Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Andreas Nesbakken Berger and Torstein Molland

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

Master's thesis in Computer Science
Supervisor: Jingyue Li
June 2020

NTNU
Norwegian University of Science and Technology

Andreas Nesbakken Berger and Torstein Molland

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

Master's thesis in Computer Science
Supervisor: Jingyue Li
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Acknowledgment

We would like to thank Associate Professor Jingyue Li at the Department of Computer Science at The Norwegian University of Science and Technology for his support on this project and his feedback and advice. We would also like to thank Associate Professor Babak A. Farshchian at the Department of Computer Science at The Norwegian University of Science and Technology for his feedback on our research plan.

Abstract

Web security is an important part of any web-based software system. XML External Entity attacks is one of the biggest security risks for web applications, both according to OWASP and MITRE. A successful XML External Entity attack can have severe consequences like denial of service, remote code execution, and information extraction. There has been done little research into tool support for fixing of XML External Entity attacks. There has been done some research into detection of XXE and fixing of other vulnerabilities like SQL Injection and Cross-Site Scripting, but not XML External Entities. To be better able to advance the field of automatic vulnerability fixing, we wish to find out how XXE vulnerability detection can be improved, and how automatic fixing of these vulnerabilities can be done. To be able to evaluate a new auto fixing tool we also aim to research how a test bed can be designed for evaluating auto fixing tools. Based on a structured literature review a number of vulnerability detection techniques were discovered. We aim to use these techniques to improve the detection in state of the art auto fixing tools, and then implement automatic fixing of these vulnerabilities. Before creating the detection and auto fixing, a test bed will be created containing test cases vulnerable to XXE that can evaluate the effectiveness of the detection and the fixes. Our research contributes a novel instance tracking method to detect XXE vulnerabilities for the tool FindSecBugs, a novel extension for the tool adding auto fixes for the detected vulnerabilities, and a novel test bed for the evaluation of the detection and the auto fixes of XXE. Our analysis of different detection methods also shows that instance tracking is an effective way to improve the detection of XXE. We also discuss both the effectiveness and shortcomings of using abstract syntax trees (ASTs) for fixing vulnerable code, and a design for test beds that are used specifically for evaluation of auto fixes and detection.

Sammendrag

Programvaresikkerhet er en viktig del av ethvert moderne programvaresystem. XML External Entity angrep er en av de største sikkerhetsrisikoene for web applikasjoner, både i følge OWASP og MITRE. Et vellykket XML External Entity angrep kan ha alvorlige konsekvenser, som tjenestenektangrep, remote code execution, og informasjonsekstraksjon. Det har blitt gjort lite forskning på verktøystøtte for fiksing av XML External Entity angrep. Det har blitt gjort noe forskning på deteksjon av XXE, og fiksing av andre sårbarheter som SQL Injection og Cross-Site Scripting, men ikke for XML External Entities. For å forbedre verktøystøtte for automatisk fiksing av sårbarheter, ønsker vi å finne ut hvordan deteksjon av XXE sårbarheter kan forbedres, og hvordan automatisk fiksing av disse sårbarhetene kan bli utført. For å evaluere det nye auto-fiksing verktøyet ønsker vi også å utforske hvordan test beds kan designes for evaluering av auto-fiks verktøy. Basert på et strukturert litteratursøk ble en rekke deteksjonsteknikker for sårbarheter identifisert. Vi ønsker å bruke disse teknikkene til å forbedre deteksjonen av sårbarheter, og deretter implementere automatisk fiksing av disse sårbarhetene. Før deteksjon og autofiksing kan implementeres, lagde vi en test bed som inneholder test cases som er sårbare mot XXE, og som har muligheten til å evaluere om fiksene klarte å gjøre koden sikker og om funksjonaliteten i koden ble bevart. Vår forskning bidrar med en ny instance tracking metode for å detektere XXE sårbarheter i FindSecBugs, en utvidelse til FindSecBugs som legger til automatisk fiksing for de detekterte sårbarhetene, og en ny test bed for å evaluere deteksjonen og fiksing av XXE. Vår analyse av ulike deteksjonsmetoder viser også at instance tracking er en effektiv måte å forbedre deteksjon av XXE. Vi diskuterer også fordeler og ulemper ved å bruke abstrakte syntaks-trær for å fikse sårbarheter i kildekode. Vi presenterer også et design for en test bed som kan brukes spesifikt for å evaluere autofikser og deteksjon.

Table of Contents

List of Tables

List of Figures

Abbreviations

API	=	Application Programming Interface
AST	=	Abstract Syntax Tree
ASIDE	=	Application Security plugin for Integrated Development Environment
AUC	=	Area Under the Curve
BCEL	=	Byte Code Engineering Library
CFG	=	Control Flow Graph
CSRF	=	Cross-Site Request Forgery
CWE	=	Common Weakness Enumeration
DBMS	=	Database Management System
DFG	=	Data Flow Graph
DOM	=	Document Object Model
DOS	=	Denial Of Service
DTD	=	Document Type Definition
ESAPI	=	Enterprise Security API
ESVD	=	Early Security Vulnerability Detector
IDE	=	Integrated Development Environment
HTTP	=	HyperText Transfer Protocol
JDK	=	Java Development Kit
JVM	=	Java Virtual Machine
LIFO	=	Last In First Out
OWASP	=	Open Web Application Security Project
PHP	=	PHP Hypertext Preprocessor
ROC	=	Receiver Operating Characteristic
SAT	=	Static Analysis Tool
SQL	=	Structured Query Language
SQLi	=	SQL Injection
SSRF	=	Server-Side Request Forgery
SSO	=	Security Sensitive Operations
TP	=	True Positive
FP	=	False Positive
FN	=	False Negative
UML	=	Unified Modeling Language
XSS	=	Cross Site Scripting
XSLT	=	Extensible Stylesheet Language Transformations
XXE	=	XML External Entities

Chapter 1

Introduction

In the prestudy [89], it was found that a lot of research has been done into detection and auto fixing of security vulnerabilities. As shown in Table A.1, numerous tools and approaches for detecting security vulnerabilities were found. However, most of these focused on the detection of SQL Injections, Cross-Site Scripting attacks, and other injection vulnerabilities. Additionally, the maturity of different tools was found to differ substantially. For instance, detection and auto fixing of injection-based vulnerabilities were found to be quite mature, whereas detection of XXE was found to be lacking. Existing tools for detecting XML related vulnerabilities were identified, however, the detection mechanism was found to employ only the most basic detection mechanism based on pattern matching. No auto fix mechanism was found for XML External Entities.

In the prestudy, it was also found that many researchers evaluate their auto fixing tools on sub-optimal test beds. As shown in Table A.1, many evaluate their approaches on open source software or proprietary test beds making it difficult to reproduce their results. These test beds also do not provide information about the effectiveness of the fixes. Others evaluate their approaches on test beds created specifically for evaluating the performance of the detection mechanism. However, these test beds make it difficult to evaluate the performance of auto fixes. The variation in evaluation methods makes it challenging to compare different detection and auto fixing tools from different researchers.

The Open Web Application Security Project (OWASP) top 10 project [125] shows an overview of the top 10 software vulnerability categories. XML External Entity attacks is ranked as the fourth most critical security risk to web applications. This is due to both the popularity and the severe consequences of such an attack being successfully carried out. XXE can be used for information extraction, Server Side Request Forgery (SSRF), denial of service attacks, and remote code execution. MITRE classifies XXE as part of the top 25 most dangerous software errors in their list from 2019 [86].

In a study done by Späth et al. [120], all the XML parsers in Java were found to be vulnerable to XXE by default. This requires the developer to manually add lines of code to make the parsers secure every time the parser is used to mitigate the vulnerability. This means that a developer who uses an XML parser without changing the default settings

will be vulnerable to XXE without knowing it. Present-day tools can detect and show the location of XXE vulnerabilities in code. However, these tools require developers to know how to correctly change settings of the XML parser used to make them secure. This can be challenging since mitigation strategies vary from parser to parser, and the resources online do not always agree on the best mitigation strategy.

For the project, three research questions are proposed:
RQ1: How can a test suite for evaluating web sec auto fixes be designed for XML External Entity attacks?
RQ2: How can detection of XXE be improved?

RQ2.1: What are the shortcomings of existing static analysis tools for the detection of XML External Entity attacks?

RQ2.2: How can the detection of XML External entities be improved using different techniques?
RQ3: How can auto fixing of XML External Entities be implemented using an IDE plugin?

A prestudy [89] was conducted the results of which are the basis for this master thesis. In the prestudy, a structured literature review to identify the strengths and weaknesses of different tools and approaches for detecting and auto fixing software vulnerabilities, as well as different tools for auto fixing classical software bugs, was conducted. An evaluation of auto fixing tools on the market today was also performed. The literature review was done to established the state of the art in detection and auto fixing of software vulnerabilities and provided the starting point for the research done in this master thesis.

In the prestudy, FindSecBugs was found to be the tool that seemed the most promising to extend with auto fix functionality focusing on XML related vulnerabilities. It was also found to employ detection based on pattern matching. To improve the detection of XXE in FindSecBug the different source code analysis techniques discovered through the literature review such as data flow analysis was used. Each of the detection techniques was preliminarily evaluated to identify the detection capabilities of the technique. Instruction based data flow analysis was found to not provide enough information to improve the detection. Instance tracking was found to perform better than the existing pattern matching approach. The most promising technique, instance tracking, was chosen and implemented. This implementation was evaluated on a larger test set to evaluate the performance of the detection. An auto fix mechanism was also implemented for XXE using FindSecBugs based on modifying the abstract syntax tree which was the most promising method identified in the prestudy.

A test bed was designed to be an extension of the Juliet Test Bed [95] since this was discovered to be the most commonly used and most complete test bed based on control flow variants. New vulnerable test cases for XXE needed to be added since these were missing in Juliet. Then tests for checking if the test cases are still vulnerable and functional were added. The output of running these tests will show how well an auto fixing tool performed without requiring developers to manually check if an auto fix is correct. Therefore, the test bed was used for evaluating the new auto fixing tool created in this research.

The main contributions of this thesis are a novel detection mechanism for XML External Entities based on instance tracking, a novel auto fix mechanism for XXE based on

modifying the abstract syntax tree, and a novel test bed for evaluating auto fix tools. All of these are improvements to the state of the art in detection and auto fixing of web security vulnerabilities. The instance tracking mechanism is generalizable and capable of being extended with detection capabilities for other instance-based vulnerabilities without much effort. The corresponding auto fix mechanism is generalizable and capable of being extended with auto fix capabilities for other vulnerabilities based on vulnerable instances. Finally, the test bed is designed in an extensible manner, making the creation of additional test cases effortless. The test bed is tailored for the evaluation of auto fixes of security vulnerabilities in Java source code. This test bed allows researchers to automatically check if their fixes are effective mitigating XXE and if they preserve the intended functionality of the code.

The rest of the thesis is organized as follows. In chapter 2 a brief overview of the background information for the thesis is presented. In chapter 3 the related work which this thesis is in the context of is shown. chapter 4 shows how the research has been designed. chapter 5 shows how the research has been implemented. In chapter 6 the results and evaluation of the test bed, the novel instance tracker approach for XXE, and the AST based auto fix approach for XXE are shown. In chapter 7 the strengths and weaknesses of each of these are discussed. Finally, the conclusion and future work are presented in chapter 8.

Chapter 2

Background

In this chapter, the necessary background for the thesis will be presented. First, an overview of different static code analysis techniques will be given. Then a description of the Java virtual machine will follow. Next, abstract syntax trees are explained. A description of OWASP top 10 will then be given as well as a description of the importance of mitigating XXE vulnerabilities. An explanation of Common Weakness Enumeration (CWE) for XML is given, alongside a description of the detection and auto fixing tools that have been extended. The basis of this background section is similar to the one in the prestudy, hence why part of the background from the prestudy has been used [89].

2.1 Static Code Analysis

Static code analysis is defined as a tool that can be used to examine the source code of a program without attempting to execute it [22]. The analysis can be performed both on program source code which has the advantage that the static analyzer checks the exact program written by the programmer [76] and on the compiled code. Since compilers optimize the code, the resulting compiled code may not reflect the source code, which means that the analysis can take compiler optimizations into account and detect problems as a result of the optimization performed. Another advantage of analyzing the compiled source is that it is considerably faster.

The tools compare with manual audits because they are faster, which means they can audit larger volumes of code more frequently compared to manual audits. They also capture security knowledge in the tool, which means that users of the tool do not need to have the same level of expertise as a human auditor. This means that it is easy to start using a static analysis tool.

Static code analysis tools can make use of the control flow graph to analyze the control flow of the program. The Data Flow Graph (DFG) can also be used to analyze the data flow within a program, which means that the tool tries to infer the possible values that variables might have at certain points in the program. This means that the tool can consider more of the relationships between function, the order of execution, and the context of the data flow

within the program. Most static analysis tools make use of patterns or rules to identify vulnerabilities, which means that the tools can identify most, but not all vulnerabilities. It also means that the tools need to be regularly updated with new tools and strategies to find new vulnerabilities.

Chess and McGraw [22] mention how static analyzers may be undecidable in the worst case which means that the output of a static analyzer will require human evaluation. It is difficult for the tool to know which problems are of most importance to the developer in terms of the acceptable level of risk. There may also be flaws in the analysis, which means that the tool can produce false negatives or false positives. False negatives mean that the program contains a bug the tool is unable to detect, whereas false positives mean that the tool reports a bug the program does not include. It is therefore desirable for the tool to be sound, which means that given a set of assumptions, it produces no false negatives. This, however, may lead to a huge number of false positives.

2.1.1 Pattern Matching

Pattern matching is a technique used for detecting bugs based on known patterns originally proposed by Knuth, Morris, and Pratt [68]. This is a common technique used to detect bugs in source code [96]. Pattern matching algorithms for code analysis use regular expressions to detect issues and are commonly used in tools for detecting code style issues called linters of which Eslint is an example [46]. With linters being based on pattern matching, it is typically easy for developers to extend and modify them to add new rules. All one has to do is add a new regular expression for incorrect code, and optionally the desired transformed pattern of the code after fixing the issue. An effective way to find these patterns is by documenting actual bugs found in production and create patterns that would have detected them, to discover similar bugs [24].

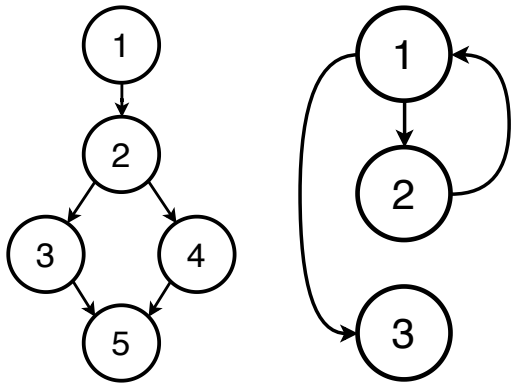
Pattern matching is limited by not being able to know the context of the statements being examined. This means that pattern matching can detect missing input sanitation easily, but does not know if the variable is sanitized later during execution. This limits pattern matching to either only focus on very specific issues, where a pattern will always be incorrect, or risk raising a large number of false-positive warnings. Although pattern matching alone is not robust enough to be used for security vulnerability detection it is often used as an early step in static code analysis to create a call graph or abstract syntax tree.

Pattern matching can be used to detect bugs in a code-base. As long as a bug can be explained fully by a regular expression or finite automata, they can be detected using pattern matching. Examples of this are using a package with known vulnerabilities or security misconfigurations. It is also possible to use pattern matching to detect unsafe methods, like the C functions `strcpy()`, `gets()`, or `sprintf()`, or SQL commands that use a concatenation of strings. The problem is that the regular expression does not know if the inputs to these functions have been checked before the functions are called.

2.1.2 Control Flow

The control flow of an imperative program determines in which order individual statements, instructions, or function calls of a program are executed or evaluated [48]. This

means that the code is executed in a certain order determined by the control flow statements such as if statements, switch statements, or function calls. This control flow can be represented by a directed graph where the nodes represent basic blocks and the edges represent control flow paths. The basic blocks are linear sequences of program instructions having one entry point and one exit point [26]. Control flow analysis is a static analysis of the expression and data relationships in a program that can be used to determine the control flow of a program. The output of such an analysis is a control flow graph. The control flow graph can be used to determine control flow relationships in a program.



(a) Control Flow Graph of an if-statement

(b) Control Flow Graph of a while-loop

Figure 2.1: Examples of control flow graphs

The graphs in Figure 2.1 are the results of an if statement in (a) causing the split at node 2 into two paths that come together at 4, when the if-block is over. (b) shows a loop were 1 checks some condition and goes to 2 as long as the conditional is true, and goes to 3 if it is not.

A control flow graph (CFG) G represents the control flow of a program as a directed graph with nodes $N = \{n_0, n_1, n_2, \ldots, n_l\}$, a single entry node n_0, and a list of edges E. For the graph in Figure 2.2

$$n_0 = 1 \tag{2.1}$$

$$N = \{1, 2, 3, 4, 5, 6, \} \tag{2.2}$$

$$E = \{(1,2), (1,7), (2,3), (3,4), (3,5), (4,6), (5,6), (6,3), (6,7), (7,2)\} \tag{2.3}$$

2.1.3 Data Flow

Allen and Cocke [2] argue the importance of data flow analysis by explaining that for every variable definition, it is usually of interest to know what uses will be affected by the definition of a particular variable. Similarly, given the use of a variable, it is of interest

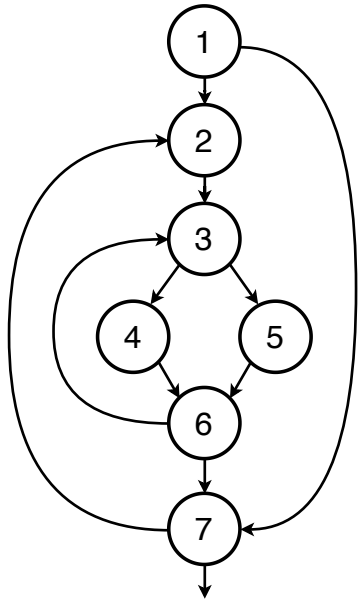


Figure 2.2: Control flow graph example from Allen and Cocke [2]

to know which definitions of data items that can potentially supply values to it. In other words, the data flow of a program shows how data flows through the program, and when it is defined and when it is used. They continue by explaining how the control flow graph can be used to derive and express data flow relationships. An expression or part of an expression that modifies a data item in some way is a data definition. Likewise, a data use is an expression or part of an expression that refers to some data item without modifying it. A data use is potentially affected by a data definition if the data items are the same and the result of the definition is available to the use of the data.

Figure 2.3 shows an example program called power, which takes as an input an integer x and an integer y, and computes x^y with the corresponding control flow graph (CFG) is shown in the middle. To the right is the data flow graph for the same example. The example is based on figure 3 and 9 from [123], using the same nomenclature. d_l^x denotes the variable x being defined, and u_l^x denotes the variable x being used at point l in the program. Using the definition points and the use points, the sets $DEF(l)$ denoting the definitions at point l, and $USE(l)$ denoting the use at point l can be constructed. These are shown in the figure.

An approach to data flow analysis is proposed by Harrold and Soffa [54], which uses a standard iterative data flow analysis to compute the variable use and the definitions that can supply values. They also abstract intraprocedural definitions and use the information for every procedure from the control flow graph to compute interprocedural definition-use pairs that cross the boundary of different procedures. In other words, the control flow graph is used in data flow analysis to observe how variables are used. Improvements have been made on this algorithm through approximations, for instance, Duesterwald, Gupta,

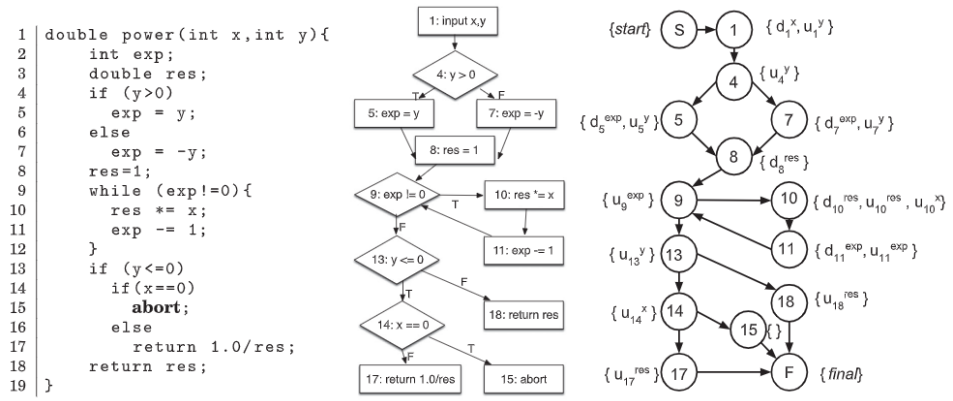


Figure 2.3: Example of code with corresponding control flow graph and data flow graph based on figure 3 and 9 from [123]

and Soffa [40] which uses a demand-driven data flow analysis technique and performs a goal-oriented search instead of exhaustive information propagation.

One of the major drawbacks of data flow analysis is that for each variable use, the definitions that can potentially supply the values to it needs to be computed. This makes it difficult to make the data flow analysis algorithm scalable and suitable approximations need to be made [123].

Definitions

The definitions below are used to describe the flow of variables through their life cycle in data flow analysis.

- n_i Node i
- d definition
- R_i definitions which reach n_i
- U_i upwards exposed uses at n_i
- A_i available definitions at n_i
- L_i definitions active at n_i

$$L_i = R_i \cap U_i \tag{2.4}$$

$$R_i = \bigcup_p A_p, \text{for all immediate predecessors of node } n_i \tag{2.5}$$

$$A_i = (R_i \cap PB_i) \cup DB_i \tag{2.6}$$

Node	R_i	U_i	L_i
1	Ø	X	Ø
2	X_7	X	X_7
3	X_5 X_7	X	X_5 X_7
4	X_5 X_7	X	X_5 X_7
5	X_5 X_7	Ø	Ø
6	X_5 X_7	X	X_5 X_7
7	X_5 X_7	Ø	Ø

Table 2.1: Values from data flow analysis, example by Allen and Cocke [2]

The table created by conducting data flow analysis on a control flow graph like Figure 2.3 is shown in Table 2.1. This table shows what variables are defined at which node, active variables at each node, and upwards exposed variables at each node. See A Program Data Flow Analysis Procedure by Allen and Cocke [2] for more details. This information allows the analysis algorithm to analyze the variables based on definition and usage pairs.

2.1.4 Bytecode Analysis

The Java Virtual Machine (JVM) doesn't know anything about the Java programming language. It only knows about the binary class file format, which contains instructions (bytecodes), a symbol table, and other ancillary information [32]. A class file must adhere to strong structural constraints imposed upon it by the Java Virtual Machine to be able to precisely describe the representation of a class or interface. This means that all code run on the JVM needs to first be compiled down to bytecode by a compiler, which then is interpreted by the JVM. One of the main benefits of compiling Java code down to a bytecode representation which is then run on the Java Virtual Machine is that it doesn't specify anything about the inner workings of the Java Virtual Machine. If the implementation supports all the operations specified in the class files it means that the JVM will be able to run the program.

It is possible to perform an analysis of the bytecode generated by the Java compiler [142]. Optimizations performed by the compiler can be taken advantage of when analyzing bytecode. Due to specific features of JVM to perform control flow analysis on Java bytecode, control flow techniques must be applied both at the intraprocedural level and at the interprocedural level. The resulting control flow graphs can be used by analysis techniques such as data flow analysis or dependency analysis.

The Byte Code Engineering Library (BCEL) is a library for analyzing, creating, and manipulating Java class files, and thus Java bytecode [47]. The library represents classes as objects which in turn contain all the symbolic information of the class including methods, fields, and bytecode instructions. These are represented as an abstract syntax tree (AST). This makes it easier to work with Java bytecode and allows for easier manipulation and analysis of the bytecode. ASM is a different Java bytecode manipulation and analysis framework [9]. It provides bytecode transformation and analysis algorithms that enable complex transformations and code analysis tools to be built. It is designed with performance in mind and performs better than BCEL [10]. FindBugs is an example of a tool for

finding bugs in Java code which uses BCEL to implement the bug pattern detectors used by the tool [58]. The tool implements both control sensitive analyzation methods and data flow analysis.

2.1.5 Source, Sink, and Sanitizer

Taint analysis is a form of information-flow analysis that attempts to find if values from untrusted methods or parameters can flow into operations that are sensitive to security [132]. A variety of tools to perform taint analysis exist such as static code analyzers and dynamic code analysis. These tools may also use different approaches such as early detection and late detection.

Tripp et al. [132] introduce a definition of source, sink, and sanitizer. A source is a method where the return value is considered tainted. A sink is a method where security-sensitive computations are performed, and which is vulnerable to attack via tainted data. A sanitizer is a method that changes the insecure input to produce an output that is considered secure.

2.1.6 Early and Late Detection

The timing of feedback from auditing tools is important to developers due to the increased cost of fixing vulnerabilities later on in development. There are two main categories of feedback timing for static development tools: early detection and late detection [117]. Early detection gives feedback to developers as they are coding. This means that this type of tool is often implemented as an Integrated Development Environment (IDE) plugin. These tools often notify developers by highlighting lines of code while the developer is coding. This can make the process of fixing the vulnerability quicker, since the mistake is noticed right away, rather than after the code is completed. Late detection tools require that developers run the tool manually after finishing coding. This process can often take days or weeks, and can, therefore, leave mistakes in code that become more difficult to fix as the code base grows around it.

2.2 Java Virtual Machine

The Java Virtual Machine is based upon a stack-based architecture [32]. It operates on the primitive types which are the numeric types, the boolean type, and the returnAddress type, and on the referenced types which are class types, array types, and interface types. Each of the threads of a Java Virtual Machine has a private Java Virtual Machine stack that stores frames. A frame consists of local variables, an operand stack, and a reference to a run-time constant pool. A new frame is created when a method is invoked and destroyed when the method invocation completes either normally or abruptly.

The local variables can hold a value of the types boolean, byte, char, short, int, float, reference, or returnAddress. Local variables are also used to pass parameters on a method invocation where the first parameter starts at local variable 0 and so on. The operand stack is a last-in-first-out (LIFO) stack of instructions for the Java Virtual Machine. An operand may load or store constants or values from local variables or fields onto the operand stack,

or consume operands from the operand stack, operate on them, and push the result back to the operand stack. The run-time constant pool contains both numeric literals known from compile-time and method or field references that are resolved at run time. It is constructed on a class by class, or interface by interface basis.

The five different instructions that can be used to invoke methods are shown in Table 2.2. For the *invokevirtual*, *invokestatic*, and *invokespecial*, the instructions take the index into a run-time constant pool as its argument which gives the internal form of the binary name of the class type of the object, the name of the method to invoke, and the descriptor of that method. Both the *invokevirtual* and the *invokespecial* first push a reference to *this* on the stack before the invocation instruction is pushed meaning that the resulting frame that is created gets a reference to *this*. Additionally, if the invocation should consume any parameters these are pushed to the stack after pushing the reference to *this* to the stack. The arguments then become the initial values of the local variables in the resulting frame.

Constructors appear in the bytecode as a method with the name <*init*>, which is a name supplied by the compiler used. To create a new Java Virtual Machine class instance the *new* instruction is used. Then the instance variables of the class and all of the super classes are initialized to their default values, before the *invokespecial* method of the new class is invoked.

Instruction	**Description**
invokevirtual	Invokes the method of an instance of an object. This dispatches on the (virtual) type of the object.
invokeinterface	Invokes the method of an interface. This looks for the appropriate method implemented by a particular run-time object
invokespecial	Invokes the method of an instance of an object which requires special handling. Examples of this is an instance initialization method, a private method, or a superclass method
invokestatic	Invokes a static method in a named class
invokedynamic	Invokes the method which is the target of the call site object bound to the invokedynamic instruction [32]

Table 2.2: The different method invocation instructions as specified by [32]

Descriptors for fields and methods are specified using a grammar notation. For fields, this descriptor represents the type of a class, an instance, or a local variable. Field descriptors can be interpreted as the 10 different types shown in Table 2.3 For methods, the descriptor represents the types of parameters that the method takes. Return descriptors of a method represent the type of the value that the method returns.

BaseType Character	Type	Interpretation
B	byte	signed byte
C	char	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L ClassName ;	reference	an instance of class ClassName
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

Table 2.3: The different field descriptor types recognized by the JVM [32]

To access class fields the opcodes *getstatic*, *putstatic*, *getfield*, and *putfield* are used. These operands take an index into a run-time constant pool as its argument similar to the invocation instructions detailed above. The first two are used to access static fields, whereas the latter two are used to access fields of class instances.

The *code* attribute is a variable-length attribute in the *attributes* table of a *method_info* structure which is part of the *class* file. This attribute contains the Java Virtual Machine instructions and the necessary information for a method. As part of this, the *code* attribute may include a *LineNumberTable* attribute which is an optional variable-length attribute that can be used to determine which part of a *code* array corresponds to a given line number in the source file which the bytecode was generated from.

2.3 Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a near-source intermediate representation of the source code built by a parser which retains the essential structure of a parse tree but eliminates many of the internal nodes that represent nonterminal symbols in a grammar [28]. In other words, an AST eliminates extraneous nodes from the parse tree and retains only the precedence and meaning of the expression. Since the AST is a near-source representation, the parser can build the AST directly.

During the second stage of a compiler's front end, a parser determines if the input program is a valid sentence in the programming language [27]. This is done by building a derivation of the input program using a context-free grammar. A concrete model of the program is then built if the input stream is determined to be a valid program. This representation is called a parse tree and is a graphical representation of a sequence of

rewriting steps that starts with the start symbol of the grammar and ends with a sentence in the language which is defined by a context-free grammar. An AST can then be built by the parser using the parse tree.

Syntax directed editors can make use of the AST to generate source code. These are called source-to-source systems. Compilers and interpreters also use ASTs. For instance, compilers can use details exposed in the AST to perform optimizations. Both source-level abstractions and assembly level abstractions are possible. Since source-level ASTs may lack the information required to translate a statement into assembly code, a low-level AST can be used instead.

The abstract syntax tree is the basis for manipulating Java code in Eclipse e.g. through refactoring, quick fixes, and quick assist [25]. The AST model used in Eclipse is comparable to the Document Object Model (DOM) of an XML file, which means changes can be made to the tree model and then reflected in the Java source code. The Eclipse IDE has a built-in parser called ASTParser which can be used to parse the Java source code into an AST representation of the source. This is done using a Java model, which is a set of classes that model the objects associated with creating, editing, and building a Java program. The model elements are represented by an in-memory object model to represent the structure of a Java program. All the nodes of the AST subclass the ASTNode class, and each subclass is a specialized element of the Java Programming Language. Org.eclipse.jdt.core.dom contains the Java DOM/AST classes, which contains an Application Programming Interface (API) that can be used for manipulating the source code of a Java program as a structured document. Eclipse also includes an ASTRewrite class, which allows Eclipse to write the changes made to the AST back to the Java source code.

2.4 Evaluation Metrics

To evaluate the performance of static code analysis tools the number of true positives (TP), false positives (FP), and false negatives (FN) can be counted. A true positive is a vulnerability that was present in the test set and which was detected. A false positive is a vulnerability that was detected, but which was not present in the test set. A false negative is a vulnerability that was present in the test set, but which was not detected.

Standard metrics for evaluating the detection performance such as precision and recall can be calculated using these numbers. Precision calculates the number of correctly identified vulnerabilities out of the total number of identified vulnerabilities. The equation for calculating the precision is shown in Equation 2.7. Recall calculates the numbers of correctly identified vulnerabilities out of the total number of vulnerabilities present in the test set. The equation for calculating the recall is shown in Equation 2.8.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{2.7}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{2.8}$$

2.5 OWASP Top 10 2017

The Open Web Application Project, OWASP [105], is a nonprofit dedicated to helping organizations develop, operate, and maintain secure software. The OWASP top 10 project [125] was started to raise awareness about web application security but has grown to be used by many as a web security standard.

