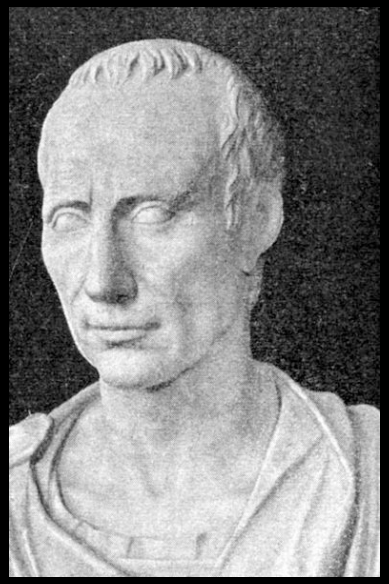
Defending Against the *Sneakers* Scenario

CRYPTOGRAPHIC AGILITY

Bryan Sullivan, Security Program Manager, Microsoft SDL

Crypto systems get broken



be sure to drink your ovaltine

Why assume that current algorithms really are unbreakable, unlike every other time in the history of cryptography?

Consequences

- Change code
- Rebuild
- Retest
- Deploy patches to *n* users

- Pretty big window of attack…

Other concerns

- Export controls
- International regulations
- FIPS-140

Solution

- Plan for this from the beginning
- Assume the crypto algorithms you use will be defeated in your application's lifetime
- Code your apps in a cryptographically agile manner
  - Or code-review apps for crypto agility if you're of the pentester persuasion and not a dev

Steps toward crypto agility

- Step 1: Avoid hardcoded algorithms

Abstraction

- Want one of these?
- Are you sure?

Three Cryptographically Agile Frameworks*

.NET

JCA

Java Cryptography Architecture

CNG

Cryptography API Next Generation

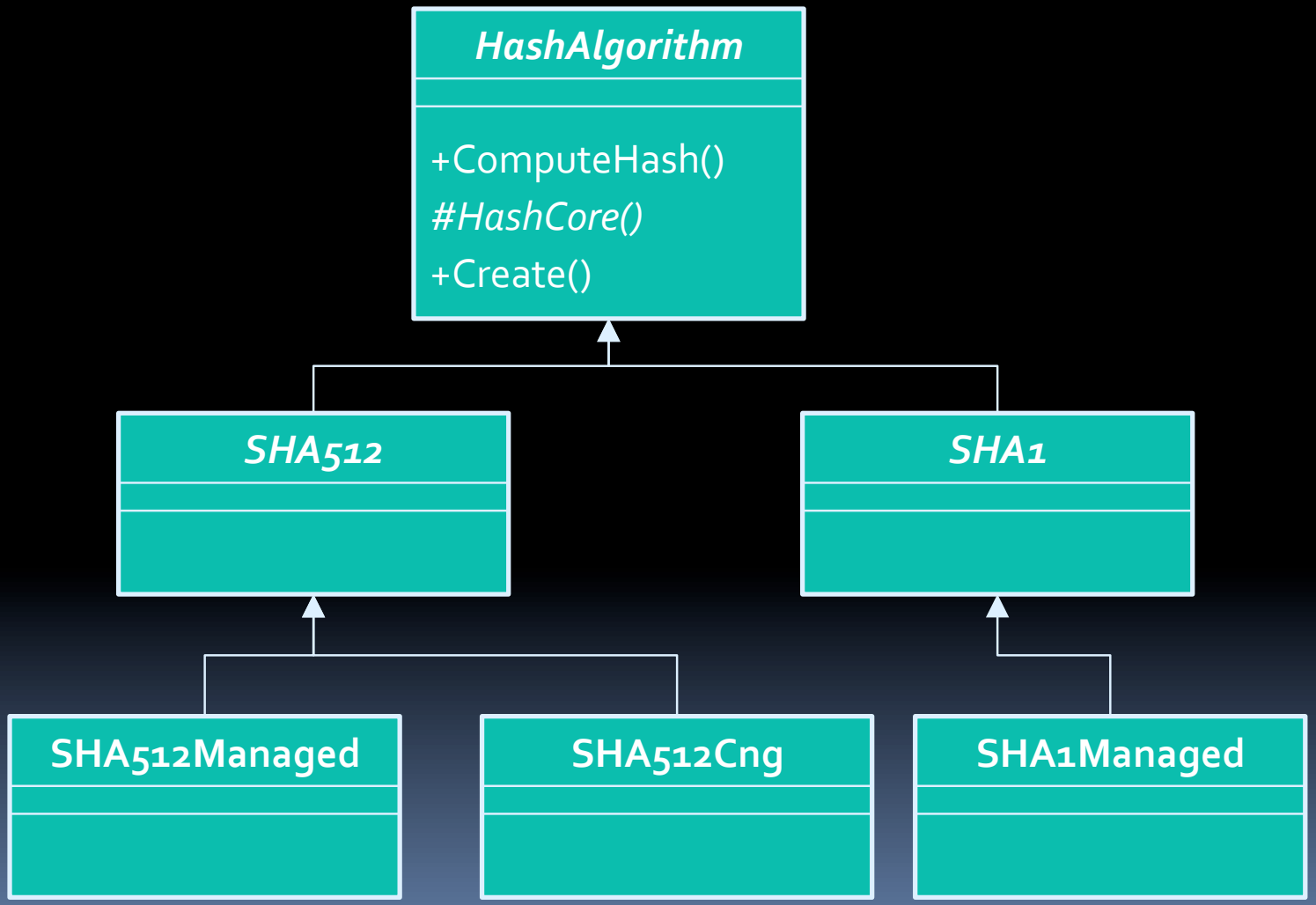
*If used correctly…

.NET Cryptography

.NET top-level abstract classes

- SymmetricAlgorithm
- AsymmetricAlgorithm
- HashAlgorithm
  - KeyedHashAlgorithm
    - HMAC
- RandomNumberGenerator

.NET Crypto Architecture



.NET examples

- Non-agile:

```
MD5Cng hashObj = new MD5Cng();
byte[] result =
  hashObj.ComputeHash(data);
```

.NET examples

- More agile:

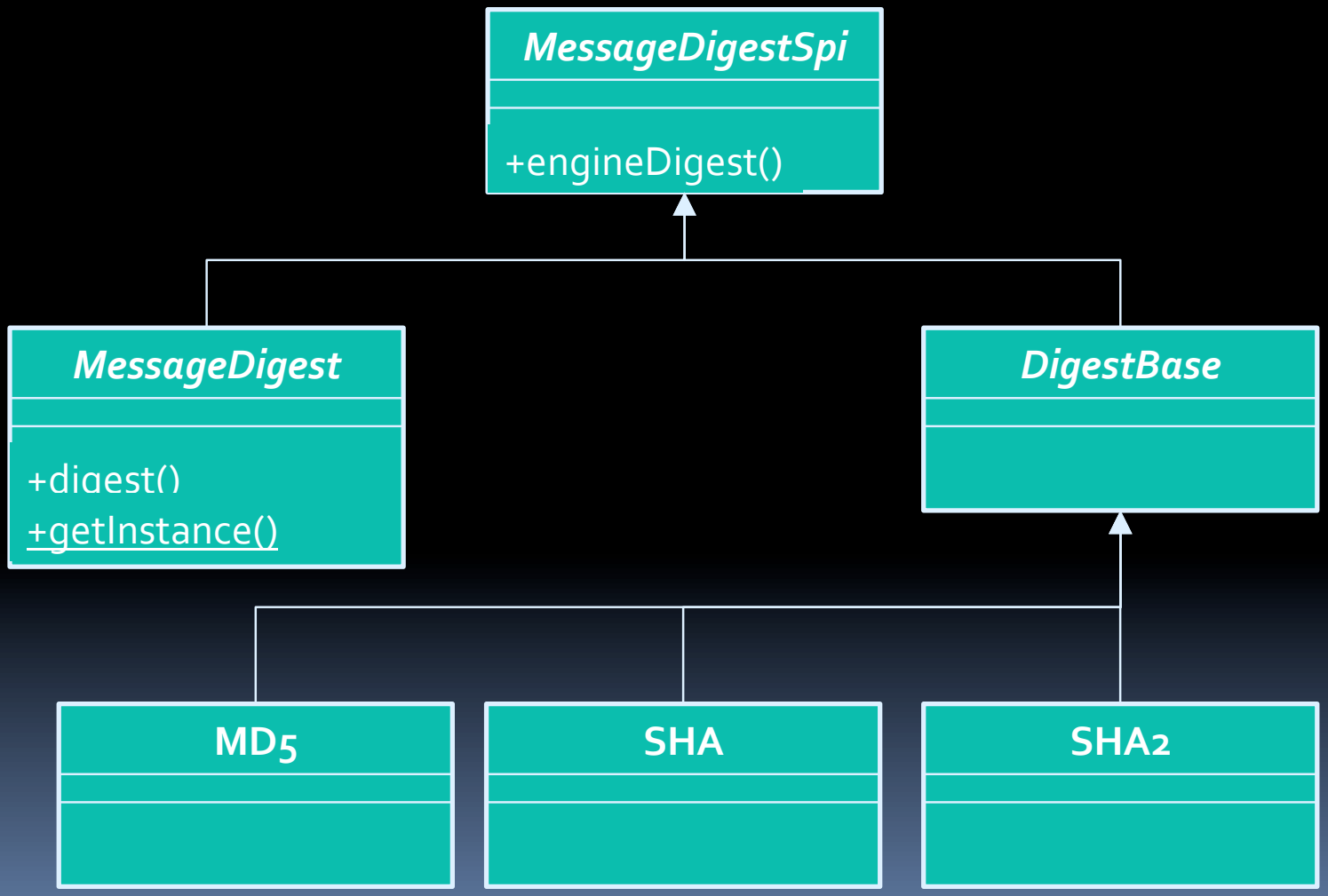
```
HashAlgorithm hashObj =
  HashAlgorithm.Create("MD5");
byte[] result =
  hashObj.ComputeHash(data);
```

Java Cryptography Architecture (JCA)

JCA top-level classes

- javax.crypto.Cipher
- javax.crypto.KeyAgreement
- java.security.KeyFactory
- javax.crypto.KeyGenerator
- java.security.KeyPairGenerator
- javax.crypto.Mac
- java.security.MessageDigest
- javax.crypto.SecretKeyFactory
- java.security.SecureRandom
- java.security.Signature

JCA Architecture

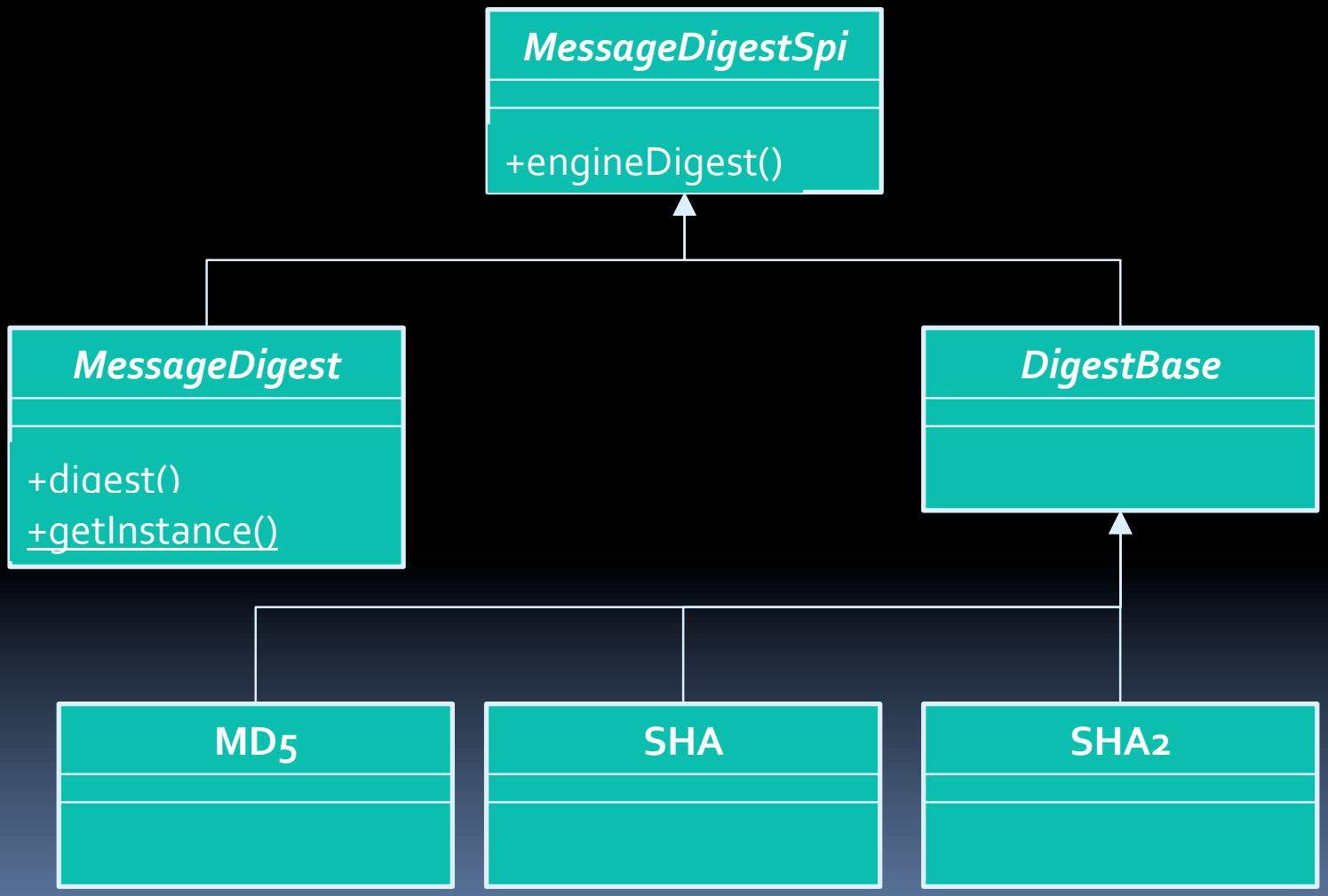


JCA example

- More agile (by default, this is great!):

```
MessageDigest md =
  MessageDigest.getInstance("MD5");
byte[] result = md.digest(data);
```

