Summary of misuses expressed in jGuard

December 22, 2021

1 RQ1-a

1.1 ByteArrayOutputStream misuses

In the MuBench dataset, the most common misuse is related to using an `ObjectOutputStream` with an `ByteArrayOutputStream` as a target. MuBench reports 36 misuses about invoking the underlying instance's `toByteArray()` before properly flushing or closing the wrapping object output stream.

It is possible to guard against this misuse with jGuard. A guard on the underlying stream can represent whether it has been wrapped or not. Its `toByteArray()` method can then ensure that no surrounding stream is still active:

```java
public verified class
     VerifiedByteArrayOutputStream
    extends OutputStream {
  private ByteArrayOutputStream output;
  public guard hasPendingWrites;

  public VerifiedByteArrayOutputStream() {
    output = new ByteArrayOutputStream();
  }
  public verified void attach()
      when returns then sets
           hasPendingWrites {}
  public void write(int i) throws
       IOException {
    output.write(i);
  }
  public verified byte[] toByteArray()
      require !hasPendingWrites {
    return output.toByteArray();
  }
}
```

```java
public verified class
     VerifiedObjectOutputStream
    extends ObjectOutputStream {
  private VerifiedByteArrayOutputStream
       out;

  public VerifiedObjectOutputStream(
       VerifiedByteArrayOutputStream out)
      throws IOException {
    super(out);
    this.out = out;
    out.attach();
  }
  public verified void flush()
      throws IOException
      when returns then resets out.
           hasPendingWrites {
    super.flush();
  }
}
```

A constructed test verified that the two classes detect the reported misuse.

MuBench reports similar misuses for when an `ByteArrayOutputStream` is wrapped in an `DataOutputStream`. This is the fifth most prevalent misuse in MuBench. A similar construction can be used to guard against this constellation as well.

1.2 Using Long.parseLong

In this misuse, offending code used `parseLong` without proper exception handling. In scenarios where the input to `parseLong` comes from a user input, it may be desirable to handle the exception to provide a helpful error message.

As this misuse is related to exception handling, it cannot be described with jGuard.

1.3 Misuses on Map

MuBench reports two common misuses related to the `Map` class. One of this misuses relates to the fact that its `get()` method may return null, which needs to be checked by the calling code. This misuse, of which there are three instances in MuBench, cannot be detected by jGuard.

Another misuse related to `Map` is calling `get()` with a nullable key. Some implementations throw in this case, so this should always be avoided. With jGuard, a guard against this is simple:

```
class NullCheckedMap<K, V> extends AbstractMap<K, V> {
  private Map<K, V> inner;
  NullCheckedMap(Map<K, V> inner) { this.inner = inner; }
  public Set<Map.Entry<K, V>> entrySet() { return inner.entrySet(); }

  public verified V get(Object key) require key != null {
    return inner.get(key);
  }
}
```

Finally, another misuse involves updating a `HashMap` concurrently, which may corrupt its data. This misuse also cannot be detected with jGuard.

1.4 Misuses on String

When converting strings to a byte array representation (or vice-versa), a charset should be specified. Default methods without a charset parameter may use an undefined charset, which can lead to incompatibilities.

Forgetting to declare a charset is a common misuse in MuBench which can be detected by forbidding usages of the problematic methods:

```
public verified class String implements CharSequence, ... {
  public verified String(byte[] bytes) require false {}
  public verified byte[] getBytes() require false {
    // ...
  }
  // other members left unchanged.
}
```

1.5 Misusing StringTokenizer

The `StringTokenizer` class can be used to split strings at configurable delimiters. Similarly to the API provided by `Iterator`, one needs to check that further tokens are available before requesting one. Consequently, a guard with jGuard can be constructed similarly to the approach chosen for iterator misuses:

```
public verified class VerifiedStringTokenizer
    extends StringTokenizer {
  private guard checkedMoreTokens;

  public verified boolean hasMoreTokens()
      when returns true then sets checkedMoreTokens {
```

```
    return super.hasMoreTokens();
  }
  public verified String nextToken()
      requires checkedMoreTokens {
    return super.nextToken();
  }
}
```

1.6 Misuses on ResultSet

It is the responsibility of a developer to close `ResultSet`s obtained from an SQL connection.

This misuse can be detected with jGuard by creating a guard with a finalized state that must be met. A scaffold for a potential result set class guarding against this is shown below:

```
public verified class ResultSet {
  private guard hasBeenClosed finally set;
  public verified void close()
      when returns then sets hasBeenClosed {

  }
}
```

Results for RQ3-a: JDK classes cannot be changed, which means that using the verified classes presented here would require additional effort from a developer.
However, it was shown that jGuard is expressive enough to describe guards against prevalent misuses happening in practice. Out of the total 139 misuses MuBench contains for the ten most-misused JDK classes, it was possible to express guards for 124 (89.2 %) of them.

2 RQ1-b

2.1 Using DES

The Data Encryption Standard (DES) is a symmetric encryption algorithm. By modern standards, the algorithm is considered insecure and its usage is discouraged [?].