2.5.1 A1:2017-Injection

Malicious inputs are injected into queries or commands. This can be used by an attacker to bypass authentication or steal information. Software is vulnerable if user-supplied data is not sanitized or validated properly. It can be avoided using prepared statements and bound variables. An example of an SQL injection attack is typing `1 or 1 = 1; --` as a username when signing in. When used in a function similar to the one in Listing 1, this will bypass password checks later in an SQL query as shown in Listing 2.

```
username = getRequestString("username");
password = getRequestString("userpassword");

sql = 'SELECT * FROM Users
    WHERE Username ="' + username + '"
    AND Password ="' + password + '"'
```

Listing 1: Code Vulnerable to SQL injection

```
SELECT UserId, Name, Password
FROM Users
WHERE Username = 1 or 1 = 1;
-- AND Password = any_password;
```

Listing 2: Result of injected SQL

2.5.2 A2:2017-Broken Authentication

Attackers can gain authenticated access to a service without knowledge of the required credentials. This can be achieved by brute force attacks, guessing weak or default passwords, or token theft. To avoid these attacks services should limit login attempts and have a strong password policy.

2.5.3 A3:2017-Sensitive Data Exposure

Sensitive data is leaked, not by an attacker breaking encryption, but by them finding data that was accidentally left unprotected by developers. This can happen when software uses unencrypted communication or encryption with default keys. It can also occur when sensitive information is exposed in code comments or variables that can be accessed in runtime.

2.5.4 A4:2017-XML External Entities (XXE)

XML External Entities are attacks against applications that incorrectly parse XML. Vulnerable code is exploited by an attacker inputting XML that contains an external entity that is parsed by a misconfigured XML parser. Applications can be targeted by filling in the code from Listing 3 in an input field, which would lead to the contents of the password file to be returned to an attacker. Applications should either disallow XML external entities and Document Type Definition (DTD) processing [82] or sanitize inputs to avoid XEE.

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include parse="text" href="file:///etc/passwd"/>
</foo>
```

Listing 3: XML External Entity to extract passwords from password file

2.5.5 A5:2017-Broken Access Control

Some applications allow attackers to bypass access control or elevate their privileges. This can be achieved for example by changing an id in a URL, application state, or in HTML if this change is accepted without a new authentication for the new user id. To avoid this web apps should deny access to non-public data by default, and perform rate-limiting and log failed login attempts.

2.5.6 A6:2017-Security Misconfiguration

A system is set up incorrectly, leaving it vulnerable to known attacks. This can happen if default passwords are used, updates are not installed or incorrect security settings are used for frameworks. To mitigate these vulnerabilities, it is recommended to avoid unnecessary features in platforms and to use automatic auditing of configurations and versioning of packages.

2.5.7 A7:2017-Cross-Site Scripting (XSS)

XSS is the second most common vulnerability in OWASP top 10, found in two-thirds of web applications. Attacker supplied input is parsed as JavaScript in browsers and the malicious code is executed. There are three ways of targeting users' browsers: Stored XSS, Reflected XSS, and DOM XSS. A code example for an XSS attack is shown in Listing 4.

In a reflected XSS attack, the script is injected into a request. These attacks are often executed by spreading links with a script in the URL. The script is executed when the page is loaded using the malicious link.

Stored, also called persistent, XSS attacks store scripts in a server's database. This makes it possible for the script to be executed repeatedly. This also makes it possible for this type of XSS to replicate itself as users are affected. An example of stored XSS is shown in Figure 2.4.

In DOM-based XSS the attack payload executes as a result of modifying the DOM environment [125]. In a DOM-based XSS, the response from the server is not altered, but the code executes differently in the client's browser due to changes to the DOM.

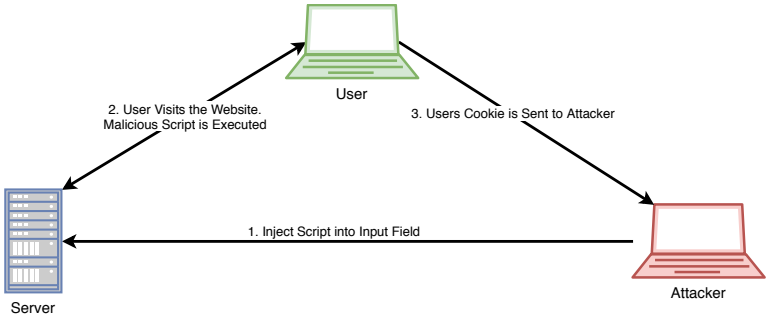


Figure 2.4: Stored XSS attack to send user's cookie to attacker

```
<script type="text/javascript">
    // malicious script
</script>
```

Listing 4: Example code for XSS attack

2.5.8 A8:2017-Insecure Deserialization

Applications that deserialize malicious objects without checking the integrity of the object. Serialization is used for purposes like HTML form data, cookies, caching, and file systems. These attacks can be used to bypass access control or to modify objects and data structures to achieve remote code execution on the victim's server.

2.5.9 A9:2017-Using Components with Known Vulnerabilities

If developers do not keep their software up to date with the latest patches and updates, their applications can be vulnerable to attacks that have already been patched. Potentially vulnerable software includes database management systems (DBMSs), libraries, APIs, operating systems, and web frameworks. To avoid these issues developers must keep systems and dependencies update. This process can be assisted by automated tools.

2.5.10 A10:2017-Insufficient Logging and Monitoring

Missing logging of events makes it easier for attackers to make continuous attempts over an extended period at compromising a system. This can be a problem if suspicious volumes of API calls and failed login attempts are not logged, or if no alert is given in real-time. One way of effectively testing if a system is monitoring activity well enough is to conduct a penetration test, and check if the event was detected with sufficient information for developers.

2.6 Other Web Application Vulnerability Classifications

2.6.1 Session management

Cross-Site Request Forgery(CSRF)

In a Cross-Site Request Forgery attack, the victim is made to send a request to a website where they are already authenticated [84]. This is typically done by sending a URL with a script tag or an image with JavaScript in the source field. An example of a CSRF attack is shown in Figure 2.5 where a user first signs in to a banking website, then open a malicious website with an image with a source that is the URL for sending money to another user. The website does not check if the request was intentionally sent by the user, and sends money to the attacker.

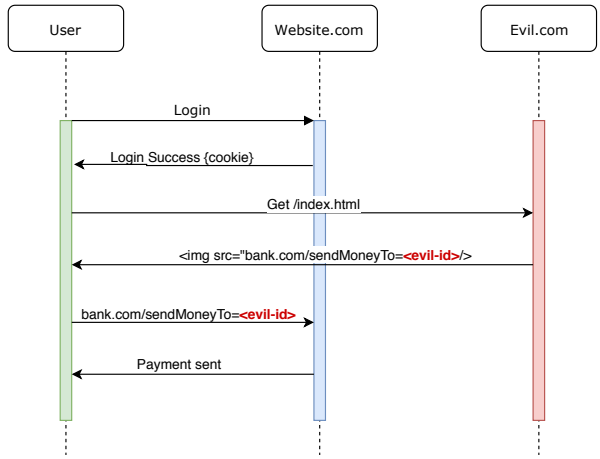


Figure 2.5: CSRF Attack

Session fixation

Session Fixation can happen when a user is authenticated without invalidating existing session cookies. Doing this allows an attacker to force a known session onto a user, which will give the attacker authenticated access to the victim after they authenticate. The attacker can set the victims token using XSS or exploiting an unencrypted connection. An example of how a session fixation attack can be carried out is shown in Figure 2.6.

2.7 Importance of Mitigating XXE

Extensible Markup Language (XML) is a well-defined meta-language which allows one to encode a description of an XML document's storage layout and logical structure [133]. These XML documents in turn are built up of entities, which can contain either parsed or unparsed data. However, these parsers may be vulnerable to XML External Entities

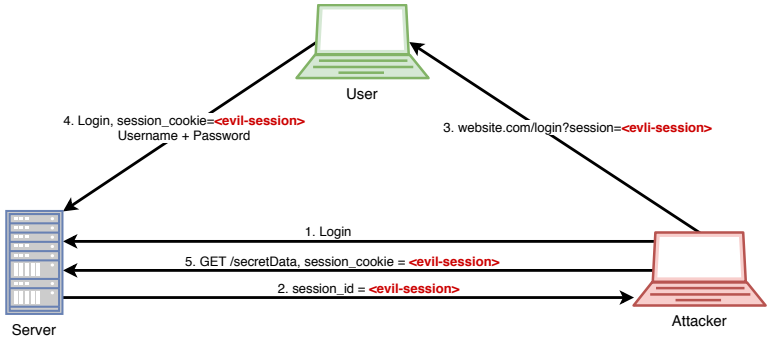


Figure 2.6: Session Fixation Attack

attacks, as described in subsection 2.5.4. An application is vulnerable if XML is accepted directly, or if XML uploads are accepted by the application [125].

Späth et al.[120] evaluated 30 XML parsers from popular programming languages of which 10 were Java-based. They found that all the parsers were vulnerable to XXE attacks. For most of the parsers, the vulnerability was present due to an incorrect configuration of the XML parser. The vulnerability can also be mitigated by filtering the input or correctly sanitizing the input before providing it to the XML parser. They mention how some features might not work as expected when applying countermeasures. For instance, disabling DTD processing limits the loading of external DTD.

Jan, Nguyen, and Briand[59] studied 13 different XML parsers. The integrated parsers in the most popular programming languages were selected. These parsers were evaluated by invoking each parser and providing XML input files as input, which contained Billion Laugh and XXE attacks. An overview of the vulnerability of each parser they looked at can be seen in Table 2.4. Out of the 13 parsers they looked at, eight of them were for Java, two for Python, one for Perl, one for PHP Hypertext Preprocessor (PHP), and one for C# and Javascript. The Java parsers were found to be the most widely used ones, with over 250 000 projects identified on GitHub making use of these parsers. Out of the Java-based parsers, six of the eight were found to be vulnerable to either the billion laughs attack or XXE attacks in general. Furthermore, of the over 700 classes from 628 open source project they inspected manually, all but one was found to use a vulnerable XML parser configured in a vulnerable way. They also found that most developers use the parser with a default configuration. This shows how widespread vulnerable use of XML parsers are and shows the importance of identifying and mitigating XXE vulnerabilities in XML parsers.

Most XXE vulnerabilities can be prevented by ensuring that the XML parsers are updated to the latest version as well as correctly configuring the parsers to disable XXE and document type definition (DTD) processing [125]. Since most parsers have a vulnerable configuration by default, it is important to identify where these parsers have been used in the code and to apply the appropriate configuration options. Whitelisting, sanitization, and input validation can also be employed to prevent XXE attacks.

Parser	Vul. to BIL	Vul. to XXE	Language
JDOM2	Yes	Yes	Java
NanoXML	Yes	Yes	Java
NanoXML-LITE	No	No	Java
Std-DOM	Yes	Yes	Java
Std-SAX	Yes	Yes	Java
Std-STAX	No	No	Java
WOODSTOX	No	No	Java
XERCES-JDOM	Yes	Yes	Java
LXML-ETREE	No	No	Python
Std-ETREE	Yes	No	Python
PERL(XML::LibXML)	Yes	Yes	Perl
PHPDOM	No	No	PHP
MSXML (DOMDocument)	Yes	Yes	C#, Javascript
Total	8	7	-

Table 2.4: Summary of vulnerable parsers in popular programming languages as identified by Jan, Nguyen, and Briand[59]

2.7.1 Mitigation Strategies for XXE Vulnerabilities

The different XML parsers support a wide variety of features [3, 4, 35, 34]. This can make it difficult to know which features to set to make a parser secure since XML parsers for Java are made secure by disabling vulnerable features [125]. Different mitigation strategies are presented by Oracle [30] and OWASP [127]. The attributes suggested by these will be used to determine if a parser is secure or not.

The different mitigation strategies presented by OWASP summarizes the features that need to be set [127]. The covered XML parsers alongside their attributes and values are shown in Table 2.5. The XMLDecoder parser has not been listed due to being considered fundamentally insecure and, therefore, should not be used. The JAXB Unmarshaller parser requires the input to be sent through a separate secure parser first to be secure.

Attribute	Value	Description	Parser
http://apache.org/xml/features/ disallow-doctype-decl	true	Disallows DTD DOCTYPE declaration	DocumentBuilder, SAXParser, DOM4J, XMLReader, SAXReader, SAXBuilder
http://xml.org/sax/features/ external-general-entities	false	Disables inclusion of external general entities	DocumentBuilder, SAXParser, DOM4J, XMLReader, SAXReader, SAXBuilder

http://xml.org/sax/features/ external-parameter-entities	false	Disallows inclusion of external parameter entities or the external DTD subset	DocumentBuilder, SAXParser, DOM4J, XMLReader, SAXReader, SAXBuilder
http://apache.org/xml/features/ nonvalidating/load-external-dtd	false	Ignores external DTD completely	DocumentBuilder, SAXParser, DOM4J, XMLReader
.setXIncludeAware(boolean state)	false	Disables XInclude processing	DocumentBuilder, SAXParser, DOM4J
.setExpandEntityReferences(boolean expandEntityRef)	false	Disables expansion of entity reference nodes	DocumentBuilder
javax.xml.stream. isSupportingExternalEntities	false	Disallows resolving externally parsed entities	XMLInputFactory
XMLInputFactory .SUPPORT_DTD	false	Requests a processor that does not support DTD	XMLInputFactory
XMLConstants .ACCESS_EXTERNAL _DTD	""	Restricts access to external DTDs and extenal entity references	TransformerFactory, Validator, SchemaFactory, SAXTransformerFactory
XMLConstants .ACCESS_EXTERNAL _STYLESHEET	""	Restricts access to external references set by the stylesheet processing	TransformerFactory, SAXTransformerFactory
XMLConstants .ACCESS_EXTERNAL _SCHEMA	""	Restricts access to external references set by the schemaLocation attribute, Import and Include element	Validator, SchemaFactory
.setEntityResolver(EntityResolver er)	er	Specifies the EntityResolver (er) to be used to resolve entities in the XML document which is parsed	DocumentBuilder, XMLReader, SAXParser

Table 2.5: The different attributes suggested by OWASP that can be configured for the different factories used to initialize the XML parsers, and for XML parsers that are initialized directly [127]

Different mitigation strategies are also presented by Oracle [30]. These have been summarized in Table 2.6. It is worth noting that for the parsers that support the feature *XMLConstants.FEATURE_SECURE_PROCESSING*, this feature has been turned on by default except for the Transformer parser. However, external access is still allowed to all protocols which means that the parsers are still vulnerable to XXE attacks. Setting this attribute explicitly to true disallows external access effectively mitigating the XXE vulnerability. When a security manager is present, then this feature is set explicitly to true for all supported parsers. Oracle also recommends disabling DTD if it is not needed.

Attribute	**Value**	**Description**	**Parser**
XMLConstants.FEATURE_ SECURE_PROCESSING	true	Instructs the XML processors to try and process XML securely. Enforces processing limits and restricts external access	DocumentBuilder, SAXParser, XMLReader, Transformer
XMLInputFactory.SUPPORT_ DTD	false	Disables DTD processing	XMLStreamReader, XMLEventReader, FilteredReader
XMLInputFactory.IS_ SUPPORTING_EXTERNAL_ ENTITIES	false	Restricts external access	XMLStreamReader, XMLEventReader, FilteredReader
.setEntityResolver(EntityResolver er)	er	Specify custom entity resolver	DocumentBuilder, XMLReader, SAXParser

Table 2.6: The different attributes suggested by Oracle that can be configured for the different factories used to initialize the XML parsers, and for XML parsers that are initialized directly [30]

2.8 Common Weakness Enumeration (CWE)

Common Weakness Enumeration (CWE) [85] is a formal list of common software security weaknesses that can occur in software architecture, design, code, or implementation that can lead to exploitable security vulnerabilities. The primary purpose of the list is to serve as a common language for describing software security weaknesses. The list is composed by the CWE community, which includes individual researchers and representatives from organizations from across the industry, academia, and government. The main difference

between a software weakness and a software vulnerability is that software weaknesses can lead to software vulnerabilities that can be used by a hacker to gain access to a system or a network. The different CWE entries have a category ID, which is assigned to a collection of weaknesses sharing common attribute(s), a compound element ID, which is assigned to meaningful aggregations of several weaknesses, a view ID, which describes different ways of viewing the CWE entry and a weakness ID, which is an ID for the software weaknesses themselves. These identifiers are used to categorize and to group the weaknesses. Different CWE entries can be classified within the OWASP top 10 categories and give specific examples of actual vulnerabilities for that category. For example, CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'), is an example of an injection attack, which fits under the OWASP top 10 A1 - injection category.

The CWE entries CWE-611 [87] and CWE-776 [88] denote the security vulnerabilities related to parsing XML. These entries correspond to OWASP top 10 A4 – XML External Entities. CWE-611 denotes the vulnerability that occurs when an XML document which contains external entities outside of the sphere of control, is processed which leads to these documents becoming part of the output. This vulnerability exploits the Document Type Definitions (DTD) feature within XML documents which allows the definition of external XML entities to be defined. For instance, an attacker can use this to embed the */etc/passwd* file in Unix-based systems into the output of the processed XML as shown in Listing 5. CWE-776 denotes the improper restriction of recursive entity references in Document Type Definitions [88]. An attacker can exploit this to create a large number of nested recursive entities that lead to an explosive growth in the data that needs to be parsed. leading to denial of service. An example of CWE-776 is the billion laughs attack shown in Listing 6.

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

Listing 5: An example of CWE-611 showing an attacker attempting to include the external entity */etc/passwd*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Listing 6: An example of CWE-776 showing a Billion Laughs Denial Of Service (DOS) attack

2.9 Static Analysis and Automatix Code Fixing in IDEs

The open source IDE Eclipse [43] offers automatic fixes for problems found by the IDE as the code is being written and after the code has been compiled by Eclipse. These automatic fixes are highlighted with a light bulb indicator next to the affected lines in the IDE, which the developer can click on to get access to the auto fixes if these are available. The automatic fixes include changing the method signature, or the return statement of a function to correctly correspond to each other, as well as changing the type of a variable to the correct one among others. A full list of the available automatic fixes can be found in [42]. A selection of the available fixes will be available in a pop-up dialogue as shown in Figure 2.7. In this example, the return statement returns a variable of the type Integer, whereas the method signature is of type void. Eclipse offers to either change the method type or to change the return statement to fix the error.

Other IDEs such as JetBrains' IntelliJIDEA [61] and Microsoft's Visual Studio [83] also support static code analysis to identify the style, quality, and other code issues. However, both IntelliJIDEA's and Visual Studio's built-in static code analysis are proprietary, and the source code cannot be inspected to identify how the respective IDEs implements the static code analysis.

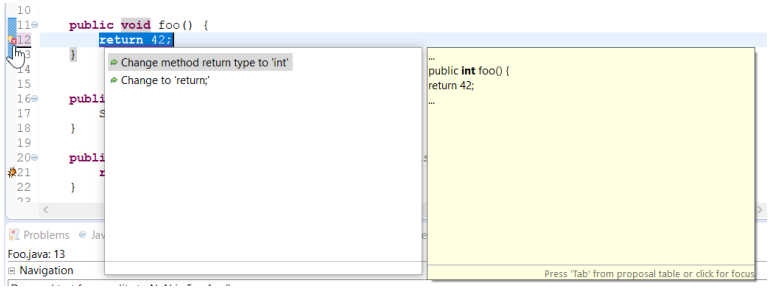


Figure 2.7: Example of quick fix offered by Eclipse

2.10 Description of Tools to be Extended

In this section, the project structure of the tools FindSecBugs [5] and SpotBugs [121], the successor to FindBugs [58], that in the prestudy [89] was found to be the most promising to extended for the master thesis will be explained. An overview of how the two tools' source code is structured will be given, as well as what functionality is available.

2.10.1 FindSecBugs Project Structure

FindSecBugs is divided into four separate modules as shown in Figure 2.8. The main module, where the different detectors are implemented, is findsecbugs-plugin [112]. This module depends on the other modules find-secbugs-samples-kotlin, findsecbugs-test-util, findsecbugs-samples-java, findsecbugs-samples-deps, and findsecbugs-samples-jsp as shown in Figure 2.9 with the blue lines representing normal dependencies and the green lines

representing test dependencies. The findsecbugs-samples-java and findsecbugs-samples-kotlin modules contain vulnerable test code for testing the detectors. The findsecbugs-samples-deps package contains mock implementations of the Java libraries used to test the different detectors which means that these libraries don't have to be downloaded to use the samples of vulnerable test code in FindSecBugs. Lastly, the findsecbugs-test-util module implements utility classes to help with testing the vulnerability detectors in FindSecBugs.

- `findsecbugs-plugin`
- `findsecbugs-samples-deps`
- `findsecbugs-samples-java`
- `findsecbugs-samples-kotlin`
- `findsecbugs-test-util`

Figure 2.8: Top-level project structure of FindSecBugs

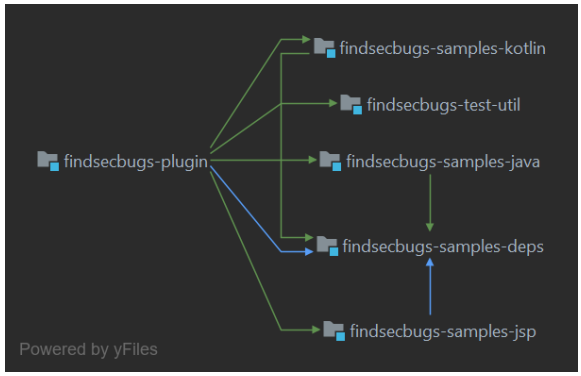


Figure 2.9: Uml diagram showing how the main findsecbugs-plugin module depends on the modules findsecbugs-samples-kotlin, findsecbugs-test-util, findsecbugs-samples-java, findsecbugs-samples-deps, and findsecbugs-samples-jsp. The blue lines represent normal dependencies, and the green lines represent test dependencies

FindSecBugs describe all bugs by specifying which detectors report which bugs in findsecbugs-plugin/metadata/findbugs.xml. The message displayed to the end user containing information about the detector reporting the bug, and information on how to mitigate the vulnerability is described in findsecbugs-plugin/metadata/messages.xml. To add a new detector, corresponding entries have to be inserted into findbugs.xml to let FindSecBugs know which detector reports which vulnerabilities, and into messages.xml to let FindSecBugs know how to present information about the detector and the identified vulnerability to the user.

Maven is used to handle dependencies, compilation, and running the tests present in FindSecBugs. FindSecBugs employ a test driven development strategy, meaning that a

test for a detector is first created which fails, and then a detector is written that makes this test pass. Classes containing examples of vulnerable test code is placed into either the findsecbugs-samples-java module, or the findsecbugs-samples-kotlin module depending on whether the vulnerable code is written in Java or in Kotlin. A mock implementation may have to be added to findsecbugs-samples-deps if external libraries are used in the test. Then test cases are written in a package with a suitable name in the findsecbugs-plugin module using the test utilities described in findsecbugs-test-util. As mentioned above, corresponding entries have to be added to findbugs.xml and messages.xml. Finally, a detector to handle the desired vulnerability is implemented to make the test cases pass.

2.10.2 SpotBugs Project Structure

SpotBugs is divided into 8 separate modules as shown in Figure 2.10. The main module where the core functionality lies is the spotbugs-spotbugs module [55]. The spotbugs-annotations module includes all the different annotations supported by Spotbugs, however, this package is mostly deprecated. The spotbugs-ant module includes Java classes that help set up tasks to run Spotbugs functionality with Ant. The packages SpotbugsTestCases, test-harness, test-harness-core, and test-harness-jupiter are used to test Spotbugs and the detectors. The SpotbugsTestCases module is a collection of vulnerable test code. The test-harness and test-harness-core package sets up the test framework used to test Spotbugs. Finally, the test-harness-jupiter package helps integrate SpotBugs tests with JUnit. Gradle is used to handle dependencies, compilation, and running the test cases in SpotBugs.

- `spotbugs-annotations`
- `spotbugs-ant`
- `spotbugs-spotbugs`
- `spotbugs-tests`
- `spotbugsTestCases`
- `test-harness`
- `test-harness-core`
- `test-harness-jupiter`

Figure 2.10: Top-level project structure of FindSecBugs

Spotbugs includes numerous different base bug detectors which provide core functionality and which can be extended to implement new detectors [56]. These are divided into visitor-based detectors and CFG-based detectors. The visitor-based detectors traverse the features of a classfile such as fields, methods, and instructions in a top-down manner while simultaneously decoding the symbolic information. When one of these features are encountered, a callback method is invoked. By overriding these callback methods new detectors can be implemented using the visitor pattern. The CFG-based detectors represent

the Java methods using a CFG representation allowing one to analyze the control flow and to implement data flow analysis. Base detectors implementing these two kinds of base detectors, as well as other generic classes for doing static analysis are implemented in the edu.umd.cs.findbugs.ba package.

Overview of Base Detector Classes

SpotBugs includes base classes that provide helper functionality which can be used to implement a new detector for a vulnerability without needing to write a lot of the base functionality for scanning the bytecode [55]. An overview of these is presented in Table 2.7.

The *ByteCodeScanningDetector* class extends the *DismantleBytecode* class and provides a concrete implementation of the *DismantleBytecode* class. The main purpose of the DismantleByteCode class is to take the bytecode apart using Apache BCEL, and create an internal representation of the bytecode for SpotBugs. Therefore, using the *ByteCodeScanningDetector* base detector class allows one to use this internal representation of the bytecode, but it does not provide any other functionality.

The *OpcodeStackDetector* collects approximate information about the values at the operand stack for each code location. It extends the *ByteCodeScanningDetector* and is thus given the functionality of both DismantleByteCode and a specific implementation of this. The OpcodeStackDetector visits the bytecode instructions of the class, method, and fields, and calls the abstract *sawOpcode* function whenever a new opcode is found. It also attempts to keep track of which operands are on the operand stack at any given moment.

The *PreorderVisitor* class allows one to traverse the contents of a Java class by calling the respective accept methods. In other words, it allows one to traverse the content of a Java class by using the visitor pattern. More specifically, the *PreorderDetector* class implements a detector using preorder traversal of the classfile which can be extended to implement a detector using this preorder traversal and the visitor pattern.

The *AnnotationVisitor* class is similar to the *PreorderVisitor* class but also visits annotation on classes, fields, methods, and method parameters. The *AnnotationDetector* class is thus also similar to the *PreorderDetector* class in that it can be used to implement a detector that uses preorder traversal of the classfile including visiting annotations by applying the visitor pattern.

The *ByteCodePatternDetector* class can be used to detect specific patterns of bytecode instructions while taking into account the control flow and uses of fields and values. It visits the class context and looks for the desired pattern in each method of the class.

The *CFGDetector* class is a base class for detectors that analyzes the control flow graph. It does this by visiting the control flow graph of each method, allowing one to perform analysis on each point in this graph.

The *ResourceTrackingDetector* class provides base functionality for detectors that wish to find where a particular kind of resource is not cleaned up or closed properly. The base classes implementing this provides the base class with information about what kinds of resources are tracked by the detector.

Base Detector	Description
OpcodeStackDetector	Base class for Detectors that want to scan the bytecode of a method and use an opcode stack
BytecodeScanningDetector	Base class for Detectors which want to extend DismantleBytecode
PreorderVisitor	Interface to make the use of a visitor pattern programming style possible. I.e. a class that implements this interface can traverse the contents of a Java class just by calling the 'accept' method which all classes have
PreorderDetector	Base class for Detectors that want to perform a preorder traversal of the classfile
AnnotationVisitor	Similar to PreorderVisitor, but also visits annotation on classes, fields, methods, and method parameters
AnnotationDetector	Base class for Detectors which want to perform a preorder traversal of the classfile including visiting Annotations
ByteCodePatternDetector	Base class for Detectors that are based on a ByteCodePattern. This class allows patterns of bytecode instructions to be detected, taking into account control flow, and the uses of fields and values
CFGDetector	Base class for detectors that analyze the control flow graph
ResourceTrackingDetector	Base class for Detectors that wish to find methods where a particular kind of resource is not cleaned up or closed properly

Table 2.7: An overview of the base detector classes present in SpotBugs [55]

Data Flow Analysis Framework

SpotBugs includes a generic data flow analysis framework that allows any kind of object to be used as a data flow value [56]. When performing data flow analysis in SpotBugs, SpotBugs symbolically executes the method using a control flow graph representation of the method. The nodes consist of BasicBlocks which are made up of InstructionHandles, and the edges are made up of the bytecode instructions that connect a source block and a target block e.g. goto, ifcmp, as well as exception handlers. The analysis attempts to estimate data flow values that are true for each location by modeling the data flow values as a lattice having a meet operator for merging values, a top value that will always result in a top value when merging with another value, and a bottom value that will always result in a bottom value when merging with other values. A data flow location is defined by an InstructionHandle and the BasicBlock it occurred in.

The different data flow analysis classes provided by SpotBugs are shown in Figure 2.11. The main difference between these is the level of which the transfer function operates at. All data flow analysis classes inherit from the *DataflowAnalysis* interface which defines methods for creating and manipulating data flow values. The *BasicAbstractDataflowAnal-*

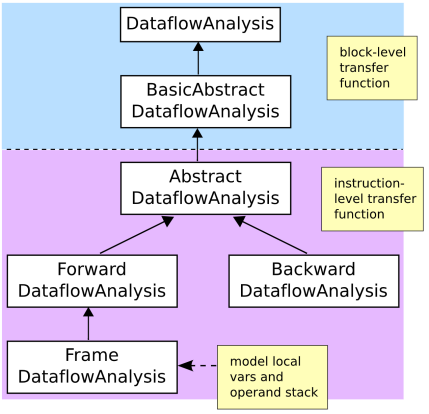


Figure 2.11: Overview of the data flow analysis framework in spotbugs [57]

ysis base class provides a starting point for defining data flow analysis and handles caching of start and result facts for basic blocks [55] and can be used to perform data flow analysis where the transfer function should be at the block-level. The *AbstractDataflowAnalysis* class extends the *BasicAbstractDataflowAnalysis* class and provides additional information for performing data flow analysis by modeling the instructions within basic blocks. The *ForwardDataflowAnalysis* class, and the *BackwardDataflowAnalysis* both extend the *AbstractDataflowAnalysis* and differ primarily in that one initializes the *AbstractDataflow-Analysis* class for performing forwards data flow analysis, and the other initializes the *AbstractDataflowAnalysis* class for performing backward data flow analysis. Therefore, if either forward or backward data flow analysis modeling the effect of instructions on data flow values is desired, these two classes should be extended. The *FrameDataflowAnalysis* class implements base functionality for modeling the local variables and the operand stack within a Java frame and uses a *Frame* class to represent Java stack frame values at a single CFG location. The *AbstractFrameModelingVisitor* class can then be used to model the effect of bytecode instructions on a Frame.

Chapter 3

Related Work

In this chapter related research into auto fixing web security vulnerabilities in Java. The related work was discovered and examined through the structured literature review performed in the prestudy [89].

3.1 Existing Detection and Auto Fix Tools for Software Security

There exists a variety of different tools for detecting security flaws in code based on both static and dynamic analysis approaches [16]. Tools that are state of the art for detecting security vulnerabilities, and fixing them through code analysis were identified through the prestudy [89]. In this section, existing tools for detecting and automatically fixing security vulnerabilities in Java will be presented.

3.1.1 ASIDE

Application Security plugin for Integrated Development Environment, ASIDE [102], is a static analysis tool for Java and PHP that can be installed in the Eclipse IDE. ASIDE was created by OWASP, but has not been updated since 2012 [101]. ASIDE has three prototype implementations: ASIDE CodeRefactoring for Education, ASIDE CodeAnnotate which consists of two implementations, ASIDE JavaCodeAnnotate and ASIDE PHPCodeAnnotate. This analysis will focus on ASIDE CodeRefactoring for Education since it revolves around web application security vulnerabilities like CSRF and broken access control. ASIDE's architecture is described in Figure 3.1.

Implementation

Figure 3.1 shows an overview of the design of ASIDE. It takes in an abstract syntax tree and a set of security sensitive operations (SSO rules). These are used to create annotations

that can be used during static analysis to detect vulnerabilities. The tool uses the OWASP Enterprise Security API (ESAPI) to provide the auto fix suggestions and sanitization options [102]. ESAPI defines a standard interface that provides secure implementations of standard API methods that are considered insecure [103]. Inspecting the source code of the tool reveals that the tool uses the visitor pattern.

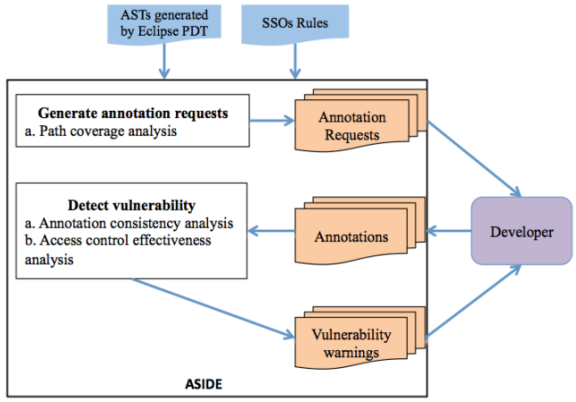


Figure 3.1: ASIDE architecture by Zhu et al. [143]

Vulnerabilities

The vulnerabilities covered by ASIDE are listed below. As can be seen, ASIDE covers the vulnerabilities input validation, broken access control, and Cross-Site Request Forgery.

- Improper input validation and/or filtering
- Broken access control
- Cross-Site Request Forgery (CSRF)

3.1.2 ESVD

Early Security Vulnerability Detector (ESVD) [117] is a vulnerability scanner developed as a plugin for Eclipse [116]. The plugin supports the identification of 11 different vulnerabilities as identified by OWASP including command injection, cookie poisoning, and SQL injection. They were selected because they all occur due to untrusted input from the user. The plugin utilizes static code analysis to perform the analysis without needing to run the code. This means that the tool can provide feedback to the developer on possible vulnerabilities as the developer writes the code. This is often referred to as early detection since the tool can be run while the code is being written and does not require the program to be fully implemented.

Implementation

The tool makes use of context-sensitive data flow analysis, which means that the tool can consider the context of the program in terms of its variables and methods when searching for vulnerabilities. This is then combined with three heuristics: A list of entry points, a list of exit points, and a list of sanitization points. An entry point is a point in the source code where untrusted input may enter the application from outside of the application. An exit point, or a sink, is a point in the source code where untrusted output may exit the application. A sanitization point is a point in the source code where a method or a class transforms the untrusted input into a trusted output. The developer of the tool has compiled a list of entry points, exit points, and sanitization points, which is then used to report vulnerabilities when an entry point reaches an exit point without going through a sanitization point. To provide auto fix suggestions, the OWASP ESAPI library is used. Inspecting the source code shows that the tool uses the visitor pattern. The described architecture is shown in Figure 3.2.

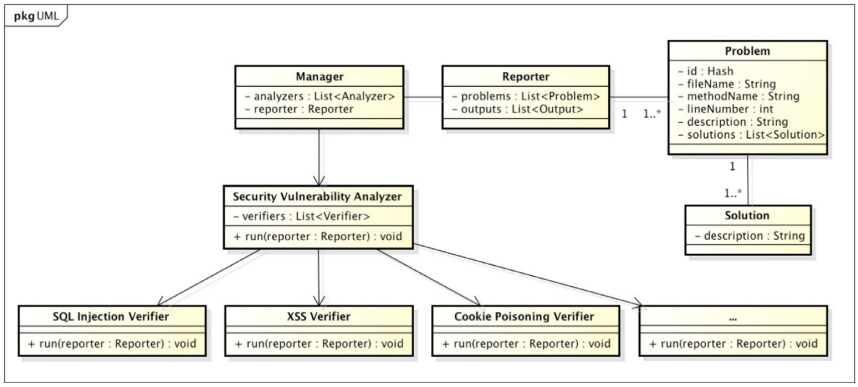


Figure 3.2: ESVD architecture as described by Sampaio and Garcia [117]

Vulnerabilities

The vulnerabilities covered by ESVD are listed below. As can be seen, ESVD focuses on vulnerabilities caused by untrusted input.

- Command Injection
- Cookie Poisoning
- Cross-Site Scripting (XSS)
- HyperText Transfer Protocol (HTTP) Response Splitting
- LDAP Injection
- Log Forging

- Path Traversal
- Reflection Injection
- Security Misconfiguration
- SQL Injection
- XPath Injection

3.1.3 FindSecBugs

FindSecBugs [5] is an extension to SpotBugs, a more general code auditing tool that is the spiritual successor to FindBugs. FindSecBugs detects bugs in bytecode using Apache BCEL to look for known bug patterns. The tool has 134 patterns used to detect CWE and OWASP vulnerabilities. FindSecBugs can be installed into IDEs or used as a part of a continuous integration pipeline.

To help solve vulnerabilities, FindSecBugs suggests potential solutions to the reported bugs. The prompt explains the bug, points out the vulnerable code, and suggests a code pattern that will solve the problem. This means that FindSecBugs does not have automatic bug fixing, but does have a system that assists users in solving the issue.

Implementation

According to the creator of FindBugs Hovemeyer and Pugh [58], FindBugs is implemented using the visitor pattern for bug detection. The implementation of the visitor pattern used by SpotBugs, the successor to FindBugs, is shown in Listing 7. Each detector checks each class of the application being analyzed. The implementation strategies used by the detector were divided into four categories:

- **Class structure and inheritance hierarchy only:** Only the overall structure of the class. It does not examine the code.
- **Linear code scan:** Scans bytecode linearly, using the visited instructions to drive a state machine. Do not use control flow information.
- **Control Sensitive:** Analyze a control flow graph.
- **Data flow:** Uses data flow as well as control flow to analyze code.

Vulnerabilities

FindSecBugs supports 134 different bug patterns ranging from vulnerabilities caused by untrusted input, to use of predictable pseudorandom number generators, and unsafe hash equals. A full list of the covered vulnerabilities can be found in [6].

