C ONLINE APPENDIX

C.1 Code Snippets

```
Class T { String algo="AES/CBC/PKCS5Padding";
T mthd1(){ algo = "AES";  return this;} T mthd2(){ algo="DES"; return this;} }
Cipher.getInstance(new T().mthd1().mthd2());
```

Listing 4. Method Chaining ($\mathbf{OP}_5$).

```
val = new Date(System.currentTimeMillis()).toString();
new IvParameterSpec(val.getBytes(),0,8);}
```

Listing 5. Predictable/Non-Random Derivation of Value ($\mathbf{OP}_6$)

```
void checkServerTrusted(X509Certificate[] x, String s)
throws CertificateException {
if (!(null != s && s.equalsIgnoreCase("RSA"))) {
     throw new CertificateException("not RSA");}
```

Listing 6. Exception in an *always-false* condition block ($\mathbf{OP}_7$).

```
public boolean verify(String host, SSLSession s) {
  if(true || s.getCipherSuite().length()>=0)}
    return true;} return false;}
```

Listing 7. False return within an *always true* condition block ($\mathbf{OP}_8$).

```
interface ITM extends X509TrustManager { }
abstract class ATM implements X509TrustManager { }
```

Listing 8. Implementing an Interface with no overridden methods.

```
new HostnameVerifier(){
  public boolean verify(String h, SSLSession s) {
    return true; } };
```

Listing 9. Inner class object from Abstract type ($\mathbf{OP}_{12}$)

```
new X509ExtendedTrustManager(){
  public void checkClientTrusted(X509Certificate[] chain, String a) throws
       CertificateException {}
  public void checkServerTrusted(X509Certificate[] chain, String authType)throws
       CertificateException {}
  public X509Certificate[] getAcceptedIssuers() {return null;} ...};
```

Listing 10. Anonymous Inner Class Object of `X509ExtendedTrustManager` (F10)

```
void checkServerTrusted(X509Certificate[] certs, String s)
   throws CertificateException {
 if (!(null != s || s.equalsIgnoreCase("RSA") || certs.length >= 314)) {
   throw new CertificateException("Error");}}
```

Listing 11. Specific Condition in `checkServerTrusted` method (F17)

```
abstract class AHV implements HostnameVerifier{} new AHV(){
  public boolean verify(String h, SSLSession s)
        return true;}};
```

Listing 12. Anonymous Inner Class Object of An Empty Abstract Class that implements `HostnameVerifier`

```
abstract class AbstractTM implements X509TrustManager{} new AbstractTM(){
  public void checkServerTrusted(X509Certificate[] chain, String authType) throws
      CertificateException {}
    public X509Certificate[] getAcceptedIssuers() {return null;}}};
```

Listing 13. Anonymous inner class object with a vulnerable checkServerTrusted method

```
interface IHV extends HostnameVerifier{} new IHV(){
  public boolean verify(String h, SSLSession s) return true;}};
```

Listing 14. Anonymous Inner Class Object of an Interface that extends HostnameVerifier

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(128); SecretKey secretKey=keyGen.generateKey();
```

Listing 15. Misuse case requiring a trivial new operator

```
if (!className.contains("android."))
    classNames.add(className.substring(1, className.length() - 1)); return classNames;
```

Listing 16. CryptoGuard's code ignoring names with "android"

```
if(!(true || arg0==null || arg1==null)) {
    throw new CertificateException();}
```

Listing 17. Generic Conditions in checkServerTrusted

```
this.name = name == null ? "AES" : name;
this.mode = mode == null ? "CBC" : mode;
this.pad = pad == null ? "PKCS5Padding" : pad;
this.string = StringUtils.format("%s/%s/%s", this.name, this.mode, this.pad);
```

Listing 18. Transformation String formation in Apache Druid similar to **F2** which uses AES in CBC mode with PKCS5Padding, a configuration that is known to be a misuse [36, 79].

