Automatic Detection and Fixing of Java XXE Vulnerabilities Using Static Source Code Analysis and Instance Tracking

Torstein Molland, Andreas Nesbakken Berger, and Jingyue Li[0000−0002−7958−391X]

Norwegian University of Science and Technology
torstmol@alumni.ntnu.no, aberger@alumni.ntnu.no, jingyue.li@ntnu.no

Abstract. Web security is an important part of any web-based software system. XML External Entity (XXE) attacks are one of web applications' most significant security risks. A successful XXE attack can have severe consequences like Denial-of-Service (DoS), remote code execution, and information extraction. Many Java codes are vulnerable to XXE due to missing the proper setting of the parser's security attributes after initializing the instance of the parser. To fix such vulnerabilities, we invented a novel instance tracking approach to detect Java XXE vulnerabilities and integrated the approach into a vulnerability detection plugin of Integrated Development Environment (IDE). We have also implemented auto-fixes for the identified XXE vulnerabilities by modifying the source code's Abstract Syntax Tree (AST). The detection and auto-fixing approaches were evaluated using typical Java code vulnerable to XXE. The evaluation results showed that our detection approach provided 100% precision and recall in detecting the XXE vulnerabilities and correctly fixed 86% of the identified vulnerabilities.

Keywords: Software security · Instance tracking · Code auto-fixing · XML External Entity · Abstract Syntax Tree

1 Introduction

According to OWASP 2017 [20], XXE attacks [7] [19] are ranked as the fourth most common security risk to web applications. In OWASP 2021 [22], XXE is merged into the security misconfiguration category. XXE can be used for information extraction, Server Side Request Forgery (SSRF), Denial-of-Service (DoS) attack, and remote code execution. The two popular XML vulnerabilities related to the parsing of XML documents are CWE-611 and CWE-776. CWE-611 denotes the vulnerability that occurs when an XML document contains external entities outside of the sphere of control, causing the software to process the XML document to embed incorrect documents into its output [1]. CWE-776 denotes the improper restriction of recursive entity references in Document Type Definitions (DTD) [2]. The code below in Listing 1 shows a Java XML

parser SAXParser (Simple API for XML parser) being instantiated with default parameters. This parser is vulnerable to XXE.

```
1 InputStream is = new FileInputStream(filePath);
2 SAXParserFactory f = SAXParserFactory.newInstance();
3 SAXParser p = f.newSAXParser();
4 PrintHandler h = new PrintHandler();
5 p.parse(is, h);
```

Listing 1: Vulnerable code

If an XML parser is vulnerable to XXE, parsing an XML input like the one in Listing 2 will extract information from the system parsing the XML. In the example shown in Listing 2, the *passwd* file of a Unix system will be read.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!DOCTYPE foo [
    <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "file:///etc/passwd" >
  ]>
3    <foo>&xxe;</foo>
```

Listing 2: Example XML input

By adding the third line in the code, as shown in Listing 3, the SAXParser is made secure and is not vulnerable to XXE.