```
for each analysis pass in the execution plan do
    for each application class do
        for each detector in the analysis pass do
            request ClassContext for the class from the AnalysisContext
            apply the detector to the ClassContext
        end for
    end for
end for
```

Listing 7: Visitor pattern implementation in SpotBugs as described by SpotBugs [121]

3.1.4 JoanAudit

JoanAudit [130] is a vulnerability detector for Java which focuses on detecting common injection vulnerabilities such as XSS, SQL injection, and XML injection. It uses static code analysis together with an approach referred to as security slicing. Security slicing first uses static analysis to identify the input sources alongside the sinks. Then program slicing and code filtering are applied to obtain only the source code that contains the statements necessary for auditing vulnerabilities related to each sink. The statements that don't require auditing are pruned away. The architecture for JoanAudit can be seen in Figure 3.3.

Implementation

The tool takes as input the bytecode of a Java application and a pre-defined set of signatures for input sources and sinks. Then a system dependency graph capturing inter-procedural data-, control- and call dependencies is constructed. Using this, the tool prunes away irrelevant functions such as functions assumed to be known free for security issues. A chop for each sink is then generated, containing all the program statements that influence a sink, while sinks not affected by any input source are pruned away. Flow analysis is then performed on each chop to check if any of the paths in the chop can be pruned away due to proper usage of sanitization functions. Context analysis is then applied to the remaining paths. During this step, the tool attempts to use the context describing how the data from a source is used in a sink, which is then used to apply an appropriate sanitization function. The tool claims that this automatic fix is guaranteed to properly fix the given vulnerability because the fix is only applied if the data flow is directly from a source to a sink, and if the context of the input to the source can be determined.

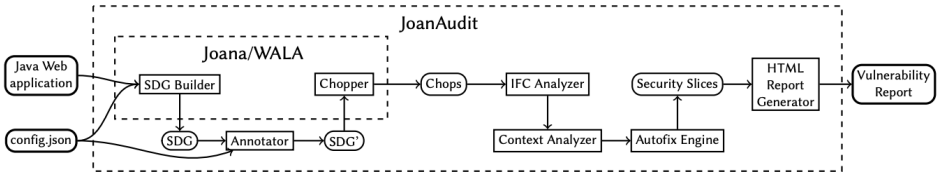


Figure 3.3: JoanAudit architecture by Thomé et al. [130]

Vulnerabilities

The vulnerabilities covered by JoanAudit are listed below. As can be seen, JoanAudit focuses on vulnerabilities caused by untrusted input.

- XSS
- SQL injection
- XML injection
- XPath injection
- LDAP injection

3.1.5 LAPSE+

LAPSE+ [104] is a tool for detecting vulnerabilities in Java EE applications developed by OWASP. It is available as an Eclipse plugin. The tool makes use of static code analysis to detect the source and sink of a vulnerability. In the context of LAPSE+, a source is a point in the code where untrusted data can be injected. A sink is a process that modifies data in order to manipulate the behavior of the application. The tool identifies a vulnerability as the possibility for a vulnerability to reach a sink from a source.

Implementation

LAPSE+ utilizes three main steps in its detection process. First, all the possible vulnerability sources are detected. These are the points in code where untrusted data may enter. Then, all the vulnerability sinks are identified. These are all the points in the code where untrusted data can propagate to other places in the application. The last step is the provenance tracker where the tool figures out if a vulnerability source can be reached from a vulnerability sink by doing backward propagation through different assignations. The tool identifies a vulnerability if this is possible.

Vulnerabilities

The vulnerabilities covered by LAPSE+ are listed below. These are caused by the injection of untrusted data into the application.

- Parameter Tampering
- URL Tampering
- Header Manipulation
- Cookie Poisoning
- SQL Injection
- Cross-site Scripting (XSS)

- HTTP Response Splitting
- Command Injection
- Path Traversal
- XPath Injection
- XML Injection
- LDAP Injection

3.1.6 Snyk

Snyk [119] is a source code auditing tool focused on detecting vulnerable dependencies. The tool is free to use but has premium tiers that come with extra features, like Docker container security scanning, Jira integration, and improved reports. Snyk's source code is available on GitHub [118] and can be installed into projects as well as customized for an organization's needs.

Implementation

Snyk is implemented both as an IDE plugin and as a CI/CD tool. The IDE plugin can be installed in major IDEs like Eclipse, Microsoft Visual Studio Code, and IntelliJ, and code managers like GitHub or GitLab.

When scanning for vulnerabilities, Snyk first scans for manifest files containing the project's dependencies. This is done by selecting and parsing files based on simple patterns since these files have a strict format and naming scheme. The dependencies are then checked against a database with known vulnerabilities and automatically corrected if any vulnerabilities are detected.

The Snyk plugin is open source, however, the Snyk vulnerability database is closed source and does not offer any documentation of their system. This means that analysis of the vulnerability database part of the tool can only be done at a high level, and only with some assumptions.

Vulnerabilities

Snyk keeps a database of identified vulnerabilities in packages and dependencies. The tool then scans for packages containing these known vulnerabilities, identifying outdated packages or packages containing vulnerabilities.

3.2 Detection and Auto Fix Methods

In this section, the different detection methods within the field of software security will be presented. Different auto fix methods were also identified both classical auto fixes focusing on logical bugs or programming error, but also software security related auto fixes. First, a presentation of the different detection techniques will be given. Then the different auto fix approaches will be detailed.

3.2.1 Overview of Detection Methods

There exists a variety of different techniques within the field of software security for detecting software vulnerabilities. Many of the existing tools for detecting software vulnerabilities apply an approach based on static code analysis to identify the vulnerabilities [81, 140, 117, 136, 104, 119, 5, 130, 139]. Approaches based on dynamic analysis are also used to identify software vulnerabilities [140, 134, 97, 69]. Most of the tools use pattern matching in conjunction with static code analysis to identify and mark code as vulnerable [136, 104, 119, 5, 11, 77]. Tools that make use of context aware data flow analysis, and thus which can take the application context into account also exist [117, 141, 113, 5, 130, 113]. In more recent times, attempts have been made at using machine learning in order to create tools to fix software security vulnerabilities [53, 52]. Lastly, there have been a few attempts at creating tools that can automatically fix security vulnerabilities [140, 130, 129]. For a more extensive list of different approaches identified through the literature review, see Table A.1.

3.2.2 Auto Fixing Based on Bytecode Analysis

As explained in chapter 2, a tool based on static code analysis can be used to examine the source code of a program without attempting to execute it. Most existing tools apply this technique [117, 136, 104, 119, 5, 130] since it allows the detector to identify vulnerabilities while the code is being written and does not require the code to be ran. Other benefits of static code analysis include analyzing the code without needing to provide runtime parameters and early detection as opposed to late detection [117]. Since the code does not need to be able to run for the analysis tool to perform the detection, the code can be analyzed while it is in an incomplete state. Bytecode analysis can be used to create automatic fixes by manipulating the abstract syntax tree representation of the code. This is done by [14, 73, 136].

Most of the tools available for Java perform static code analysis on the bytecode as opposed to the source code itself [5, 130]. According to Logozzo and Fähndrich [75], this is because analysis on the bytecode level brings many simplifications to the analysis. There are fewer cases to handle and a higher degree of independence from the source syntax. Bytecode analysis also brings the analysis closer to the code that is actually executed, enabling the analyzer to analyze libraries whose source code is not available and avoid redoing work performed by the compiler. Another benefit of analyzing bytecode as opposed to the Java source code is that the analyzer can work with any language that compiles to the Java bytecode. For instance, FindSecBugs [5] works with Kotlin, Groovy, and Scala in addition to the Java language, making the tool more capable than if the tool analyzed the Java source code directly.

3.2.3 Dynamic Analysis for Auto Fixing

Dynamic analysis is an approach that evaluates the application at runtime by submitting input and evaluating the returned results [8]. Dynamic analysis is used by [134, 97, 69]. A benefit of using dynamic analysis is that the program can be evaluated without knowing how the source code was implemented. This means that the tool can be run on software

where the source code is not available. Dynamic analysis is also more precise when compared to static analysis. However, it can be difficult to know where in the source code the fault lies, which makes identifying the piece of source code contributing to the vulnerability difficult. The program also needs to be fully working in order for the dynamic analysis to take place [8].

Dynamic analysis can be used together with static analysis to create automatic fixes. This is done by Yan et al. [140]. Their approach is to combine static and dynamic analysis. Existing static memory leak detectors for the C language is used to automatically fix the leaky allocation site reported by the detector along all leaky paths. The leaky allocation sites are then instrumented with dynamic checks to ensure safe fixing at runtime.

3.2.4 Pattern Matching

Pattern matching is one of the most common approaches to static code analysis [96]. It is the approach used by [136, 104, 119, 72, 66, 14] and one of more approaches used by [117, 5] which also uses data flow analysis. Pan, Kim, and Whitehead [108] identified 27 common bug patterns for Java, showing that many common bugs can be identified through pattern matching. Pattern matching can be used to automatically fix security vulnerabilities by replacing the vulnerability identified using pattern matching with a non-vulnerable version. This is done by [136, 72, 66]. Pattern matching does, however, lead to a lot of false positives as identified by the studies [71, 107]. Pattern matching is also unable to follow the tainted values through the program to find where unsanitized input enters and exits the program [117].

3.2.5 Data Flow Analysis for Auto Fixing

Data flow analysis is another approach used by static code analysis tools. Tools that use data flow analysis are able to take the context of the flow of the data within the application into context, which means the tool is able to make informed decisions about how data enters and exits the application, and whether or not the tainted data was correctly sanitized [132]. Data flow is used by [117, 58, 5, 130, 141, 113] to help the detection of vulnerabilities. A weakness of using data flow analysis is the complexity of performing the analysis [117]. The approaches identified through the literature review did not use data flow directly to auto fix software vulnerabilities but used it to assist in detecting the vulnerability, which then was fixed using pattern matching [141, 130].

3.2.6 Machine Learning Approaches

In more recent time vulnerability detection [52] and auto fixing of vulnerabilities [53] have been done using machine learning. Harer et al. [52] describe a data-driven machine learning approach where different machine learning model was trained using generated features based on the output of Clang static analyzer to detect vulnerabilities. Measuring the Area Under Curve (AUC) of the ROC curve (receiver operating characteristic curve) metric, they get a result of 0.82, and using the AUC of the precision-recall curve, they get a result of 0.32. They mention that a weakness of using machine learning approaches is that there does not exist a labeled dataset that can be used for training the machine

learning models. Harer et al. [53] use a generative adversarial network to repair software vulnerabilities by turning bad source code into good source code. The tool was tested on the SATE IV dataset [94] and was found to repair source code errors in the dataset.

3.2.7 Other Security Auto Fixing Approaches

Thomas, Williams, and Xie [129, 128] claims to have constructed an algorithm for automatically generating prepared statements to remove SQL injection vulnerabilities. Based on their experimental results performed on four open source web applications they were able to successfully replace 94 % of the SQL injection vulnerabilities present in the projects. Dysart and Sherriff [41] implement a tool for PHP using this algorithm. The tool was evaluated on phpBB, an open source forum system, where it found 663 SQL statements, of which 328 were found to be possible SQLi vulnerabilities, all of which were successfully fixed using prepared statements. A weakness of the algorithm is the assumptions made, for instance, it cannot handle data structures that shift pointers when getting data, or statements dependent on multiple files.

Chen and Li [21] created a tool that examines the client host according to information security requirements to find out if the system configuration meets the requirements. The tool modifies the Windows registry to register the correct parameters according to 31 types of misconfigurations. In their testing, the tool is more efficient than examining the client hosts manually. The author does not discuss any weaknesses of the approach.

3.2.8 Classical Auto Bug Fixes

In this section different classical approaches to fixing bugs not related to security is detailed. Classical approaches to bug fixes are of interest because the existing approaches could be re-used to fix security related vulnerabilities.

Code Search

Studies [72, 73, 62, 138, 65] use code search to identify similar code in order to create patches. They create a database of existing patches and use semantic code search to identify which patch to apply. Liu et al. [72] creates code fixes by matching faulty code with a given pattern from a database of mined fix patterns. The repair action is performed on an abstract syntax tree representation of the program. Jeffrey et al. [60] maintains a database of bug-fixing scenarios. These bug fixes are then used to apply patches in debugging situations. They also use a machine learning approach to learn about new debugging situations. Liu et al. [72] argues that the main strength of using code search is that knowledge about existing ways to fix bugs can be applied in a new context. However, this is also the main weakness, since the bug cannot be fixed if the database does not include the knowledge necessary.

Approaches Based on Pattern Matching

Classical approaches for automatically fixing bugs also make use of pattern matching [66, 14]. Kim et al. [66] generates patches based on 10 fix templates identified by inspecting

human written patches. They then use test cases to calculate the fitness of each patch candidate in order to identify which patch to apply using an Abstract Syntax Tree representation of the program. Test cases are used to identify if it is a successful patch. Balachandran [14] takes as input the output of a static analysis tool named Review bot. The code is turned into an abstract syntax tree representation of the code. Transformations are then performed on the abstract syntax tree according to a collection of known patterns to auto fix the code. Both authors argue that pattern-based automatic program repair can use templates to fix known patterns of vulnerabilities, however, bugs that do not fit the pattern cannot be fixed.

Genetic programming

Approaches [91, 135] make use of genetic programming to automatically create patches. Nguyen et al. [91] searches over the space of replacements from the other expressions in the program using genetic programming to device a patch. The tool then derives a repair constraint from a set of tests and then generates a valid repair by solving the repair constraint. They argue that the strengths of using genetic programming are that the technique can evolve variants of the patch to fix a variety of bugs. However, the technique is slow and has a low success rate in their testing. Weimer et al. [135] evolves program variants for locating and repairing bugs in C programs using genetic programming by using an abstract syntax tree representation of the program. Test cases are used to verify the repairs. They identified a success rate of 54%. However, they also found that the performance of generating the patches is low.

Other approaches

Xuan et al. [137] uses angelic fix location together with an instance of a Satisfiability Modulo Theory problem in order to fix buggy if-then-else statements given a buggy program and test suites with at least one failing test suite. It then tries to find a solution to the Satisfiability Modulo Theory problem in order to generate a patch. Muntean et al. [90] also uses a Satisfiability Modulo Theory problem in order to fix buffer overflow vulnerabilities. They argue that the benefit of using their approach is that it is faster than symbolic execution. However, angelic fix location may lead to infinite loops, and the approach requires test cases to be correct.

Nguyen et al. [92] creates a tree-based representation of a PHP program using symbolic execution, which estimates the possible HTML client page outputs. A mapping between any text in the HTML page and PHP code location is created. An HTML validation tool is then used to find validation errors in an HTML page, and the fixes are propagated to the PHP code. They argue that the benefit of using symbolic execution and a mapping between the PHP code and the HTML page is that it achieves high accuracy and high performance. However, they use an approximation of the symbolic execution which means that all possible paths are not represented.

Coker and Hafiz [23] uses program transformations to fix integer related vulnerabilities. The first transformation explicitly introduces casts to make integer use clear. The second transformation replaces arithmetic operations with safe functions to detect overflows and overflows at runtime. The third transformation changes the types of integer to

fix signedness and widthness problems. They argue that the approach is capable of handling all types of C integer problems. They do not discuss any weaknesses of the approach.

3.2.9 General Limitations of Existing Auto Fix Tools

Johnson et al. [63] performs a study in which they investigated why developers are not using static analysis tools and what would need to be changed in order for developers to start using them. 20 participants with developer background were interviewed of which all mentioned the beneficial use of static code analysis tools but pointed out that the high rate of false positives and the way the warnings were presented provided too high of a barrier for using the tools in practice. They point out the importance of having helpful links to detailed examples of how to fix the issue in the error reported by the tool, and further points out that there is a lack of or ineffectively implemented quick fixes. Most of the participants in this study expressed an interest in getting code suggestions or quick fixes from the tool. They also wanted the possibility of previewing how a possible fix would affect their code before applying it e.g. by viewing the difference before and after applying the fix in the editor to show how the code would be affected.

A different study by Baset and Denning [16] evaluated 17 tools, of which two of the 17 tools evaluated were found to provide automatic fixes, namely ASIDE and Codepro AnalytiX, showing that few existing tools provide automatic fixes. They highlight that there is a big difference in the quality and thoroughness in the analysis of the plugins. Other tools they evaluated can be seen in Table 3.1. They explain that the plugins they focused on check for the most common input validation vulnerabilities, but only three of them (ESVD, FindBugs, and LAPSE+) are capable of identifying more than 6 of the vulnerabilities they focused on, showing that most tools do not cover many vulnerabilities.

Vulnerability checks	CWE	Android Lint	ASIDE	Checkmarx	ESVD	Findbugs	Fortify	FxCop	Goanna Studio	Klocwork Insight	LAPSE+	SecureAssist	SonarLint	Veracode
Improper input validation	20	-	✓	-	-	✓	-	-	-	-	-	-	-	-
Command injection	77	-	-	✓	✓	✓	-	-	✓	-	✓	-	-	-
OS Command Injection	78	-	-	✓	-	✓	-	-	✓	-	✓	-	-	-
Cross-site Scripting	79	-	-	✓	✓	✓	✓	✓	-	✓	✓	-	-	-
SQL Injection	89	-	-	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	✓
LDAP injection	90	-	-	-	✓	✓	✓	-	-	-	✓	✓	-	-
XML injection	91	-	-	-	-	-	-	-	-	-	✓	-	-	-
Unsafe Reflection	470	✓	-	-	✓	-	-	-	-	-	-	-	-	-
XPath injection	643	-	-	✓	✓	✓	-	-	-	-	✓	✓	-	-

Table 3.1: Input-validation related vulnerability Coverage by Baset and Denning [16]

Codepro AnalytiX used to be available on the following URL `https://developers.google.com/java-dev-tools/codepro/doc/`, but seems to have been deprecated. Using an archived version of the URL, it seems like the latest release of the project was in September of 2010 [49]. A proposal was made to turn Codepro AnalytiX into an

open source Eclipse project according to Eclipse Foundation, Inc [44] and Google, Inc [50], but this never got past the proposal phase. Since the source code for the tool cannot be obtained, we conclude that this project cannot be used in order to identify its approach to quick fixes, nor to verify which quick fixes the tool provided.

3.3 Existing Test Beds

As shown in Table A.1, many tools evaluated in the literature review were evaluated on different test beds, making comparison difficult. Many tools were evaluated against open source software where some vulnerabilities were known, and some on custom made and vulnerable test beds, but none of the security test beds were created for auto fixing, and thus lacked any mechanism for testing effectiveness of fixes. A common test bed with checks for security fixes is required for a thorough evaluation and comparison of different software security auto fix tools.

3.3.1 Test Beds Used for Evaluations of Vulnerability Detection and Fixing Approaches

As shown in Table A.1, many tools are evaluated on vastly different test beds. The different test beds used are listed in Table 3.2. This table does not describe evaluations using home-made code, or open source software. No documentation could be obtained for DroidBench which is why it is not detailed any further. The documentation for Stanford SecuriBench was found to be lacking in detail, and the test suite was only used once and it is therefore not focused on.

Test Bed	Design	Language
Juliet [95]	Collection of vulnerable functions	Java, C/C++
WebGoat [106]	Complete application	Java
DroidBench testsuite	-	Java
Stanford SecuriBench [122]	Collection of real-life applications	Java
Defects4J [64]	Collection of vulnerable functions	Java
ManyBugs [78]	Collection of vulnerable functions with associated test cases	C
Bugs.jar [114]	Collection of vulnerable functions	Java

Table 3.2: Test beds discovered through literature review

Juliet

The Juliet Test Suite is a popular test suite used for source code analysis of security vulnerabilities [51, 90, 71]. Juliet Test Suite for Java contains 28,881 vulnerable test cases for a large number of different CWE classes, making it the largest test bed used for the evaluation of static analysis tools. The test bed is divided by CWE category and each vulnerable class inherits from a superclass that contains common logic between all the different vulnerable classes. This is done to make both development and evaluations easier using the test bed.

WebGoat

WebGoat is a deliberately vulnerable web application created for teaching security [106]. WebGoat is designed as a complete application, as opposed to a set of vulnerable functions like Juliet. The application is designed to be more realistic than a set of vulnerable methods and can be attacked during runtime through a web interface. Though the application is not primarily designed for static analysis it was used by multiple researchers [139, 129, 117], primarily for its realistic design.

Defects4J

Defects4J is a test set consisting of a collection of vulnerable functions gathered from real-world applications. The tools authors Just, Jalali, and Ernst [64] note that this is done to allow the test bed to be extensible.

Bugs.jar

Bugs.jar is a test bed comprised of 1,158 bugs from real open source Java programs [114]. The test bed is designed as a set of vulnerable functions designed to allow modification and addition of new vulnerabilities.

Open Source Software and Proprietary Test Beds

Many tools are evaluated with proprietary test sets [53, 19] or open source projects [81, 140, 113, 141]. These tests are often not made public and are also not very clearly described. Most of these only explain how many test cases they have and what sort of vulnerability is covered. They lack information about how the insecure functions are constructed. There is little information about how many functions are tested that are not vulnerable. This means that many do not test for false positives. It is also unclear how complicated the data flow in the tested functions was. It is therefore hard to tell if these tools can detect if data is cleaned in a different function that is called by the evaluated function, or if the software is constructed to always only call a function with sanitized variables.

[107, 1, 39] are easier to replicate due to using open source and publicly available test sets, however, due to not disclosing how they treated false positives, e.g. if they had some sort of ground truth it is still difficult to know where these results were obtained from.

[107, 1, 39, 80] evaluates on open source code, namely PuTTY, Nmap and Wireshark. [20] does not specify which test set was used to generate the test cases. [16] compares tools

based on what the tools claim to be able to do, but doesn't run the tools to evaluate them. [13] evaluates only one tool, Coverity Prevent, on four anonymized products. [17] designs a custom benchmark suite to rank Static Analysis Tools (SAT) and compares this suite with OWASP BSA and SAMATE. In general, it is difficult to replicate the results done by [80, 20, 13, 17], since they do not specify the details of the custom test sets that were generated. [107, 1, 39] are easier to replicate due to using open source and publicly available test sets, however, due to not disclosing how they treated false positives, e.g. if they had some sort of ground truth it is still difficult to know where these results were obtained from. Other studies are evaluations done by the creator of the tool themselves [141, 117, 130, 136, 132]. [141, 130, 136] evaluates on real-world open source applications. [117] evaluates on people in terms of ease of use, and benchmark applications. [136] evaluates on a custom application, Apache Roller.

3.3.2 Classical Test Beds

ManyBugs [78] is a test set for C that implements validation of auto fixes. The test bed consists of a collection of buggy functions, and each function has associated tests that can check the functionality of the function. This means that the test bed can check the correctness of repairs.

3.3.3 Metrics used for Evaluation

The metrics collected by researchers vary drastically. Evaluations based on well-known test sets often use standard metrics, like recall and precision. Evaluations on open source software typically present a limited set of metrics in their results. Many only list the number of bugs found, but since the total number of bugs is unknown they cannot give recall or precision.

A significant number of papers [77, 111, 134, 73, 74] also only claim to have fixed a number of vulnerabilities, but without explaining what vulnerability or how many vulnerabilities were missed, or how many false positives were found before the correct one was detected.

Many also make generalized claims to be better than other tools or better than the state of the art [117, 98, 130]. A few studies [139, 53, 138] also obfuscate their results so much that they are hard to understand, or they present no concrete results at all.

3.3.4 Evaluation Methodology

The tools proposed were all tested in different ways. Some use public test sets like the Juliet Test Suite [95], but all have different ways of evaluating the results of the tests. It is unknown how robust these tests are. It is for example not clear if tests using the Juliet Test Suite check if vulnerabilities were detected in the correct place in a given test file. E.g. it is possible that a tool detected an SQL injection vulnerability in a test case that is vulnerable to injection attacks, but that the detected vulnerability was a false positive somewhere else. This is an issue since the Juliet suite does not clarify where the problem is, or what the solution is. There are also no regression tests to check that a suggested solution does not break the intended functionality.

In general, the evaluations of the tools examined are hard to reproduce. The test sets are mostly not made public, and the way the test sets were used is often not well explained. Many of the tools are also closed source, or even not published at all, making it impossible to evaluate them properly.

3.4 Studies into the Prevalence of XML External Entities

Jan, Nguyen, and Briand [59] studied the presence of the Billion Laughs attack and XML External Entities attack in 13 popular parsers. These parsers were chosen due to being included with the programming languages Java, Python, PHP, Perl, and C#. Popular open source parsers were also selected for evaluation. Using search queries, the open source hosting systems GitHub and Google code were analyzed for the prevalence of these XML parsers. They found that the parsers together had been used over half a million times as shown in Table 3.3.

The CPU and memory usage of the parsers were evaluated when parsing billion laugh example files of different sizes. On the smallest example, they observed a ram usage of 33MB, whereas on the largest example they experienced ram usage of 8Gb which was the limit of their test. They found that the CPU usage ranged from less than 2 minutes to 51 minutes depending on the parser. They also checked if open source systems that use the vulnerable XML parsers DocumentBuilder and SAXParser remember to apply mitigation strategies to prevent these attacks. 1000 Java source files were selected. Then, all the Java projects that did not parser XML inputs were filtered. After filtering, they were left with 628 open source projects containing 749 Java files of which 735 were found to be vulnerable (98.13%) not setting any of the attributes they were looking for.

Parser	Query	GitHub	Google Code
JDOM2	org.jdom2.input.SAXBuilder	2,861	9,380
NanoXML	net.n3.nanoxml.IXMLParser	1,410	291
NanoXML-LITE	nanoxml.XMLElement	6,057	4,380
Std-DOM	javax.xml.parsers.DocumentBuilder	112,638	58,900
Std-SAX	javax.xml.parsers.SAXParser	43,307	11,200
Std-STAX	javax.xml.stream.XMLStreamReader	84,826	4,840
WOODSTOX	org.codehaus.stax2.XMLStreamReader2	252	251
XERCES-JDOM	org.apache.xerces.parsers.DOMParser	3,444	1,440

Table 3.3: How frequently different parsers for Java have been used as described by Jan, Nguyen, and Briand [59].

Späth et al. [120] analyzed known attacks on 30 different XML parsers executing 1459 different attacks to identify vulnerabilities in the default configuration of these parsers. They also parser features that can be set to prevent the different attacks. These were classified into the categories prevention, counteraction, and limitation. For the attacks presented, using the prevention mitigation strategy by disabling the insecure parser features was applicable in all cases. Of all the Java parsers they evaluated, all were vulnerable to denial of service attacks such as the billion laughs attack, external entity attacks through

file system access, and Server-Side Request Forgery (except for the KXml parser). Using a custom entity resolver mitigates the XXE vulnerability and some SSRF attacks by filtering the input. Setting secure features such as *disallow-doctype-decl* to true disables all DTD processing mitigating all these attacks. Setting the feature *EXPAND_ENTITYREF* to false mitigates the denial of service attacks, the XXE attacks, and some SSRF attacks. however not all the features work as expected due to different parser API's supporting setting features but not implementing them.

Tiwari and Singh [131] surveyed possible attacks on web services and composite web services. They found that DOS attacks due to large payloads are common for web services that rely on SOAP messages and XML syntax. These attacks exploit the resource consuming nature of XML parsing, mentioning that DOM model parsers are more susceptible to large payloads when compared to other XML parsing techniques. Mitigation strategies include restricting the length of XML elements and the number of elements.

Oliveira, Laranjeiro, and Vieira [100] implemented a tool based on WS-Attacker for testing the security of web service frameworks by dynamically evaluating them. The tool contains the nine attack types coercive parsing, malformed XML, malicious attachment, oversized XML, soap array attack, XML bomb, XML document size, repetitive entity expansion, and XXE. These attacks can be run against a live version of the service to be tested. Evaluating Apache Axis 1 and Apache Axis 2, they found that both were vulnerable to numerous of the vulnerabilities tested including oversized XML and XML document sizer vulnerabilities. An extension to the dynamic testing tool WS-attacker for testing DOS attacks against XML parsers was also created by Falkenberg et al. [45]. In their evaluation, all the parsers were vulnerable to XXE attacks.

Chapter 4

Research Design

In this chapter, the design of the research will be presented. First, the purpose of the research will be presented. Then the method used to answer the research questions is detailed. The participants of the research and their roles in the research will then be listed. The research paradigm will then be presented before finally the deliverables from the research performed will be listed. Parts of the research design is reused from the prestudy [89].

4.1 Motivation

This project aims to improve the detection and fixing of XML External Entity vulnerabilities in Java source code. A successful XXE attack can have multiple severe consequences such as denial of service, information extraction, and remote code execution. All XML parsers in Java are vulnerable by default and they have to be made secure manually. It is therefore important to focus on helping developers mitigate XXE vulnerabilities.

XXE is listed as number 4 on OWASP's list of top 10 most critical vulnerabilities in web applications. XML External Entity reference is also on the CWE Top 25 most dangerous software errors listed in Table B.4.

Web security tools for developers mostly focus on the detection of vulnerabilities rather than fixing them. FindSecBugs [5] it a popular tool that provides detection of vulnerabilities using pattern matching. Since previous research has focused on the detection of vulnerabilities, there are currently no test beds created for auto fixing tools. Because of this, evaluations of auto fixing tools often use various code bases for evaluation.

Previous research has focused on the detection of vulnerabilities, not fixing them. Most of this research has also been focused on detecting SQLi and XSS, while little research has been done into detection or fixing of XXE. There are currently no tools that provide auto fixes for XXE and detection of XXE is not yet mature, mostly relying on pattern matching which is not enough to consistently detect XXE.

4.2 Research Questions

Based on the results of the literature review from the prestudy the following research questions were selected:

RQ1: How can a test suite for evaluating web sec auto fixes be designed for XML External Entity attacks?
RQ2: How can detection of XXE be improved?
RQ2.1: What are the shortcomings of existing static analysis tools for the detection of XML External Entity attacks?
RQ2.2: How can the detection of XML External entities be improved using different techniques?
RQ3: How can auto fixing of XML External Entities be implemented using an IDE plugin?

4.3 Research Method

The research topic given by the project supervisor was creating an auto fixing tool for web security vulnerabilities as an IDE plugin. To gain an understanding of the state of the art in the field of vulnerability auto fixing, a literature review was conducted in the prestudy. A summary of this prestudy can be found in Appendix B. The results of the literature review were used to select appropriate research questions for the master thesis. An overview of the research method is shown in Figure 4.1.

The first research question will be answered by creating a test bed similar to Juliet in design but altered to accommodate the validation of fixes as well as validation of the functionality after the fixes are applied. The test bed will be evaluated based on how well it works for collecting the necessary metrics to evaluate an auto fixing tool.

The second research question will be answered by evaluating the best existing XXE detection tool to discover its limitations. Then a selection of vulnerability detection methods discovered in the prestudy will be evaluated to find out which technique works best to detect more vulnerabilities. The best candidate technique will then be implemented and evaluated on a test bed. The number of true positives, false positives, and false negatives will be collected from the evaluation, and recall and precision will be calculated using the test bed we designed. These will then be compared to the evaluation results for the best existing tool.

The third research question will be answered by creating an auto fixing tool based on FindSecBugs. FindSecBugs was discovered to be the most promising vulnerability detection tool for extension through the literature review. The auto fixing will be implemented using techniques discovered in the prestudy. The auto fixing will be evaluated on the test bed to discover how well it performs. The number of correct fixes, incorrect fixes (that try to fix a bug, but fail), and bugs that are missed completely (e.g. false negatives) will be presented using the test bed we designed. The distinction between incorrect fixes and missed fixes will be used to argue whether the problem was lacking detection or lacking auto fixing.

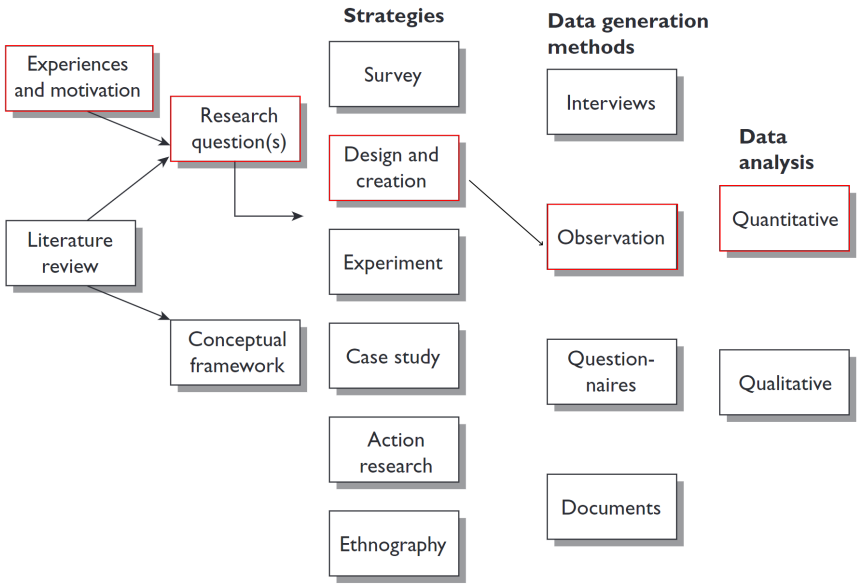


Figure 4.1: Visualization of research method used for the project

4.4 Participants

Torstein Molland and Andreas Berger, both master students in computer science, are the main participants of the research project. Their task is to plan and perform the research. The project supervisor, Jinguye Li, associate professor at the Department of Computer Science at NTNU, will monitor the progress of the research. Li will also contribute with his experience within the field of software security.

4.5 Research Paradigm

The research uses the scientific method and therefore is within the positivist paradigm. First, a test bed will be created. This will be used to evaluate the existing tools to find out what is missing. Then the findings of the literature review will be used to select possible detection and auto fixing techniques to use. These techniques will be implemented and evaluated to find out which performs best. This project aims to find patterns and regularities in fixing web security vulnerabilities through experimentation. A hypothesis will be proposed and attempted refuted. Conclusions will be drawn from the data generated. The hypothesis will be refuted if other researchers cannot replicate the experiments or get different results.

4.6 Final Deliverables and Dissemination

The research results will be presented in a master's thesis and in a research paper submitted to The Asia-Pacific Software Engineering Conference (APSEC). The research paper is listed in Appendix K. The source code for the test bed, the new vulnerability detectors, and the auto fixing tool are available on the projects GitHub page [18].

Chapter 5

Research Implementation

In this chapter, the research methods used to answer the research questions will be presented. Prior to this master thesis, a prestudy was conducted where the state of the art within software security auto fixing was identified through a structured literature review. The results of this literature review were used to identify the research questions which have been answered in this master thesis. RQ1 was answered by implementing a test suite design suggested in the prestudy using the result of analyzing the strengths and weaknesses of different test beds identified through the literature review. RQ2 was split into two sub-research questions. RQ2.1 was answered by performing an evaluation of FindSecBugs, which is the tool that was found to be the most capable in the prestudy. RQ2.2 was answered by improving the existing tool using techniques identified in the prestudy, as well as new techniques. RQ3 was answered by implementing a new auto fix mechanism for XXE vulnerabilities using auto fix techniques identified in the prestudy. For both RQ2 and RQ3 the evaluations were performed using this test suite to generate quantitative data.

5.1 RQ1: How can a Test Suite for Evaluating Web Sec Auto Fixes be Designed for XML External Entity attacks?

In the prestudy, the state of the art in evaluating auto fixing tools and methods were identified through a structured literature review. The result of this review showed that different test suites for evaluating the performance of the detection of software security vulnerabilities have been researched. A test bed for evaluating the performance of classical bug auto fixes was also found. However, no test bed for evaluating the auto fix performance of software security auto fix tools was identified. No test bed for evaluating the detection performance of XXE vulnerabilities were found either. Juliet Test Suite was found to be the most complete test suite for evaluating the detection performance of static analysis tools for software security and seemed the most promising to extend with automatic verification of both the functionality of and the security of auto fixes, as well as test cases for XXE

vulnerabilities. RQ1 was created as a result of this. RQ1 was answered by designing and creating a new test bed for automatically evaluating the functionality of and the security of software security auto fixes with focus on XXE vulnerabilities based on flow variants from Juliet Test Suite. The test bed was evaluated by using the detection and auto fix methods implemented as part of RQ2 and RQ3, then running the evaluation mechanism in the test bed, and finally checking if the test bed generated sufficient quantitative data to be able to evaluate the performance of the detection and auto fixing tools. It was discovered that the flow variants from Juliet Test Suite could not sufficiently detect XXE vulnerabilities. Therefore 11 additional test cases with more complex data flows were implemented. These aimed to find out the limits of intraprocedural analysis in FindSecBugs, i.e. the test cases were limited to one method and instead introduced multiple instances of parsers and other objects.

5.2 RQ2: How can Detection of XXE be Improved?

In the prestudy, existing tools were identified through a structured literature review. The result of this showed that not a lot of research had been done into the detection of XXE vulnerabilities. It was discovered that FindSecBugs is the best performing tool for detection, which is why it was chosen as the tool to evaluate. RQ2, as well as its sub research questions, were created as a result of this. RQ2.1, What are the shortcomings of existing static analysis tools for the detection of XML External Entity attacks, was answered by evaluating FindSecBugs, since it was found to be the tool that seemed the most promising in the prestudy. This was done by evaluating on a test suite created with the same flow variants as the Juliet Test Suite [93] to gather quantitative data on the performance of the existing detectors within FindSecBugs. The number of true positives, false positives, and false negatives was used to calculate the precision and recall. The existing detectors were also evaluated on the entirety of Juliet Test Suite to gather quantitative data on the detection performance on test cases with no vulnerabilities. The time performance was evaluated by measuring the execution time on the entirety of Juliet Test Suite and the test bed with XXE vulnerabilities.

In the prestudy, the strengths and weaknesses of software security tools were also identified focusing on the different techniques used by these tools. RQ2.2, How can the detection of XML External entities be improved using different techniques, was answered by designing and creating improved detectors for XXE vulnerabilities based on the strengths identified in the prestudy. The new detectors were evaluated using the same test suite as RQ2.1 to be able to compare the results to each other by analyzing the quantitative data. The techniques were first evaluated based on how well the data they provided could be used to improve upon the existing detectors in FindSecBugs. Then, the most promising technique was fully implemented within FindSecBugs and evaluated on all the tests for XXE created for the evaluation. The detectors were also evaluated on the entirety of Juliet Test Suite, similarly to the evaluation for RQ2.1, to see if the new detection would increase the detectors' false positive rate since Juliet does include XXE vulnerabilities. The number of true positives, false positives, and false negatives was used to calculate the precision and recall. The quantitative data was then assessed to identify strengths and weaknesses compared to the existing detection mechanism for XXE vulnerabilities in FindSecBugs.

The time performance was evaluated by measuring the execution time on the entirety of Juliet Test Suite and the test bed with XXE vulnerabilities.

5.3 RQ3: How can Auto Fixing of XML External Entities be Implemented using an IDE Plugin

In the prestudy, strengths and weaknesses of different auto fix tools were identified through a structured literature review both focusing on software security but also on classical bug fix tools. The result of this review showed that no auto fix mechanism for XXE vulnerabilities had been researched. Existing classical bug fix tools and auto fix tools for software security were found to use abstract syntax trees to perform their quick fixes and were found to perform well. Additionally, FindSecBugs was found to be the most promising tool to extend with auto fix functionality, since it was discovered to be an extensible tool minimizing the groundwork needed to be done. RQ3 was created as a result of this. RQ3 was answered by designing and creating a new auto fix mechanism for XXE vulnerabilities based on modifying the abstract syntax tree. The auto fixes were implemented using parts of [67], which contains fixes for classical Java bugs detected by SpotBugs. This repository was used to get a working setup for connecting FindSecBugs to the Eclipse auto fix API and does not contain code for auto fixing XXE or other web security vulnerabilities. The auto fix implementation was evaluated using the same test suite as in RQ2 on the number of successful fixes, the number of incorrect fixes, and the number of missed fixes. Missed fixes and incorrect fixes were distinguished between to separate lacking detection and lacking auto fixing from each other since a missed fix suggests the vulnerability was missed by the detector, while an incorrect fix suggests an error in the auto fixing mechanism. The quantitative data generated from this evaluation was then used to assess the performance of the auto fixes. The time performance was evaluated by measuring the time to auto fix each vulnerability in the test bed with XXE vulnerabilities.

Chapter 6

Research Results

This chapter presents the results of the research conducted in this project. First, the design of the test bed will be presented. Then the evaluation of the existing XXE detection is detailed. Next, the improved XXE detection is presented and evaluated. Finally, the auto fix mechanism for XXE vulnerabilities is presented and evaluated.

6.1 RQ1: How can a Test Suite for Evaluating Web Sec Auto Fixes be Designed for XML External Entity attacks?

One of the results of the literature review conducted in the prestudy, as shown in Appendix B, was that there was a lack of test beds designed for evaluation of automatic fixes of software security vulnerabilities. Therefore, such a test bed was designed as part of the prestudy. In this master thesis, this design was used to implement a test bed for the evaluation of automatic fixes of software security vulnerabilities with focus on XXE vulnerabilities. This test bed was used to evaluate the detection of XXE. This test bed was also used to evaluate the automatic fixes of XXE vulnerabilities, to find out how well the test bed performed for evaluating the correctness of auto fixes for XXE.

6.1.1 Design of Existing Test Beds

To ensure the test bed had a representative enough number of test cases for the different flow variants tested it was designed based on the flow variants from the Juliet Test Suite. The overall design of the test suite is shown in Figure 6.1. The flow variants within Juliet Test Suite are shown in section D.2. Juliet Test Suite was found in the prestudy to be the most common test bed used for evaluating web security detection tools. It is designed as a collection of vulnerable test cases categorized by their corresponding CWE codes using the different flow variants described. Not all flow variants are applicable to test all vulnerabilities [93].

Since Juliet Test Suite does not include test cases for XEE attacks against different XML parsers [93], these were added using the previously mentioned relevant flow variants. These test cases are classified by the CWE codes CWE-611 and CWE-776 and were implemented for the parsers DocumentBuilder, XMLStreamReader, XMLEventReader, FilteredReader, SAXParser, XMLReader, and Transformer since these were found to be the XML parsers the existing detectors in FindSecBugs detected vulnerabilities from as shown in subsection 6.2.1. Each test case tests a different flow variant and has a *good* method that tests the flow variant using a secure XML parser and a *bad* method that tests the flow variant using an insecure XML parser.

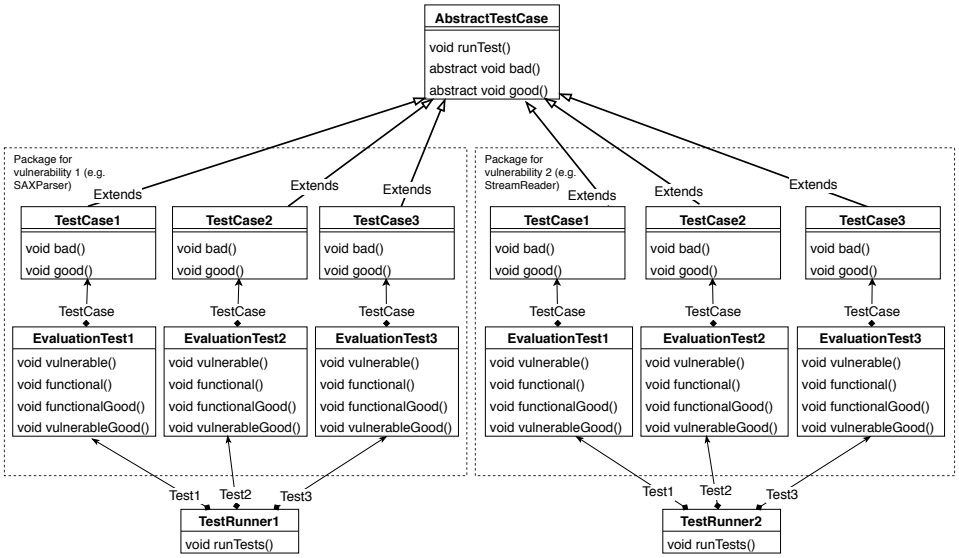


Figure 6.1: Overview of test bed design

6.1.2 Explanation of Juliet Style Test Cases

The test bed includes test cases created to be similar in data flow to those found in the Juliet Test Suite. These test cases were created for the seven XML parsers included in the test bed. The first 17 variants from Juliet shown in Table D.2 were chosen since these are the ones that are applicable to XXE and only include intraprocedural data flows.

6.1.3 Explanation of Instance Based Test Cases

The flow variants within Juliet Test Suite were found to cover different control flows related to if statements, switch statements, and loops well. However, control flows and data flows for invoking methods on an instance were found to be missing. This included different ways of initializing an object as well as different ways of invoking methods on an object instance which may affect the detection and auto fix performance of a tool. 11

additional test cases were enumerated to better cover these cases. These are shown in Table D.1.

- Six test cases with variations of class field and method variable an example of which is shown in Listing 8
- Four test cases with multiple parsers being made secure and insecure in the same method an example of which is shown in Listing 9
- One test case where an object with the same secure method as an XML parser, and an XML parser. If instances are not tracked, it is impossible to know if the *.setFeature()* method has been called on the factory or on a different object. An example of which is shown in Listing 10