```
Class T {
  int i = 0;
  cipher = "AES/GCM/NoPadding";
  public void A(){
      cipher = "AES/GCM/NoPadding";
  }
  public void B(){
      cipher = "AES/GCM/NoPadding";
  }
  public void C(){
      cipher = "AES/GCM/NoPadding";
  }
  public void D(){
      cipher = "AES";
  }
  public String getVal(){
      return cipher
  }
}
Cipher.getInstance(new T().A().B().C().D().getVal() ) ;
```

Listing 19. Iterative Method Chaining

```
Class T {
    int i = 0;
    cipher = "AES/GCM/NoPadding";
    public void A(){
    if ( i == 0){
        if (i == 0){
            if(i == 0){
```

```
                cipher = "AES";
            }
            else{
                cipher = "AES/GCM/NoPadding";
            }
        }
        else{
            cipher = "AES/GCM/NoPadding";
            }
    } else{
        cipher = "AES/GCM/NoPadding";

    }}
    public String getVal(){
        return cipher
    }
}
Cipher.getInstance(new T().A().getVal() ) ;
```

Listing 20. Iterative Conditionals

```
Class T {
    int i = 0;
    cipher = "AES/GCM/NoPadding";
    public String A(){
        return "D";
    }
    public String B(){
        return "E";
    }
    public String C(){
        return "S";
    }
    public void add(){
        cipher = A() + B() + C();
    }
    public String getVal(){
        return cipher
    }
}
Cipher.getInstance(new T().add().getVal() ) ;
```

Listing 21. Method Builder

```
T secure = new T();
T insecure = new T().mthd2();
secure = insecure;
Cipher.getInstance(secure.getVal());
```

Listing 22. Object Sensitive, using the object created in Listing A.1

```
String cryptoVariable = "AES";
char[] cryptoVariable1  = cryptoVariable.toCharArray();
javax.crypto.Cipher.getInstance(String.valueOf(cryptoVariable1));
```

Listing 23. Build Variable

```
javax.crypto.Cipher.getInstance("secureParamAES".substring(11));
```

Listing 24. Substring

