if the expected size and actual size fail to match.

If `opts.pickAlgorithm` is provided, it will be used by `Integrity#pickAlgorithm` when deciding which of the available digests to match against.

Example

```
const integrity =
        ssri.fromData(fs.readFileSync('index.js'))

ssri.checkStream(
  fs.createReadStream('index.js'),
  integrity
)
// ->
// Promise<{
//   algorithm: 'sha512',
```



```
// Async stream functions
ssri.checkStream(fs.createReadStream('./
        my-file'), integrity).then(...)
ssri.fromStream(fs.createReadStream('./
        my-file')).then(sri => {
  sri.toString() === integrity
})
fs.createReadStream('./my-
        file').pipe(ssri.createCheckerStream(sr

// Sync data functions
ssri.fromData(fs.readFileSync('./my-
        file')) // === parsed
ssri.checkData(fs.readFileSync('./my-
        file'), integrity) // =>
        'sha512'
```

Features

- Parses and stringifies SRI strings.
- Generates SRI strings from raw data or Streams.
- Strict standard compliance.
- `?foo` metadata option support.
- Multiple entries for the same algorithm.
- Object-based integrity hash manipulation.
- Small footprint: no dependencies, concise implementation.
- Full test coverage.
- Customizable algorithm picker.

Contributing

The ssri team enthusiastically welcomes contributions and project participation! There's a bunch of things you can do if you want to contribute! The Contributor Guide has all the information you need for everything from reporting bugs to contributing entire new features. Please don't hesitate to jump in if you'd like to, or even ask us questions if something isn't clear.

API

```
> ssri.parse(sri, [opts]) -> Integrity
```

Parses `sri` into an `Integrity` data structure. `sri` can be an integrity string, an `Hash`-like with `digest` and `algorithm` fields and an optional `options` field, or an `Integrity`-like object. The resulting object will be an `Integrity` instance that has this shape:

```
{
  'sha1': [{algorithm: 'sha1', digest:
        'deadbeef', options: []}],
  'sha512': [
    {algorithm: 'sha512', digest:
        'c0ffee', options: []},
    {algorithm: 'sha512', digest:
        'bad1dea', options: ['foo']}
  ],
}
```



```
> ssri.checkData(data, sri, [opts]) ->
Hash|false
```

Verifies `data` integrity against an `sri` argument. `data` may be either a `String` or a `Buffer`, and `sri` can be any subresource integrity representation that `ssri.parse` can handle.

If verification succeeds, `checkData` will return the name of the algorithm that was used for verification (a truthy value). Otherwise, it will return `false`.

If `opts.pickAlgorithm` is provided, it will be used by `Integrity#pickAlgorithm` when deciding which of the available digests to match against.

If `opts.error` is true, and verification fails, `checkData` will throw either an `EBADSIZE` or an `EINTEGRITY` error, instead of just returning false.

Example

```
const data = fs.readFileSync('index.js')
ssri.checkData(data,
        ssri.fromData(data)) // ->
        'sha512'
ssri.checkData(data, 'sha256-
        l981iLWj8kurw4UbNy8Lpxqdzd7UOxS50Glhv8F
ssri.checkData(data, 'sha1-
        BaDDigEST') // -> false
ssri.checkData(data, 'sha1-BaDDigEST',
        {error: true}) // -> Error!
        EINTEGRITY
```

Example

```
ssri.fromStream(fs.createReadStream('index.js'),
        {
  algorithms: ['sha1', 'sha512']
}).then(integrity => {
  return
        ssri.checkStream(fs.createReadStream('index
        integrity)
}) // succeeds
```

```
> ssri.create([opts]) -> <Hash>
```

Returns a Hash object with `update(<Buffer or string>[,enc])` and `digest()` methods.

The Hash object provides the same methods as crypto class Hash. `digest()` accepts no arguments and returns an Integrity object calculated by reading data from calls to update.

It accepts both `opts.algorithms` and `opts.options`, which are documented as part of `ssri.fromData`.

If `opts.strict` is true, the integrity object will be created using strict parsing rules. See `ssri.parse`.

Example

```
const integrity =
        ssri.create().update('foobarbaz').digest()
integrity.toString()
// ->
// sha512-
        yzd8ELD1piyANiWnmdnpCL5F52f10UfUdEkHywVZeqT
```


If `opts.single` is truthy, a single `Hash` object will be returned. That is, a single object that looks like `{algorithm, digest, options}`, as opposed to a larger object with multiple of these.

If `opts.strict` is truthy, the resulting object will be filtered such that it strictly follows the Subresource Integrity spec, throwing away any entries with any invalid components. This also means a restricted set of algorithms will be used – the spec limits them to `sha256`, `sha384`, and `sha512`.

Strict mode is recommended if the integrity strings are intended for use in browsers, or in other situations where strict adherence to the spec is needed.