```
// Test case 1
InputStream inputStream = new FileInputStream(filePath);
// Factory initialized into method variable
SAXParserFactory factory = SAXParserFactory.newInstance();
// parser initialized into class field
parser = factory.newSAXParser();
PrintHandler handler = new PrintHandler();
parser.parse(inputStream, handler);
```

Listing 8: Example of test case with variation of class field and method variable

```
// Test case  7
SAXParserFactory factory = SAXParserFactory.newInstance();

SAXParser parser1 = factory.newSAXParser();
PrintHandler handler1 = new PrintHandler();
parser1.parse(inputStream, handler1); // Insecure

factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

SAXParser parser2 = factory.newSAXParser();
PrintHandler handler2 = new PrintHandler();
parser2.parse(inputStream, handler2); // Secure
```

Listing 9: Example of test case with multiple parsers. The first is insecure and the second is secure.

```
// Test case 11
Bar b = new Bar();
// Calls the setFeature method with correct parameters
// but on the wrong object
b.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

InputStream inputStream = new FileInputStream(filePath);
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
PrintHandler handler = new PrintHandler();
parser.parse(inputStream, handler); // Insecure
```

Listing 10: Example of test case with a different object with the same secure method as the XML parser

6.1.4 Included XML Parsers

The test bed includes test cases for the seven Java XML parsers shown in Table 6.1. It was discovered that not all parsers could feasibly include all the 11 test cases enumerated in Table D.1 because the data flows described did not make sense. For XMLStreamReader, FilteredReader, and EventReader, cases 9 and 10 were excluded since those cases require parsing twice, something that does not make sense for these parsers since they parse using iterators and consume the iterator after finishing. Thus, trying to parse again would yield no items in the iterator and the parsing would be skipped. XMLReader lack cases 1, 2, 3, and 5 because these require instantiating the parser and its factory separately. This is not possible with XMLReader since the parser is created directly.

Parser	No. of Juliet Style cases	No. of Custom Cases
SAXParser	17	11
DocumentBuilder	17	11
EventReader	17	9
FilteredReader	17	9
TransformerFactory	17	11
XMLReader	17	7
XMLStreamReader	17	9
Total	119	67

Table 6.1: Test cases included for each XML parser

6.1.5 Evaluation Process using the Test Bed

The test bed is designed with validation of fixes in mind. The way the test bed evaluates fixes is by using a set of tests that check for correct functionality and checks the security of the test cases in the test bed. The validation is performed as shown in Figure 6.2. The tests are run after the auto fixes are applied and output a list of insecure functions and functions where functionality has been broken. Since the automatic evaluation only tests for exceptions, case 7 to 10 needs to be evaluated manually since these have multiple parsers that can raise exceptions. When evaluating detection using the test bed manual evaluation of correctness is required. The automatic evaluation only works for auto fixes since the mechanism is based on JUnit tests.

Before running the automated validation included in the test bed researchers must first manually apply their fixes. With the tool proposed in this research, that is done by running FindSecBugs and either selecting functions to fix by opening the file they are in, or by clicking a button to automatically fix all vulnerabilities for an XML parser. Since the fixes can be applied to all test cases with the press of one button and the evaluation is automatic, an auto fixing tool can be evaluated within a couple of minutes.

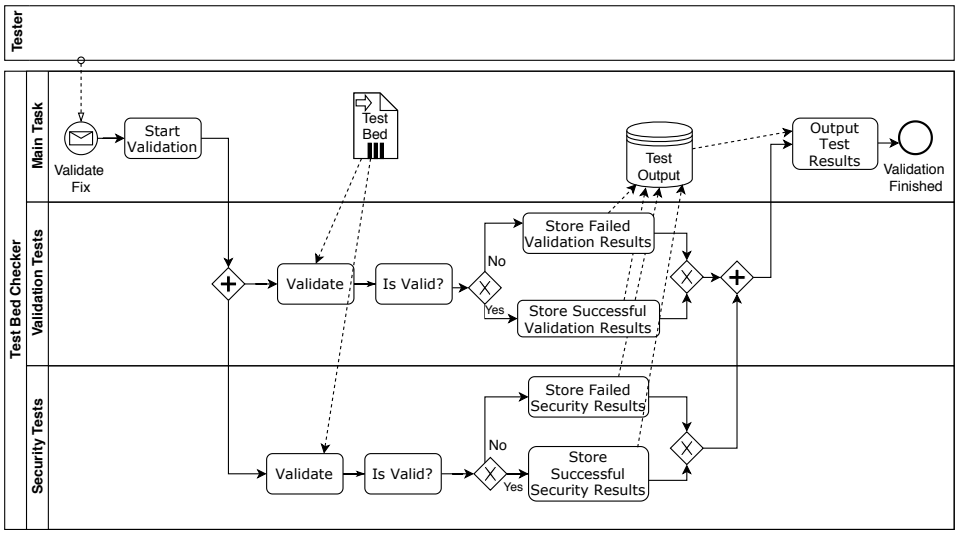


Figure 6.2: Process view of test bed

6.1.6 Testing the Functionality After Applying the Auto Fixes

To make sure the auto fixes do not break the code after fixing a security vulnerability, the test bed implements tests that check for the proper functionality of the code. Since each test case is a function that parses an XML document from a given file path, the function can be tested by parsing a file with no external entities or DTD. These XML documents should be parsed correctly after the vulnerability has been fixed since no dangerous features of the XML parsers need to be used. The file used to validate functionality is listed in Listing 11, and the output of parsing it is expected to be the contents of the <foo>-tags, i.e. “test”.

```
<?xml version="1.0"?>

<foo>test</foo>
```

Listing 11: Example of safe XML file

6.1.7 Testing the Effectiveness of the Auto Fixes

After the auto fixes are applied the test bed checks the security of the parsers in the test cases. This is done by attempting to parse an XML document with external entities trying to read from a file. If the file read is successful and the function returns the contents of the file, the parser is still vulnerable to XXE. If the parser throws an exception when trying to read the file, the parser is secure. The file used for this is listed in Listing 12. The content of the file being read by the external entity is the string “vulnerable”. That means that if the parser outputs the string “vulnerable” when parsing the XML document, the parser is vulnerable to XXE. Since the external entity called ‘xxe’ should not be parsed, the test

case parsing it should raise an exception. The tests validating the security of the test cases therefore also check for raised exceptions to see if the parser is configured correctly.

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

Listing 12: Content of XML file used to evaluate protection against XML External Entities

6.1.8 Output of Automatic Evaluation of Fixes

After running the detection and performing auto fixes on the test bed, the tests for validating the security and the functionality of the test cases in the test bed can be run. An example of a test case to evaluate the security of a parser is shown in Listing 13. These outputs a list of the test cases that were broken as a result of running the auto fixes, and a list of the test cases that are still vulnerable to XXE. This can be used to calculate various metrics such as true positives, false positives, and false negatives, and therefore also precision and recall. Since all test cases are functional before fixes are applied, any test case listed with broken functionality will have been broken by the auto fix. Test cases listed as still vulnerable to XXE after the fix indicates a missed or incorrect fix. If a test case starting with the name *good* is listed as insecure or with broken functionality it means that there has been applied a “fix” to a test case that was already secure and that the ”fix” negatively affected the security and/or functionality of a function that should not have been touched by an auto fixing tool. Conversely, if a test case starting with the name *bad* is listed as secure and the functionality is intact, then a fix has been applied successfully.

```
@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();
  }
  assertFalse(vulnerable, "Parser should not be vulnerable to XXE");
}
```

Listing 13: Test for checking security of parser

6.2 RQ2.1: What are the Shortcomings of Existing Static Analysis Tools for the Detection of XML External Entity Attacks?

In this section, the results for RQ2.1 will be presented. First, the approach used by the existing detectors in FindSecBugs will be presented. Then the results of evaluating these detectors on the test bed presented in section 6.1 will be shown. Finally, the strengths and weaknesses of the existing approach will be detailed in light of these evaluation results. The latest version of FindSecBugs at the time of writing is version 1.10.1 which was released on the 17th of October 2019 [7]. This version was used in all of the evaluations in this section.

6.2.1 Analyzing the Approach used by Existing XXE Detectors in FindSecBugs

Three detectors for identifying XXE vulnerabilities were found in FindSecBugs. XxeDetector reports vulnerabilities for the parsers SAXParser, XMLReader, and DocumentBuilder. TransformerFactoryDetector reports vulnerabilities for the Transformer parser. XMLStreamReaderDetector reports vulnerabilities for the parsers XMLStreamReader, XMLEventReader, and FilteredReader.

The detectors work in a similar fashion which is summarized in the pseudocode shown in Listing 14. For implementation-specific details see Appendix E. First, a stack of opcodes for the method under consideration is analyzed to find where a parser has been used to parse an XML document. This is done by comparing the class name of the opcode and the method signature with that of the parser. The TransformerFactoryDetector finds where the parser has been initialized instead of where it has been used. Then the control flow graph of the method under consideration is obtained. The name of all the called methods is compared to the name of the secure methods for the parser. If only one call is required, the detector marks the parser as safe if this is found. If multiple calls are required, then the detector attempts to identify all of these.

XxeDetector only checks the first parameter of the method, which means that for the secure calls that require two parameters, the second one is disregarded of. For many of the parameters in subsection 2.7.1 the second parameter indicates whether to enable or disable the feature. Thus, not checking the second parameter means that the vulnerable feature may be enabled as opposed to disabled. TransformerFactoryDetector and XMLStreamReaderDetector checks all the required parameters at most two. XMLStreamReaderDetector also includes extra logic to get the Boolean value of a Boolean object. Except for TransformerFactoryDetector, all the detectors report the vulnerability where the parser is used. TransformerFactoryDetector reports the vulnerability where the parser is initialized. TransformerFactoryDetector also reports two separate bug patterns for DTD vulnerabilities and XSLT vulnerabilities.