```
1 InputStream is = new FileInputStream(filePath);
2 SAXParserFactory f = SAXParserFactory.newInstance();
3 f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
4 SAXParser p = f.newSAXParser();
5 PrintHandler h = new PrintHandler();
6 p.parse(is, h);
```

Listing 3: Secure code

Besides being vulnerable to information leakage, the vulnerable parser shown above is also vulnerable to DoS and remote code execution attacks. These attacks could be performed by inputting different XML into the parser. The fixed parser is not vulnerable to any of these attacks.

The study [19] showed that *"all tested Java parsers are vulnerable to instances of DoS, SSRF, and File System Access (FSA) except KXml which is not vulnerable to any attack vector"*. This means that a developer who uses an XML parser without changing the default settings will be vulnerable to XXE without knowing it. This requires the developer to manually add code lines to make the parser secure every time to mitigate the XXE vulnerability. Jan et al. [11] studied the presence of the Billion Laughs (BIL) and XXE attacks in 13 popular parsers. The studied parsers were chosen because they were included in Java, Python, PHP, Perl, and C#. They found that the parsers together had been used over half a million times. It was concluded that *"BIL and XXE attacks are successful in many modern XML parsers. Among the ones selected for experimentation, more than half are vulnerable."* They also checked if open source systems that used the vulnerable XML parsers DocumentBuilder and SAXParser remembered to apply mitigation strategies to defend against these attacks, and found that

98.13% of open source projects had not implemented the known mitigation and were vulnerable to BIL or XXE attacks.

Oliveira et al. [16] extended WS-Attacker for testing the security of web service frameworks. After evaluating Apache Axis 1 and Apache Axis 2, they found that both Apache Axis 1 and Apache Axis 2 were vulnerable to many of the vulnerabilities tested. An extension to the dynamic testing tool WS-attacker for testing DoS attacks against XML parsers was created by Falkenberg et al. [10]. In their evaluation, all the parsers were vulnerable to XXE attacks.

To our knowledge, no existing tool provides sufficient auto-detection and auto-fix functionality for the XXE vulnerabilities. Our research motivation is to further advance software security by improving the detection and auto-fixing of XXE vulnerabilities. We limit our focus on detection and auto-fixing of XXE vulnerabilities in Java code. Java is the second most popular programming language to develop web application [4]. To detect XXE vulnerabilities in Java code, we invented a novel detection mechanism called instance tracking. We proposed the instance tracking approach based on two insights: 1) All Java XML parsers are instance based [5]; 2) Most Java XML parsers are vulnerable and need to be fixed using the known mitigation approaches, i.e., correctly setting the attributes of the parser, after instance initialization [11]. Our implemented vulnerability auto-fixing is based on modifying the AST [3] of the source code. The purpose of auto-fixing is to add code to set the parser's attributes correctly after instance initialization. To help developers detect and auto-fix the XXE vulnerabilities in the first place, we implemented our approaches in FindSecBugs [8] as a proof-of-concept. To evaluate our implementations, we have also extended the test cases in the Juliet Test Suite [14] by adding more test cases to evaluate detection and auto-fixes of XXE vulnerabilities. Our contributions are threefold: 1) A novel instance tracking detection mechanism to detect XXE vulnerabilities, 2) A novel mechanism to auto-fix XXE vulnerabilities based on modifying the source code's AST, 3) A novel test bed to evaluate detection and auto-fixing of XXE vulnerabilities.

The rest of the paper is organized as follows. In Section 2, the research design and implementation is detailed. In Section 3, we explained the test bed and the evaluation results. In Section 4, our methods' strengths and weaknesses are discussed. In Section 5, the conclusion and future work are presented.

2 Research design and implementation

More than 98% of open source projects studied by Jan et al. [11] had not implemented the known mitigation against XXE attacks, i.e., correctly setting the parser's security attributes. Although implementing the mitigation was not tricky, many developers overlooked it. We hope IDE plugins that can help developers detect and auto-fix such vulnerabilities in the first place will significantly reduce the number of XXE vulnerabilities in many projects. In this study, we want to develop IDE plugins to detect and auto-fix prevalent XXE vulnerabili-

ties, i.e., CWE-611, CWE-776, and the ones listed in [11], namely, i.e., various vulnerabilities that can be exploited by BIL and XXE attacks.

2.1 Using instance tracking to detect XXE vulnerability

FindSecBugs already supports detecting vulnerabilities in the seven XML parsers we focus on in this study. However, FindSecBugs' approach is limited to pattern matching. The pattern matching approach does not track which instance the secure method calls, i.e., the ones to set the parser's security attributes, have been called on. Therefore, the pattern matching approach cannot know whether the XXE mitigations, i.e., correctly setting the parser's security attributes, have been appropriately implemented.

We implemented the instance tracking approach to track secure method calls on a particular instance. The instance tracker requires a list of initialization instructions of the instances to track. The instance tracking approach begins with a stack of opcodes acquired from Java bytecode using the Byte Code Engineering Library (BCEL). First, we identify if the opcode is an initialization of an XML parser instance. The identification is made by matching the initialization instruction of the XML parser we focus on, e.g., *DocumentBuilder*. If the opcode is instance initialization, we add the location of code the instance is initialized to a list. The list contains all the XML instances which are tracked and need to be checked as vulnerable or secure in later analysis. If the opcode is not instance initialization, we check if the opcode matches an invocation of a method of interest of the tracked instances. A method of interest is a method that can be vulnerable if the security attributes of a corresponding XML parser are not set correctly before the method is called. For example, the methods of interest for the *SAXParserFactory* include *SAXParserFactory.newSAXParser, SAXParser.parse*, and *SAXParser.getXMLReader*, and the methods of interests for *DocumentBuilderFactory* include *DocumentBuilderFactory.newDocumentBuilder* and *DocumentBuilder.parse*. The list of methods of interest of each XML parser is derived from the OWASP 2017 [20] and Oracle documents. If a method of interest is found for a tracked instance, we attach the return value of the method of interest to the tracked instance. If there are multiple invocations of the methods of interest on the same tracked instance, we will keep only the return value of one of them. For example, in the code in Listing 4, *documentBuilder1* and *documentBuilder2* are invocation of the *DocumentBuilderFactory.newDocumentBuilder* method of the same *DocumentBuilderFactory* instance. In such a case, we attach only *documentBuilder1* to the tracked *DocumentBuilderFactory* instance.

For every method of interest attached to the tracked instance, we check if proper security attributes of the tracked XML parser instance are set. In some cases, only a singular call is needed to set the security attributes. For example, if the method of interest is *newDocumentBuilde*r and it is attached to the tracked instance *DocumentBuilderFactory*, we will check if the *DocumentBuilderFactory.setFeature* method is called with correct parameter setup, i.e., *XMLConstants.FEATURE_SECURE_PROCESSING* is set with the value "true ", be-

fore calling the *DocumentBuilderFactory.newDocumentBuilder* method. In some other cases, multiple calls have to be made to set all relevant security attributes. For example, for the *XMLStreamReader* parser, the *setProperty* method has to be called several times to set up multiple parameters, e.g., *SUPPORT_DTD* and *IS_SUPPORTING_EXTERNAL_ENTITIES*, to make the invocation to its *createXMLStreamReader* method to be secure.

After all the methods of interest of the tracked instance are checked with security attributes setup, the tracked instances without all security attributes set correctly before the method of the interest of the instance are called will be reported as vulnerable.