JCA Architecture

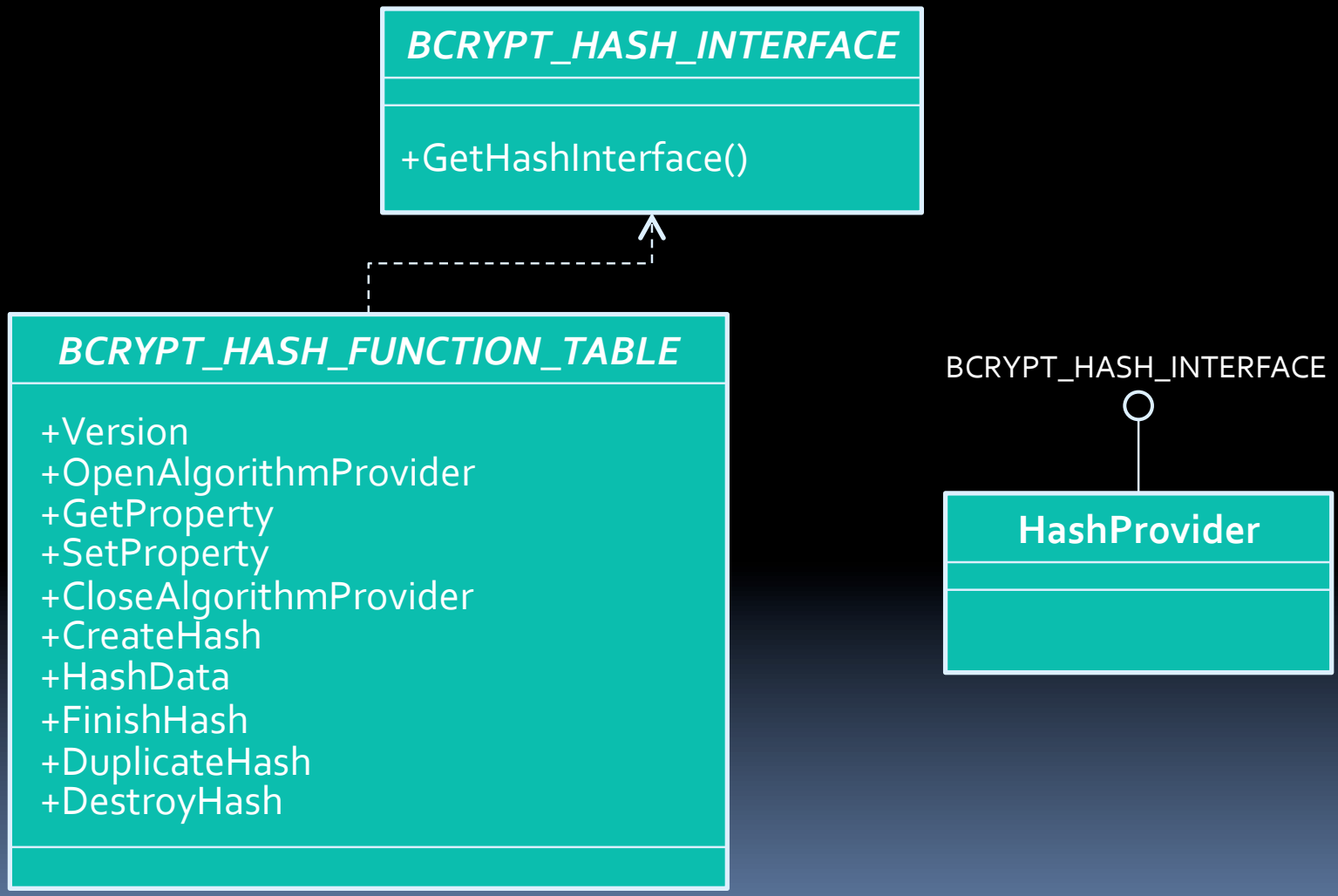


Cryptography API: Next Generation (CNG)

CNG agile capabilities

- Key generation and exchange
- Object encoding and decoding
- Data encryption and decryption
- Hashing and digital signatures
- Random number generation

CNG Architecture



CAPI example

- Non-agile:

```
HCRYPTPROV hProv = 0;
HCRYPTHASH hHash = 0;
CryptAcquireContext(&hProv,
  NULL, NULL, PROV_RSA_FULL, 0);
CryptCreateHash(hProv, CALG_MD5,
  0, 0, &hHash);
CryptHashData(hHash, data, len, 0);
```

CNG example

- More agile:

```
BCRYPT_ALG_HANDLE hAlg = 0;
BCRYPT_HASH_HANDLE hHash = 0;
BCryptOpenAlgorithmProvider(&hAlg,
  "MD5", NULL, 0);
BCryptCreateHash(hAlg, &hHash, …);
BCryptHashData(hHash, data, len, 0);
```