```
byte[] cryptoTemp = "12345678".getBytes();
javax.crypto.spec.IvParameterSpec ivSpec = new
     javax.crypto.spec.IvParameterSpec.getInstance(cryptoTemp,"AES");
```

Listing 25. Constant IV

C.2 Additional Implementation and Evaluation Details

C.2.1 Expanded rationale for choosing certain operators. We prioritized misuse cases for inclusion in MASC that are discussed more frequently in the artifacts. For instance, when implementing restrictive operators (Sec. 6.1), we chose the misuse of using `AES` with `ECB` mode, or using `ECB` mode in general, as both misuse cases were frequently mentioned in our artifacts (*i.e.,* in 2 and 11 artifacts respectively). Additionally, we chose the misuse of using `DES` mode, since several crypto-detectors did not consider that not specifying a mode explicitly for encryption defaults to ECB mode. Similarly, we chose the misuse cases of using `MD5` algorithm with the `MessageDigest` API for hashing (5 artifacts), and digital signatures (5 artifacts). When implementing flexible mutation operators (Sec. 6.2), we observed that the majority of the misuse cases relate to improper SSL/TLS verification and error handling, and hence chose to mutate the `X509TrustManager` and `HostnameVerifier` APIs with $\mathbf{OP}_7$ – $\mathbf{OP}_{13}$.

C.2.2 Do we need to optimize the number of mutants generated? MASC generates thousands of mutants to evaluate crypto-detectors, which may prompt the question: should we determine exactly *how many* mutants to generate or optimize them? The answer to this question is no, for two main reasons.

First, MASC generates mutants as per the mutation-scope applied, *i.e.,* for the exhaustive scope, it is natural for MASC to seed an instance of the same mutation at every possible/compilable entry point (including internal methods) in the mutated application. Similarly, for the similarity scope, we seed a mutant besides every similar "usage" in the mutated application. Therefore, in all of these cases, every mutant seeded is justified/necessitated by the mutation scope being instantiated. Any reduction in mutants would require MASC to sacrifice the goals of its mutation scopes, which may not be in the interest of a best-effort comprehensive evaluation. Second, in our experience, the number of mutants does not significantly affect the time to seed taken by MASC. That is, MASC took just 15 minutes to seed over 20000 mutants as a part of our evaluation (see Section 10). Moreover, once the target tool's analysis is complete, we only have to analyze the unkilled mutants, which is a far smaller number than those originally seeded (Section 10). Therefore, in our experience, there is little to gain (and much to lose) by reducing the number of mutants seeded; *i.e.,* we want to evaluate the tools as thoroughly as we can, even if it means evaluating them with certain mutation instances/mutants that may be effectively similar.

That said, from an analysis perspective, it may be interesting to dive deeper into the relative effectiveness of individual features (*i.e.,* operators as well as scopes), even if they are all individually necessary, as each mutation operator exploits a unique API use characteristic, and scopes exploit unique code-placement opportunities, and any combination of these may appear in real programs. However, it would be premature to determine relative advantages among scopes/operators using the existing evaluation sample (i.e., 9 detectors evaluated, 13 open-source apps mutated, 19 misuse cases instantiated, with 12 operators). For instance, mutating other misuse cases, or evaluating another tool, or using a different set of open source apps to mutate, may all result in additional/different success at the feature-level (although overall, MASC would still find flaws, and satisfy its claims). We defer such an evaluation to determine the relative advantages of different mutation features to future work, as described in Section 11.

C.2.3 Further details regarding confirming killed mutants. Matching the mutation log generated by MASC with the reports generated by crypto-detectors is challenging because crypto-detectors often generate reports in heterogeneous and often mutually incompatible ways; *i.e.,* GCS, LGTM, ShiftLeft, and more recently, CogniCrypt generate text files following the recently introduced Static Analysis Results Interchange Format (SARIF) [74] format. However, CryptoGuard, Tool_X, SpotBugs and QARK generate reports in custom report formats, downloadable as HTML, CSV, or

Table 8. Mutants analyzed vs detected by crypto-detectors

Tool	Input Type	Analyzed	Detected
CryptoGuard	`apk` or `jar`	45,763	25,299
Xanitizer [1]	Java Src Code & `jar`	17,788	17,774
CogniCrypt	`apk` or `jar`	23,601	4,576
Tool$_X$ [2]	Android or Java Src Code	9,774	8,547
SpotBugs	`jar`	17,702	13,848
QARK	Java Src Code or `apk`	46,324	7
LGTM	Java Src Code	34,846	21,474
GCS	Java Src Code	34,846	21,440
ShiftLeft	Java Src Code	46,252	35,200
Snyk	Java Src Code	47,002	40,877
DeepSource	Java Src Code	47,002	17,028
Codiga	Java Src Code	26,725	0
SonarQube	Java Src Code	13,749	11,601
Amazon CodeGuru Security	Java Src Code	46,967	840

[1]Xanitizer has been merged to another product and is not included in this extension,
[2]We did not obtain license for this extended study