Example

```
ssri.parse('sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEF
        r/pAe1DM+JI/A+line3jUBgzQ7A==?foo') //
        Integrity object
```

```
> ssri.stringify(sri, [opts]) -> String
```

This function is identical to `Integrity#toString()`, except it can be used on *any* object that `parse` can handle – that is, a string, an `Hash`-like, or an `Integrity`-like.

The `opts.sep` option defines the string to use when joining multiple entries together. To be spec-compliant, this *must* be whitespace. The default is a single space (`' '`).

If `opts.strict` is true, the integrity string will be created using strict parsing rules. See `ssri.parse`.

Example

```
// Useful for cleaning up input SRI
        strings:
ssri.stringify('\n\rsha512-
        foo\n\t\tsha384-bar')
// -> 'sha512-foo sha384-bar'

// Hash-like: only a single entry.
ssri.stringify({
  algorithm: 'sha512',
  digest:'9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEFyht3
        r/pAe1DM+JI/A+line3jUBgzQ7A==',
  options: ['foo']
})
// ->
// 'sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEFyht3
        r/pAe1DM+JI/A+line3jUBgzQ7A==?foo'

// Integrity-like: full multi-entry
        syntax. Similar to output of
        `ssri.parse`
ssri.stringify({
  'sha512': [
    {
      algorithm: 'sha512',
      digest:'9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEFyht3
        r/pAe1DM+JI/A+line3jUBgzQ7A==',
      options: ['foo']
    }
  ]
```


Example

```
const integrityObj =
        ssri.fromData('foobarbaz', {
  algorithms: ['sha256', 'sha384',
        'sha512']
})
integrity.toString('\n')
// ->
// sha256-
        l981iLWj8kurw4UbNy8Lpxqdzd7UOxS50Glhv8F
// sha384-
        irnCxQ0CfQhYGlVAUdwTPC9bF3+YWLxlaDGM4xb
// sha512-
        yzd8ELD1piyANiWnmdnpCL5F52f10UfUdEkHywV

> ssri.fromStream(stream, [opts]) ->
Promise<Integrity>
```

Returns a Promise of an Integrity object calculated by reading data from a given `stream`.

It accepts both `opts.algorithms` and `opts.options`, which are documented as part of `ssri.fromData`.

Additionally, `opts.Promise` may be passed in to inject a Promise library of choice. By default, ssri will use Node's built-in Promises.

If `opts.strict` is true, the integrity object will be created using strict parsing rules. See `ssri.parse`.

Example

```
ssri.fromHex('75e69d6de79f',
        'sha1').toString() // 'sha1-
        deadbeef'
```

```
> ssri.fromData(data, [opts]) ->
Integrity
```

Creates an `Integrity` object from either string or `Buffer` data, calculating all the requested hashes and adding any specified options to the object.

`opts.algorithms` determines which algorithms to generate hashes for. All results will be included in a single `Integrity` object. The default value for `opts.algorithms` is `['sha512']`. All algorithm strings must be hashes listed in `crypto.getHashes()` for the host Node.js platform.

`opts.options` may optionally be passed in: it must be an array of option strings that will be added to all generated integrity hashes generated by `fromData`. This is a loosely-specified feature of SRIs, and currently has no specified semantics besides being `?`-separated. Use at your own risk, and probably avoid if your integrity strings are meant to be used with browsers.

If `opts.strict` is true, the integrity object will be created using strict parsing rules. See `ssri.parse`.


```
})
// ->
// 'sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEF
        r/pAe1DM+JI/A+line3jUBgzQ7A==?foo'
```

```
> Integrity#concat(otherIntegrity,
[opts]) -> Integrity
```

Concatenates an `Integrity` object with another IntegrityLike, or an integrity string.

This is functionally equivalent to concatenating the string format of both integrity arguments, and calling `ssri.parse` on the new string.

If `opts.strict` is true, the new `Integrity` will be created using strict parsing rules. See `ssri.parse`.

Example

```
// This will combine the integrity
        checks for two different
        versions of
// your index.js file so you can use a
        single integrity string and
        serve
// either of these to clients, from a
        single `<script>` tag.
const desktopIntegrity =
        ssri.fromData(fs.readFileSync('./
        index.desktop.js'))
```

```
const mobileIntegrity =
        ssri.fromData(fs.readFileSync('./
        index.mobile.js'))

// Note that browsers (and ssri) will
        succeed as long as ONE of the
        entries
// for the *prioritized* algorithm
        succeeds. That is, in order for
        this fallback
// to work, both desktop and mobile
        *must* use the same `algorithm`
        values.
desktopIntegrity.concat(mobileIntegrity)
```

```
> Integrity#merge(otherIntegrity,
[opts])
```

Safely merges another IntegrityLike or integrity string into an `Integrity` object.

If the other integrity value has any algorithms in common with the current object, then the hash digests must match, or an error is thrown.

Any new hashes will be added to the current object's set.