Still looks hardcoded to me…

- .NET

```
HashAlgorithm.Create("MD5");
```

- JCA

```
MessageDigest.getInstance("MD5");
```

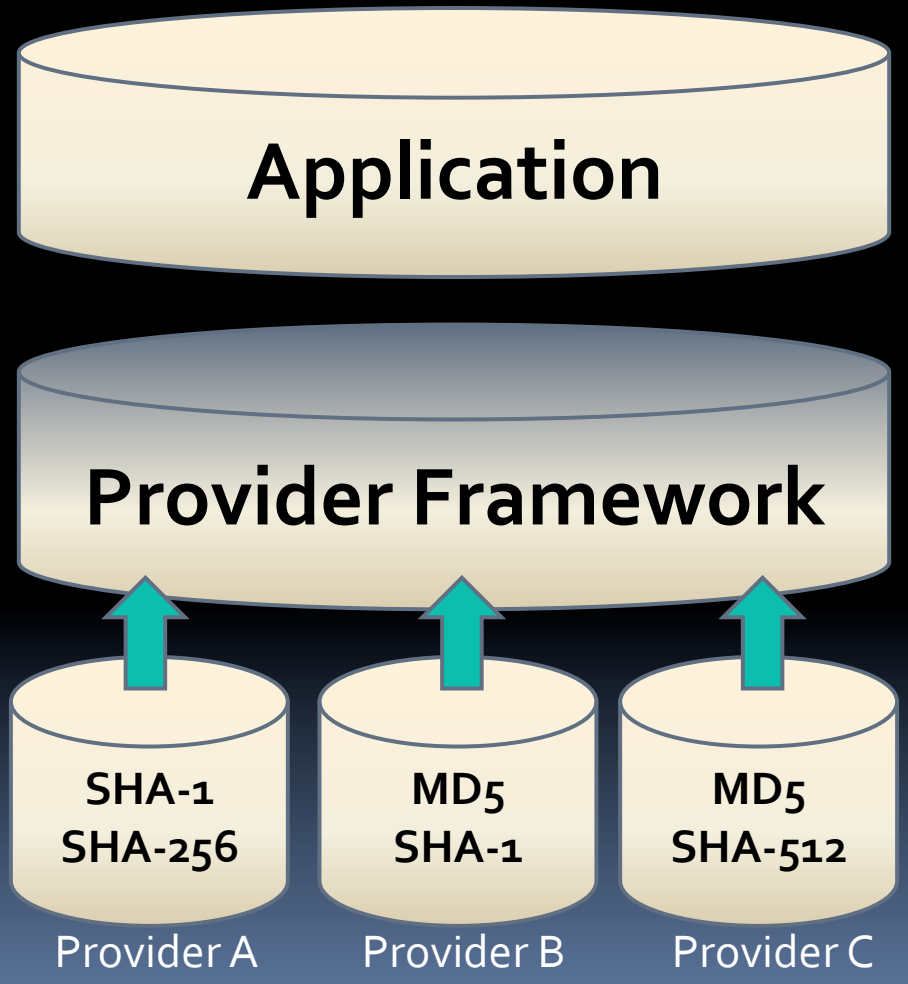
- CNG

```
BCryptOpenAlgorithmProvider(&hAlg, "MD5", NULL, 0);
```

Steps toward crypto agility

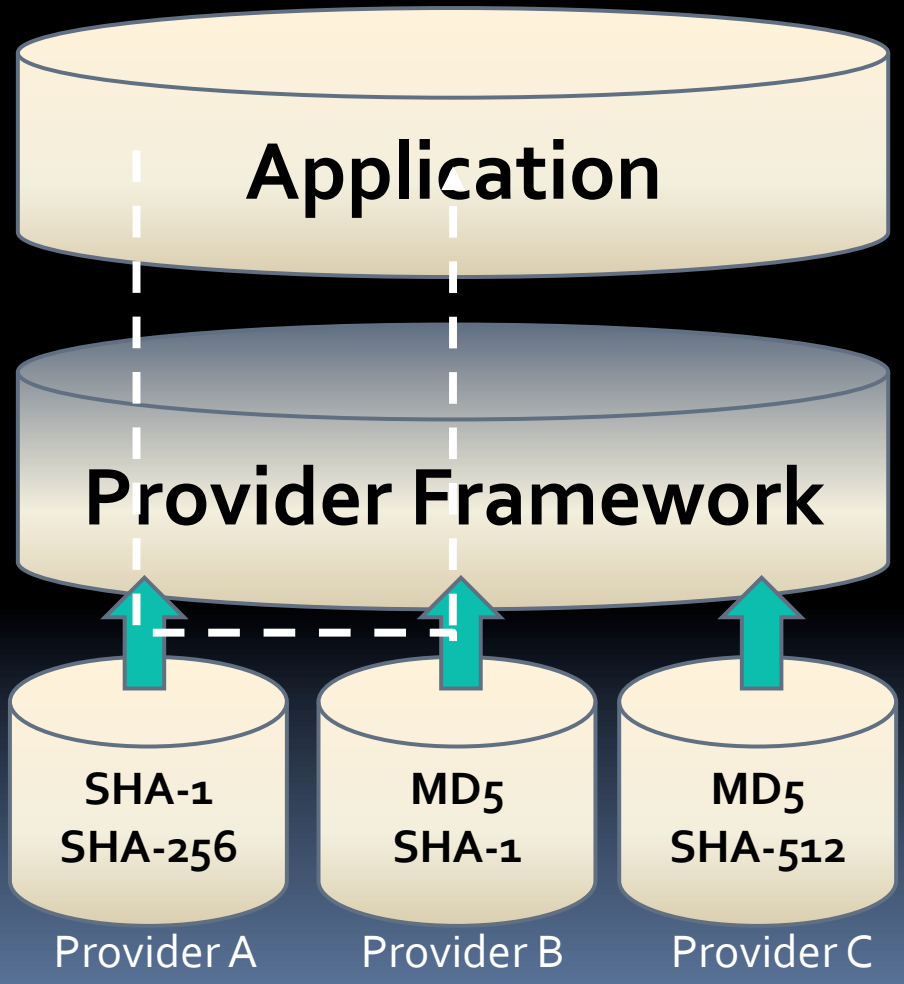
- ~~Step 1: Avoid hardcoded algorithms~~
- Step 2: Reconfigure the algorithm provider