```
1 DocumentBuilderFactory dbFactory =
      DocumentBuilderFactory.newInstance();
2 DocumentBuilder documentBuilder1 =
      dbFactory.newDocumentBuilder();
3 documentBuilder1.parse(<inputFile>);
4 DocumentBuilder documentBuilder2 =
      dbFactory.newDocumentBuilder();
5 Document doc = documentBuilder2.parse(<inputFile>);
```

Listing 4: An example code

2.2 Auto-fixing identified XXE vulnerability

We traverse and modify the code's AST to auto-fix the instance-related vulnerabilities. There are many different APIs and features that need to be set for different parsers to secure. An auto-fix mechanism will help reduce the complexity, time, and effort spent identifying the different parsers' correct fixes. The approach includes three steps.

1. Find node to insert auto-fix. In this step, we find the AST node to insert the auto-fix by using the location of the vulnerability reported by a vulnerability detector. First, the AST node of the vulnerable source code line is obtained. Then, a check is made to identify if the AST node is a call of a method of interest (e.g., SAXParser.parse, as explained in Section 2.1) or a call of the XML parser initialization (e.g., SAXParserFactory.newInstance). If the call is a method of interest, then the auto-fix approach attempts to traverse the predecessors of the AST node until it finds the node where the XML parser is initialized. The location to insert auto-fix on is the node to initialize the XML parser. For the vulnerable parser example shown in Listing 1, the AST node of *p.parse()* is found to be initialized by *f.newSAXParser()*, and *f* is found to have been initialized by *SAXParserFactory.newInstance()*. The type of the *SAXParserFactory.newInstance()* node is SAXParserFactory. Thus, the node *SAXParserFactory.newInstance()* is the location where the auto-fix should be inserted.
2. Prepare node for auto-fix insertion. In this step, a check is first made to identify if there are multiple methods invoked on the AST node. If there are, then the AST node of the initialization of the instance and the remaining calls are split up using an auxiliary variable. For example, for *SAXParserFactory.newInstance().newSAXParser().parse()*, the fix should be inserted

on the AST node between *SAXParserFactory.newInstance()* and *newSAXParser().parse()*. Hence, the *SAXParserFactory.newInstance()* is first stored in a temporary variable, e.g., *f*, and then the remaining calls are called on this variable, e.g., *f.newSAXParser().parse()*.

3. Apply auto-fix. In this step, we modify the AST by inserting the AST nodes corresponding to the missing calls of methods to set the security attributes correctly after the instance's initialization. The result of this is equivalent to inserted line 3 in the secure code example shown in Listing 3. When inserting the missing calls to set the security attributes, we set the corresponding security parameters' values as specified by OWASP 2017 [20] and Oracle. The parameters and their values after auto-fixing are shown in Table 1. For each possible missing call to add, we pre-defined the necessary imports related to the call. The imports are added automatically by calling the ASTUtil() function of the FindSecBugs, after the missing call related to the imports are inserted.

Table 1: The security parameters and their settings after auto-fixing. The functions to call are either setFeature or setProperty, Value T=Ture, F=False

Parser	**Parameter, Value**
DocumentBuilder	XMLConstants.FEATURE_SECURE_PROCESSING, T
XMLStreamReader	XMLInputFactory.SUPPORT_DTD, F
XMLStreamReader	XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, F
XMLEventReader	XMLInputFactory.SUPPORT_DTD, F
XMLEventReader	XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, F
FilteredReader	XMLInputFactory.SUPPORT_DTD, F
FilteredReader	XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, F
SAXParser	XMLConstants.FEATURE_SECURE_PROCESSING, T
XMLReader	http://apache.org/xml/features/disallowdoctype-decl, T
XMLReader	http://apache.org/xml/features/nonvalidating/load-external-dtd, F
XMLReader	http://xml.org/sax/features/externalgeneral-entitiesG, F
XMLReader	http://xml.org/sax/features/externalparameter-entities, F
Transformer	XMLConstants.FEATURE_SECURE_PROCESSING, T

3 Evaluation

One challenge of implementing such a plugin is that there exists no solid test bed that we can use to evaluate it. The flow variants from the Juliet Test Suite could not sufficiently detect XXE vulnerabilities. The Juliet Test Suite only covers test cases with one parser wrapped in different control flows, e.g., for-loops and if-statements. So, we implemented a test suite based on the Juliet Test Suite for evaluating our XXE vulnerability detection and auto-fixing plugin. We first chose 17 flow variants from Juliet Test Suite V1.3 [15] to create our test cases

(so called existing Juliet Test Cases). The test cases were created for the seven Java XML parsers, namely, **DocumentBuilder**,**XMLStreamReader**, **XMLEventReader**, **FilteredReader**, **SAXParser**, **XMLReader**, and **TransformerFactory**. We implemented additional test cases (so called added test cases) for each of the seven parsers with more complex data flows to cover the XXE vulnerabilities listed in [11]. For each of the seven Java XML parser, we added up to 11 test cases. These test cases included different ways of initializing an object and invoking methods on an object instance, which may affect the XXE vulnerability detection and auto-fix performance. Examples of our added test cases are as follows.

- Six test cases (Test case 1 to Test case 6) with variations of class field and method variable were added. In the test case example in Listing 5, *SAXParser p* and *SAXParserFactory f* are either a class field or a method variable:

```
1 // Test case 1
2 InputStream is = new FileInputStream(filePath);
3 // Factory initialized into method variable
4 SAXParserFactory f = SAXParserFactory.newInstance();
5 // parser initialized into class field
6 p = f.newSAXParser();
7 PrintHandler h = new PrintHandler();
8 p.parse(is, h);
```

Listing 5: An example of test cases 1 to 6

- Four test cases (Test case 7 to Test case 10) with multiple parsers that were made secure and insecure in the same method were added. In the test case example in Listing 6, *SAXParser p1* is vulnerable. *SAXParser p2* is secure, since the factory f has been made secure prior to initializing *SAXParser p2*:

```
1 // Test case  7
2 InputStream is = new FileInputStream(filePath);
3 SAXParserFactory f = SAXParserFactory.newInstance();
4 SAXParser p1 = f.newSAXParser();
5 PrintHandler h1 = new PrintHandler();
6 p1.parse(is, h1); // Insecure
7 f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,
                true);
8 SAXParser p2 = f.newSAXParser();
9 PrintHandler h2 = new PrintHandler();
10 p2.parse(is, h2); // Secure
```

Listing 6: An example of test cases 7 to 10

- One test case (Test case 11) where an XML parser and an object with the same secure method as the XML parser were added. In the test case shown in Listing 7, the *setFeature()* method should be called on the factory. However, the *setFeature()* method is called on an irrelevant object.

```
1 // Test case 11
2 Bar b = new Bar();
3 // Calls the setFeature method with correct parameters
4 // but on the wrong object
5 b.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,
                true);
6 InputStream is = new FileInputStream(filePath);
7 SAXParserFactory f =  SAXParserFactory.newInstance();
8 SAXParser p = f.newSAXParser();
9 PrintHandler h = new PrintHandler();
10 p.parse(is, h); // Insecure
```

Listing 7: Test cases 11

Some test cases do not apply to a particular XML parser. For example, test cases that test if the detector can detect multiple uses of the same parser are not applicable to test the detection performance of the *XMLStreamReader* parser. This is because *XMLStreamReader* uses an iterator to parse the XML [9]. When the iterator reaches the end of the XML document, it cannot be reset. This means that to parse an XML document multiple times, the old parser instance cannot be used, and a new parser needs to be created. Test cases that test if the detector can correctly identify secure instances from vulnerable instances created using the same factory cannot test the detection performance on the *XMLReader* parser. It is because an *XMLReader* parser is initialized directly without first initializing a factory [6]. Each added test case had only one method but could include multiple instances of parsers and other objects. This means that each test case may include more than one XXE vulnerability to be detected and fixed. The number of vulnerabilities of each test case for the seven parsers is shown in Table 2.

Table 2: Number of XXE vulnerabilities in each added test case

Parser	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**Sum**
DocumentBuilder	1	1	1	1	1	1	1	2	2	4	1	16
XMLStreamReader	1	1	1	1	1	1	1	2	0	0	1	10
XMLEventReader	1	1	1	1	1	1	1	2	0	0	1	10
FilteredReader	1	1	1	1	1	1	1	2	0	0	1	10
SAXParser	1	1	1	1	1	1	1	2	2	4	1	16
XMLReader	0	0	0	1	0	1	1	2	2	4	1	12
Transformer	2	2	2	2	2	2	2	4	4	8	2	32

We have also implemented test cases for checking if the code is still vulnerable to XXE after auto-fix. An example test case is shown in Listing 8.