This misuse can be detected with jGuard language extensions. In BouncyCastle, the `DESEngine` class is responsible for providing the DES algorithm. Calling `Cipher.getInstance("DES")` will use this class as an implementation. Since this constitutes a misuse, a suitable check is to forbid creating instances of the implementation class altogether. First, a meta variable describing whether DES is allowed was introduced. It defaults to false, but allows users to enable DES if desired. Second, the `DESEngine()` constructor is converted to a verified constructor requiring that the meta variable is set to true. This prevents a usage of DES with the default configuration:

```
public verified class DESEngine implements BlockCipher {
  protected static final int BLOCK_SIZE = 8;
  private int[] workingKey = null;
  meta boolean desAllowed = false;

  public verified DESEngine() require desAllowed { }

  // rest of the class is left unchanged
}
```

MuBench contains 9 instances of this misuse. Both CogniCrypt and the annotated DES implementation presented here caught all of them in a test run.

2.2 Using the ECB mode for encryption

When requesting a symmetric encryption scheme without specifying further transformations, most implementations default to using the Electronic Code Book (ECB) mode. In this mode, messages are split into fixed-length blocks which are then encrypted or decrypted independently [**?**]. This mode is not generally considered secure [**?**], as the output of two blocks may be correlated to learn about the original message.

Hence, it is considered a bad practice to use symmetric encryption algorithms with ECB. At the same time, this potentially-insecure mode is used a lot. In an automated analysis of 11 748 Android applications, Egele et al. found that 7656 applications were using this encryption mode [**?**].

To detect these misuses, the `BaseBlockCipher` class from BouncyCastle has been converted to a verified class with the following additions.

```
public verified class BaseBlockCipher extends BaseWrapCipher
    implements PBE {
  meta boolean allowECB = false;
  // ...
  protected verified void engineInit(int opmode, Key key,
      AlgorithmParameterSpec params, SecureRandom random)
    throws InvalidKeyException, InvalidAlgorithmParameterException
    require allowECB || modeName != null && !modeName.equals("ECB") {
      //...
    }
}
```

In MuBench, 9 misuses either explicitly use ECB mode or fallback to ECB because this mode is used by default when no other mode is specified. The presented checks cannot distinguish between the two cases, but all 9 misuses were correctly reported when running the offending applications with the verified implementation. CogniCrypt also detected all 9 cases.

2.3 Re-initializing Cipher

According to MuBench, the common `Cipher` class is not meant to be initialized multiple times. Instead of re-initializing a cipher, developers should create new new instance and initialize that one. On the other hand, other operations on a cipher may be invoked multiple times.

To detect this misuse, the `BaseBlockCipher` class from BouncyCastle is converted to a verified class. This class is responsible for wrapping all encryption algorithms into the interface expected by JCA. An introduced guard `isInitialized` represents whether the cipher has been initialized. When `engineInit` is called, the implementation should check that the guard has not already been set. The guard should be set once the method returns. With jGuard language constructs, this behavior can be expressed using requirements and consequences:

```
public verified class BaseBlockCipher extends BaseWrapCipher
    implements PBE {
  private guard isInitialized;

  @Override
  protected verified void engineInit(
    int opmode,
    Key key,
    final AlgorithmParameterSpec params,
    SecureRandom random
  )
    throws InvalidKeyException, InvalidAlgorithmParameterException
    require !isInitialized
    when returns then sets isInitialized {
    // Implementation left unchanged
  }
  // other members left unchanged
}
```

MuBench reports three misuse under this description, all of which were detected by this guard. CogniCrypt found two of the three instances. In the misuse not detected by CogniCrypt, `init` was called on the same instance, but in different methods. As CogniCrypt analyzes the control flow of a single method, it was unable to detect this.

2.4 Re-initializing Mac

Similarly to the misuse on the `Cipher` class, the `Mac` class is not generally meant to be re-initialized either. A working guard can be obtained similarly as well. In BouncyCastle, the `BaseMac` is responsible for wrapping MAC implementations for JCA. Hence, it can be converted to a verified class with a guard ensuring it's not initialized multiple times:

```
public verified class BaseMac extends MacSpi implements PBE {
  private guard isInitialized;

  @Override
  protected verified void engineInit(Key key,
      final AlgorithmParameterSpec params)
    throws InvalidKeyException,
        InvalidAlgorithmParameterException
    require !isInitialized
    when returns then sets isInitialized {
      // implementation unchanged
    }

  // other members were left unchanged.
}
```

MuBench contains one instance of this misuse, which was detected when running the offending code against the guarded BouncyCastle library.

CogniCrypt was unable to detect this misuse. This is because there is only `init` invocation in the source code, but the containing method is called multiple times. A control flow analysis of this method does not reveal the problematic usage.

2.5 Insecure RSA configurations

Using the RSA encryption algorithm with a PKCS v1 encoding configuration can reveal information about the encrypted message and is considered insecure [?].

To detect insecure usages of this encoding, the `CipherSpi` class implementing asymmetric ciphers in BouncyCastle was converted to a verified class.