JCA Provider Framework



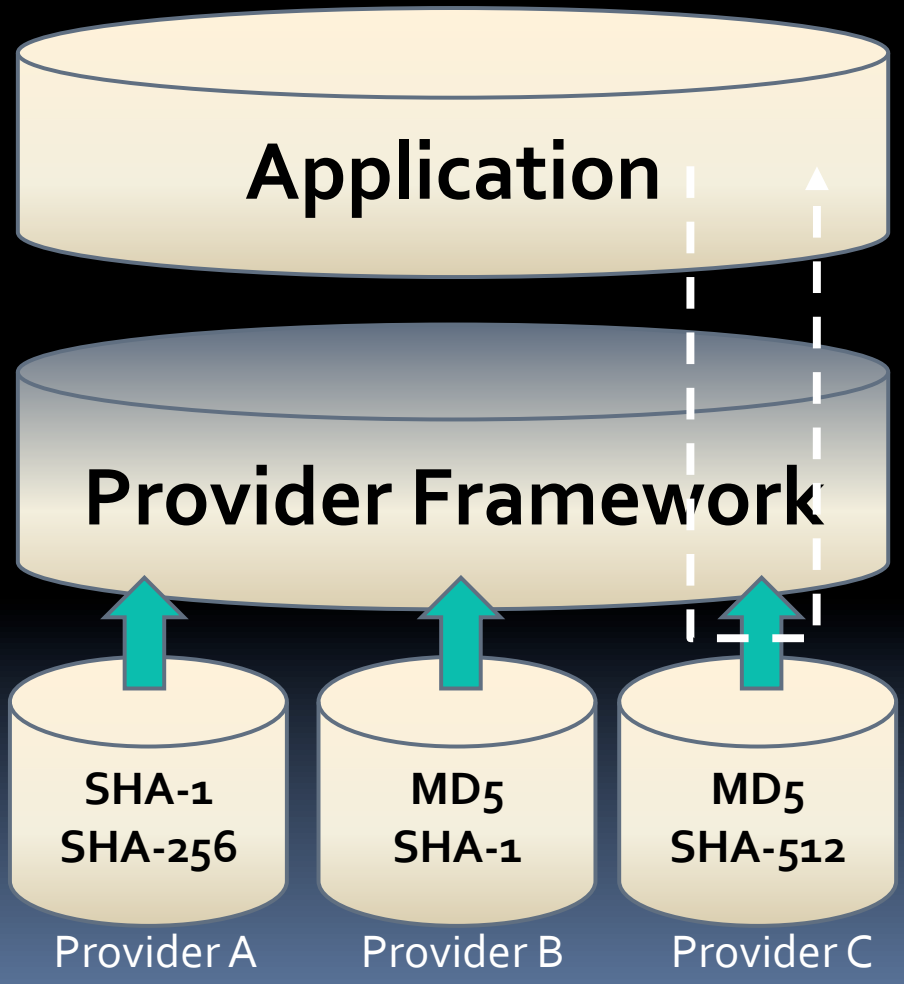
JCA Provider Framework

MessageDigest.
getInstance
("MD5");



JCA Provider Framework

```
MessageDigest.
getInstance
("MD5",
"Provider C");
```



Configure providers

- Option #1: Modify java.security file (static)

```
security.provider.1=
   sun.security.provider.Sun
security.provider.2=
   sun.security.provider.SunJCE
…
```

Configure providers

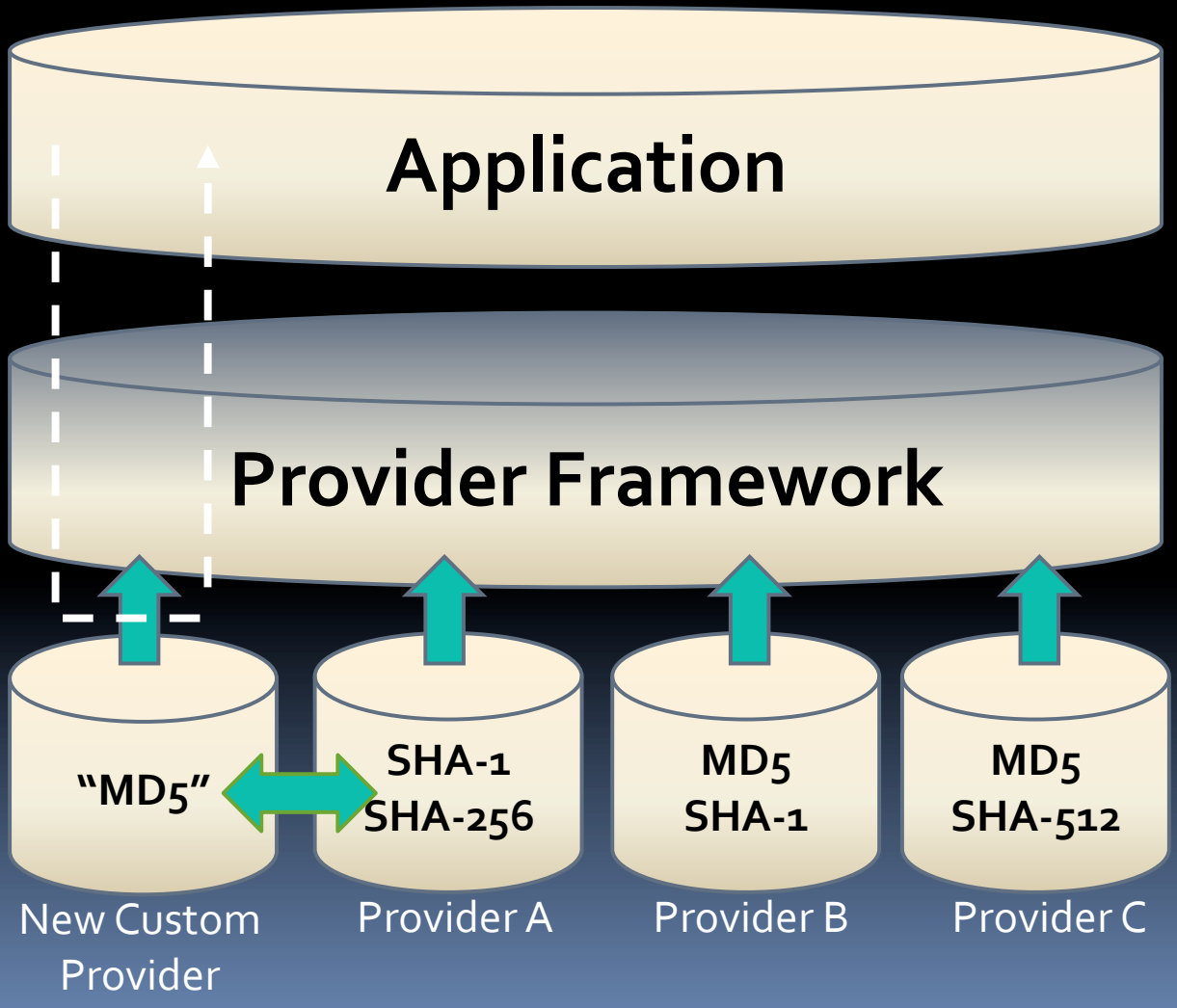
- Option #2: Add in code (dynamic)

```
java.security.Provider provider =
   new MyCustomProvider();
Security.addProvider(provider);
```

Scenario #1: Bad provider

```
security.provider.1=foo
security.provider.2=bar
```