```
@Test
public void vulnerable() {
  Boolean vulnerable = true;
  try {
   CWE611_XML_External_Entities__SAXParser_01 parser
    = new CWE611_XML_External_Entities__SAXParser_01();
   String res = parser.bad("bad.xml");

   if(res.equals("vulnerable")) {
     vulnerable = true;
   } else {
     vulnerable = false;
   }

 } catch (SAXParseException e) {
       vulnerable = false;
 }catch (Throwable e) {
   e.printStackTrace();
 }
```

```
19  assertFalse(vulnerable,
20    "Parser should not be vulnerable to XXE");
21 }
```

Listing 8: Test case after auto-fix

To verify if the vulnerability has been fixed, the test attempts to parse an XML file, i.e., the *bad.xml* file, with external entities referring to another file. The path the external entitiy refers to here is *file.txt*, as shown in Listing 9.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE foo [
 <!ELEMENT foo ANY >
 <!ENTITY xxe SYSTEM "file:///some/path/file.txt" >
 ]>
6 <foo>&xxe;</foo>
```

Listing 9: An XML with external entities

The content of the file *file.txt* is a string "vulnerable". If the test returns the contents of the file, i.e., *file.txt*, the external entity is referring to, which indicates that the parser is still vulnerable to XXE. If the parser throws a specific exception, e.g., *SAXParserException*, when trying to read *file.txt*, then it shows that the parser is secure. Since the external entity called should not be parsed, the test case parsing it should raise an exception. Therefore, the tests validating the security of the test cases also check for raised exceptions to see if the parser is configured correctly. In addition, the test cases check if the fixes preserved the original functionality of the code before fixing, i.e., the auto-fixes do not bring negative impacts. To verify if the code's functionality is preserved after the fix, we run a test parse on an XML file without external entities. After auto-fixing, the XML file should be parsed correctly without returning any exceptions. If the XML file's content is returned, it shows that the test case's functionality is preserved after the auto-fix. After running all the test cases, the output is a list showing the functions that are broken and the functions that are still vulnerable to XXE after the auto-fix is applied. This data can then be used to calculate the number of successful, missed, and incorrect fixes.

We evaluated how well FindSecBugs detects these vulnerabilities before and after using the instance tracking approach. The tests are based on the existing Juliet test cases and our added test cases. FindSecBugs using existing Juliet test cases return 100% precision and recall before and after implementing the instance tracking approach. The true positives (TP), false positives (FP), false negatives (FN), precision, and recall of test FindSecBugs on our added test cases before implementing the instance tracking approach are shown in Table 3. The precisions of the detection results of FindSecBugs XXE detectors on these added test cases are still high. However, the high number of false negatives shows that the detectors failed to detect a lot of vulnerabilities. All of the detectors were able to handle added test cases 1 to 6, which tested different ways of initializing the parser but fell through on added test cases 7 to 11. The reason for the false negatives in added test cases 7 to 11 is that the existing FindSecBugs XXE detectors do not keep track of which instance the secure method calls have been

called on. The pseudocode of the FindSecBugs XXE detector is shown in Listing 10 and illustrates that it overlooks the insecure use of multiple parsers within the same method. The FindSecBugs XXE detectors will try to identify the secure call, i.e., the calls to set the security attributes of a parser. Once the secure call is found, the parser will be set as secure. Without tracking all the instances, FindSecBugs XXE detectors will miss other insecure instances.

```
for each opcode in method do
    if name of opcode equals name of parse method
      for each method call in method_callgraph do
        if the method name equals name of a secure call
          check parameters of method to see if
         the parameter values are correct
         end if
      end for
    end if
  end for
if not all secure calls are found
    report vulnerability
end if
```

Listing 10: FindSecBugs XXE detector pseudocode

Table 3: Summary of evaluating FindSecBugs using the added test cases before implementing the instance tracking approach