```
for each opcode in method do
    if name of opcode equals name of parse method
        for each call in method_callgraph do
            if name equals name of secure function
                check parameters of method
            end if
        end for
    end if
end for
if not all secure calls found
    report vulnerability
end if
```

Listing 14: Pseudocode for the existing detectors in FindSecBugs

6.2.2 Evaluation of the Existing XML Detectors in FindSecBugs

To test the performance of the existing detectors the test bed detailed in section 6.1 was used. The test bed consists of test cases that are based on control flow and data flow variants from Juliet Test Suite, and test cases based on different ways of initializing an instance and calling secure or insecure methods on it. Each of the test cases has been created for the parsers supported by FindSecBugs. When performing the evaluation it became clear that the existing detectors were able to handle the test cases based on the flow variants from Juliet Test Suite well, whereas they fell through on the test cases that tested different ways of initializing instances. Therefore, the evaluation results on these test cases have been shown in more detail to show exactly where the existing detectors fell through.

Detection Results for DocumentBuilder

The result of evaluating on the instance based test cases for the DocumentBuilder parser is shown in Table 6.2. The false negatives on test case seven through 10 are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are due to the detector not differentiating between the parser and a separate object with the same secure method.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	0	0	1

Case	TP	FP	FN
Case 8	0	0	2
Case 9	0	0	2
Case 10	0	0	4
Case 11	0	0	1

Table 6.2: Result of evaluating the existing detector for the *DocumentBuilder* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for SAXParser

The results of evaluating on the instance based test cases for the SAXParser parser is shown in Table 6.3. The false negatives on test case seven through 10 are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are due to the detector not differentiating between the parser and a separate object with the same secure method.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	0	0	1
Case 8	0	0	2
Case 9	0	0	2
Case 10	0	0	4
Case 11	0	0	1

Table 6.3: Result of evaluating the existing detector for the *SAXParser* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for XMLStreamReader

The results of evaluating on the instance based test cases for the XMLStreamReader parser are shown in Table 6.4. Test case nine and test case 10, which tests 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 this parser uses an iterator to parse the XML [37]. 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.

The false negatives on test case seven and eight are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are

due to the detector not differentiating between the parser and a separate object with the same secure method.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	0	0	1
Case 8	0	0	2
Case 11	0	0	1

Table 6.4: Result of evaluating the existing detector for the *XMLStreamReader* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for XMLEventReader

The results of evaluating on the instance based test cases for the XMLEventReader parser is shown in Table 6.5. Test case nine and test case 10 are not applicable to XMLEventReader for the same reasons as XMLStreamReader described in section 6.2.2. The false negatives on test case seven and eight are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are due to the detector not differentiating between the parser and a separate object with the same secure method.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	0	0	1
Case 8	0	0	2
Case 11	0	0	1

Table 6.5: Result of evaluating the existing detector for the *XMLEventReader* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for FilteredReader

The results of evaluating on the instance based test cases for the FilteredReader parser is shown in Table 6.6. Test case nine and test case 10 are not applicable to FilteredReader for

the same reasons as XMLStreamReader described in section 6.2.2. The false negatives on test case seven and eight are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are due to the detector not differentiating between the parser and a separate object with the same secure method.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	0	0	1
Case 8	0	0	2
Case 11	0	0	1

Table 6.6: Result of evaluating the existing detector for the *FilteredReader* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for Transformer

The results of evaluating on the instance based test cases for the Transformer parser are shown in Table 6.7. Note that for each vulnerability in these test cases, the detector reports two vulnerabilities. As mentioned in subsection 6.2.1, this is because the detector differentiates between whether the parser is vulnerable to DTD processing or Extensible Stylesheet Language Transformation (XSLT) attacks and reports a separate bug pattern for each vulnerability. The test cases used to evaluate the detector are vulnerable to both bug patterns.

Case	TP	FP	FN
Case 1	2	0	0
Case 2	2	0	0
Case 3	2	0	0
Case 4	2	0	0
Case 5	2	0	0
Case 6	2	0	0
Case 7	0	0	2
Case 8	2	0	2
Case 9	0	0	4
Case 10	6	0	2
Case 11	0	0	2

Table 6.7: Result of evaluating the existing detector for the Transformer parser in FindSecBugs on the instance based test cases described in Table D.1.

The false negatives on test case seven through 10 are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are due to the detector not differentiating between the parser and a separate object with the same secure method.

Detection Results for XMLReader

The results of evaluating on the instance based test cases for the XMLReader parser are shown in Table 6.8. Test cases one through three and test case five tests if the detector can correctly identify secure instances from vulnerable instances created using the same factory. An XMLReader parser is initialized directly without first initializing a factory [36]. Therefore, these test cases are not applicable to test the detection performance on the XMLReader parser.

The false negatives on test case seven through 10 are due to the detector not handling the use of multiple parsers and the detector not handling parsers that have been configured to be explicitly vulnerable within the same method. The false negatives on test case 11 are due to the detector not differentiating between the parser and a separate object with the same secure method.

Case	TP	FP	FN
Case 4	1	0	0
Case 6	1	0	0
Case 7	0	0	1
Case 8	0	0	2
Case 9	0	0	2
Case 10	0	0	4
Case 11	0	0	1

Table 6.8: Result of evaluating the existing detector for the *XMLReader* parser in FindSecBugs on the instance based test cases described in Table D.1. Only test case four, six, and seven through 11 are applicable to this parser

Summary of Detection Results

The detection performance on the instance based test cases is identical for all the parsers handled by the XxeDetector. These are DocumentBuilder, SAXParser, and XMLReader. The detection performance for XMLStreamReaderDetector is identical to that for XxeDetector. XMLStreamReaderDetector handles the parsers XMLStreamReader, XMLEventReader, and FilteredReader. These detectors correctly identify the vulnerabilities in test cases one through six which tests initializing a parser into a class field or a variable and calling the secure methods. This is the simplest form of initializing a parser. However, the detectors were not able to handle test cases seven through 11. These tests initializing multiple parsers, calling both secure and insecure methods on the parser, and calling the secure method on a separate object.

TransformerFactoryDetector performs identically to XxeDetector and XMLStream-

ReaderDetector for all of the instance based test cases, except for test case eight and test case 10. The detector has a higher number of true positives for these. Test case eight tests if the detector can detect vulnerabilities for parsers created by a factory that is first vulnerable, then secure, and then vulnerable again. Test case 10 is a variant of test case eight where the vulnerable parsers are used more than once. The reason for the higher number of true positives is due to the detector toggling a boolean value depending on the second boolean attribute set when setting the *XMLConstants.FEATURE_SECURE_PROCESSING* to determine if this attribute has been set. Therefore, since this feature is first enabled, and then disabled, the detector toggles the boolean and coincidentally ends up in a state where it successfully reports the vulnerabilities. The number of true positives for test case eight and 10 can, therefore, be regarded as a coincidence.

FindSecBugs had high precision but varying recall for the instance based test cases. The number of true positives, false positives, false negatives, precision, and recall for each of the parsers handled by the existing detectors for the instance based test cases have been summarized in Table 6.9. While there is a high number of true positives, and a low number of false positives showing that the detectors are able to successfully identify vulnerabilities in many cases, there is also a high number of false negatives. This means that of the vulnerabilities the detector identifies, all of them are actual vulnerabilities as shown by the precision, however, the detectors also miss a lot of vulnerabilities as shown by the recall. This shows that the existing detectors are able to handle different ways of initializing the different parsers, but that they are not able to handle more complex data and control flows.

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%

Table 6.9: Summary of the true positives, false positives, and false negatives after evaluating the existing detectors on the instance based test cases in Table D.1

FindSecBugs had 100% precision and recall for all XML parsers for the Juliet style tests. The result of evaluating the existing detectors on the test cases based on flow variants from Juliet Test Suite has been summarized in Table 6.10. These flow variants test if the detector is able to handle wrapping the initialization of the parsers in different control flow constructs such as ifs, loops, and switch statements. For more detail see section D.2. These test cases test a single instance of a parser and a single call to the secure method. This shows that the existing detectors are able to handle different flow constructs where only one parser instance has been used. Additionally, the detectors were evaluated on the entirety of Juliet Test Suite which does not include support for XXE. No false positives or false negatives were found.

Parser	TP	FP	FN	Precision	Recall
DocumentBuilder	17	0	0	100%	100%
XMLStreamReader	17	0	0	100%	100%
XMLEventReader	17	0	0	100%	100%
FilteredReader	17	0	0	100%	100%
SAXParser	17	0	0	100%	100%
XMLReader	17	0	0	100%	100%
Transformer	34	0	0	100%	100%

Table 6.10: Summary of the true positives, false positives, and false negatives after evaluating the existing detectors on the Juliet style test cases

The results of evaluating the performance of the existing detectors for XXE have been summarized in Table 6.11. The execution time was measured on a PC with 16G of memory and a 3.9GHz CPU using Windows 10 pro. The test bed detailed in section 6.1 and the existing Juliet Test Suite was evaluated separately to show the execution time for test cases without XXE vulnerabilities and test cases with. FindSecBugs allows enabling and disabling specific detectors. Therefore, only the XXE vulnerability detectors were enabled. This is useful for comparing the detection performance of the new detectors presented in RQ2.2. The difference in execution time for a cold run and a hot run is due to caching performed by FindSecBugs.

Test suite	LOC	Execution time cold run	Execution time hot run
Test cases with XXE vulnerabilities	24,087	4.6s	1.5s
Juliet Test Suite	5,143,930	84.1s	79.1s

Table 6.11: Execution time for the existing detectors

6.3 RQ2.2 How can the Detection of XML External Entities be Improved using Different Techniques?

Different analysis techniques, such as pattern matching, data flow analysis, and control flow analysis, were identified in the prestudy in Appendix B. The existing detectors for identifying XXE vulnerabilities in FindSecBugs employ limited control flow analysis using pattern matching. To improve the detection of XXE a tool needs to **check if an instance of a parser has been made secure before it is used by identifying when a secure or a vulnerable method has been called on the instance**. The tool has to analyze the calls performed on the parser through the execution of the method. This problem requires an analysis of the flow of data.

Two data flow analysis based approaches were evaluated to find out which would work

best for detecting the XXE vulnerabilities FindSecBugs missed. First, instruction based data flow analysis will be detailed. This should perform better in theory due to modeling the effect of each instruction within each basic block thus being able to take into account the sequence of calls and the values of these calls within the method. However, this analysis was not capable of knowing which parser instance the calls have been called on. Therefore, a novel instance tracking approach was implemented. This approach tracks the instances and the calls alongside their parameters performed on the instances through the execution of the method. This approach improves the state of the art in detecting XXE vulnerabilities by being able to identify vulnerabilities from multiple parsers within the same method and by being able to identify vulnerabilities correctly even if a parser and a different instance with the secure methods are called within the same method. As will be shown, the approach can easily be generalized to other vulnerabilities.

6.3.1 Instruction Based Data Flow Analysis Approach

The instruction-based data flow analysis approach evaluated is based on the observation that the calls and the parameters these calls have been called with can be modeled as data flow values. Thus, both the order of the sequence of the calls within a method and the parameters the calls are called with can be stored in these data flow values for each basic block of the control flow graph of the method. At the end of the method, in the last basic block, the sequence of all the calls made within the method, and the parameters of these calls can be found. For the implementation-specific details see Appendix F. For XXE these calls can be identified as secure or insecure calls which in turn can be used to determine if a parser is vulnerable. The approach has been summarized in the following points:

1. Create a map from instruction to Calls to record method name, class name, and signature
2. For each block in the control flow graph, initialize data flow values as an empty list of calls
3. Execute data flow algorithm: For each instruction in the basic block, check if it is a method call, and add the call and the parameters of the call to the data flow value
4. Check the sequence of the calls to identify if the calls have been called with correct parameters in the correct sequence to make the parser secure

6.3.2 Evaluation Results of Instruction Based Data Flow Analysis

Evaluating the instruction based data flow analysis on the test cases from section 6.1 revealed that this type of analysis was not capable of detecting many of the vulnerabilities present in the test bed. For the test cases including the use of multiple parsers (test cases seven through 10) this analysis method was not able to keep track of which was secure and which was insecure. The results presented show why this is the case and why instruction based data flow analysis is not capable of significantly improving the detection of XXE vulnerabilities. The results presented here are for the detection performance on SAXParser. Similar results were obtained for the other parsers.

When analyzing the instruction based data flow for the vulnerable method, the *bad* method, in test case one, the analyzer can correctly detect that the secure function has not been called. In Figure 6.3, the result of evaluating the instruction based data flow analysis on the vulnerable version of test case one of the instance based test cases is shown. The sequence of calls recorded at the end of the method is shown in the highlighted area. As can be seen, the approach is successfully able to identify the sequence of the calls performed in the method and that the secure function for the parser has not been called. For SAXParser, this means that the method setFeature has not been called before a new instance of SAXParser has been obtained and used to parse an XML document using the method newSAXParser.

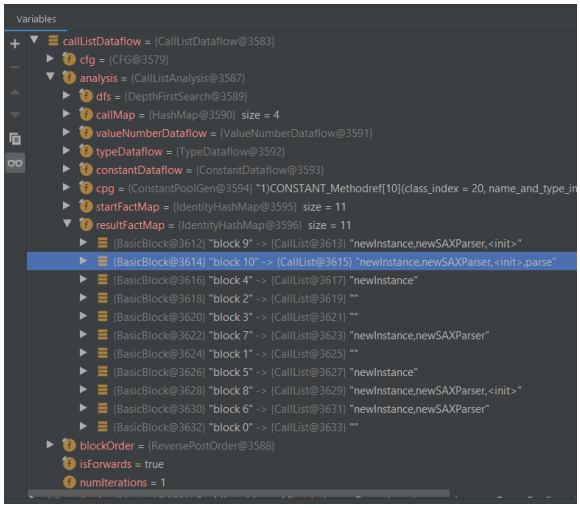


Figure 6.3: The result of evaluating the instruction based data flow analysis on the vulnerable version of test case one, the *bad* method in Table D.1. The sequence of recorded calls at the end of the method is shown in the highlighted area. It is successfully able to determine that no call to the secure method setFeature has been performed which means the parser use is vulnerable

The result of evaluating the instruction based data flow on the secure version of test case one of the instance based test cases (the *good* method) is shown in Figure 6.4. The sequence of calls recorded at the end of the method is shown in the highlighted area. Compared to the vulnerable version of test case one shown in Figure 6.3, the method setFeature has been called before a new instance of SAXParser has been obtained using the newSAXParser method. This shows that the instruction based data flow approach records the sequence of calls needed to identify if the parser secure. In Figure 6.5 the setFeature call, which is call number two in Figure 6.4 has been expanded to show the values recorded for this call. The parameters of interest are highlighted. As can be seen, the approach is also able to identify the parameters the secure method (setFeature) has been called with which can be used to successfully determine if the method has been called with the correct parameters.

Similar positive results were obtained for the other parsers for test case one through six of the instance based test cases and for all of the Juliet style test cases, where the

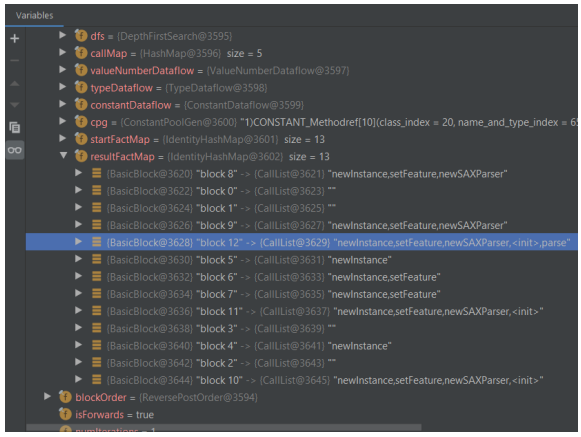


Figure 6.4: Result of evaluating instruction based data flow analysis on the secure version of test case one, the *good* method in Table D.1. The sequence recorded calls at the end of the method is shown in the highlighted area. It is successfully able to determine that a call to the secure method setFeature has been performed before the parser use, which means that the parser use is secure

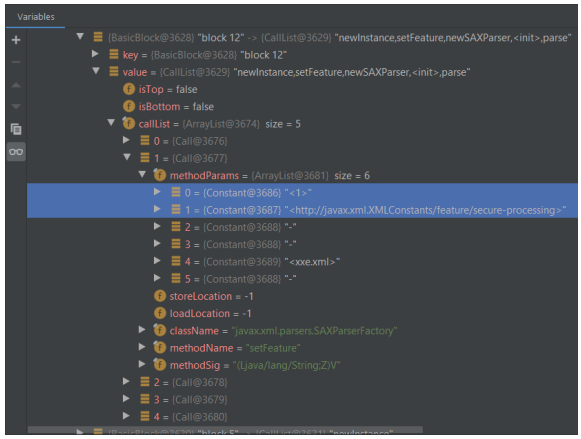


Figure 6.5: The parameters the setFeature call, which is call number two shown in Figure 6.4, has been called with is highlighted in blue. This shows that the data flow value successfully records the parameters the method has been called with

sequence of calls and their parameters recorded was enough to determine if the parsers initialized were secure or vulnerable. In these test cases, only one parser is initialized and used. As mentioned previously, the data flow values capture the sequence of when the factory is initialized, then if the secure method has been called, and finally when the parser has been initialized using the factory. If the secure method has been called, the corresponding data flow value for this call captures the parameters of the secure method e.g. xmlconstants.FEATURE_SECURE_PROCESSING and 1 as shown by the highlighted values in Figure 6.5.

However, when evaluating on test cases seven through 11 of the instance based test cases it became evident that the approach was not significantly better than state of the art. In these test cases, multiple parsers are initialized and used. As an example, the result of evaluating on test case nine for SAXParser is shown in Figure 6.6. Test case nine first initializes an insecure factory, then an insecure parser is initialized using the factory, then this parser is used twice, then the factory is made secure, then a secure parser is initialized, and finally this secure parser is used twice. The sequence of calls recorded at the end of the method is shown in the highlighted area. The instruction based data flow approach is able to identify that the first parse call is from the first vulnerable parser. However, the approach is unable to determine if the first or the second parser is used for the two latter parse calls. Thus, the instruction based data flow analysis can determine that the first two parser uses are vulnerable, but not whether the two last parser values are used since it does not know which instance these are invoked on.

Therefore, this approach was determined to not perform significantly better than state of the art. It is able to identify the sequence of calls and can identify which parameters these calls have been invoked with, but it is not able to identify which parser use corresponds to which parser when multiple parsers are used within the same method. Therefore, it was decided that a different approach is necessary.

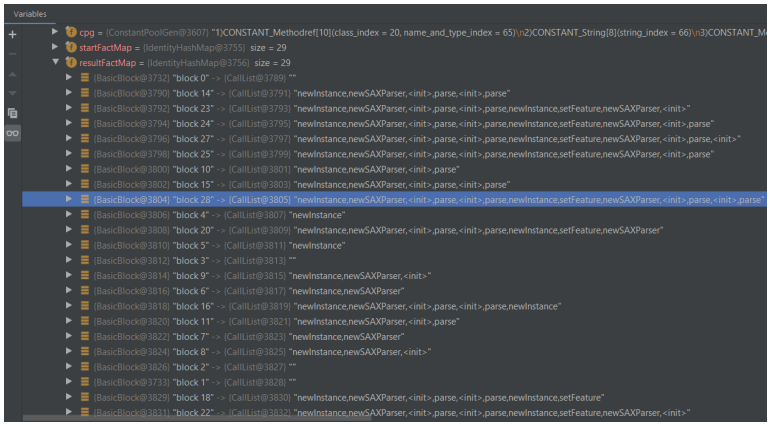


Figure 6.6: The result of evaluating instruction based data flow analysis on the vulnerable version, the *bad* method, of test case 9 in Table D.1. The sequence of recorded calls at the end of the method is shown in the highlighted area. The data flow value does not contain enough information to determine if the last two parse calls are invoked on the first vulnerable parser or on the second secure parser

6.3.3 Instance Tracking Analysis Approach

In this section, a novel instance tracking approach is presented. Evaluating the existing detectors in FindSecBugs and the instruction based data flow analysis showed that the main weakness of both was that they were not capable of knowing which instance the secure or vulnerable methods have been called on. Therefore, a detector that can track the

instance and the methods called on this instance is needed.

The main instance tracking approach has been summarized in the flow chart shown in Figure 6.7. The approach can be broken down into four steps:

1. Find the instances to track
2. Combine instances that should be treated as the same instance
3. Identify the secure and vulnerable calls called on the tracked instances
4. Report the vulnerabilities found for each instance

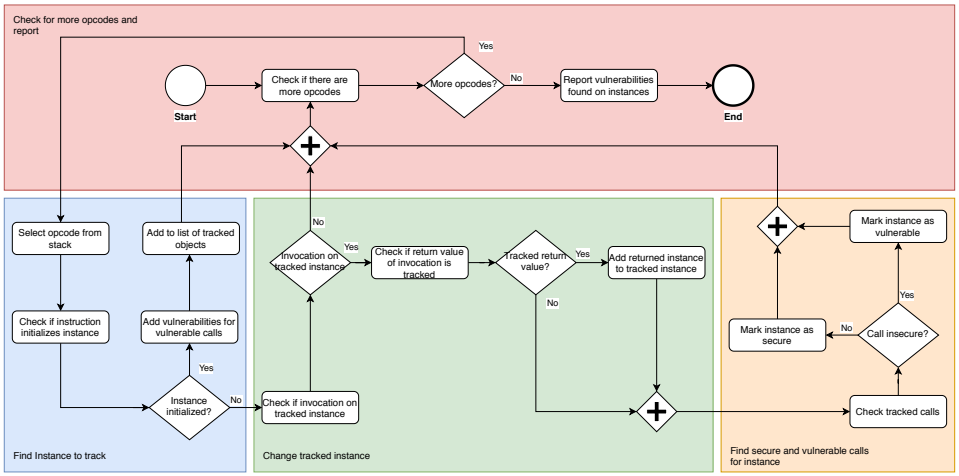


Figure 6.7: Flow chart for the instance tracking approach

The instance tracking approach begins with a stack of opcodes. First, the initialization opcodes are used to find which instances to track the calls of. These are identified by their initialization instruction. Then the method call opcodes which return a new instance are analyzed to find which instances should be treated as the same instance. For the example shown in Listing 15, this means that documentBuilder1 and documentBuilder2 are both treated as part of the dbFactory instance. Afterward, the calls performed on the tracked instances are analyzed to identify the vulnerable and secure calls. This is then used to report the vulnerabilities found for each instance.

```
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder1 = dbFactory.newDocumentBuilder();
documentBuilder1.parse(<inputFile>);

DocumentBuilder documentBuilder2  = dbFactory.newDocumentBuilder();
Document doc = documentBuilder2.parse(<inputFile>);
```

Listing 15: Example of DocumentBuilder instances that are treated as part of the DocumentBuilderFactory instance

The main instance tracking approach has been summarized in more detail in the pseudocode shown in Listing 16. The instance tracker requires a list of initialization instructions for the instances to track, a list of return values for each instance that should be combined and considered the same, and a list of calls to track for each instance. The instance tracker analyzes the stack of opcodes to find opcodes corresponding to the initialization of an instance using the list of initialization instructions. If an instance is found, then the source line location of where it was initialized is stored. The initial vulnerabilities for the instance are determined by the calls that vulnerable if not found. These are added to the instance found.

```
for each opcode in method
  if opcode is in list of initialization instructions
    add instance to list of tracked instances
    add vulnerabilities to the instance
  end if
  if opcode is invocation instruction
    if invocation invoked on tracked instance
      add the return value to the tracked instance
    end if
    check if opcode corresponds to tracked call
  end if
end for

for each tracked instance
    for each vulnerability
        report vulnerability
    end for
end for
```

Listing 16: Pseudocode for instance tracking

When an opcode corresponding to an invocation is found, the instance it has been invoked on is obtained from the operand stack. If this invocation results in a new instance, the list of return values is used to determine if this instance should be treated as part of the tracked instance. The vulnerabilities on the instance retrieved from the operand stack is also stored on this returned instance to keep track of which vulnerabilities are present on each tracked instance and the instances that are part of these at any given time.

The call checking subroutine shown in Listing 17 is then used to check if any of the calls invoked on the instances have been found. Singular calls and multiple calls are differentiated between. A singular call is a call which can be either vulnerable if not found or secure if found. The stack parameters are used to determine if such a call has been found. A multiple call consists of multiple singular calls. All of these calls have to be found for the vulnerability reported by a multiple call to be removed. Each of the singular calls that makes up a multiple call has an opposite that allows checking if one call undoes the effect of a previously found call. Finally, the vulnerabilities found are reported. This is done after all the opcodes within a method has been checked.

```
if invocation is singular tracked call
  if parameters of invocation corresponds to tracked call
    mark call as found
    if call is secure:
        remove the vulnerability reported by it
    end if
    if call is vulnerable:
        add the vulnerability reported by it
    end if
  end if
end if
if invocation is multiple tracked call
  for each singular tracked call
    if parameters of invocation corresponds to tracked call
      if singular call is secure
        mark call as found
      end if
      if singular call is vulnerable
        mark call as not found
        add the vulnerability reported by the multiple tracked call
      end if
    end if
  end for
  if all singular calls for multiple tracked call found
    remove the vulnerability reported by it
  end if
end if
```

Listing 17: Pseudocode for call checking subroutine used by the instance tracker

6.3.4 Evaluation of Instance Tracking Based XXE Detectors

To test the performance of the new detectors based on the instance tracking approach the test bed detailed in section 6.1 was used. To better compare with the existing detectors, new detectors based on the instance tracking approach for all the parsers supported by the existing ones were evaluated. The implementation details are shown in Appendix G. Similar to the evaluation performed for the existing detectors, the new detectors were found to handle all the test cases based on the flow variants from the Juliet Test Suite. Therefore, the instance based test cases will be presented in more detail to better compare to the detection performance of the existing state of the art detectors for XXE.

Detection Results for DocumentBuilder

The result of evaluating on the instance based test cases for the DocumentBuilder parser is shown in Table 6.12.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0

Case	TP	FP	FN
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	2	0	0
Case 9	2	0	0
Case 10	4	0	0
Case 11	1	0	0

Table 6.12: Result of evaluating the new detector for the DocumentBuilder parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for SAXParser

The result of evaluating on the instance based test cases for the SAXParser parser is shown in Table 6.13.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	2	0	0
Case 9	2	0	0
Case 10	4	0	0
Case 11	1	0	0

Table 6.13: Result of evaluating the new detector for the *SAXParser* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for XMLStreamReader

The result of evaluating on the instance based test cases for the XMLStreamReader parser is shown in Table 6.14. As mentioned in section 6.2.2, test case nine and 10 are not applicable to XMLStreamReader.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0

Case	TP	FP	FN
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	2	0	0
Case 11	1	0	0

Table 6.14: Result of evaluating the new detector for the *XMLStreamReader* parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for XMLEventReader

The result of evaluating on the instance based test cases for the XMLEventReader parser is shown in Table 6.15. As mentioned in section 6.2.2, test case nine and 10 are not applicable to XMLEventReader.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	2	0	0
Case 11	1	0	0

Table 6.15: Result of evaluating the new detector for the XMLEventReader parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for FilteredReader

The result of evaluating on the instance based test cases for the FilteredReader parser is shown in Table 6.16. As mentioned in section 6.2.2, test case nine and 10 are not applicable to FilteredReader.

Case	TP	FP	FN
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0

Case	TP	FP	FN
Case 7	1	0	0
Case 8	2	0	0
Case 11	1	0	0

Table 6.16: Result of evaluating the new detector for the FilteredReader parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for Transformer

The result of evaluating on the instance based test cases for the Transformer parser is shown in Table 6.17. Note that for each vulnerability in these test cases, the detector reports two vulnerabilities as mentioned in section 6.2.2.

Case	TP	FP	FN
Case 1	2	0	0
Case 2	2	0	0
Case 3	2	0	0
Case 4	2	0	0
Case 5	2	0	0
Case 6	2	0	0
Case 7	2	0	0
Case 8	4	0	0
Case 9	4	0	0
Case 10	8	0	0
Case 11	2	0	0

Table 6.17: Result of evaluating the new detector for the Transformer parser in FindSecBugs on the instance based test cases described in Table D.1

Detection Results for XMLReader

The result of evaluating on the instance based test cases for the XMLReader parser is shown in Table 6.18. As mentioned in section 6.2.2, test case one through three and test case five are not applicable to XMLReader.

Case	TP	FP	FN
Case 4	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	2	0	0
Case 9	2	0	0
Case 10	4	0	0
Case 11	1	0	0

Table 6.18: Result of evaluating the new detector for the XMLReader parser in FindSecBugs on the instance based test cases described in Table D.1

Summary of Detection Results

The new detectors based on the instance tracking approach are able to detect all the vulnerabilities in the instance based test cases. The detectors detect the vulnerabilities in test case one through six which tests initializing an instance into a class field or a variable and calling the secure methods. The detectors are also capable of detecting vulnerabilities in test case seven through 11 which tests multiple parsers and calling secure and vulnerable calls on these.

Both the precision and recall of the instance tracking based detector are 100% for all test cases in the test bed. It should be noted that the scores are high because the test bed only includes test cases with intraprocedural flows. This limitation of the test bed is because the focus of the auto fixing tool, as described in chapter 4, is intraprocedural and is dependent on proper detection. The number of true positives, false positives, and false negatives for each of the parsers on the instance based test cases is summarized in Table 6.19. As can be seen, there is a high number of true positives, no false positives, and no false negatives. This shows that the new detectors are able to handle more complex data and control flows. Similar results were obtained when evaluating the new detectors on the test cases based on flow variants from Juliet Test Suite which is summarized in Table 6.20. As can be seen, there is a high number of true positives, no false positives, and no false negatives. This shows that the instance tracking based approach is able to handle different control flow constructs. The detectors were also evaluated on the entirety of Juliet Test Suite, which does not include support for XXE. No false positives or false negatives were found.

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%

Table 6.19: Summary of the true positives, false positives, and false negatives after evaluation of the instance tracking based detectors on the instance based test cases in Table D.1

Parser	TP	FP	FN	Precision	Recall
DocumentBuilder	17	0	0	100%	100%
XMLStreamReader	17	0	0	100%	100%
SAXParser	17	0	0	100%	100%
XMLReader	17	0	0	100%	100%
Transformer	34	0	0	100%	100%

Table 6.20: Summary of the true positives, false positives, and false negatives after evaluation of the instance tracking based detectors on the Juliet style test cases

Compared to the execution time of the existing detectors in FindSecBugs shown in Table 6.11, the execution time of the new detectors is 31% slower when examining a cold run on the whole Juliet Test Suite. The results of evaluating the performance of the new detectors for XXE have been summarized in Table 6.21. The execution time was measured on a PC with 16G of memory, 3.9GHz CPU using Windows 10 pro. The test bed detailed in section 6.1 and the existing Juliet Test Suite was evaluated separately to show the execution time for test case without XXE and test cases with. FindSecBugs allows enabling and disabling specific detectors. Therefore, only the XXE detectors were enabled. The difference in execution time for a cold run and a hot run is due to caching performed by FindSecBugs.

Test suite	**LOC**	**Execution time cold run**	**Execution time hot run**
Test cases with XXE vulnerabilities	24,087	4.21s	1.75s
Juliet Test Suite	5,143,930	110s	98.9s

Table 6.21: Execution time for instance tracking detectors

6.3.5 Evaluation on Built in Test Cases in FindSecBugs

As mentioned in section 2.10, FindSecBugs employs a test driven development strategy. Test cases for testing the detection of vulnerabilities for the parsers DocumentBuilder, SAXParser, Transformer, XMLStreamReader, XMLEventReader, FilteredReader, and XMLReader were already present. These tests include the bare minimum for initializing one of the XML parsers, and test the different safe function the existing detectors look for. Both examples of vulnerable code and non-vulnerable code are tested. The test framework verifies that the vulnerabilities are reported on the line that is expected.

During the implementation of the new detectors using the recommendations from OWASP and Oracle as described in subsection 2.7.1 it was discovered that some of the test cases in FindSecBugs were wrongly implemented using the wrong parameters. An overview of the affected methods is shown below. These test cases were changed to correspond to the recommendations by OWASP and have been published to GitHub alongside the new detectors [18].

- DocumentBuilderSafeProperty.safeManualConfiguration
- DocumentBuilderSafeProperty.unsafeManualConfig1
- DocumentBuilderSafeProperty.unsafeManualConfig2
- DocumentBuilderSafeProperty.unsafeManualConfig3
- DocumentBuilderSafeProperty.unsafeManualConfig4

- SaxParserSafeProperty.safeManualConfiguration

With the new detectors handling more cases than the old ones, more test cases were added to handle these. These are shown below. For DocumentBuilder and XMLReader test cases were added to test if the detector could identify the use of a custom entity resolver which is considered safe. For SAXParser similar test cases to *DocumentBuilderSafeProperty.unsafeManualConfig1* through *DocumentBuilderSafeProperty.unsafeManualConfig5* were added to test if the detector successfully reported the vulnerability if one of the multiple attributes which together makes the parser secure was missing.

- DocumentBuilderSafeProperty.unsafeManualConfig5
- DocumentBuilderSafeEntityResolver.receiveXMLStream
- SaxParserSafeProperty.unsafeManualConfig1
- SaxParserSafeProperty.unsafeManualConfig2
- SaxParserSafeProperty.unsafeManualConfig3
- SaxParserSafeProperty.unsafeManualConfig4
- SaxParserSafeProperty.unsafeManualConfig5
- XmlReaderSafeEntityResolver.receiveXMLStream

6.4 RQ3: How can Auto Fixing of XML External Entities be Implemented using an IDE Plugin

In this section, a novel auto fix approach for instance based vulnerabilities based on traversing and modifying the AST is presented. In the prestudy shown in Appendix B, no auto fix mechanism for XXE was found. Therefore, this approach was used to create auto fixes for these vulnerabilities. This is useful for developers who are not domain experts to help them mitigate XXE by inserting the fixes at the correct location in the code. Given a detection mechanism, it is desirable with an auto fix mechanism to make mitigating the vulnerabilities easier. As shown in subsection 2.7.1, there are many different APIs and features that need to be set for the different parsers to make them secure. Having an auto fix mechanism will help reduce the complexity, time, and effort spent identifying the correct fixes for the different parsers. The auto fixes have been evaluated on the test bed presented in section 6.1 and was found to perform well.

6.4.1 AST Based Auto Fix Approach for XML External Entities

The AST based auto fix approach has been summarized in Figure 6.8. For implementation specific details see Appendix I. The approach can be broken down into three steps:

1. Find the node to insert the auto fix on

2. Prepare the node for the auto fix insertion

3. Apply the auto fix

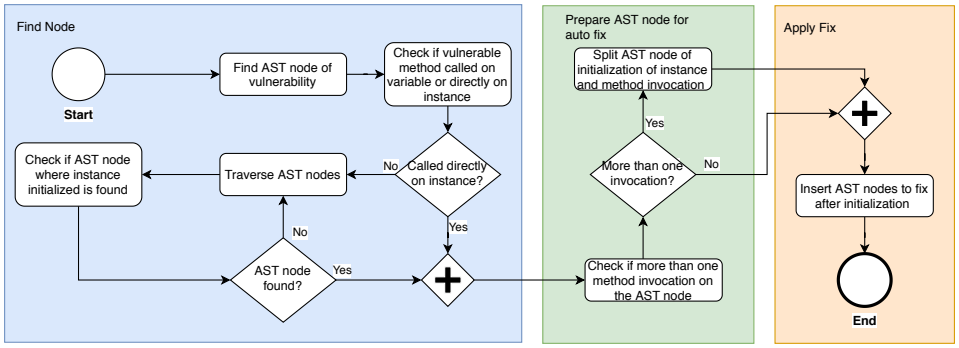


Figure 6.8: Flow chart for the auto fixing approach

The auto fix approach begins with the location of the vulnerability to be auto fixed which is reported by a vulnerability detector. The corresponding AST node for the source code line is found by traversing the AST of the method. Then, a check is made to identify if the vulnerable method is called on a variable or directly on an instance. If the vulnerable method is called on a variable, then the auto fix approach attempts to traverse the predecessors of the AST nodes of this variable until it finds the AST node of the variable where the instance to be auto fixed was initialized to. It does this by matching the names of the variable the vulnerability is reported on with the previous method calls that led to this variable being created. The type of the node is used to determine when an instance where the auto fix should be inserted has been found.

When this node is found, or if the vulnerable method is called directly on an instance, then a check is made to identify if there are multiple methods invoked on this 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.

Finally, the AST nodes corresponding to the missing method calls that make up the auto fix are inserted after the initialization. The necessary imports are also added as part of this step.

6.4.2 Evaluation of AST based Auto Fixes

To test the performance of the AST based auto fixes for XXE the test bed detailed in section 6.1 was used. The implementation details of the auto fixes are shown in Appendix I. The output of the detectors for XXE based on the instance tracking approach shown in subsection 6.3.3 was used. The implementation details of these detectors are shown in Appendix G. The auto fixes were found to handle the flow variants from Juliet Test Suite well. Therefore, the instance based test cases will be presented in more detail to better be able to show the strengths and weaknesses of the AST based auto fix approach.

Auto Fix Results for DocumentBuilder

The result of evaluating on the instance based test cases for the DocumentBuilder parser is shown in Table 6.22. The incorrect fixes are due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable.

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	1	0	1
Case 9	2	0	0
Case 10	3	0	1
Case 11	1	0	0

Table 6.22: Result of evaluating the auto fixes for the DocumentBuilder parser in FindSecBugs on the instance based test cases shown in Table D.1

Auto Fix Results for SAXParser

The result of evaluating on the instance based test cases for the SAXParser parser is shown in Table 6.23. The incorrect fixes are due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable.

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	1	0	1
Case 9	2	0	0
Case 10	3	0	1
Case 11	1	0	0

Table 6.23: Result of evaluating the auto fixes for the SAXParser parser in FindSecBugs on the instance based test cases shown in Table D.1

Auto Fix Results for XMLStreamReader

The result of evaluating on the instance based test cases for the XMLStreamReader parser is shown in Table 6.24. As mentioned in section 6.2.2 test case nine and test case 10 are not applicable. The incorrect fixes are due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable.

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	1	0	1
Case 11	1	0	0

Table 6.24: Result of evaluating the auto fixes for the XMLStreamReader parser in FindSecBugs on the instance based test cases shown in Table D.1

Auto Fix Results for XMLEventReader

The result of evaluating on the instance based test cases for the XMLEventReader parser is shown in Table 6.25. Due to the same reasons mentioned in section 6.2.2, test case nine and test case 10 are not applicable. The incorrect fixes are due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable.

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	1	0	1
Case 11	1	0	0

Table 6.25: Result of evaluating the auto fixes for the XMLEventReader parser in FindSecBugs on the instance based test cases shown in Table D.1

Auto Fix Results for FilteredReader

The result of evaluating on the instance based test cases for the FilteredReader parser is shown in Table 6.26. Due to the same reasons mentioned in section 6.2.2, test case nine and test case 10 are not applicable.

The auto fixes for FilteredReader made three incorrect fixes, one in test case four, one in test case six, and one in test case eight. The incorrect fix in test case eight was due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable. For test case four and test case six the incorrect fixes were due to the auto fix fixing one of the insecure factories used to initialize the FilteredReader, but not both. As an example, an excerpt of test case four and test case six is shown in Listing 18. As can be seen, two calls to XMLInputFactory.newInstance() are made to create one FilteredReader. The incorrect fixes in these two test cases were due to the auto fix fixing one of the insecure factories, but not both. For test case four, the first call to XMLInputFactory is made secure by the auto fix but not the second call. For test case six the second call to XMLInputFactory, referenced by the factory variable is made secure, but not the first call.