Scenario 2: Bad algorithm

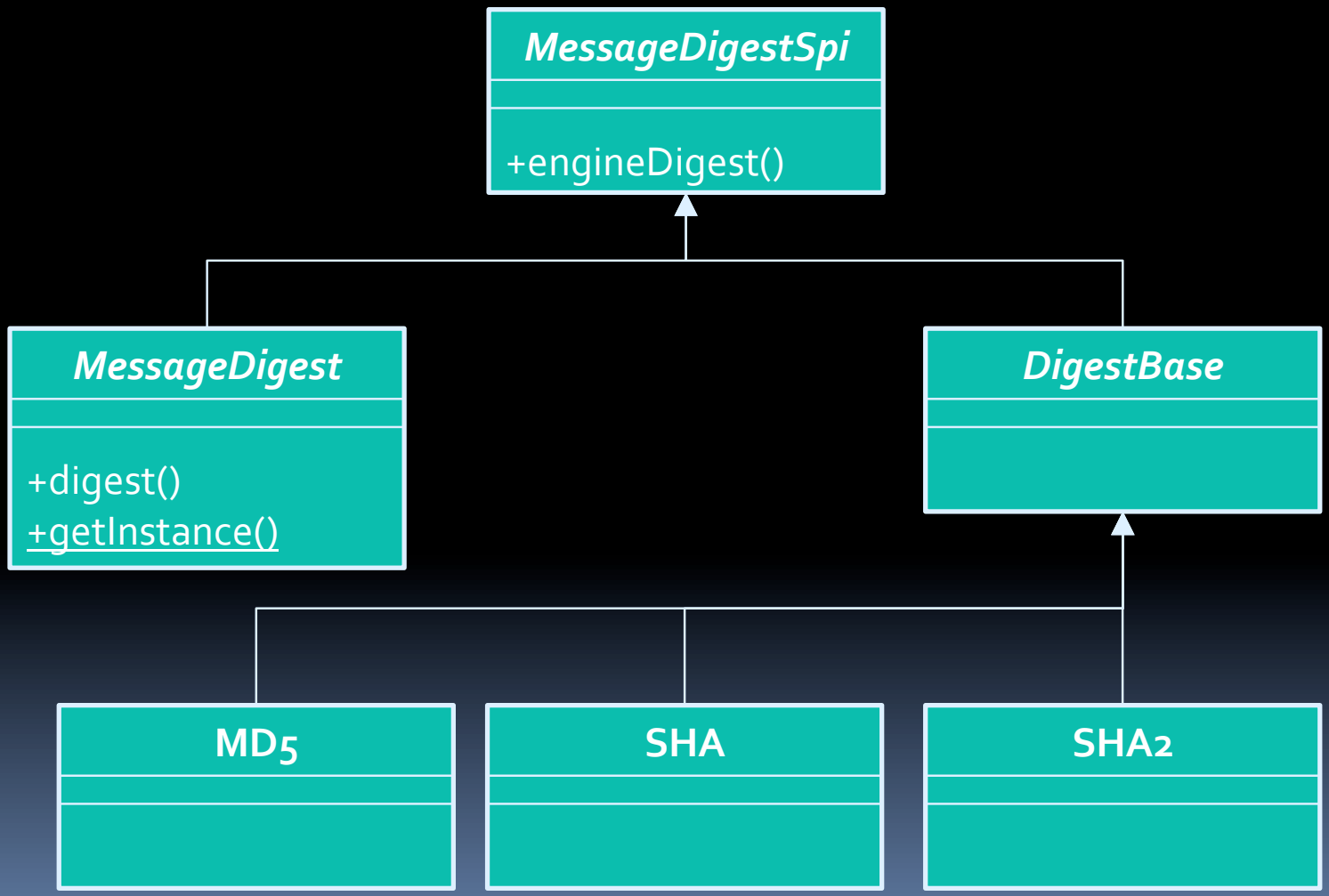


Custom provider

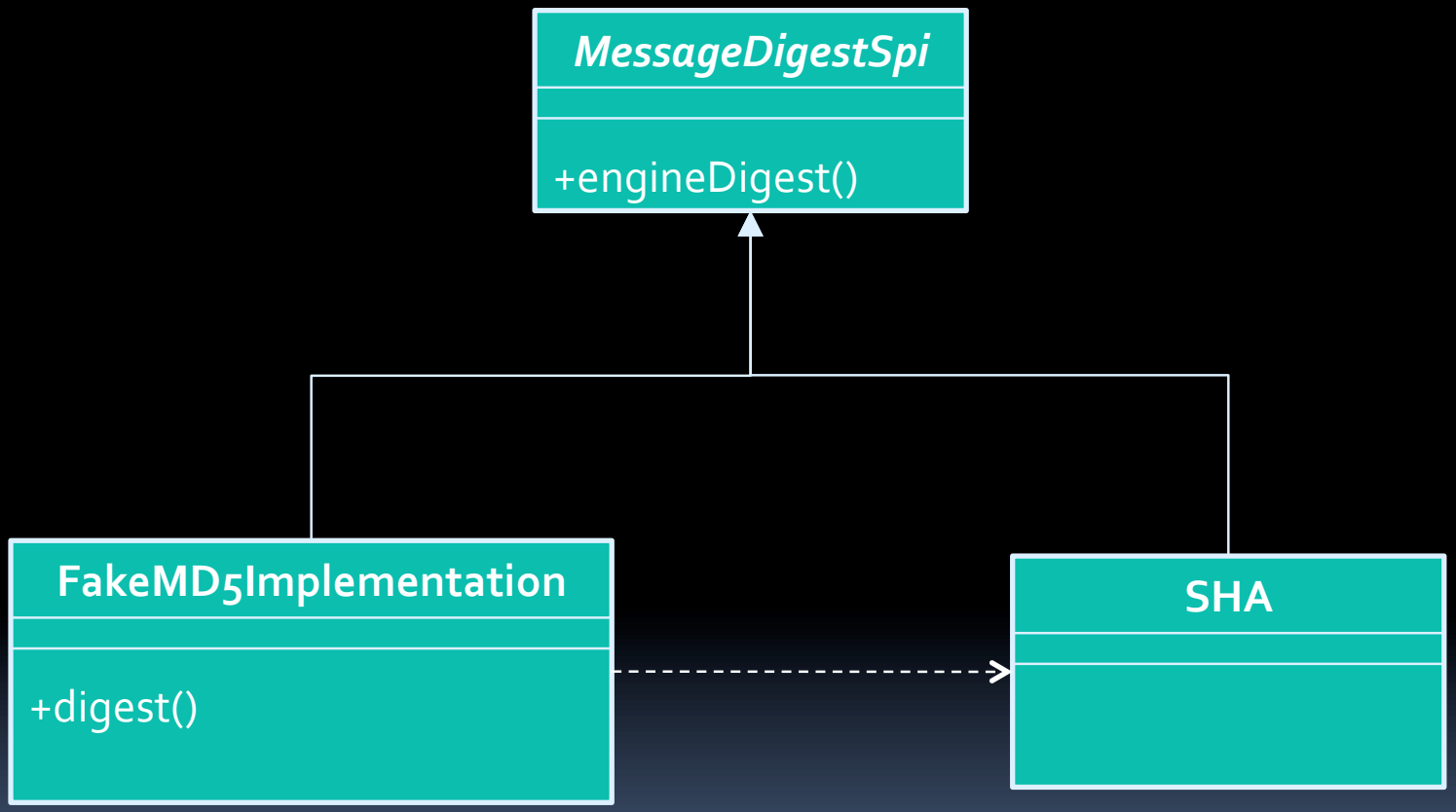
```
public class Provider
  extends java.security.Provider {

  put("MessageDigest.MD5",
  "MyFakeMD5Implementation");
}
```