Parser	**TP**	**FP**	**FN**	**Precision**	**Recall**
DocumentBuilder	6	0	10	100%	38%
XMLStreamReader	6	0	4	100%	60%
XMLEventReader	6	0	4	100%	60%
FilteredReader	6	0	4	100%	60%
SAXParser	6	0	10	100%	38%
XMLReader	2	0	10	100%	17%
Transformer	20	0	12	100%	63%
Sum	52	0	54	100%	48%

A summary of the evaluation of FindSecBugs after using the instance tracking approach on the added test cases is shown in Table 4. The instance tracking-based vulnerability detectors identified all XXE vulnerabilities in the added test cases without reporting any false positives. The instance tracking-based detectors were also evaluated on the existing Juliet test cases. The results illustrate that the instance tracking-based detectors are still able to identify all the XXE vulnerabilities there without reporting any false positives.

The results of evaluating the performance of FindSecBugs XXE detectors before and after using the instance tracking approach show that the overhead introduced by the instance tracking is 0.25s to check through all our test cases included XXE vulnerabilities. To run through all other Juliet test cases, which has more than five millions line of code (LOC), the overhead introduced by the instance tracking is only 20 seconds. The execution time was measured on a PC with 16Gb of memory and a 3.9GHz CPU using Windows 10 Pro.

Table 4: Summary of evaluating FindSecBugs using the added test cases after implementing the instance tracking approach

Parser	**TP**	**FP**	**FN**	**Precision**	**Recall**
DocumentBuilder	16	0	0	100%	100%
XMLStreamReader	10	0	0	100%	100%
XMLEventReader	10	0	0	100%	100%
FilteredReader	10	0	0	100%	100%
SAXParser	16	0	0	100%	100%
XMLReader	12	0	0	100%	100%
Transformer	32	0	0	100%	100%
Sum	106	0	0	100%	100%

A summary of the evaluation of the auto-fix on the added test cases created is shown in Table 5. There is a high number of successful fixes for all the parsers. There were no missed fixes. The incorrect fixes are due to the auto-fix not removing or modifying code that makes a factory explicitly vulnerable. A simplified example of such a case is shown in Listing 11. The fix is inserted on the second line, making the factory secure, which in turn makes parser *p1* secure. However, line 4 makes the factory insecure again, which in turn makes parser *p2* insecure. If line 4 is manually removed by a developer, then the fix inserted on line 2 will make parsers *p1* and *p2* secure.

```
1 SAXParserFactory f = SAXParserFactory.newInstance();
2 f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
3 SAXParser p1 = f.newSAXParser();
4 f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);
5 SAXParser p2 = f.newSAXParser();
```

Listing 11: Example of code fixed incorrectly

The instance tracking-based detectors will keep reporting these parsers as vulnerable and notify the developer that they need to remove the insecure code for the auto-fixes to be effective. Technically, our auto-fixing approach can be slightly updated to remove the insecure code, e.g., the fourth line in Listing 11. Concerning the strategy of auto-fixing, we chose to be conservative. Adding the code to set the security attributes is always safe and makes the system more secure. Deleting or changing the security attributes setup without developer's consensus can mess up the code. A future improvement could be to give the developer the recommendation to remove the vulnerable code, i.e., the fourth line in Listing 11, as a complement to the auto-fix. However, as part of our current auto-fixing solution evaluations, these fixes were still regarded as incorrect fixes.

The auto-fixes were also evaluated on existing Juliet test cases. The results of this evaluation are shown in Table 6, and illustrate that the auto-fixes could fix all the vulnerabilities in these test cases without missed or incorrect fixes.

The result of the auto-fixes' performance for XXE vulnerabilities show that auto-fixing close to 30 vulnerabilities takes around a second for each parser.

Table 5: Summary of the successful fixes, missed fixes, and incorrect fixes after evaluating the auto-fixes on the added test cases

Parser	**Successful**	**Missed**	**Incorrect**
DocumentBuilder	14	0	2
XMLStreamReader	9	0	1
XMLEventReader	9	0	1
FilteredReader	7	0	3
SAXParser	14	0	2
XMLReader	10	0	2
Transformer	28	0	4
Sum	91	0	15

Table 6: Summary of the successful fixes, missed fixes, and incorrect fixes after evaluating the auto-fixes on the existing Juliet test cases

Parser	**Successful**	**Missed**	**Incorrect**
DocumentBuilder	17	0	0
XMLStreamReader	17	0	0
XMLEventReader	17	0	0
FilteredReader	17	0	0
SAXParser	17	0	0
XMLReader	17	0	0
Transformer	34	0	0
Sum	136	0	0

4 Discussion

This section will discuss the strengths and weaknesses of our approach and compare it with related work.