text, web-based services such as Amazon CodeGuru Security, Snyk, SonarQube, and DeepSource offer results through web-based user-interface, and finally, Xanitizer generates PDFs with source code annotations. We developed a semi-automated implementation that allows us to systematically identify uncaught mutants given these disparate formats. For QARK and CryptoGuard, we wrote custom scripts to parse and summarize their reports into a more manageable format, which we then manually reviewed and matched against MASC's mutation logs. For SARIF formatted reports, we used a VSCode based SARIF viewer [86] that allows iterative searching of logs and tool reports by location. For CogniCrypt, SpotBugs, and Xanitizer, we performed the matching manually since even though they used custom Text or PDF formats, they were generated in such a way that manual checking was trivial. This process is in line with prior work that faces similar challenges [6]. As more tools move to standard formats such as SARIF (which is being promoted by analysis suites such as Github Code Scan) and being adopted by crypto-detectors (*e.g.,* Xanitizer and CogniCrypt adopted SARIF after our 2022 study concluded), we expect the methodology to be fully automated.

C.2.4 Why GCS, LGTM, and QARK fail to detect base cases. In our 2022 study, we observe that GCS and LGTM fail to detect base cases (*i.e.,* $\emptyset$ in Table 3) for **FC3** – **FC5**, although they claim to find SSL vulnerabilities in Java, due to incomplete rulesets (*i.e.,* the absence of several SSL-related rules) [43]. However, we noticed that there was an SSL-related experimental pull request for GCS's ruleset [44, 51] and even upon integrating it into GCS and LGTM, we found both tools to still be vulnerable to the base cases. In our current study, even after we used the Github Code Security with both the *default*, and *security-extended* test suites as of April 2024, it was unable to detect misuse related to `X509TrustManager` and `HostnameVerifier`. Similarly, QARK fails to detect base cases for all flaws in **FC1** and **FC2**, because of its incomplete ruleset [82].

C.3 Types of cases that our SLR approach may miss

Our SLR approach involves manually analyzing each document in an attempt to include all misuse cases, but this extraction of misuse cases is often affected by the context in which they are expressed. For instance, CogniCrypt's core philosophy is whitelisting, which is reflected throughout its papers and documentation. However, there are two ways in which whitelisting is expressed in the paper, one concerning functionality, and another security, *i.e.,* cases of desired behavior expressed in the paper may not always indicate a security best-practice. For instance, the ORDER keyword in the CrySL language initially caused us to miss the PBEKeySpec misuse (now included in the taxonomy), because as defined in the paper, ORDER keyword allows defining "*usage patterns*" that will not break functionality. Thus, as the "usage" patterns were not security misuses (or desired behaviors for security), we did not include them as misuse cases in the taxonomy.However, in a later part of the paper, the ORDER keyword is used to express a security-sensitive usage, for PBEKeySpec, but the difference in connotation is not made explicit. This implicit and subtle context-switch was missed by both our annotators in the initial SLR, but fixed in a later iteration, and misuse cases related to the ORDER keyword were added to the taxonomy.

Similarly when labeling for misuse extraction (Sec. 5.3) we marked each misuse found in a document using common terminology (*i.e.,* labels) across all documents. Thus, if a misuse found in the current document was previously discovered and annotated with a particular label, we would simply apply the same label to the newly found instance.This standard, best-practice approach [62, 71] makes it feasible to extract a common taxonomy from a variety of documents written by different authors, who may use inconsistent terminology. However, a limitation of this generalization is that in a rare case wherein a particular example may be interpreted as two different kinds of misuse, our approach may lose context and label it as only one type of misuse. For instance, based on how the misuse of a "password stored in String" was described in most of the documents we studied, the misuse label of "using a hardcoded password" was applied to identify it across the documents. However, this results in the loss of the additional, semantically different misuse that may still be expressed in terms of a "password stored in String", that passwords should not be stored/used in a String data construct for garbage collection-related reasons. Note that this problem would only occur in rare instances wherein (1) there are multiple contexts/interpretations of the same misuse example, and (2) only one or few document(s) use the additional context. This misuse has also been included in the taxonomy.

C.4 Additional Evaluation Data

Table 9. List of Applications mutated using MASC in previous study (S&P'22 [5]), CLOC = Count Lines of Code from Java Source files only [21], Source = Source Code collected from/Originated From

ID	Name	Type	Source	CLOC
A1	2048	Android	GitHub	136
A2	BMI Calculator	Android	GitHub	145
A3	Calendar Trigger	Android	GitHub	8,553
A4	LocationShare	Android	GitHub	215
A5	NasaApodCL	Android	GitHub	706
A6	AFH Downloader	Android	GitHub	1,657
A7	A Time Tracker	Android	GitHub	2,928
A8	Kaltura Device Info	Android	GitHub	1,049
A9	Protect Baby Monitor	Android	GitHub	625
A10	Activity Monitor	Android	GitHub	1,168
A11	personalDNSfilter	Android	GitHub	8,446
A12	aTalk	Android	GitHub	254,364
A13	Car Report	Android	BitBucket	16,966
	Apache Qpid™ Broker-J			
J14.1	Broker-J - AMQP/JDBC	Java	Apache	597
J14.2	Broker-J - Tools	Java	Apache	1,725
J14.3	Broker-J - HTTP	Java	Apache	24,141
J14.4	Broker-J - Core	Java	Apache	127,280