JCA Architecture



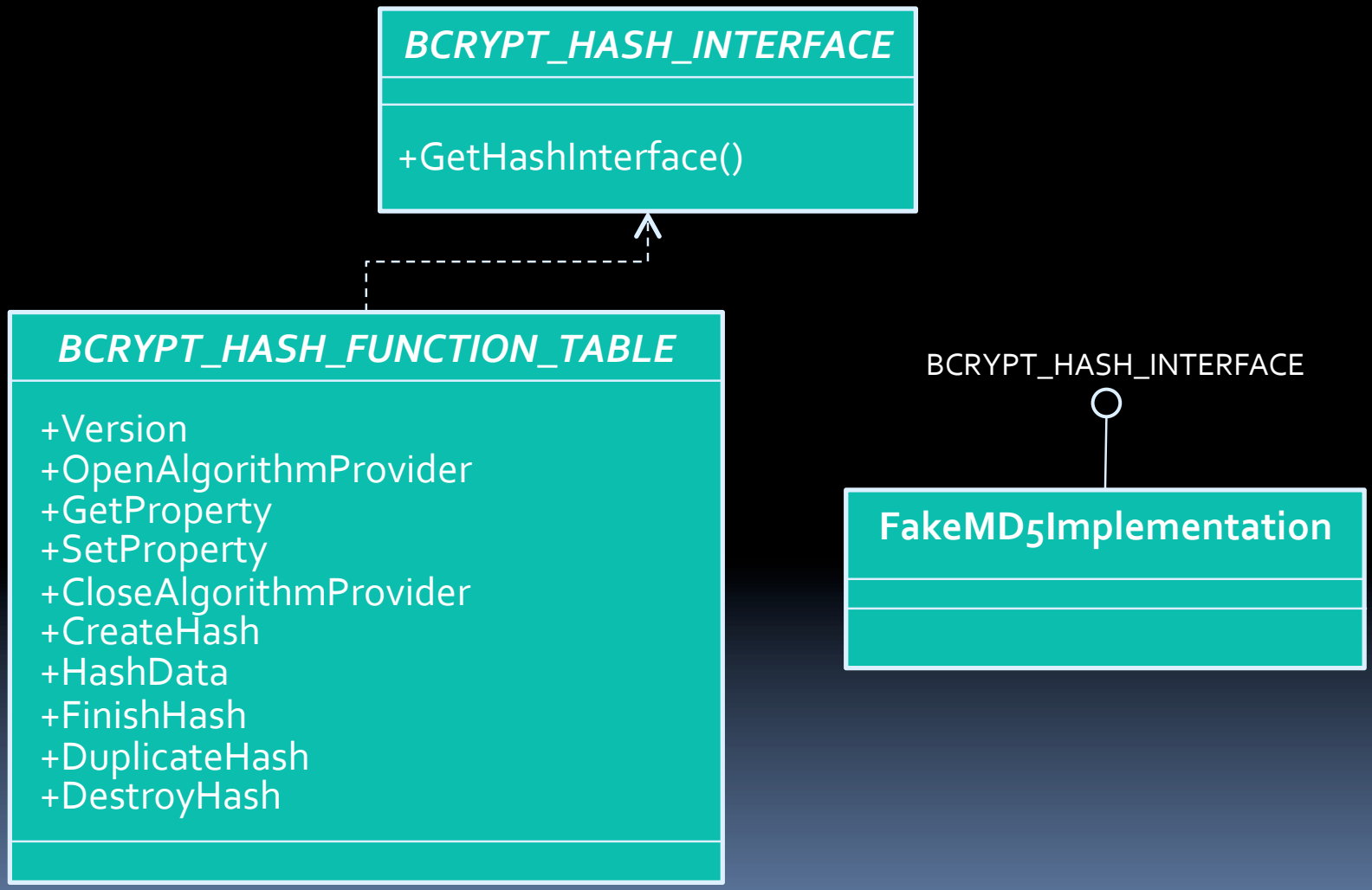
Fake implementation



CNG provider framework

- Similar to JCA, but less flexible
- Custom providers go in system folder
- Must register programmatically
  - Can only specify top or bottom of the list

Fake implementation



Registering a custom provider

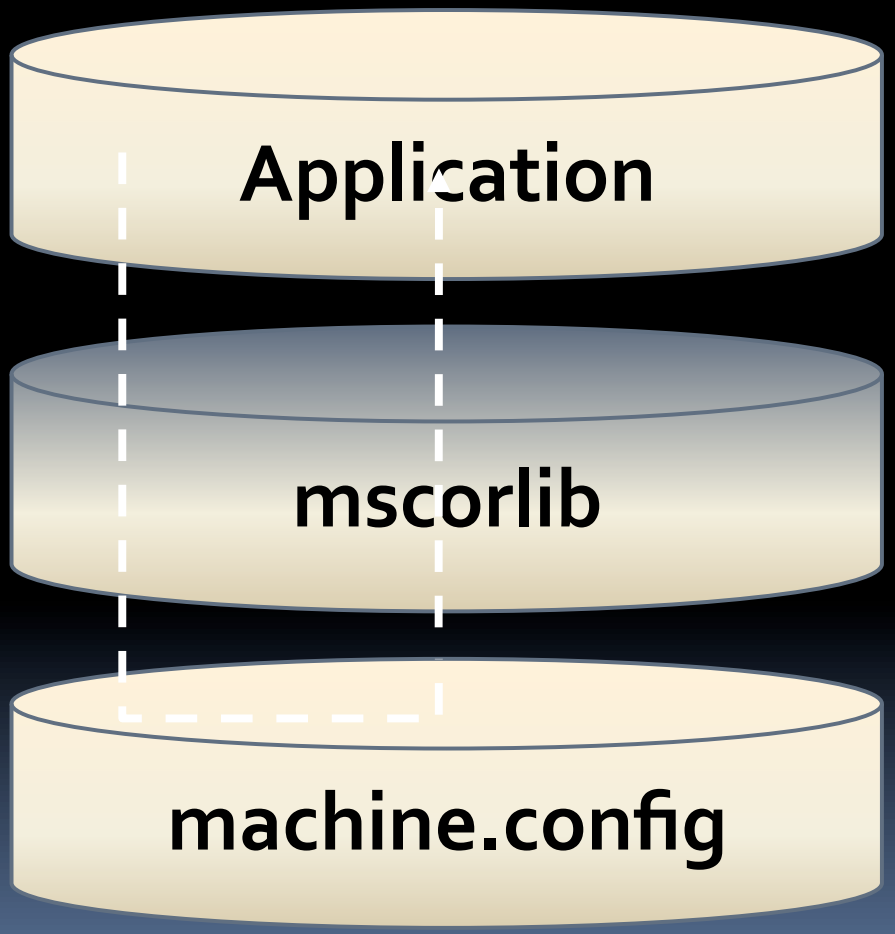
```
CRYPT_PROVIDER_REG providerReg =
    {…};
BCryptRegisterProvider(
    "FakeMD5Implementation", 0,
    &providerReg);
BCryptAddContextFunctionProvider(
    CRYPT_LOCAL, NULL,
    BCRYPT_HASH_INTERFACE, "MD5",
    "FakeMD5Implementation",
    CRYPT_PRIORITY_TOP);
```

Avoid hardcoded implementation

```
BCRYPT_ALG_HANDLE hAlg = 0;
BCryptOpenAlgorithmProvider(
  &hAlg,
  "SHA1",
  "Microsoft Primitive Provider",
  0);
```

.NET

```
HashAlgorithm.
Create("MD5")
```



Altering machine.config

```
<configuration>
 <mscorlib>
  <cryptographySettings>

   <nameEntry name="MD5"
    class="MyPreferredHash" />

   <cryptoClasses>
    <cryptoClass
     MyPreferredHash="SHA512Cng, …" />
   </cryptoClasses>
```

Remapping algorithm names is dangerous

MD5 ⇨ SHA-1

- This is a good thing, right?
- What could possibly go wrong?