4.1 Pros and cons of our XXE vulnerability detector

All the existing Juliet test cases and the added test cases in our study are handled by instance tracking. The instance tracking approach handles parameters, secure and insecure method calls, and singular and multiple calls to determine if a parser is secure. The sequence of the calls on the instance is kept track of, which means that the approach can identify when an instance is vulnerable and secure. Because it keeps track of the calls performed on different instances, it can also know which parsers within a method are vulnerable and secure. Compared with existing Java XXE vulnerability detectors, e.g., those implemented in FindSecBugs, the instance tracking approach can handle more complex control and data flow variants in addition to the simplest forms of Java XXE vulnerabilities with no false positives. Another strength of the instance tracking we implemented for XXE is that it is generalizable to other vulnerabilities with insecure

instances, such as insecure cookies. In Java, cookies are not set as secure by default [21]. After creating a cookie, the method *setSecure(true)* needs to be called on the cookie instance to make it secure. Therefore, the cookie can be viewed as a vulnerable instance if it misses the *setSecure* call. Such a vulnerability can be detected and auto-fixed by the instance tracking approach presented in this paper. Other instance-based vulnerabilities, e.g., missing httpOnly attributes, can also be handled by the instance tracking approach due to this being an instance-based vulnerability. In this study, we demonstrated the instance tracking approach using FindSecBugs. We can also implement the instance tracking approach based on other Java program analysis frameworks, e.g., Soot [12]. The instance tracking approach can detect the most popular Java XXE vulnerabilities identified by Jan et al. [11], which cover 98.13% of their studies on open source projects.

A weakness of the instance tracking-based detection is that it takes longer to run than the existing XXE detectors. However, our evaluation results show that the overhead is trivial.

4.2 Pros and cons of our XXE vulnerability auto-fixing tool

Existing auto-fixing tools provide unspecific auto-fix suggestions and list all the available auto-fixes. ASIDE [23] and ESVD [18] use ESAPI to provide their auto-fixes. However, neither of them presents the auto-fixes applicable to a certain vulnerability but instead lists all the auto-fixes ESAPI supports, making it difficult for developers to know which auto-fix to apply. Our auto-fix tool allows bulk auto-fixes of Java XXE vulnerabilities. When fixing source code using ASTs, developers are assured that the code change will not break the code's semantics. This means that the fix will not leave any incorrect tokens such as curly braces or commas. Our implementation of the auto-fix approach has been done in a generalizable manner. Thus, several missing functions and security feature setup shown in Table 1 can be added.

Our auto-fixes are backward compatible with the existing XXE vulnerability detectors in FindSecBugs, which means that they can be applied using the existing detectors and the instance tracking-based detectors we have implemented. The issue of making our tool backward compatible with FindSecBugs is that FindSecBugs reports a vulnerability where it can be exploited, not where the fix should be inserted. These two locations may differ. For the example shown in Listing 1, the vulnerability is reported on line 5 on the SAXParser instance, but the fix is inserted on line 3 on the SAXParserFactory instance. Thus, our auto-fix tool has to traverse the AST to find the location of the instance to insert the fix.

As explained in Section 2.2, our AST-based auto-fix approach does not fix parsers that have been made explicitly vulnerable through calls to insecure methods because of choice of being conservative when auto-fixing code. The insecure method calls need to be manually removed by a developer for the fixes to be effective.

4.3 Pros and cons of our test bed

Compared with existing test beds, e.g., Juliet Test Suite [15], WebGoat [17], and ManyBugs [13], our test bed allows researchers to quickly and easily make a thorough evaluation of their XXE detectors and auto-fixing tool. The test bed also has more robust testing of intraprocedural data flows than the Juliet Test Suite. Ideally, we shall use a large Java application to evaluate our approach and tool. However, to our knowledge, no existing Java applications cover all the test cases we have added in our test bed. It means that the Java application may not bring better insights on the precision and recall of the tool.

Our implementation of the instance-based vulnerability detection and auto-fixing was based on FindSecBugs and the Eclipse IDE and its APIs. This could mean that the results discovered for adding auto-fixes to Eclipse-based on FindSecBugs might not be applicable for other analysis tools and other IDEs. We used generalized approaches such as modifying an AST and using data flow analysis, which can possibly be generalized to other IDEs.