```
// Test case 4
reader = XMLInputFactory.newInstance().createFilteredReader(
    XMLInputFactory.newInstance()
        .createXMLStreamReader()
    );

// Test case 6
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader reader = XMLInputFactory.newInstance()
    .createFilteredReader(
        factory.createXMLStreamReader()
    );
```

Listing 18: Initialization of the *FilteredReader* in test case four and test case 6 from Table D.1

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	0	0	1
Case 5	1	0	0
Case 6	0	0	1
Case 7	1	0	0
Case 8	1	0	1
Case 11	1	0	0

Table 6.26: Result of evaluating the auto fixes for the FilteredReader parser in FindSecBugs on the instance based test cases shown in Table D.1

Auto Fix Results for Transformer

The result of evaluating on the instance based test cases for the Transformer parser is shown in Table 6.27. The incorrect fixes are due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable.

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 1	1	0	0
Case 2	1	0	0
Case 3	1	0	0
Case 4	1	0	0
Case 5	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	1	0	1
Case 9	2	0	0
Case 10	3	0	1
Case 11	1	0	0

Table 6.27: Result of evaluating the auto fixes for the Transformer parser in FindSecBugs on the instance based test cases shown in Table D.1

Auto Fix Results for XMLReader

The result of evaluating on the instance based test cases for the XMLReader parser is shown in Table 6.28. As mentioned in section 6.2.2, test case one through three and test case five are not applicable to XMLReader. The incorrect fixes are due to the auto fix not removing or modifying code that makes a factory explicitly vulnerable.

Case	Successful fixes	Missed fixes	Incorrect fixes
Case 4	1	0	0
Case 6	1	0	0
Case 7	1	0	0
Case 8	1	0	1
Case 9	2	0	0
Case 10	3	0	1
Case 11	1	0	0

Table 6.28: Result of evaluating the auto fixes for the XMLReader parser in FindSecBugs on the instance based test cases shown in Table D.1

6.4.3 Summary of Auto Fix Evaluation Results

The performance of the auto fixes on the instance based test case is identical for test case one through seven and eleven for all of the parsers except for FilteredReader. FilteredReader had incorrect fixes in test cases four, six, and eight, due to this parser being slightly different in how it is used as mentioned in section 6.4.2. All of the auto fixes had lower performance on test case eight through ten compared to test case one through seven and eleven. This is due to these test cases including parsers that are made explicitly secure and then vulnerable. The incorrect fixes were made when attempting to fix a parser that had been explicitly made insecure. The auto fix was inserted directly after the factory was initialized making the factory secure. However, the code making the factory explic-

itly insecure was not removed or modified. Therefore, the developer needs to manually remove the code making the factory explicitly insecure for the auto fix to be successfully applied. The new detectors described in subsection 6.3.3 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. These fixes were still regarded as incorrect fixes since a future improvement would be to give the developer the option to automatically remove the vulnerable code as part of the auto fix.

All parsers are auto fixed with a high number of successful fixes, but all of them have some incorrect fixes. None of the parsers have any missed fixes. A summary of the evaluation on the instance based test cases can be seen in Table 6.29. There is a high number of successful fixes and a low number of missed and incorrect fixes for all the parsers. The number of successful fixes, missed fixes, and incorrect fixes for the parsers DocumentBuilder, SAXParser, and Transformer are identical. This is due to the similar way these parsers are implemented. The auto fixes were also evaluated on the test cases based on flow variants from Juliet. The result of this evaluation is shown in Table 6.30. As can be seen, the auto fixes were successfully able to fix all the vulnerabilities in these test cases.

Parser	Successful fixes	Missed fixes	Incorrect fixes
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	14	0	2

Table 6.29: Summary of the successful fixes, missed fixes, and incorrect fixes after evaluating the auto fixes on the instance based test cases shown in Table D.1

Parser	Successful fixes	Missed fixes	Incorrect fixes
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	17	0	0

Table 6.30: Summary of the successful fixes, missed fixes, and incorrect fixes after evaluating the auto fix mechanism on the Juliet style test cases

When auto fixing all XXE vulnerabilities of each type, the execution took between 604ms and 861ms for between 27 and 34 test cases. The result of evaluating the performance of the auto fixes for XXE is shown in Table 6.31. The execution time was measured

on a PC with 16G of memory and a 3.9GHz CPU using Windows 10 pro. These numbers were obtained by fixing all vulnerabilities for each parser using the test bed detailed in section 6.1. As can be seen, auto fixing close to 30 vulnerabilities takes less than a second for each parser.

Parser	Number of vulnerabilities auto fixed	Execution time
DocumentBuilder	33	668ms
XMLStreamReader	27	604ms
XMLEventReader	27	764ms
FilteredReader	27	684ms
SAXParser	34	861ms
XMLReader	29	799ms
Transformer	33	807ms

Table 6.31: Execution time for auto fixes

6.4.4 Dependence of Auto Fixes on Correctness of Detection

The auto fixes use the vulnerability location reported by the detectors as a starting point for identifying where to perform the auto fix. If the detector reports a false positive, then the auto fix will use this as an input and end up fixing the wrong vulnerability. This may lead to incorrect fixes that will break the functionality of the code being fixed. If a detector has a lot of false negatives, then the vulnerability is not identified and the auto fixes cannot be applied. Therefore, the precision and recall of the detectors need to be high in order for the auto fixes to be useful.

As shown in Table 6.9 and Table 6.10, the existing detectors had high precision but low recall. This means that for all the vulnerabilities the detectors reported all of them were correct, however, a lot of vulnerabilities were missed. Therefore, the detection needed to be improved for the auto fixes to be useful. As shown in Table 6.19 and Table 6.20, the new detectors had high precision and high recall, making the auto fixes more useful since all of the vulnerabilities found are actual vulnerabilities, and fewer vulnerabilities were missed.

Chapter 7

Discussion

In this chapter, the research results presented in chapter 6 will be compared with related work. First, the test bed will be compared with other test beds used for evaluations of security-based source code analysis tools, and test beds used for evaluation of automatic fixes of other code bugs. Then the improved XXE detection will be compared to other detection tools and techniques. Last the auto fixing technique presented in this thesis will be compared to other code repair techniques and implementations. This chapter also lists the most important threats to validity and explains how these threats were mitigated.

7.1 RQ1: How can a Test Suite for Evaluating Web Sec Auto Fixes be Designed for XML External Entity Attacks?

This section contains discussions about how well the test beds function for the evaluation of auto fixing tools.

7.1.1 Comparison with Classical Auto Fixing Test Beds

The automatic evaluation of auto fixes is the main contribution of the test bed. No such feature exists for security auto fixing tools. The automatic evaluation makes checking the performance of auto fixing tools much simpler than it would be without this feature. It was discovered through the literature review that many tools are evaluated on various code bases and use various metrics. This test bed allows researchers to quickly and easily do a thorough evaluation of their auto fixing tool.

ManyBugs [70] is a test bed for C code bugs that does implement validations for auto fixes. This is the only test bed discovered through our literature review that is designed with validations in mind. The test cases in the test bed are not relevant for our purposes since they focus on common C bugs, not web security in Java. The addition of automatic

evaluation of fixes allows researchers to apply fixes to all vulnerabilities, and then evaluate the effectiveness of all fixes within minutes.

7.1.2 Comparison With Other Software Security Test Beds

Multiple existing test beds for software security were discovered through the prestudy. These test beds were focused on evaluating detection not auto fixing vulnerabilities. Our test bed was based on the flow variants and overall design of the Juliet Test Suite. This helps ensure that the test bed covers as many flow variants as possible and avoids the test bed being tailored to the auto fixing tool being evaluated.

The limitation of putting each vulnerability in a separate function in a test case makes it difficult to find ways to make realistic methods with advanced data flow. This results in some of the test cases being unnatural, for instance instantiating multiple parsers in a row. These more complex usages are more natural in test beds designed as complete test beds, like WebGoat [106]. Although some test cases seem somewhat out of place, we argue that they test how the tools handle complex data flows just as well as a complete application would. It is also easier to systematically test different combinations of parsers with different control flows in a test bed consisting of a collection of test cases.

The design also makes it easier to add new flow variants for existing vulnerabilities as well as new vulnerabilities to the test bed. This is the primary reason many existing test beds use this design. Juliet's design made it easy to add new test cases for this project, something that would have been more difficult for a test bed designed as a complete app.

Comparison with Juliet Test Suite

The overall design of the test bed was designed to be as similar as possible to the Juliet Test Suite. This is done to ensure better coverage for relevant data flows present in Juliet but with added tests for XML External Entity vulnerabilities. There are still a few key differences between our test bed and Juliet.

One major difference between our test bed and Juliet is the addition of 11 instance-based test cases. Though not present in the Juliet Test Suite, these test cases are somewhat similar to test cases in the Juliet Test Suite for C. These cases have tests where variables are added to the stack or heap, similar to how our test bed has class and method variables.

Our test bed also has test cases with multiple parsers in the same test case. This is not present in Juliet. Juliet's variants concern wrapping vulnerable code in different code structures like conditionals, loops, etc. but does not explore the effects of having multiple parsers within the same method on detection and auto fixing tools.

7.1.3 Strengths and Weaknesses of Test Bed Design

Ease of Evaluating Auto Fixes

The main strength of the test bed proposed in this research compared to other test beds is the automatic evaluation of auto fixes. This saves a lot of time when evaluating using this test bed compared to when auto fixing on test beds without this feature. Evaluating auto fixes using the Juliet Test Suite requires manual evaluation of each auto fix. Evaluations

of auto fixes on applications without a known number of vulnerabilities require manual verification of fixes to see if they are correct. These types of evaluations also cannot give the number of false negatives since there is no ground truth.

More Robust Testing of Intraprocedural Flows

Another benefit of this test bed is that it has more robust testing of intraprocedural data flows for instance based vulnerabilities. The test bed includes not only test cases similar to Juliet, but also a number of other test cases to test more complex data flows. Juliet does not have any cases where instances are made secure and then insecure.

The test bed created for this project also covers the XXE better than other test beds. Juliet does not include any test cases for XXE. WebGoat has some, but not many examples of XXE. The test bed used for our evaluation contains a large collection of XXE cases for seven of the most used XML parsers in Java.

Lack of Interprocedural Data Flows

The test bed does not include test cases with interprocedural data flows. This was done since the detection and auto fixing tools only cover intraprocedural analysis. All test cases requiring interprocedural analysis would fail and would not give any results that are not already known. This limitation of the test bed results in better results in the evaluation of our tool than a similar evaluation with a test bed including interprocedural data flows. This was taken into account during the evaluation. This type of test case would need to be added to the test bed if it is going to be used to evaluate an XXE auto fixing tool that includes interprocedural analysis.

Evaluation of Functions with multiple parsers

If a test case has more than one vulnerable parser in the same method the automated tests cannot check if both have been fixed. The tests only check the output of one XML parser. This means that it is technically possible to have one secure parser that refuses to parse external entities and another insecure parser that is in the same method, and this would go unnoticed by the automated tests if the return value comes from the secure parser. This does not occur in any of the tests based on the Juliet Test Suite but does occur in some of the 11 added tests for each parser. Since the tests only evaluate the returned values from the function parsing XML it is still possible to, for example, use XML for remote code execution or denial of service with another parser that does not need to be used for the return value of the function being tested.

Impossible to Evaluate XML Entity Expansion for Newer Versions of Java

The test bed design supports evaluating XML Entity Expansion, however, the Java Development Kit (JDK) imposes a default limit of 64000 on the number of entity expansions since JDK 7u45 and JDK 8 [30]. Therefore testing if the parser is vulnerable to the billion laughs DOS attack shown in Listing 6 does not make sense for newer Java versions, since the parsers are inherently secure from these attacks. If one wants to evaluate the detection

and fixing of XML Entity Expansion, an older version of Java has to be used. Therefore, a Java code base with an older version of Java would be better suited for this.

Duplicate Code

There is a substantial amount of duplicate code in the test bed. This is a result of having many very similar test cases with similar automated evaluations for each case. This is common for test beds with this design but could be improved to make the test beds more understandable for researchers. One way to reduce the amount of code in the test bed would be to extract common functionality to a superclass that the specific cases could extend. This would reduce the amount of code but would force test cases to be very similar in design by forcing them to implement a common interface. This could limit how well the test bed covers relevant test cases.

The Thoroughness of Evaluation of Security

The tests used to ensure that the test cases are secure after applying fixes could be more thorough. To be sure that an auto fix actually fixes a security vulnerability the test bed would need more automated tests. This was not implemented since checking if a function is vulnerable to XXE is a research topic on its own. This has been researched by [120] who found a large collection of tests that could be used to evaluate the security of parsers. The automated tests should be sufficient and do check properly for XML External Entities, but other relevant, parser specific vulnerabilities could be omitted.

7.1.4 Comparison with Related Work

Most test beds used for evaluation of auto fixing tools were proprietary code bases [53, 19]. Many used open source software [81, 140, 113, 141, 107, 1, 39, 80], and some used home made code bases [80, 20, 13, 17]. All the test beds used can be found in Table A.1. Only a few used properly documented test beds [51, 38, 139, 117, 73].

No existing test bed focusing on automatic evaluation of auto fixes of security vulnerabilities was found. The Juliet Test Suite contains a collection of web vulnerabilities. WebGoat is a full application containing web vulnerabilities. However, neither of these include automatic verification of either the vulnerability of or the security of test cases. Furthermore, Juliet Test Suite did not contain test cases for XXE.

The only test bed with an automatic evaluation of auto fixes was found to be ManyBugs [70], which only has test cases in the C programming language. This test bed is a collection of vulnerable functions with associated test cases for testing the correctness of a fix by verifying the functionality of them. However, it does not verify the security of the fix since it is made for classical auto fixing. Furthermore, it does not contain different flow variants for testing software security for different vulnerabilities in a repeatable manner across different vulnerabilities.

Test suites have been made for testing the security of different XML parsers focusing on testing how vulnerable different parsers are to different attacks as mentioned in section 3.4. However, none of the studies identified focused on testing the detection performance of static analysis tools, but rather the vulnerabilities present within the parsers.

For instance [120] evaluated 1459 attacks on 30 different XML parsers to identify which ones the parsers were vulnerable to, whereas [131] surveyed different potential attack on web services with a focus on DOS attacks, and [100, 45] implemented a tool to run various XML attacks against a live web server. None of these were test cases with repeatable flow variants for either testing the detection performance of static analysis tools, or for evaluating the functionality and security of auto fixes.

We, therefore, argue the novelty of test bed since it tests the detection of XXE for different flow variants, includes additional flow variants for testing different ways of initializing an object and testing subsequent functions invoked on this instance, and verifies both the functionality of and tests the security of automatic fixes. The verification of the functionality and checking the security of the test cases both before and after performing an auto fix are automated, which was not found to have been done previously.

7.2 RQ2.1: What are the shortcomings of existing static analysis tools for the detection of XML External Entity attacks?

In this section, the strengths and weaknesses of the existing approach used to detect XXE in FindSecBugs will be presented. The approach will also be compared to other vulnerability detection tools.

7.2.1 Strengths and Weaknesses of the Existing Detection of XXE in FindSecBugs

Detection of Simple XXE Vulnerabilities with No False Positives

The main strength of the way detection is implemented in FindSecBugs is that it is able to handle the simplest forms of XXE with no false positives. The existing detectors handle test case one through six of the instance based test cases, and all the test cases based on flow variants from Juliet Test Suite. For the instance based test cases, the first six are the simplest test cases only differing in whether the instance has been initialized into a field or a variable. For the Juliet style test cases, they primarily differ in the control flow constructs used. This shows that the existing detectors are able to handle the simplest variants of XXE where the vulnerability occurs once within the same method and where different control flow constructs are used with no complex data or control flow. Compared to the detectors for other vulnerabilities in FindSecBugs, the recall and precision of the XXE detectors are much higher, as mentioned by Oyetoyan et al. [107].

Simple Detection Approach

The main strength of the approach used by the existing detectors is that it is very simple and straight forward for developers to understand. A double pattern matching approach is used. First, a parser use is found, then the secure calls are found. Except for Xxedetector, all the parameters of these calls are checked to determine if the parser use is secure. Very little

effort is required for creating detectors that are capable of identifying many vulnerabilities. The execution time of the detectors is also low due to the simplicity of the detectors. The other approaches used for source code analysis shown in section 3.2 are more complicated than the pattern matching used for the detection of XXE by FindSecBugs.

Lacking Detection for Complex Data Flows

The main weakness of the XXE detection approach used by FindSecBugs is that it cannot handle more complex data and control flow. As shown in Table D.1, test cases seven through 10 test the use of multiple parsers within the same method, and initializing multiple parsers within the same method using a factory that is made both secure and insecure. All of the existing detectors fell through on these test cases using more complex data and control flow. As noted in section 6.2.2, the TransformerFactoryDetector detector was able to handle test cases eight and 10 by a coincidence, hence it can be regarded as not handling this test case, since if the secure calls had been exchanged then the detector would not have been able to handle the test case. The existing detectors are also not able to handle the use of a parser and a separate instance in the same method whose secure method is similar. This is tested by test case 11. All of the existing detectors fell through on this test case. This type of flow requires data flow analysis to be detected. As shown in section 3.2, this type of analysis is used by other tools to detect other vulnerabilities, but not currently to detect XXE.

Inconsistent Implementations

Internal inconsistency in how the approach has been implemented in FindSecBugs was also found. TransformerFactoryDetector toggles a boolean when secure and insecure calls are found allowing it to detect vulnerabilities in a sequence of consecutive calls happening after each other. An example of this is shown in Listing 19. Neither XxeDetector nor XMLStreamReaderDetector is capable of this. XxeDetector also does not check the second parameter of a call and only considers the first which makes it incapable of differentiating between secure and insecure calls for the features in subsection 2.7.1 where the second parameter differentiates the secure call from the insecure one. TransformerFactoryDetector also reports the vulnerability where the parser is initialized, not where it is used. This may confuse developers who may assume that initializing the parser is itself vulnerable when it is parsing an XML document that is vulnerable. The taint analysis implemented by FindSecBugs is an example of a set of detectors with a consistent implementation making the detectors perform comparably.

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);
// This parser use is regarded as vulnerable
// by TransformerFactoryDetector
Transformer transformer = factory.newTransformer();
```

Listing 19: Example of subsequent calls handled by TransformerFactoryDetector

7.2.2 Comparison with Related Work

No evaluation of FindSecBugs' detection of XXE was found in the prestudy. This is because there is no good test bed for this evaluation. Previous evaluations of other vulnerability detectors in FindSecBugs have used similar research methods to ours where a test bed is used to evaluate the detection performance. The metrics collected by researchers were discovered to vary drastically, as shown in chapter 6. As shown in Table A.1, many other researchers only collected the number of true positives for instance [11, 73, 77, 134], others also included false positives for instance [81, 115], but few collected false negatives [38, 136, 129, 65]. Therefore many researchers could not calculate the recall or precision of the tools they were evaluating. This was a result of many researchers using poor test beds with no ground truth. Because no other evaluations of XXE detection in FindSecBugs exist, we argue the novelty of our analysis into the shortcomings of the existing static analysis tools for detecting XML related vulnerabilities.

7.3 RQ2.2: How can the Detection of XML External Entities be Improved using Different Techniques?

7.3.1 Strengths and Weaknesses of Instruction Based Data Flow Analysis

Analyze Sequence of Method Calls

The main strength of the instruction based data flow analysis is that it is able to capture the sequence of the calls within the method. This is not possible using simpler analysis approaches like pattern matching. Therefore, it can determine a sequence of secure and vulnerable calls as either ultimately secure or insecure. This approach is also able to identify a generalized number of parameters. This means that it can be used to identify secure or vulnerable calls that require an arbitrary number of parameters. Instruction based data flow analysis is able to determine where the secure call has been made within the method instead of just somewhere in the method. This means that it can determine when a parser has been initialized and identify the secure or vulnerable calls performed after the parser has been initialized.

Lacking Instance Tracking

The main weakness of the approach is that it is not capable of knowing which instance calls are invoked on thus not knowing which secure calls are for which instance. It also gets confused by two instances with similar secure calls because it does not know which of these they were invoked on. The approach also only works on the intraprocedural level. To handle these data flows, the analysis tool needs to be extended with an instance tracking mechanism.

Benefits Compared to Pattern Matching Approach Used by FindSecBugs

Compared to the existing approach, the instruction based data flow analysis handles a generalized number of parameters and can determine where in the method a secure or vulnerable call has been made. Due to the internal inconsistency within FindSecBugs, it is also able to better determine if subsequent calls made after one another are ultimately vulnerable or secure. While TransformerFactoryDetector is able to handle this, it is more of a coincidence. XxeDetector and XMLStreamReaderDetector is not capable of this. However, evaluating the detection performance of the instruction based data flow approach on the test cases in section 6.1 revealed that it did not perform significantly better than the existing approach. This means that an approach that is capable of tracking calls performed on each instance is needed.

7.3.2 Strengths and Weaknesses of Instance Tracking

Handles More Complex Data Flows

The main strength of the new detector implementations based on the instance tracking approach is that they are able to handle more complex control and data flow variants in addition to the simplest forms of XXE with no false positives, compared to FindSecBugs. All of the Juliet style test cases and all the instance based test cases are handled by this approach.

Handles Multiple Parameters, Secure and Insecure Calls, and Multiple Parsers

Another strength of the instance tracking based approach is that a generalized number of parameters for each call can be detected allowing for generalizability. Both secure and insecure calls can be tracked, and singular and multiple calls can be differentiated between. Additionally, the sequence of the calls called on the instance is kept track of, which means that the approach is able to identify when an instance is vulnerable and when it is secure. Due to keeping track of the calls performed on different instances, it is also capable of knowing which parsers within a method that is vulnerable and which is secure.

Generalizable Detectors

The implementation of the instance tracker has also been done in a generalized manner. The core algorithm has been extracted to a superclass allowing subclasses to supply the algorithm with which initialization instructions to look for, which return values to look for, where to report the vulnerability, and which singular and multiple tracked calls to look for. This was done to improve upon the internal inconsistency found within FindSecBugs for the existing detectors. Extracting the similarities to a common superclass ensures that the detectors using this approach all use it correctly and in the same manner. Having extracted and generalized the approach also makes it easy to implement additional detectors. For instance, a new detector for identifying insecure cookies was implemented as shown in Appendix H by using this generalized implementation. This shows that instance tracking is not only applicable to detect XXE, but it is applicable to other vulnerabilities as well.

Lack of Interprocedural Analysis

The main weakness of the instance tracking based approach is that it performs only intraprocedural analysis and not interprocedural analysis. However, it was designed to perform only intraprocedural analysis to focus on supporting the auto fixing tool, thus, this is a reasonable weakness considering it was not implemented. FindSecBugs includes limited support for interprocedural analysis [56] and does not support context sensitive interprocedural analysis at all [57]. Performing interprocedural analysis can still be achieved by ”faking it” [57] through the use of two detectors where one pass is done over the Java class under consideration to collect information for use in a second detector. However, since this support is lacking at best this was decided to be out of scope for this thesis.

Performance Penalty for More Advanced Analysis

Another weakness of the instance tracker is that it takes longer to run compared to the existing detectors. However, it is only 31% slower than FindSecBugs' XXE detectors on 5 million lines of code and the performance is comparable on 24K lines of code. Therefore, this should not be too much of a problem. Additionally, this is only a one-time penalty due to FindSecBugs only scanning files that have changed after the initial run.

7.3.3 Extending FindSecBugs for Detection Compared to Making Stand Alone Tool

FindSecBugs is Easy to Extend

FindSecBugs proved easy to extend with detection functionality and worked well as a baseline for implementing the new detectors. While the new detectors do not inherently use any of the detection mechanisms provided by FindSecBugs, but instead build upon the functionality within SpotBugs which FindSecBugs is an extension to, FindSecBugs is set up with a testing framework that proved quite useful in ironing out the kinks of the detection mechanism. The test cases enumerated in Table D.1 could be integrated directly into FindSecBugs providing immediate feedback on the detection performance of both the existing and the new detectors.

Extending FindSecBugs Requires Backwards Compatibility

Using FindSecBugs also meant that a choice between being backward compatible with regards to where the bugs are reported needed to be made. As discussed in subsection 6.4.4, for the auto fix mechanism later implemented it would have been more useful to report where the fix should have been inserted as opposed to where the bug occurred. However, a choice was made to keep reporting the bugs on the same lines as the current detectors in FindSecBugs. A discrepancy was found between the existing XML detectors where the Transformer detector reported the bug on the line where the parser was initialized, whereas the remaining XML detectors reported the bug where the bug occurred. We decided to change the new transformer detector to be more consistent with the rest of the existing XML detectors, reporting the vulnerability where the parser is used as opposed to where it is initialized. Maintaining backward compatibility will also reduce the overhead

of developers adopting the new detectors since visibly it will not change anything within the IDE except for more vulnerabilities being detected.

Pros and Cons of Test Driven Development in FindSecBugs

Using the test-driven development method in FindSecBugs helps ensure that the detectors that are implemented can detect the vulnerability patterns that have been tested for. However, it also shows the importance of these test cases being correct. If the test cases are wrong then the detection mechanism implemented will be wrong. Additionally, if improvements are made upon existing detectors or new detectors are implemented then it is not unreasonable that the test cases are assumed to be correct. If these are wrong, then the new detection mechanism will also be wrong because of this.

7.3.4 Different Mitigation Strategies for XXE Vulnerabilities

Both OWASP and Oracle give recommendations for best practices in web security, but their recommendations differ in multiple ways.

OWASP and Oracle Recommendations

Oracle and OWASP both describe different attributes that can be set to make an XML parser secure as shown in subsection 2.7.1. However, where OWASP specifies each attribute that should be set alongside the value recommended, Oracle mentions more top-level features that can be enabled which in turn enables or disables multiple underlying attributes. For instance, enabling feature secure processing implicitly disallows DTD processing and restricts external access effectively mitigating both CWE-611 and CWE-776. Similarly, using a security manager implicitly enables the feature secure processing.

Impact on Developers

In general, the main difference between the suggested mitigation strategies from OWASP and Oracle is that the strategies presented by OWASP are more specific, whereas the strategies presented by Oracle are easier to remember. Therefore, for the detection part, we chose to detect the attributes from both OWASP and Oracle. Since the mitigation strategies from Oracle incurs less overhead for the developer to both remember and verify, for the auto fix part the parameters recommended by Oracle were used. In general, it would be easier for developers to know which features to set if both OWASP and Oracle agreed.

7.3.5 Comparison with Related Work

Most previous work into software security vulnerability detection has focused on SQL Injections and Cross-Site Scripting. Little research has been done on the detection of XXE. For instance, Baset and Denning [16] found that most existing tools cover SQLi and XSS vulnerabilities well.

Previous research into the detection of other web security vulnerabilities also shows that data flow analysis performs better than pattern matching. This is the case for vulnerabilities including complex data flows, such as SQL injections and XSS.

Dynamic analysis is another vulnerability detection technique that was not evaluated in this research. That is because it is a run-time analysis and does not provide pointers to vulnerabilities in source code. This is required to be able to auto fix the detected vulnerabilities.

Machine learning approaches were found to have been used for vulnerability detection. However, no publicly available test set for training such models for the detection of XML External Entities was found. Therefore, this approach was not attempted for detecting XXE.

We, therefore, argue the novelty of our evaluation of different techniques for improving the detection of XXE vulnerabilities, and our new instance tracking mechanism.

7.4 RQ3: How can Auto Fixing of XML External Entities be Implemented using an IDE Plugin

In this section, the auto fixes for XXE vulnerabilities will be discussed. First, the strengths and weaknesses will be presented. Then the strengths and weaknesses of backward compatible auto fixes will be discussed. Finally, the auto fix approach will be compared to related work.

7.4.1 Strengths and Weaknesses of using AST Based Auto Fixes

Correct Auto Fix Suggestion

The main strength of the auto fixes is that they provide vulnerability specific auto fix suggestions. This means that for each vulnerability, a specific auto fix for mitigating that vulnerability is suggested. This allows bulk auto fixes to be performed. ASIDE and ESVD suggests all auto fixes for each detected vulnerability. This means that to apply the auto fix, the developer has to manually select the correct one from a list of all auto fixes provided by the tool, since the tools don't know which auto fix is correct. Additionally, this means that bulk auto fixes cannot be applied. This makes it difficult for developers to both identify the correct auto fix to apply and to fix more than one vulnerability.

Ensures Correctness of Code Structure

When applying fixes to source code using ASTs, developers are assured that the code change will not break the semantics of the code. This means that the fix will not leave any incorrect tokens such as curly braces or commas. This is something that developers would have to check for themselves when creating auto fixing tools that change code as a string. Auto fixing approaches based on machine learning have a large risk of breaking the code structure when applying a change if there are no manual tests in place to avoid this.