Steps toward crypto agility

- ~~Step 1: Avoid hardcoded algorithms~~
- ~~Step 2: Avoid hardcoded implementations~~
- ~~Step 3: Reconfigure the algorithm provider~~
- Step 3 (alternate): Avoid default algorithm names

Unique algorithm names

- .NET

```
HashAlgorithm.Create(
 "ApplicationFooPreferredHash");
```

- JCA

```
MessageDigest.getInstance(
 "ApplicationBarPreferredDigest");
```

- CNG

```
BCryptOpenAlgorithmProvider(&hAlg,
"ApplicationFooPreferredHash", …);
```

Steps toward crypto agility

- ~~Step 1: Avoid hardcoded algorithms~~
- ~~Step 2: Avoid hardcoded implementations~~
- ~~Step 3: Reconfigure the algorithm provider~~
- ~~Step 3 (alternate): Avoid default algorithm names~~
- Step 3 (alternate #2): Pull algorithm name from secure configuration store

Unique provider vs. config

Unique provider

- Pros
  - Security to perform this action already part of the system
- Cons
  - Probably prohibitive in terms of implementation cost

Configuration store

- Pros
  - Much easier to implement
- Cons
  - Must remember to secure the store!

DEMO

What went wrong?

- Changing the algorithms is one thing…
- …but changing stored data is another.

Steps toward crypto agility

- ~~Step 1: Avoid hardcoded algorithms~~
- ~~Step 2: Avoid hardcoded implementations~~
- ~~Step 3: Reconfigure the algorithm provider~~
- ~~Step 3 (alternate): Avoid default algorithm names~~
- ~~Step 3 (alternate #2): Pull algorithm name from secure configuration store~~
- Step 4: Store and consume algorithm metadata

What metadata to store

- Hashes
  - Algorithm name
  - Salt size
  - Output size
  - (Max input size)
- Size considerations
  - Local variables (ie source code)
  - Database columns

What metadata to store

- Symmetric encryption
  - Algorithm name
  - Block size
  - Key size
  - Mode
  - Padding mode
  - Feedback size

What metadata to store

- Asymmetric encryption
  - Algorithm name
  - Key sizes
  - Key exchange algorithm
  - Signature algorithm

What metadata to store

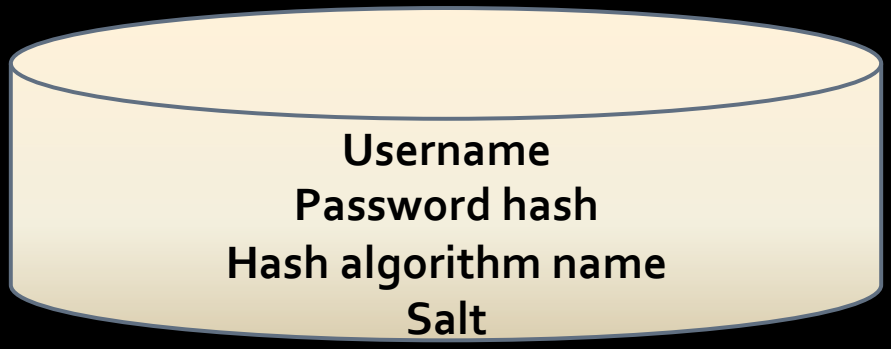
- MAC
  - Algorithm name
  - Key size
  - Key derivation function
    - Function algorithm
    - Salt size
    - Iteration count
  - Output size
  - (Max input size)

MS-OFFCRYPTO

- Office Document Cryptography Structure Specification

- http://msdn.microsoft.com/en-us/library/cc313071(office.12).aspx

Consuming metadata: Authn



- Pull metadata from database for user
- Instantiate the same algorithm originally used
- Create hash from supplied password & compare
- If authentic, prompt for new password
- Store in new format

Storage considerations

- This is wasteful

DocId	EncryptedContents	Algorithm	KeySize	Mode
1	sdfER35wef23SDDp…	AES	256	CBC
2	pOlo89X13WasM8oi…	AES	256	CBC
3	45TrooSd2ZaZ23lk…	RC2	64	ECB

Storage considerations

- This is better

DocId	EncryptedContents	AlgorithmId
1	sdfER35wef23SDDp…	1
2	pOlo89X13WasM8oi…	1
3	45TrooSd2ZaZ23lk…	2

AlgorithmId	Algorithm	KeySize	Mode
1	AES	256	CBC
2	RCS	64	ECB

Wrap-up

Other frameworks

- Bouncy Castle
  - Missing factory/provider functionality
- OpenSSL
  - Not OO
- CAPI
  - Providers need to be signed by Microsoft
  - Algorithms stored as integers, not strings
- Common Crypto
  - Not OO

Summary

- .NET
  - Never hard-code classes, use abstract classes and factory pattern
- JCA
  - Never name specific provider in getInstance()
  - Never dynamically add providers
- CNG
  - Never name specific implementation in BCryptOpenAlgorithmProvider

Summary

- Reconfiguring a global algorithm name is extremely dangerous
  - Use as last resort and a temporary fix at best
- Store and consume algorithm metadata
- Read all formats, but write only strong crypto

Q & A

More resources

- http://www.microsoft.com/sdl
- http://blogs.msdn.com/b/sdl

- My alias: bryansul

SDL Allowed Algorithms

Algorithm Type	Banned Algorithms to be replaced in existing code or used only for decryption	Minimally Acceptable Algorithms acceptable for existing code (except sensitive data)	Recommended Algorithms for new code
Symmetric Block	DES, 3DES (2 key), DESX, RC2, SKIPJACK	3DES (3 key)	AES (>=128 bit)
Symmetric Stream	SEAL, CYLINK_MEK, RC4 (<128bit)	RC4 (>= 128bit)	None – Block cipher is preferred
Asymmetric	RSA (<2048 bit), Diffie-Hellman (<2048 bit)		RSA (>=2048bit), Diffie-Hellman (>=2048bit), ECC (>=256bit)
Hash (includes HMAC usage)	SHA-0 (SHA), SHA-1, MD2, MD4, MD5	3DES MAC	SHA-2 (includes: SHA-256, SHA-384, SHA-512)