5 Conclusion and Future Work

Studies show that Java XXE vulnerabilities are prevalent, and many of the vulnerabilities are caused by missing proper security attributes setting after instance initialization. This study proposed a novel instance tracking-based approach to enhance the existing Java XXE vulnerability detectors. We also proposed a novel auto-fix approach based on AST for fixing the identified Java XXE vulnerabilities. Our detection and auto-fixing methods are implemented as extensions of an existing IDE plugin, i.e., FindSecBugs. We evaluated our proposed approaches using a test bed, including instance-related vulnerabilities derived from several XXE CWEs. Our results show that instance tracking performed significantly better than FindSecBugs with high precision and recall values. The auto-fixes were able to fix many vulnerabilities successfully. The performance impact of instance tracking was also evaluated and shown to be negligible.

Our future work is to add support for more parsers to the test bed, detectors, and auto-fixes. Currently, FindSecBugs reports a vulnerability where it can be exploited, not where the fix should be inserted. An empirical study will be conducted to figure out where developers want the bug to be reported by the detectors. The study will give interesting results to help make the vulnerability detectors and auto-fixing tools more user-friendly. In addition, we will pilot the IDE plugins in an experimental or industrial setting to study whether IDE plugins implemented in this study help developers detect and auto-fix such vulnerabilities in the first place.

Bibliography

[1] Cwe-611: Improper restriction of xml external entity reference (2019), https://cwe.mitre.org/data/definitions/611.html, accessed Jun 01, 2020

[2] Cwe-776: Improper restriction of recursive entity references in dtds ('xml entity expansion') (2019), https://cwe.mitre.org/data/definitions/776.html, accessed Jun 01, 2020

[3] Ast (2020), https://en.wikipedia.org/wiki/Abstract_syntax_tree, accessed 21 Oct, 2020

[4] Java for web app. development (2020), https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019, accessed 21 Oct, 2020

[5] Java xml parser (2020), https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/parsers/package-summary.html, accessed 21 Oct, 2020

[6] XMLReaderFactory (Java SE 13 & JDK 13) (2020), https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/org/xml/sax/helpers/XMLReaderFactory.html, accessed May 25, 2020

[7] Xxeattack (2020), https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing, accessed 21 Oct, 2020

[8] Arteau, P.: Find Security Bugs (2019), https://find-sec-bugs.github.io/, accessed September 22, 2019

[9] Corporation, O.: XMLStreamReader (Java SE 13 & JDK 13) (2020), https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/stream/XMLStreamReader.html, accessed May 28, 2020

[10] Falkenberg, A., Mainka, C., Somorovsky, J., Schwenk, J.: A new approach towards dos penetration testing on web services. In: 2013 IEEE 20th International Conference on Web Services. pp. 491–498 (2013)

[11] Jan, S., Nguyen, C.D., Briand, L.: Known xml vulnerabilities are still a threat to popular parsers and open source systems. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. pp. 233–241 (Aug 2015). https://doi.org/10.1109/QRS.2015.42

[12] Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective

[13] Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W.: The manybugs and introclass benchmarks for automated repair of c programs. IEEE Transactions on Software Engineering **41**(12), 1236–1256 (Dec 2015). https://doi.org/10.1109/TSE.2015.2454513

[14] NIST: Juliet test suite v1.2 for java user guide (2012), https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf, accessed May 12, 2019

[15] NIST: Test suites (2017), https://samate.nist.gov/SRD/testsuite.php, accessed October 29, 2019

[16] Oliveira, R.A., Laranjeiro, N., Vieira, M.: Wsfaggressor: An extensible web service framework attacking tool. In: Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference. MIDDLEWARE '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2405146.2405148, https://doi.org/10.1145/2405146.2405148

[17] OWASP: Webgoat project (2019), https://www2.owasp.org/www-project-webgoat/, accessed December 4, 2019

[18] Sampaio, L., Garcia, A.: Exploring context-sensitive data flow analysis for early vulnerability detection. Journal of Systems and Software **113**, 337–361 (2016). https://doi.org/https://doi.org/10.1016/j.jss.2015.12.021, http://www.sciencedirect.com/science/article/pii/S0164121215002873

[19] Späth, C., Mainka, C., Mladenov, V., Schwenk, J.: Sok: Xml parser vulnerabilities. In: WOOT (2016)

[20] The OWASP Foundation: Owasp top 10 - 2017 (2017), https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_%28en%29.pdf.pdf, accessed January 21st, 2019

[21] The OWASP Foundation: Secure Cookie Flag (2020), https://owasp.org/www-community/controls/SecureFlag, accessed Jun 01, 2020

[22] The OWASP Foundation: Owasp top 10 - 2021 (2021), https://owasp.org/www-project-top-ten/, accessed August 29, 2021

[23] Xie, J., Chu, B., Lipford, H.R., Melton, J.T.: ASIDE: IDE support for web application security. In: ACM International Conference Proceeding Series. pp. 267–276 (2011). https://doi.org/10.1145/2076732.2076770