Generalizable Design

The implementation of the auto fix approach has also been done in a generalized manner similar to the implementation of the instance tracker detector. The core algorithm has been

extracted to a superclass allowing subclasses to supply the algorithm with which nodes to look for, and which AST nodes to insert to perform the auto fix. This makes it easy to implement auto fixes using this approach. For instance, auto fixes for insecure cookies were implemented as shown in Appendix J. This shows that the approach can be used to auto fix other vulnerabilities in addition to XXE. Auto fixes for SQL injections [129, 128] and XSS were found, however, the methods used were only applicable to one specific vulnerability and could not be used to auto fix other injection vulnerabilities as well.

Needs to Traverse AST to Identify Where to Insert Auto Fix

The main weakness of the AST based auto fix approach is that it cannot fix parsers that have been made explicitly vulnerable through calls to insecure methods. This is because the AST based approach currently does not identify statements that make a parser explicitly vulnerable. This is the main reason for the incorrect auto fixes for test case eight and 10 shown in subsection 6.4.2. In both tests, a parser has been explicitly made vulnerable as described in Table D.1. When attempting to perform the auto fix, the auto fix identifies the factory to insert the auto fix on and inserts the auto fix right after the factory is created. However, the code making the factory explicitly insecure is not removed or modified. The new detectors still report the remaining vulnerabilities, letting the developer choose whether to remove the insecure calls or not. If the developer removes the insecure calls the auto fixes inserted will become effective.

Lack of Fixes with Interprocedural Data Flows

Another weakness of the implementation of the AST based auto fixes is that they only support intraprocedural fixes and not interprocedural fixes. However, they were implemented to perform intraprocedural fixes, thus, this is a reasonable weakness considering it was not implemented. This is due to the detectors only supporting intraprocedural analysis.

7.4.2 Strengths and Weaknesses of Backwards Compatible Auto Fixes

Seamless Extension of FindSecBugs

One of the strengths of the auto fix approach is that they are backward compatible with the existing detectors in FindSecBugs. This means that developers can use the existing detectors with the novel auto fixes for XXE. As shown in section 6.2, the existing detectors were unable to handle more complex data and control flow. However, since the precision is high, the auto fixes can use the input from these detectors to sufficiently perform the auto fixes for the identified vulnerabilities. Using the input from the instance based detectors detailed in section 6.3 will allow for more vulnerabilities to be fixed compared to using the existing detectors due to the higher number of recall.

Dependence on Input From Detectors

One of the main weaknesses of the auto fix approach, however, is also that they are backward compatible with the existing detectors. For many of the XXE vulnerabilities, the vulnerability occurs on a different instance than where the auto fix should be inserted as

shown in the specific implementations explained in Appendix I. The existing detectors report the vulnerability where it occurs and not where the auto fix should be inserted. This means that the auto fixes have to traverse the AST to identify where the fix should be inserted. Since the AST lacks the necessary information to perform data flow analysis [28], the traversal part of the auto fixes has to use pattern matching to traverse from one AST node to another. In most cases this is sufficient, however, it makes the auto fix approach more complicated. To mitigate this weakness, the vulnerability could be reported on the source code line where the auto fix should have been inserted. This would remove the need for the auto fix to identify which instance to insert the vulnerability on. However, developers would not know where the vulnerability occurred only where the auto fix should have been applied. Making this change would likely reduce the number of incorrect fixes.

7.4.3 Comparison with Related Work

Existing auto fixing tools provide unspecific auto fix suggestions and list all the available auto fixes. ASIDE and ESVD uses ESAPI to provide their auto fixes. However, neither of these present the auto fixes applicable to a certain vulnerability but instead lists all the auto fixes ESAPI supports making it difficult to know which auto fix to apply as mentioned in Appendix B. Neither ASIDE nor ESVD supports identifying XML vulnerabilities which means it is impossible to identify these vulnerabilities and apply the auto fixes using ESAPI using these tools.

Auto fix mechanisms were identified for SQLi and XSS as mentioned in the prestudy in Appendix B, but none were found for fixing XXE. These techniques are specific to SQLi and XSS for instance using prepared statements or inserting runtime patches. Auto fixes for classical bugs were also identified as shown in subsection 3.2.8. Of these, only AST based fixes seemed promising. No data set was found to be available for training machine learning models, satisfiability modulo problems are more tailored to incorrect if statements, and genetic programming requires a test suite which adds additional overhead. Therefore, an auto fix was implemented based on modifying the AST.

We, therefore, argue the novelty of the auto fix mechanism because it supports auto fixing XML External Entities, it presents only relevant auto fixes for the vulnerability under consideration, it supports different XML parsers, and handles each parser according to the implementation of the auto fixes for that parser, The auto fix mechanism implemented is also generalizable and can be used to auto fix other vulnerabilities as well. Such a generalized auto fix mechanism for instance related vulnerabilities based on modifying the AST was not found to have been done before.

7.5 Threats to Validity

This section details the threats to the validity of the research conducted in this thesis. These threats include biases in the way detection and auto fixing is evaluated, biases in the way the tools could be affected by the test bed, and the way biases in the prestudy could affect decisions made in the research.

7.5.1 Threats to Internal Validity

Reliance on Correctness of FindSecBugs

The soundness of our findings depends on the soundness of Eclipses APIs and FindSecBugs' underlying detection framework. If these do not perform as intended, the auto fixing tool proposed in this thesis will not perform as demonstrated in this thesis. This risk is the reason improved detection was made to be a part of this research. To reduce the risk of incorrect detection by FindSecBugs, the tool was evaluated thoroughly, and changes were made to ensure a better and more reliable detection mechanism.

It is also possible that there was selection bias in the prestudy when collecting papers to review. Missed information from the prestudy could threaten the internal validity of the results in this thesis since this information is the foundation of this study. To avoid selection bias in the prestudy, a large amount of literature was reviewed. These papers were collected using a structured literature review to avoid bias in the selection of papers.

Bias in Prestudy

The prestudy may contain threats to its validity. If the prestudy is biased it would affect the correctness of the related work presented. This could mean that the motivations behind the research could be invalid, or that the detection and auto fixing techniques examined in this study are not complete. To avoid biases in the prestudy the information was collected in a structured literature review. Relevant search terms were used and as many papers as possible were collected to avoid selection bias. To get a collection of papers as large and as representative as possible, snowballing was used in addition to the search. It is also possible that research into detection and auto fixing of security vulnerabilities is affected by publication bias. Many papers present only the positive results of their research. To mitigate this as many papers as possible were collected.

Tailored Detection and Auto Fixing for Test Bed

The auto fixes may be tailored for the test bed. This would result in better evaluation scores when evaluating the auto fixes and detection on this test bed than on other test beds. This was mitigated by focusing on using the detection and fixing techniques discovered in the literature review, and evaluating these as objectively as possible.

7.5.2 Threats to External Validity

Biased Test Bed

One threat to the external validity of our research is the limited test bed used for evaluation. The results of this evaluation cannot simply be generalized and compared to results from evaluations on other test beds. This is the result of possible selection bias when creating test cases for the test bed. To mitigate selection bias, the test bed was made to be as similar as possible to the Juliet Test Suite which was found to be the most common test bed for evaluation of security analysis tools. This should help our results be more comparable to

evaluations on other test beds. If there is a bias in the Juliet Test Bed, this bias will be present in the test bed created based on this.

The evaluations done in this study also only focused on intraprocedural data flows. The results can therefore not be directly compared to results of tools evaluated on test beds including interprocedural data flows. The lack of interprocedural test cases also gives better testing results than a test bed with these cases. It should, therefore, be noted that this study does not claim to cover interprocedural data flows, and is only a proof of concept for auto fixing XXE limited to intraprocedural data flows.

Generalizability of FindSecBugs and Eclipse

Another threat is that the study was based on FindSecBugs and the Eclipse IDE and its APIs. FindSecBugs might be implemented differently than other analysis tools, and Eclipse could have APIs that are not found in other IDEs. 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. To avoid this, generalized approaches such as modifying an AST and using data flow analysis were used.

Chapter 8

Conclusion and Future Work

8.1 Summary of Related Work

A summary of the related work for this project is shown below. It was discovered through a structured literature review in the prestudy phase of the project as detailed in Appendix B. The related work consists of existing tools for auto fixing, research into auto fixing, information about test beds as well as testing methodologies used by other researchers, and studies into the prevalence of XML External Entities.

8.1.1 Research into Detection

A lot of research has been done on tools for detecting web security vulnerabilities. Various techniques, such as static analysis, dynamic analysis, pattern matching, data flow analysis, and machine learning approaches have been used. However, the majority of the research was found to focus on injection related vulnerabilities such as SQLi and XSS. These vulnerabilities are covered well by existing tools and research into them. Few tools for detecting XML related vulnerabilities were identified.

Some research uses dynamic analysis to detect vulnerabilities [134, 97, 69]. This approach is not applicable to auto fixing because it cannot report the line the vulnerability occurred on. This is required for auto fixing.

Machine learning approaches have been attempted for the detection of security vulnerabilities [52, 53]. This type of detection is limited by the availability of data sets with vulnerable and secure code. No such data set exists for XXE.

8.1.2 Research into Fixing

Few existing tools for auto fixing of software security vulnerabilities were identified. Most research focuses on auto fixing SQLi and XSS with specific auto fixes only applicable to these vulnerabilities. However, no tools that auto fix XXE vulnerabilities in Java were

identified. A common problem discovered in existing auto fixing tools for software security was that they did not provide specific enough auto fix suggestions. Some tools provide all possible mitigation of other unrelated security vulnerabilities for a detected vulnerability. This limits these tools to only be usable by experts who already know how to mitigate vulnerabilities.

Classical auto fix techniques were also identified. These used approaches based on code search, pattern matching, genetic programming, and satisfiability modulo theory problems. Many of the tools using code search and pattern matching worked on the AST instead of the source code which was found to perform well.

8.1.3 Test Beds and Testing Methods

There are multiple existing test beds for software security that the test bed created as part of this research drew inspiration from. The most common test bed for evaluating web security vulnerability detection is Juliet, which the new test bed is based on. WebGoat is another test bed designed as a web application with multiple vulnerabilities. This design makes the test beds more realistic than Juliet and other test beds designed as collections of vulnerable functions. Neither of these test beds are designed for evaluation of auto fixes. ManyBugs is a test bed for C code bugs that does implement validations for auto fixes. This is the only test bed discovered through our literature review that is designed with validations in mind. The test cases in the test bed are not relevant for our purposes since they focus on common C bugs, not web security in Java. Most research into detection and fixing of security vulnerabilities were evaluated on either open source software or proprietary test beds. None of the test beds discovered through our literature review covered XXE.

8.2 Research Motivation

XML External Entity attacks are ranked as the fourth most critical security risk to web applications by OWASP [125]. A successful XML External Entity attack can have severe consequences, such as for information extraction, Server-Side Request Forgery, denial of service attacks, and remote code execution. The Common Weakness Enumeration classifies XXE as part of the top 25 most dangerous software errors in their list from 2019 [86]. The XML parsers in Java are all vulnerable to XEE by default and require developers to manually make them secure [120]. The settings needed to make the different parsers secure vary from parser to parser and different sources give different guidelines on what should be done to mitigate XXE. This makes it difficult for developers to fix parsers and highlights the need for tool support to fix vulnerabilities. There has been done a substantial amount of research into auto fixing SQL Injection and Cross-Site Scripting attacks, however, a lack of research into auto fixing of XXE was found. The existing detection of XXE was also found to be based only on basic detection techniques such as pattern matching, making this detection incapable of detecting more advanced data flows. Existing test beds for evaluating the detection performance of static analysis tools were found, however, no test beds suitable for evaluating software security auto fixes were found. Therefore, an improved detection mechanism, a novel auto fix mechanism, and a novel test bed focusing on XML related vulnerabilities were created.

8.3 Contributions and Conclusion

The main contributions of this research are a novel analysis of existing static analysis tools for detecting XML related vulnerabilities, an improved detection method for XXE, a novel auto fix method for XXE, and a novel test bed used for evaluation of auto fixing tools. As part of the improved detection method, a generalized instance tracker which can be used to track vulnerabilities occurring due to insecure instances was created. Similarly, as part of the auto fix method, a generalized auto fix mechanism for fixing vulnerabilities occurring due to insecure instances was created. These are elaborated in more detail in the following paragraphs.

8.3.1 RQ1: How can a test suite for evaluating web sec auto fixes be designed for XML External Entity attacks?

This master thesis presents a novel test bed for evaluation of auto fixes for web security vulnerabilities. This was one of the things discovered to be missing in previous research into tool support for web security. This test bed builds on existing test beds, mainly the Juliet Test Suite, by reusing their flow variants to ensure coverage of relevant control and data flows. While inspecting the existing flow variants in Juliet Test Suite it was found that they did not include flow variants related to how instances are initialized, nor how different methods are invoked on the instances. Therefore additional test cases were identified and implemented to cover these. The test bed covers all the XML parsers present in Java which are supported by existing detection tools making it possible to use in future research for comparing different detection and auto fixing tools.

8.3.2 RQ2.1: What are the shortcomings of existing static analysis tools for the detection of XML External Entity attacks?

The tool FindSecBugs was evaluated on a novel test bed for testing the detection capabilities of static code analysis tools for XML related vulnerabilities. Our novel results show that the existing detection performed well on test cases based on Juliet Test Suite, but fell through on new test cases created to test vulnerabilities related to insecure instances. The existing detection mechanism was found to primarily use pattern matching which is not able to take the context of the parser instances being analyzed into account. A control flow graph was used in the analysis, but only to enable pattern matching on the other methods called within a Java method and not caring about the different control flows within the method. The existing analysis was found to be incapable of reporting vulnerabilities from multiple parser instances within the same Java method as well as being easily confused by similar function calls used for different parsers if they were used together in the same method.

When analyzing the test driven development strategy used to implement the existing detectors in FindSecBugs, some of these test cases were found to have been wrongly implemented explicitly enabling vulnerabilities in the parsers instead of disabling them. The detectors for *DocumentBuilder*, *SAXParser*, and *XMLReader* were found to not check the

value of the secure attributes being set. Because of this, the existing detectors were incapable of identifying these vulnerabilities in their own test set.

8.3.3 RQ2.2 How can the Detection of XML External Entities be Improved Using Different Techniques?

Instruction based data flow analysis was researched to identify if it could be used to identify XML related vulnerabilities. It is shown that this is capable of performing the same detection as the existing detectors within FindSecBugs in addition to considering the parameters the methods are invoked with. It is also shown that it is not able to reason about different parsers within the Java method and treats them all as a long sequence of function calls.

Analyzing different XML parser implementations showed that XML parsers are vulnerable due to missing calls needed to make them secure. Both OWASP and Oracle suggest different parameters that can be set to make them secure. Treating the parser instance as either vulnerable or not vulnerable with the vulnerability state depending on the methods invoked on the instance allows the detector to know when the parser should be considered secure and when it should not be. This was realized by implementing a novel instance tracker which is supplied with which calls to look for and whether this call should make the instance vulnerable or secure if called. To implement such an instance tracker, the Java operand stack is analyzed to determine which values each method the instance tracker is looking for is called with. Additionally, the operand stack is used to determine which instances are the same and which are different. Effectively, a data flow analysis has been implemented by modeling the operand stack. The instance tracking mechanism was created to be general. Since it is not specific to XXE, it can also be applied to other vulnerabilities that can be treated as insecure or secure instances. An example implementation of this was shown by implementing an insecure cookie detector.

8.3.4 RQ3: How can auto fixing of XML External Entities be implemented using an IDE plugin

A novel auto fix method for fixing XML External Entities based on modifying the AST was implemented. Having identified that XML parsers can be treated as insecure or secure instances, this novel auto fix mechanism was implemented as a generalized auto fix mechanism for vulnerabilities occurring due to insecure instances. By supplying the auto fix mechanism with which nodes to look for and which fixes to insert the generalized auto fix mechanism handles traversing the AST to identify which node to insert the fix on and inserts the fix. Since the auto fix mechanism is not specific to XXE, it can also be applied to other vulnerabilities that can be treated as insecure or secure instances. An example implementation of this was shown by implementing auto fixes for insecure cookies.

8.4 Future Work

A test bed for automatically evaluating the functionality and security of XXE vulnerabilities were created. A generalize instance tracker detector and a generalized auto fix

mechanism for this detector was implemented. These were used to both detect and auto fix XXE. During this research different detection techniques were explored. Different existing auto fix techniques and existing test beds were also identified and evaluated through a literature review. The study discovered multiple parts of the detection and auto fixing process that can be improved by future work. The most important are listed in this section.

8.4.1 Improve Test Bed

Since FindSecBugs only supports detecting vulnerabilities for the XML parsers DocumentBuilder, SAXParser, XMLStreamReader, XMLEventReader, FilteredReader, Transformer, and XMLReader test cases for these parsers were created. However, Java has a multitude of different XML parsers as shown in section 3.4. Therefore, when detectors for these parsers are implemented corresponding test cases should be created too.

The test bed created only includes automatic validation of the functionality and automated tests for the security for the test cases made for XXE. However, Juliet Test Suite includes numerous other test cases. These can be extended with similar tests to give the same functionality to the rest of the test cases. The new tests written were created manually making the process time-consuming. If others choose to use this design for validation of auto fixes it would be recommended to automate the generation of the test scripts. This should be possible since this is how the vulnerability test cases in Juliet are created.

8.4.2 Improve Detection

Support Interprocedural Analysis

The detection mechanism implemented is only intraprocedural. As discussed, SpotBugs only supports limited interprocedural analysis. However, implementing interprocedural analysis would in turn improve the detection capabilities of the instance tracking mechanism. Interprocedural analysis would allow the detection of XXE vulnerabilities, as well as other instance related vulnerabilities, to consider using separate safe Java methods or safe Java classes to wrap around a parser implementation. This is not an unreasonable case since having a wrapper Java method or a wrapper Java class to always retrieve a safe parser would mean that the parser is always secure when used. Detecting vulnerabilities in such Java methods or Java classes would ensure that they are correctly mitigating the XXE vulnerability.

Report Bug Where it Should be Fixed

The implications of reporting the bug in different places can also be evaluated. Currently, bugs are reported where they can be exploited. However, as discussed, it could be beneficial for the auto fix to know where the fix should have been inserted. A survey can be performed to evaluate whether developers prefer to know where the vulnerability occurred, or where the fix should have been inserted. It is not obvious for a developer if a fix needs to be applied to an XML parser or to the parser's factory, before instantiating the parser.

Add Support for Additional XML Parsers

Finally, since FindSecBugs only supports detecting vulnerabilities for the XML parsers DocumentBuilder, SAXParser, XMLStreamReader, XMLEventReader, FilteredReader, Transformer, and XMLReader new detectors for these were implemented to be able to compare the performance of the new detectors with the existing detectors. However, as shown in section 3.4 there exists multiple additional parsers for Java. Since a generalized instance tracking mechanism was implemented additional detectors were not implemented since this is mainly engineering work. However, additional detectors for these parsers could be made using the instance tracking mechanism.

8.4.3 Improve Auto Fixing

Replace Vulnerable Calls with Secure Calls

If an XML parser has been made explicitly vulnerable by explicitly enabling vulnerable features such as DTD processing or XXE on the parser then the auto fix tool inserts the auto fixes before the method calls enabling these features making the auto fix ineffective. This is because the auto fix in its current state does not remove or modify code that explicitly enables vulnerable features. A better solution would be to find the method calls that enable the insecure features and either remove them completely, or replace them with secure method calls that disable the vulnerable features. This would improve the limitations discussed in subsection 7.4.1, and reduce the number of incorrect fixes shown in subsection 6.4.3.

Add Only Select Secure Methods

The auto fixes also do not consider if parts of the fix have been applied, but always inserts all the instance invocations necessary to make the instance secure. If an instance has been made partially secure the detectors are capable of reporting this, however, the auto fixes do not know about the partially secure state of the instance. Thus, if the auto fix is applied, some of the secure method invocations will be duplicated. As discussed in subsection 7.4.1, this is not an issue for XML parsers, but it could be for other auto fixes that can be based on the generalized auto fix mechanism. This can be solved by identifying which code lines are necessary to be inserted for the auto fix to be applied, instead of inserting all the code lines that are part of the auto fix.

Identify Best Place to Report Vulnerability for Auto Fixing

The auto fixes have been implemented to be backward compatible with the existing detectors in FindSecBugs, except for the detector for the *Transformer* parser as discussed in subsection 7.3.3. To stay backward compatible the vulnerability for the new detectors is reported where the vulnerability occurred instead of where the fix should be applied. This means that the current auto fixes include a lot of code to traverse the AST from the node the vulnerability is reported on to where the auto fix should be applied. If an evaluation is done on where the best place to report the vulnerability on finds out that this is where the auto fix should be applied, then the auto fixes should be revisited to update support for

this. Similarly, if such an evaluation finds out that the best place to report the vulnerability on is where the vulnerability occurred then the possibility of using two representations - one for where the vulnerability occurred and one for where the auto fix should be inserted should be looked into.

Add Support for Interprocedual Auto Fixes

An XXE auto fixing tool should also be capable of fixing vulnerabilities that require interprocedural program analysis to be identified. The main reason this has not been implemented is that the detectors were not implemented to support interprocedural analysis. However, adding auto fixing of vulnerabilities identified using such an analysis should be done when detectors that support this analysis is available. Researchers will have to identify how to find the correct method to apply the fix in. This can be solved by giving the developer the choice of which method to apply the fix in. A different solution would be to report the vulnerability on the code line the fix should be applied to as discussed above.

Bibliography

[1] H. H. AlBreiki and Q. H. Mahmoud. "Evaluation of static analysis tools for software security". In: *2014 10th International Conference on Innovations in Information Technology (IIT)*. Nov. 2014, pp. 93–98. DOI: 10.1109/INNOVATIONS.2014.6987569.

[2] F. E. Allen and J. Cocke. "A Program Data Flow Analysis Procedure". In: *Commun. ACM* 19.3 (Mar. 1976), pp. 137–. ISSN: 0001-0782. DOI: 10.1145/360018.360025. URL: http://doi.acm.org/10.1145/360018.360025.

[3] Apache. *Features*. Accessed Jun 13, 2020. 2020. URL: http://xerces.apache.org/xerces-j/features.html.

[4] Apache. *Parser Features*. Accessed Jun 13, 2020. 2020. URL: https://xerces.apache.org/xerces2-j/features.html.

[5] Philippe Arteau. *Find Security Bugs*. Accessed September 22, 2019. 2019. URL: https://find-sec-bugs.github.io/.

[6] Philippe Arteau. *Find Security Bugs*. Accessed May 12, 2020. 2020. URL: https://find-sec-bugs.github.io/bugs.htm.

[7] Philippe Arteau. *OWASP Find Security Bugs Plugin*. Accessed Jun 20, 2020. 2019. URL: https://search.maven.org/artifact/com.h3xstream.findsecbugs/findsecbugs-plugin/1.10.1/jar.

[8] Cyrille Artho and Armin Biere. "Combined static and dynamic analysis". In: *Electronic Notes in Theoretical Computer Science*. Vol. 131. May 2005, pp. 3–14. DOI: 10.1016/j.entcs.2005.01.018.

[9] *ASM*. Accessed September 22, 2019. 2019. URL: https://asm.ow2.io/index.html.

[10] *ASM Performance Benchmarks*. Accessed September 22, 2019. 2019. URL: https://asm.ow2.io/performance.html.

[11] N. Ayewah et al. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (Sept. 2008), pp. 22–29. ISSN: 1937-4194. DOI: 10.1109/MS.2008.130.

[12] Dejan Baca. “Identifying security relevant warnings from static code analysis tools through code tainting”. In: *ARES 2010 - 5th International Conference on Availability, Reliability, and Security*. 2010, pp. 386–390. ISBN: 9780769539652. DOI: 10.1109/ARES.2010.108.

[13] Dejan Baca et al. “Improving software security with static automated code analysis in an industry setting”. In: *Software - Practice and Experience* 43.3 (Mar. 2013), pp. 259–279. ISSN: 00380644. DOI: 10.1002/spe.2109.

[14] Vipin Balachandran. “Fix-it: An extensible code auto-fix component in review bot”. In: *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*. IEEE Computer Society, 2013, pp. 167–172. ISBN: 9781467357395. DOI: 10.1109/SCAM.2013.6648198.

[15] Santa Barbara and Muath Abdullah Alkhalaf. *Automatic Detection and Repair of Input Validation and Sanitization Bugs*. Tech. rep. 2014.

[16] A. Z. Baset and T. Denning. “IDE Plugins for Detecting Input-Validation Vulnerabilities”. In: *2017 IEEE Security and Privacy Workshops (SPW)*. May 2017, pp. 143–146. DOI: 10.1109/SPW.2017.37.

[17] “Benchmarking Static Analysis Tools for Web Security”. In: *IEEE Transactions on Reliability* 67.3 (Sept. 2018), pp. 1159–1175. ISSN: 00189529. DOI: 10.1109/TR.2018.2839339.

[18] Andreas Berger and Torstein Molland. *xxe-autofix-tool*. Accessed Jun 21, 2020. 2020. URL: https://github.com/Berger-and-Molland/xxe-autofix-tool.

[19] Chandrapal Chahar, Vishal Singh Chauhan, and Manik Lal Das. “Code Analysis for Software and System Security Using Open Source Tools”. In: *Information Security Journal: A Global Perspective* 21.6 (2012), pp. 346–352. DOI: 10.1080/19393555.2012.727132. URL: https://doi.org/10.1080/19393555.2012.727132.

[20] Chandrapal Chahar, Vishal Singh Chauhan, and Manik Lal Das. “Code Analysis for Software and System Security Using Open Source Tools”. In: *Information Security Journal* 21.6 (Jan. 2012), pp. 346–352. ISSN: 19393555. DOI: 10.1080/19393555.2012.727132.

[21] C. Chen and M. Li. “SecConfig: A Pre-Active Information Security Protection Technique”. In: *2008 Fourth International Conference on Networked Computing and Advanced Information Management*. Vol. 2. Sept. 2008, pp. 648–652. DOI: 10.1109/NCM.2008.79.

[22] B. Chess and G. McGraw. “Static analysis for security”. In: *IEEE Security Privacy* 2.6 (Nov. 2004), pp. 76–79. DOI: 10.1109/MSP.2004.111.

[23] Zack Coker and Munawar Hafiz. “Program transformations to fix C integers”. In: *Proceedings - International Conference on Software Engineering*. 2013, pp. 792–801. ISBN: 9781467330763. DOI: 10.1109/ICSE.2013.6606625.

[24] Brian Cole et al. “Improving Your Software Using Static Analysis to Find Bugs”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 673–674. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176667. URL: http://doi.acm.org/10.1145/1176617.1176667.

[25] Eclipse contributors. *Eclipse Documentation*. Accessed February 21, 2020. 2020. URL: https://help.eclipse.org/2019-12/index.jsp.

[26] “Control flow analysis”. In: *Proceedings of a Symposium on Compiler Optimization*. Association for Computing Machinery, Inc, July 1970, pp. 1–19. DOI: 10.1145/800028.808479.

[27] Keith D. Cooper and Linda Torczon. “Chapter 3 - Parsers”. In: *Engineering a Compiler (Second Edition)*. Ed. by Keith D. Cooper and Linda Torczon. Second Edition. Boston: Morgan Kaufmann, 2012, pp. 83–159. ISBN: 978-0-12-088478-0. DOI: https://doi.org/10.1016/B978-0-12-088478-0.00003-7. URL: http://www.sciencedirect.com/science/article/pii/B9780120884780000037.

[28] Keith D. Cooper and Linda Torczon. “Chapter 5 - Intermediate Representations”. In: *Engineering a Compiler (Second Edition)*. Ed. by Keith D. Cooper and Linda Torczon. Second Edition. Boston: Morgan Kaufmann, 2012, pp. 221–268. ISBN: 978-0-12-088478-0. DOI: https://doi.org/10.1016/B978-0-12-088478-0.00005-0. URL: http://www.sciencedirect.com/science/article/pii/B9780120884780000050.

[29] Oracle Corporation. *Class DocumentBuilderFactory (Java SE 13 & JDK 13)*. Accessed May 25, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/parsers/DocumentBuilderFactory.html.

[30] Oracle Corporation. *Java API for XML Processing (JAXP) Security Guide*. Accessed May 25, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/security/java-api-xml-processing-jaxp-security-guide.html.

[31] Oracle Corporation. *SAXParserFactory (Java SE 13 & JDK 13)*. Accessed May 25, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/parsers/SAXParserFactory.html.

[32] Oracle Corporation. *The Java Virtual Machine Specification*. Accessed September 22, 2019. 2019. URL: https://docs.oracle.com/javase/specs/jvms/se13/html/index.html.

[33] Oracle Corporation. *TransformerFactory (Java SE 13 & JDK 13)*. Accessed May 25, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/transform/TransformerFactory.html.

[34] Oracle Corporation. *XMLConstants (Java SE 13 & JDK 13)*. Accessed Jun 13, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/XMLConstants.html.

[35] Oracle Corporation. *XMLInputFactory (Java SE 13 & JDK 13)*. Accessed May 25, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/stream/XMLInputFactory.html.

[36] Oracle Corporation. *XMLReaderFactory (Java SE 13 & JDK 13)*. Accessed May 25, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/org/xml/sax/helpers/XMLReaderFactory.html.

[37] Oracle Corporation. *XMLStreamReader (Java SE 13 & JDK 13)*. Accessed May 28, 2020. 2020. URL: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/stream/XMLStreamReader.html.

[38] Erik Derr. *Understanding and assessing security on Android via static code analysis*. 2017. DOI: http://dx.doi.org/10.22028/D291-27345.

[39] Gabriel Díaz and Juan Ramón Bermejo. "Static analysis of source code security: Assessment of tools against SAMATE tests". In: *Information and Software Technology* 55.8 (Aug. 2013), pp. 1462–1476. ISSN: 09505849. DOI: 10.1016/j.infsof.2013.02.005.

[40] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. "Demand-driven Computation of Interprocedural Data Flow". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 37–48. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199461. URL: http://doi.acm.org/10.1145/199448.199461.

[41] Fred Dysart and Mark Sherriff. "Automated fix generator for SQL injection attacks". In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. 2008, pp. 311–312. ISBN: 9780769534053. DOI: 10.1109/ISSRE.2008.44.

[42] Eclipse contributors. *Quick Fix*. Accessed November 19, 2019. URL: https://help.eclipse.org/kepler/index.jsp?topic=%5C%2Forg.eclipse.jdt.doc.user%5C%2Fconcepts%5C%2Fconcept-quickfix-assist.htm.

[43] Eclipse Foundation. *Eclipse*. Accessed November 19, 2019. URL: https://www.eclipse.org/.

[44] Eclipse Foundation, Inc. *Runtime Analysis Tools (RAT)*. Accessed October 31, 2019. 2010. URL: https://www.eclipse.org/proposals/tools.rat/.

[45] A. Falkenberg et al. "A New Approach towards DoS Penetration Testing on Web Services". In: *2013 IEEE 20th International Conference on Web Services*. 2013, pp. 491–498.

[46] OpenJS Foundation. *Eslint*. Accessed October 30, 2019. 2019. URL: https://eslint.org/.

[47] The Apache Software Foundation. *Apache Commons BCEL*. Accessed September 22, 2019. 2019. URL: https://commons.apache.org/proper/commons-bcel/.

[48] Scott Frame and John W Coffey. "A Comparison of Functional and Imperative Programming Techniques for Mathematical Software Development". In: *Journal of Systemics, Cybernetics and Informatics* 12.2 (2014).

[49] Google, Inc. *Codepro AnalytiX*. Accessed October 31, 2019. 2015. URL: https://web.archive.org/web/20150919011445/https://developers.google.com/java-dev-tools/codepro/doc/history.

[50] Google, Inc. *WindowBuilder Pro Eclipse Donation FAQ*. Accessed October 31, 2019. 2012. URL: https://web.archive.org/web/20120314010806/http://code.google.com/javadevtools/eclipse-donation-faq.html.