This is useful when an integrity value may be upgraded with a stronger algorithm, you wish to prevent accidentally suppressing integrity errors by overwriting the expected integrity value.

Example

```
const data = fs.readFileSync('data.txt')
```


Example

```
ssri.parse('sha1-
        deadbeef').hexDigest() //
        '75e69d6de79f'
```

```
> ssri.fromHex(hexDigest, algorithm,
[opts]) -> Integrity
```

Creates an `Integrity` object with a single entry, based on a hex-formatted hash. This is a utility function to help convert existing shasums to the Integrity format, and is roughly equivalent to something like:

```
algorithm + '-' +
        Buffer.from(hexDigest,
        'hex').toString('base64')
```

`opts.options` may optionally be passed in: it must be an array of option strings that will be added to all generated integrity hashes generated by `fromData`. This is a loosely-specified feature of SRIs, and currently has no specified semantics besides being `?`-separated. Use at your own risk, and probably avoid if your integrity strings are meant to be used with browsers.

If `opts.strict` is true, the integrity object will be created using strict parsing rules. See `ssri.parse`.

If `opts.single` is true, a single `Hash` object will be returned.

```
ssri.parse(integrity).match('sha1-
        deadbeef')
// false
```

```
> Integrity#pickAlgorithm([opts]) ->
String
```

Returns the “best” algorithm from those available in the integrity object.

If `opts.pickAlgorithm` is provided, it will be passed two algorithms as arguments. ssri will prioritize whichever of the two algorithms is returned by this function. Note that the function may be called multiple times, and it **must** return one of the two algorithms provided. By default, ssri will make a best-effort to pick the strongest/most reliable of the given algorithms. It may intentionally deprioritize algorithms with known vulnerabilities.

Example

```
ssri.parse('sha1-WEakDigEST sha512-
        yzd8ELD1piyANiWnmdnpCL5F52f10UfUdEkHywVZeqT
        sha512
```

```
> Integrity#hexDigest() -> String
```

`Integrity` is assumed to be either a single-hash `Integrity` instance, or a `Hash` instance. Returns its `digest`, converted to a hex representation of the base64 data.


```
// integrity.txt contains 'sha1-
        X1UT+IIv2+UUWvM7ZNjZcNz5XG4='
// because we were young, and didn't
        realize sha1 would not last
const expectedIntegrity =
        ssri.parse(fs.readFileSync('integrity.t
        'utf8'))
const match = ssri.checkData(data,
        expectedIntegrity, {
  algorithms: ['sha512', 'sha1']
})
if (!match) {
  throw new Error('data corrupted or
        something!')
}

// get a stronger algo!
if (match && match.algorithm !==
        'sha512') {
  const updatedIntegrity =
        ssri.fromData(data, {
        algorithms: ['sha512'] })
  expectedIntegrity.merge(updatedIntegrity)
  fs.writeFileSync('integrity.txt',
        expectedIntegrity.toString())
  // file now contains
  // 'sha1-X1UT+IIv2+UUWvM7ZNjZcNz5XG4= sha512-
        yzd8ELD1piyANiWnmdnpCL5F52f10UfUdEkHywV
}
```

```
> Integrity#toString([opts]) -> String
```

Returns the string representation of an `Integrity` object. All hash entries will be concatenated in the string by `opts.sep`, which defaults to `' '`.

If you want to serialize an object that didn't come from an `ssri` function, use `ssri.stringify()`.

If `opts.strict` is true, the integrity string will be created using strict parsing rules. See `ssri.parse`.

Example

```
const integrity = 'sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEFyht3
        r/pAe1DM+JI/A+line3jUBgzQ7A==?foo'

ssri.parse(integrity).toString() ===
        integrity
```

```
> Integrity#toJSON() -> String
```

Returns the string representation of an `Integrity` object. All hash entries will be concatenated in the string by `' '`.

This is a convenience method so you can pass an `Integrity` object directly to `JSON.stringify`. For more info check out toJSON() behavior on mdn.

Example

```
const integrity = '"sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEF
        r/pAe1DM+JI/A+line3jUBgzQ7A==?foo"'

JSON.stringify(ssri.parse(integrity))
        === integrity
```

```
> Integrity#match(sri, [opts]) -> Hash |
false
```

Returns the matching (truthy) hash if `Integrity` matches the argument passed as `sri`, which can be anything that `parse` will accept. `opts` will be passed through to `parse` and `pickAlgorithm()`.

Example

```
const integrity = 'sha512-9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEF
        r/pAe1DM+JI/A+line3jUBgzQ7A=='

ssri.parse(integrity).match(integrity)
// Hash {
//   digest: '9KhgCRIx/
        AmzC8xqYJTZRrnO8OW2Pxyl2DIMZSBOr0oDvtEF
        r/pAe1DM+JI/A+line3jUBgzQ7A=='
//   algorithm: 'sha512'
// }
```