A simple method is added to require that a padding is set and that no PKCS v1 encoding is used:

```
private verified void verifyEngineParameters()
  require !(cipher instance RSABlindedEngine),
    !(cipher instanceof PKCS1Encoding) {

}
```

In this method, the first requirement ensures that a padding is used. The second requirements forbids using PKCS v1 encodings. The `verifyEngineParameters()` method is the invoked at the end of each other method changing underlying engine parameters.

MuBench contains a single instance of this misuse, which is both reported by the guarded implementation and CogniCrypt.

2.6 Using PBE with MD5 and DES

Password-Based Encryption, or PBE, combines a password chosen by a user with salt into a secure cryptographic key [?].

Using insecure encryption algorithms, or hashing algorithms with known collisions like MD5 [?], can lead to predictable and hence insecure keys.

In BouncyCastle, the `PBE` class is responsible for constructing a PBE implementation. To verify that md5 is not used, a requirement is added to the `makePBEGenerator` method:

```
private static verified PBEParametersGenerator makePBEGenerator(
    int type, int hash)
  requires type != MD5 {
  // implementation left unchanged
```

In the MuBench dataset, this misuse is reported once. The misuse was reported by both CogniCrypt and the verified implementation.

2.7 Using a non-random initialization vector for CBC

Cipher Block Chaining Mode, or CBC, combines previous ciphertext blocks with the next message block to avoid information about the source leaking out. When encrypting the first block, the message is combined with an Initialization Vector (IV) instead. If an IV is re-used and a message starting with the same block is encrypted multiple times, the first ciphertext block will also be the same, which is undesirable [?]. For this reason, encrypting with CBC mode should require a random initialization vector.

To determine randomness, the guarded BouncyCastle implementation can require that the initialization vector was the result of a call to `SecureRandom.nextBytes`, as those are most likely to be a secure random source.

To store whether a byte-array is random, an external state declaration has been introduced:

```
state IsRandom for byte[] {
  boolean isRandom = false;
}
```

In the common `BaseBlockCipher` class implementing symmetric block ciphers, a method has been introduced:

```
private verified void afterInit(int opmode)
  required opmode != Cipher.ENCRYPT_MODE || ivParam == null ||
    IsRandom(ivParam.getIV()).isRandom {}
```

This method ensures that, if there is an initialization vector and the cipher is used for encryption, that the IV is a random byte array. Finally, the `afterInit` method is called at the end of `engineInit` to ensure the IV is in the desired state after initialization.

This mechanism should ensure random IVs. To ensure that byte arrays generated by a `SecureRandom` instance have their `IsRandom` extended state set, a new secure random generator was added to BouncyCastle. The implementation is copied from the default implementation in the JDK, but jGuard extensions allow setting the desired state:

```
public synchronized verified void engineNextBytes(byte[] result)
  when returns then IsRandom(result).isRandom = true {
 // implementation left unchanged
}
```

Inside the test setup code, the new random provider is registered to ensure that it will get used in applications.

MuBench contains a single misuse using a non-random initialization vector. However, this misuse actually uses the CBC mode for decryption, where using a random initialization vector yields to incorrect decryption results. The guard above does not report a misuse for this source, but a misuse is detected when the source code is changed to encrypt instead. Hence, we can say that the guard is correctly detecting this misuse without flagging a similar usage that does not violate best practices.

CogniCrypt finds a misuse when the code is used for encryption, but reports a false-positive when a non-random IV is used for decryption.

2.8 Passing an empty array to Cipher.doFinal

Version 6 of the IBM JVM implementation had problems when an empty byte array was passed to `Cipher.doFinal`. Even though this problem does not occur on other platforms, programmers can't always know about the actual runtime and thus must consider this limitation regardless.

Once again, this misuse can be covered with a simple requirement. The `engineDoFinal` method in `BaseBlockCipher` is changed to the following:

```
protected verified byte[] engineDoFinal(byte[] input,
    int inputOffset, int inputLen)
  throws IllegalBlockSizeException, BadPaddingException
  require inputLen > 0 {
  // implementation left unchanged.
}
```

In the MuBench dataset, one such misuse is reported. The check identifies this misuse correctly. CogniCrypt did not identify this misuse.

2.9 Misuses that jGuard cannot guard against

While jGuard is able to guard a library implementation to detect most misuses reported by MuBench, not all misuses can be analyzed.

In two instances, a misuse is related to a missing `try/catch` mechanism, which cannot be detected by jGuard as described in section **??**. CogniCrypt did not detect the two misuses either.

In two other cases, a key used for encryption was defined statically the code. As section 2.7 shows, it is possible to enforce that an encryption key is random. However, there are valid uses for a non-random key. For instance, one might want to use a key obtained through an asynchronous key-exchange mechanism [**?**]. This high-level usage violation cannot be detected in the implementation of a library. CogniCrypt reported the two misuses, but also reported a lot of false-positives for this misuse kind in valid code.

A similar misuse reported once is initializing a cipher instance for encryption, but then using it for decryption. Neither CogniCrypt nor jGuard can detect this high-level usage constraint violation.