[51] Katerina Goseva-Popstojanova and Andrei Perhinschi. "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68 (2015), pp. 18–33. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.08.002. URL: http://www.sciencedirect.com/science/article/pii/S0950584915001366.

[52] Jacob A. Harer et al. "Automated software vulnerability detection with machine learning". In: (Feb. 2018). arXiv: 1803.04497. URL: http://arxiv.org/abs/1803.04497.

[53] Jacob Harer et al. "Learning to Repair Software Vulnerabilities with Generative Adversarial Networks". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 7933–7943. URL: http://papers.nips.cc/paper/8018-learning-to-repair-software-vulnerabilities-with-generative-adversarial-networks.pdf.

[54] Mary Jean Harrold and Mary Lou Soffa. "Interprocedual Data Flow Testing". In: *SIGSOFT Softw. Eng. Notes* 14.8 (Nov. 1989), pp. 158–167. ISSN: 0163-5948. DOI: 10.1145/75309.75327. URL: http://doi.acm.org/10.1145/75309.75327.

[55] David Hovemeyer. *SpotBugs API Documentation*. Accessed May 16, 2020. URL: https://javadoc.io/doc/com.github.spotbugs/spotbugs/latest/index.html.

[56] David Hovemeyer. *The Architecture of FindBugs*. Accessed May 13, 2020. URL: https://github.com/spotbugs/spotbugs/blob/master/spotbugs/design/architecture/architecture.tex.

[57] David Hovemeyer. *Using FindBugs for Research*. Accessed May 19, 2020. URL: https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/findbugs-tutorials/uffr-talk.pdf.

[58] David Hovemeyer and William Pugh. “Finding Bugs is Easy”. In: *SIGPLAN Not.* 39.12 (Dec. 2004), pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. URL: http://doi.acm.org/10.1145/1052883.1052895.

[59] S. Jan, C. D. Nguyen, and L. Briand. “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*. Aug. 2015, pp. 233–241. DOI: 10.1109/QRS.2015.42.

[60] Dennis Jeffrey et al. “BugFix: A learning-based tool to assist developers in fixing bugs”. In: *IEEE International Conference on Program Comprehension* (2009), pp. 70–79. DOI: 10.1109/ICPC.2009.5090029.

[61] Jetbrains. *IntelliJIDEA*. Accessed December 08, 2019. 2019. URL: https://www.jetbrains.com/idea/.

[62] T. Ji et al. “Automated Program Repair by Using Similar Code Containing Fix Ingredients”. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. June 2016, pp. 197–202. DOI: 10.1109/COMPSAC.2016.69.

[63] B. Johnson et al. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. May 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[64] René Just, Darioush Jalali, and Michael D. Ernst. “Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 437–440. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628055. URL: http://doi.acm.org/10.1145/2610384.2628055.

[65] Yalin Ke et al. “Repairing programs with semantic code search”. In: *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*. Institute of Electrical and Electronics Engineers Inc., Jan. 2016, pp. 295–306. ISBN: 9781509000241. DOI: 10.1109/ASE.2015.60.

[66] Dongsun Kim et al. “Automatic patch generation learned from human-written patches”. In: *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 2013, pp. 802–811. ISBN: 9781467330763. DOI: 10.1109/ICSE.2013.6606626.

[67] kjlubick. *fb-contrib Eclipse quick fix plugin*. Accessed November 19, 2019. URL: https://github.com/kjlubick/fb-contrib-eclipse-quick-fixes.

[68] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. “Fast Pattern Matching in Strings”. In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024. eprint: https://doi.org/10.1137/0206024. URL: https://doi.org/10.1137/0206024.

[69] T. Kwon and Z. Su. “Automatic Detection of Unsafe Dynamic Component Loadings”. In: *IEEE Transactions on Software Engineering* 38.2 (Mar. 2012), pp. 293–313. ISSN: 2326-3881. DOI: 10.1109/TSE.2011.108.

[70] C. Le Goues et al. “The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs”. In: *IEEE Transactions on Software Engineering* 41.12 (Dec. 2015), pp. 1236–1256. ISSN: 2326-3881. DOI: 10.1109/TSE.2015.2454513.

[71] Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. “Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities”. In: *Proceedings of the Evaluation and Assessment on Software Engineering*. EASE ’19. Copenhagen, Denmark: ACM, 2019, pp. 200–209. ISBN: 978-1-4503-7145-2. DOI: 10.1145/3319008.3319011. URL: http://doi.acm.org/10.1145/3319008.3319011.

[72] Kui Liu et al. “AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations”. In: *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*. Institute of Electrical and Electronics Engineers Inc., Mar. 2019, pp. 456–467. ISBN: 9781728105918. DOI: 10.1109/SANER.2019.8667970. arXiv: 1812.07270.

[73] Kui Liu et al. “LSRepair: Live Search of Fix Ingredients for Automated Program Repair”. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. Vol. 2018-December. IEEE Computer Society, July 2018, pp. 658–662. ISBN: 9781728119700. DOI: 10.1109/APSEC.2018.00085.

[74] Kui Liu et al. “Mining Fix Patterns for FindBugs Violations”. In: *IEEE Transactions on Software Engineering* (2018). ISSN: 19393520. DOI: 10.1109/TSE.2018.2884955. arXiv: 1712.03201.

[75] Francesco Logozzo and Manuel Fähndrich. “On the relative completeness of bytecode analysis versus source code analysis”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4959 LNCS. 2008, pp. 197–212. ISBN: 3540787909. DOI: 10.1007/978-3-540-78791-4_14.

[76] P. Louridas. “Static code analysis”. In: *IEEE Software* 23.4 (July 2006), pp. 58–61. DOI: 10.1109/MS.2006.114.

[77] Z. Lu and S. Mukhopadhyay. “Model-Based Static Source Code Analysis of Java Programs with Applications to Android Security”. In: *2012 IEEE 36th Annual Computer Software and Applications Conference*. July 2012, pp. 322–327.

[78] *ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs*. Accessed December 4, 2019. 2017. URL: https://repairbenchmarks.cs.umass.edu/.

[79] MaxNad. *Implement a base detector to track specific calls (#211)*. Accessed Jun 19, 2020. 2016. URL: https://github.com/find-sec-bugs/find-sec-bugs/pull/220.

[80] Ryan K. McLean. “Comparing static security analysis tools using open source software”. In: *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability Companion, SERE-C 2012*. 2012, pp. 68–74. ISBN: 9780769547435. DOI: 10.1109/SERE-C.2012.16.

[81] I. Medeiros, N. Neves, and M. Correia. “Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining”. In: *IEEE Transactions on Reliability* 65.1 (Mar. 2016), pp. 54–69. DOI: 10.1109/TR.2015.2457411.

[82] Microsoft. *Insecure DTD Processing*. Accessed September 22, 2019. 2019. URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/ca3075?view=vs-2019.

[83] Microsoft. *Visual Studio*. Accessed December 08, 2019. 2019. URL: https://visualstudio.microsoft.com/.

[84] MITRE. *CCWE-352: Cross-Site Request Forgery (CSRF)*. Accessed November 5, 2019. URL: https://cwe.mitre.org/data/definitions/352.html.

[85] MITRE. *Common Vulnerabilities and Exposures (CVE)*. Accessed September 22, 2019. 2019. URL: https://cwe.mitre.org/.

[86] MITRE. *CWE VIEW: Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors*. Accessed November 23, 2019. 2019. URL: https://cwe.mitre.org/data/definitions/1200.html.

[87] MITRE. *CWE-611: Improper Restriction of XML External Entity Reference*. Accessed Jun 01, 2020. 2019. URL: https://cwe.mitre.org/data/definitions/611.html.

[88] MITRE. *CWE-776: Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')*. Accessed Jun 01, 2020. 2019. URL: https://cwe.mitre.org/data/definitions/776.html.

[89] Torstein Molland and Andreas Berger. *Autofix feature of software security vulnerability detection IDE plugins*. Project report in TDT4501. Department of Information Security, Communication Technology, NTNU – Norwegian University of Science, and Technology, Dec. 2019.

[90] Paul Muntean et al. “Automated generation of buffer overflow quick fixes using symbolic execution and SMT”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9337. Springer Verlag, 2015, pp. 441–456. ISBN: 9783319242545. DOI: 10.1007/978-3-319-24255-2_32.

[91] Hoang Duong Thien Nguyen et al. “SemFix: Program repair via semantic analysis”. In: *Proceedings - International Conference on Software Engineering*. 2013, pp. 772–781. ISBN: 9781467330763. DOI: 10.1109/ICSE.2013.6606623.

[92] Hung Viet Nguyen et al. “Auto-locating and fix-propagating for HTML validation errors to PHP server-side code”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*. 2011, pp. 13–22. ISBN: 9781457716393. DOI: 10.1109/ASE.2011.6100047.

[93] NIST. *Juliet Test Suite v1.2 for Java User Guide*. Accessed May 12, 2019. 2012. URL: https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf.

[94] NIST. *Static Analysis Tool Exposition (SATE) IV*. Accessed June 11, 2020. 2013. URL: https://samate.nist.gov/SATE4.html.

[95] NIST. *Test Suites*. Accessed October 29, 2019. 2017. URL: https://samate.nist.gov/SRD/testsuite.php.

[96] J. Novak, A. Krajnc, and R. Žontar. "Taxonomy of static code analysis tools". In: *The 33rd International Convention MIPRO*. May 2010, pp. 418–422.

[97] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Exterminator: Automatically Correcting Memory Errors with High Probability". In: *Commun. ACM* 51.12 (Dec. 2008), pp. 87–95. ISSN: 0001-0782. DOI: 10.1145/1409360.1409382. URL: http://doi.acm.org/10.1145/1409360.1409382.

[98] Paulo Nunes et al. "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios". In: *Computing* 101.2 (Feb. 2019), pp. 161–185. ISSN: 1436-5057. DOI: 10.1007/s00607-018-0664-z. URL: https://doi.org/10.1007/s00607-018-0664-z.

[99] Briony J Oates. *Researching Information Systems and Computing*. Sage Publications Ltd., 2006. ISBN: 1412902231.

[100] Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira. "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. Montreal, Quebec, Canada: Association for Computing Machinery, 2012. ISBN: 9781450316132. DOI: 10.1145/2405146.2405148. URL: https://doi.org/10.1145/2405146.2405148.

[101] OWASP. *ASIDE GitHub Repository*. Accessed October 21, 2019. 2014. URL: https://github.com/Jing-Xie/owasp-aside.

[102] OWASP. *OWASP ASIDE Project*. Accessed September 22, 2019. 2016. URL: https://www.owasp.org/index.php/OWASP_ASIDE_Project.

[103] OWASP. *OWASP Enterprise Security API*. Accessed November 21, 2019. URL: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API.

[104] OWASP. *OWASP LAPSE Project*. Accessed September 22, 2019. 2017. URL: https://www.owasp.org/index.php/OWASP_LAPSE_Project.

[105] OWASP. *The OWASP Foundation*. Accessed September 22, 2019. 2019. URL: https://www.owasp.org/index.php/Main_Page.

[106] OWASP. *WebGoat Project*. Accessed December 4, 2019. 2019. URL: https://www2.owasp.org/www-project-webgoat/.

[107] Tosin Daniel Oyetoyan et al. "Myths and facts about static application security testing tools: An action research at telenor digital". In: *Lecture Notes in Business Information Processing*. Vol. 314. Springer Verlag, 2018, pp. 86–103. ISBN: 9783319916019. DOI: 10.1007/978-3-319-91602-6_6.

[108] Kai Pan, Sunghun Kim, and E. James Whitehead. “Toward an understanding of bug fix patterns”. In: *Empirical Software Engineering* 14.3 (June 2009), pp. 286–315. ISSN: 13823256. DOI: 10.1007/s10664-008-9077-5.

[109] M. Payer. “The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes”. In: *IEEE Security Privacy* 17.1 (Jan. 2019), pp. 78–82. ISSN: 1558-4046. DOI: 10.1109/MSEC.2018.2889892.

[110] Yu Pei et al. “Automated Program Repair in an Integrated Development Environment”. In: *Proceedings - International Conference on Software Engineering*. Vol. 2. IEEE Computer Society, Aug. 2015, pp. 681–684. ISBN: 9781479919345. DOI: 10.1109/ICSE.2015.222.

[111] N. H. Pham et al. “Detecting recurring and similar software vulnerabilities”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2. May 2010, pp. 227–230. DOI: 10.1145/1810295.1810336.

[112] David Formánek Philippe Arteau and Tomáš Polešovský. *OWASP Find Security Bugs Wiki*. Accessed May 13, 2020. 2020. URL: https://github.com/find-sec-bugs/find-sec-bugs/wiki.

[113] W. Qiang et al. “Patch-Related Vulnerability Detection Based on Symbolic Execution”. In: *IEEE Access* 5 (2017), pp. 20777–20784. DOI: 10.1109/ACCESS.2017.2676161.

[114] R. Saha et al. “Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs”. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. May 2018, pp. 10–13.

[115] Hesam Samirni et al. “Automated repair of HTML generation errors in PHP applications using string constraint solving”. In: *Proceedings - International Conference on Software Engineering*. 2012, pp. 277–287. ISBN: 9781467310673. DOI: 10.1109/ICSE.2012.6227186.

[116] Luciano Sampaio. *Early Security Vulnerability Detector - ESVD*. Accessed September 22, 2019. 2019. URL: https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/.

[117] Luciano Sampaio and Alessandro Garcia. “Exploring context-sensitive data flow analysis for early vulnerability detection”. In: *Journal of Systems and Software* 113 (2016), pp. 337–361. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.12.021. URL: http://www.sciencedirect.com/science/article/pii/S0164121215002873.

[118] Snyk. *Snyk: GitHub Repository*. Accessed October 21, 2019. 2019. URL: https://github.com/snyk/snyk.

[119] *Snyk: Open Source Security Platform*. Accessed October 21, 2019. 2019. URL: https://snyk.io.

[120] Christopher Späth et al. “SoK: XML Parser Vulnerabilities”. In: *WOOT*. 2016.

[121] SpotBugs. *SpotBugs GitHub*. Accessed November 11, 2019. URL: https://github.com/spotbugs/spotbugs.

[122] Stanford. *Introduction to Stanford SecuriBench*. Accessed December 4, 2019. 2005. URL: https://suif.stanford.edu/~livshits/securibench/.

[123] Ting Su et al. "A Survey on Data-Flow Testing". In: *ACM Comput. Surv.* 50.1 (Mar. 2017), 5:1–5:35. ISSN: 0360-0300. DOI: 10.1145/3020266. URL: http://doi.acm.org/10.1145/3020266.

[124] "Testing for software security: A case study on static code analysis of a file reader Java program". In: vol. 166 AISC. VOL. 1. 2012, pp. 529–538. ISBN: 9783642301568.

[125] The OWASP Foundation. *Owasp Top 10 - 2017*. Accessed January 21st, 2019. 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf.

[126] The OWASP Foundation. *Secure Cookie Flag*. Accessed Jun 01, 2020. 2020. URL: https://owasp.org/www-community/controls/SecureFlag.

[127] The OWASP Foundation. *XML External Entity Prevention*. Accessed May 12, 2020. 2020. URL: https://cheatsheetseries.owasp.org/bundle.zip.

[128] S. Thomas and L. Williams. "Using Automated Fix Generation to Secure SQL Statements". In: *Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007)*. May 2007, pp. 9–9. DOI: 10.1109/SESS.2007.12.

[129] Stephen Thomas, Laurie Williams, and Tao Xie. "On automated prepared statement generation to remove SQL injection vulnerabilities". In: *Information and Software Technology* 51.3 (Mar. 2009), pp. 589–598. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.08.002.

[130] Julian Thomé et al. "JoanAudit: A Tool for Auditing Common Injection Vulnerabilities". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 1004–1008. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3122822. URL: http://doi.acm.org/10.1145/3106237.3122822.

[131] S. Tiwari and P. Singh. "Survey of potential attacks on web services and web service compositions". In: *2011 3rd International Conference on Electronics Computer Technology*. Vol. 2. 2011, pp. 47–51.

[132] Omer Tripp et al. "TAJ: Effective Taint Analysis of Web Applications". In: *SIGPLAN Not.* 44.6 (June 2009), pp. 87–97. ISSN: 0362-1340. DOI: 10.1145/1543135.1542486. URL: http://doi.acm.org/10.1145/1543135.1542486.

[133] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Accessed March 3, 2020. URL: https://www.w3.org/TR/xml/.

[134] T. Wang et al. "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection". In: *2010 IEEE Symposium on Security and Privacy*. May 2010, pp. 497–512. DOI: 10.1109/SP.2010.37.

[135] Westley Weimer et al. “Automatically finding patches using genetic programming”. In: *Proceedings - International Conference on Software Engineering*. 2009, pp. 364–374. ISBN: 9781424434527. DOI: 10.1109/ICSE.2009.5070536.

[136] Jing Xie et al. “ASIDE: IDE support for web application security”. In: *ACM International Conference Proceeding Series*. 2011, pp. 267–276. ISBN: 9781450306720. DOI: 10.1145/2076732.2076770.

[137] Jifeng Xuan et al. “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs”. In: *IEEE Transactions on Software Engineering* 43.1 (Jan. 2017), pp. 34–55. ISSN: 00985589. DOI: 10.1109/TSE.2016.2560811. eprint: 1811.04211.

[138] Jiangtao Xue et al. “History-Driven Fix for Code Quality Issues”. In: *IEEE Access* 7 (Aug. 2019), pp. 111637–111648. DOI: 10.1109/access.2019.2934975.

[139] Fen Yan and Tao Qiao. “Study on the detection of cross-site scripting vulnerabilities based on reverse code audit”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9937 LNCS. Springer Verlag, 2016, pp. 154–163. ISBN: 9783319462561. DOI: 10.1007/978-3-319-46257-8_17.

[140] Hua Yan et al. “AutoFix: an automated approach to memory leak fixing on value-flow slices for C programs”. In: *ACM SIGAPP Applied Computing Review* 16 (Jan. 2017), pp. 38–50. DOI: 10.1145/3040575.3040579.

[141] Chao Zhang et al. “IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time”. In: *Computer Security – ESORICS 2010*. Ed. by Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 71–86. ISBN: 978-3-642-15497-3.

[142] Jian-jun Zhao. “Static analysis of Java bytecode”. In: *Wuhan University Journal of Natural Sciences* 6.1 (Mar. 2001), pp. 383–390. ISSN: 1993-4998. DOI: 10.1007/BF03160273. URL: https://doi.org/10.1007/BF03160273.

[143] Jun Zhu et al. “Mitigating Access Control Vulnerabilities Through Interactive Static Analysis”. In: *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. SACMAT ’15. Vienna, Austria: ACM, 2015, pp. 199–209. ISBN: 978-1-4503-3556-0. DOI: 10.1145/2752952.2752976. URL: http://doi.acm.org/10.1145/2752952.2752976.

Appendix

Appendix A

Summaries of Papers About Auto Fix Tools

The following section outlines the results of the literature review that was conducted as part of the prestudy described in Appendix B. The table includes all papers that include an evaluation of tools discovered in the literature review conducted in the prestudy. For more information about this see the prestudy [89].

When no information was given by a paper on one or more of the criteria the cell is marked with a dash (-). When little, or unclear information was presented by a paper, the information was listed in the table as concisely as possible.

Study. Ref.	**Approach**	**Strength**	**Weaknesses**	**Vulnerabilities covered**	**Evaluation test bed**	**Evaluation results**
"On the capability of static code analysis to detect security vulnerabilities" [51]	Comparison of three Static code analysis tools	No significant performance difference between tools.	*"Comparable or worse performance than random guessing"*.	22 vulnerabilities for C/C++ 19 vulnerabilities for Java	Juliet	27% of C/C++ vulnerabilities and 11% of Java vulnerabilities missed 41% of C/C++ and 21% of Java vulnerabilities detected
"Identifying security relevant warnings from static code analysis tools through code tainting" [12]	Limit existing static analysis tools to increase efficiency	Increased efficiency	Reduced capability of tools	-	Anonymized testset	-
"Testing for software security: A case study on static code analysis of a file reader Java program" [124]	Data flow Control flow	-	-	-	Custom made Java file	-
Understanding and assessing security on Android via static code analysis [38]	Static analysis of Android components	Resilient to byte code obfuscation	Low performance	Outdated components	DroidBench testsuite	97% precision 80% recall

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Model-Based Static Source Code Analysis of Java Programs with Applications to Android Security" [77]	Static analysis techniques combined with model-based deductive verification using SMT solvers	-	-	-	Android Bluetooth ChatServices application Android SMSPopup openGPStracker OpenSudoku	Found a total of 8 mistakes in open source software
"Study on the detection of cross-site scripting vulnerabilities based on reverse code audit" [139]	Reverse code auditing and static analysis	Improved time performance using reverse auditing	None listed (Did not mention false positives or negatives)	XSS	WebGoat	Found as many vulnerabilities as some taint analysis algorithm Used 65% less time no mention of false positives or negatives
"Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining" [81]	Taint analysis and data mining	Reduces false positive rates	Uses unproven AI to reduce false positives	SQLi,XSS, remote file inclusion, local file inclusion, directory traversal and path traversal, source code disclosure, PHP code injection, and OS command injection	45 open source applications	found 431 vulnerabilities, where at least 43 were false positives

Study. Ref.	**Approach**	**Strength**	**Weaknesses**	**Vulnerabilities covered**	**Evaluation test bed**	**Evaluation results**
Automatic Detection and Repair of Input Validation and Sanitization Bugs [15]	Input validation and sanitization language	-	-	Input sanitization vulnerabilities	5 open source PHP applications A number of Javascript sanitizer-function benchmarks	Generates a number of patches with no ground truth
"SecConfig: A Pre-Active Information Security Protection Technique" [21]	Examine client host according to information security requirements.	-	-	31 types of misconfigurations	*'520 end hosts and 20 rounds of experiments (each round per month)'*	*'more than the rate of 95% hosts'*
"Detecting recurring and similar software vulnerabilities" [111]	Datamining to find bugs	Could automatically 'learn' to fix new vulnerabilities	Requires training data to work	-	Red Hat Enterprise Linux ES Version 4 and Apache	Claim to have detected bugs

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection" [134]	Checksum fuzzing using dynamic taint analysis	Implements integrity checks for fixes	-	Buffer overflow, integer overflow, double free, null pointer dereference, and infinite loop	Real-world applications	Detected 27 vulnerabilities in real-world applications
"IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time" [141]	Compiler extension Data flow Using LLVM	Small performance loss No false negatives	Based on LLVM (early stages of development)	Integer-Overflow-to-Buffer-Overflow (IO2BO)	Open source applications: libtiff, ming, dillo, and gstreamer	1% performance decrease No false negatives Unknown false positive rate
"Patch-Related Vulnerability Detection Based on Symbolic Execution" [113]	Static analysis Symbolic execution data flow analysis	Does not require developers to create test cases Uses pruning method to deal with path explosion problem	Does not handle third-party library function calls Only focuses on memory vulnerabilities	Memory vulnerabilities	Patches collected from GNU binutils, GNU coreutils and OpenSSL	Analysis time proportional to the number of paths Has only one false negative

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Exterminator: Automatically Correcting Memory Errors with High Probability" [97]	Dynamic analysis	Provable low false positives and false negative rates	Great variability in performance	Heap-based memory errors	SPECint2000 suite Squid Web cache server Mozilla Web browser	Performance degradation between 0% and 132% Observed 0% false negative rate
"Automatic Detection of Unsafe Dynamic Component Loadings" [69]	Dynamic analysis	-	Code coverage problem due to using dynamic analysis	Component resolution failure Unsafe component resolution	Software for Windows and Linux	Found many vulnerabilities, but no ground truth
"The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes" [109]	Fuzzing	Realistic way of testing web applications	Hard to use for fixing	Input validation related vulnerabilities	-	-
"ASIDE: IDE support for web application security" [136]	Static code analysis Early detection	Provides auto fix suggestions	Uses pattern matching which leads to high false positive rate Auto fixes are generic, not specific	Injection vulnerabilities	Apache Roller	Identified 131 of 143 exploitable taint sources

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"On automated prepared statement generation to remove SQL injection vulnerabilities" [129]	Created a prepared statement replacement algorithm	High success rate	Limited to single file Relies on pattern matching	SQLi	Nettrust, iTrust, WebGoat, Roller (with custom SQLi unit tests). Also tested on: Stanford SecuriBench	Removes 94% of SQLIVs in the custom test set Removes 9 out of 16 SQLIVs in Stanford SecuriBench
"Exploring context-sensitive data flow analysis for early vulnerability detection" [117]	Context-sensitive data flow analysis Early detection	High recall and precision	High memory usage Lower time performance than other tools	11 injection vulnerabilities	BlueBlog PersonalBlog WebGoat Roller Pebble NCO	Better precision than tools its compared with
"An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios" [98]	Combining the output of diverse ASATs to improve performance	Improve performance by using the best tool for each vulnerability class	Recall not always improved by adding ASAT	SQLi and XSS	WordPress plugins from the online WordPress Vulnerability Database	Generally improved performance by combining tools

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Using Static Analysis to Find Bugs" [11]	Static code analysis	Uses control flow and data flow to perform more in-depth analysis	False positives	More than 300 different programming mistakes	Sun's JDK	Found 379 bugs. No ground truth
"Automated software vulnerability detection with machine learning" [52]	Different machine learning models trained on labeled dataset	-	Uses static analyzer to label unlabeled dataset	-	C/C++ packaged distributed with Debian C/C++ functions from public Github repositories	ROC AUC score of 0.82 for Github ROC AUC score of 0.76 for Debian
"Learning to Repair Software Vulnerabilities with Generative Adversarial Networks" [53]	Generative adversarial network to fix software vulnerabilities	Does not require paired examples to train	Trade-off between complex and simple discriminator	-	Generated sequence of 20 random integers Data generated from simple context free grammar SATE IV	Difficult to understand the metrics used

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"AutoFix: an automated approach to memory leak fixing on value-flow slices for C programs" [140]	Static and dynamic program analysis Graph reachability analysis on value flow graph	Auto fixes memory leaks Small overhead for fixes Guarantees memory safety	Compile time transformation, which means it is done after code is written (late detection) Only covers memory leaks	Memory leaks	5 SPEC200 benchmarks (ammp, gcc, perlbmkm, twolf, mesa) a2ps h2o redis	Fixes all leaky allocation sites reported by leak detector SABER Introduces 1.06 % overhead on average
"JoanAudit: A Tool for Auditing Common Injection Vulnerabilities" [130]	Static analysis Security slicing Auto fix	Auto fixes vulnerabilities where input used directly in sinks	Does not evaluate the auto fixes Does not present detection metrics such as precision or recall	Injection vulnerabilities	9 web applications, but does not specify which ones	Claims good runtime performance Claims to not miss any important information for security auditing
"LSRepair: Live Search of Fix Ingredients for Automated Program Repair" [73]	Semantic code search	Uses several search strategies	Code transform problem Naïve code transform	-	Defects4J	Repaired 19 bugs

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Automated Program Repair by Using Similar Code Containing Fix Ingredients" [62]	Similar code detection based on reusability metrics	Suggests more suitable fixes compared to just code search	-	-	Randomly picked six groups of similar code fragments from Tomcat70	Buggy code fragment's NCC decrease after using reusability metric
"History-Driven Fix for Code Quality Issues" [138]	Mining fix patterns from code change history	-	Can only be as good as SonarQube, since it was used for mining	Code Quality Issues (CQIs)	Unclear where the tests came from. May be the 206 GitHub projects they mined	Very obfuscated
"BugFix: A learning-based tool to assist developers in fixing bugs" [60]	Uses database that maps debug-situations to relevant bug-fix descriptions	The tool becomes better over time at predicting most relevant bug fixes	Requires initial training to create knowledge base Requires test cases	-	Subset of Siemens benchmark programs: tcas totinfo sched sched2 replace	Lists detected faults and corresponding fixes without any metrics

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Repairing programs with semantic code search" [65]	Database of code fragments encoded as SMT constraints Use constraint solving to replace buggy code with the code fragments	Uses test cases to guide semantic search Uses human written code templates	Repairs may be hard to maintain due to overfitting	-	IntroClass benchmark	Repairs 150 out of 778 defects Claims higher quality patches
"Automated Program Repair in an Integrated Development Environment" [110]	Dynamic analysis to collect program behavior and validate repairs	No false positives	Cannot provide real time fixes Low performance compared to state of the art	-	Various Eiffel code bases	Suggests fixes for 42% for 200 faults, 25% of which are proper

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Automatic patch generation learned from human-written patches" [66]	Pattern-based automatic program repair	Using templates expands the fixability of patch generation	Cannot generate predicates to satisfy branch conditions Needs bug fix template to generate a patch	-	119 bugs collected from Apache, log4j, Rhino, and AspectJ	Successfully generated 27 patches of 119
"SemFix: Program repair via semantic analysis" [91]	Genetic programming Generates patch using constraint solving	Faster than enumeration and search based techniques	Requires test suite to be available	Branch predicates	Programs from SIR repository: Tcas, Schedule, schedule2, replace, grep Coreutils	Repaired 48 of 90 test cases
"Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs" [137]	Generates patches for conditionals given a test suite	Claims angelic fix localization is faster than symbolic execution	Requires test cases to be correct Angelic fix location may lead to infinite loops	buggy IF conditions Missing preconditions	22 bugs from Apache Commons Math and Apache Commons Lang	17 of 22 bugs successfully fixed

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations" [72]	Code fixes abstracted into fix patterns	Fix patterns are more general than pattern based fixes	Not better than state of the art in terms of bugs fixed	-	Defects4J	34 out of 49 bugs fully fixed 5 out of 14 bugs partially fixed
"Auto-locating and fix-propagating for HTML validation errors to PHP server-side code" [92]	Symbolic execution Mapping algorithm	Achieves high accuracy High performance	Dependent on an external validation tool to find errors	HTML validation errors	SchoolMate TimeClock WebERP UPB AddressBook Manhali	Over 86% accuracy for all test beds
"Fix-it: An extensible code auto-fix component in review bot" [14]	Transformations according to known patterns	Extensible	-	-	User study	Fixes 43% of issues

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Automated generation of buffer overflow quick fixes using symbolic execution and SMT" [90]	Static execution Code patch patterns Satisfiability Modulo Theory Generates bug patches	Patches introduce little compile overhead High performance	Not generalizable approach for larger programs	Buffer overflow	58 C programs from Juliet Test Suite	1.97% overhead
"Automatically finding patches using genetic programming" [135]	Genetic programming	Claims to successfully repair off the shelf software Minimizes successful repairs	Requires test cases to be available High variance in success rate	-	10 open source benchmarks with known defects	Average success: 58.7%
"Automated repair of HTML generation errors in PHP applications using string constraint solving" [115]	Adding, modifies or removes statements that print string literals	Fixes most errors automatically Integrates with Eclipse High performance	Requires test cases	Incorrect constant prints	Custom test set consisting of: faqforge, webchess, schoolmate, hgb, timeclock, dnscript	Has 73% false positives for hgb Has close to 0% false positives for the others Performs 86% of patches automatically

Study. Ref.	Approach	Strength	Weaknesses	Vulnerabilities covered	Evaluation test bed	Evaluation results
"Program transformations to fix C integers" [23]	Program transformations	Handles all types of C integer problems	-	Integer problems	NIST SAMATE 5 open source programs	Modifies 13.18% of expressions on average
"Mining Fix Patterns for FindBugs Violations" [74]	Machine learning to find patterns in fixed and unfixed static analysis violations across revisions of software	The fix patterns are applicable to real bugs	Dependent on input from FindBugs	-	Defects4J	14 bugs detected were 4 fixed

Table A.1: Literature review results [89]

Appendix B

Prestudy

A prestudy [89] was conducted prior to writing this master thesis. The aim of the prestudy was to identify what was the state of the art of detection and auto fixing in the field of software security, and the state of the art of classical auto fixing not necessarily focusing on software security. Tools and approaches were also analyzed to identify how they have been evaluated in terms of what test bed has been used and which metrics have been collected. To answer these questions a literature review was conducted. Additionally, the most popular existing vulnerability detection tools were also empirically evaluated to find out which one was the most promising to be extended with auto fix functionality. The main contribution of the prestudy was a comprehensive overview of the state of auto fixing web application security vulnerabilities in terms of detection approaches, evaluation approaches, and different auto fix approaches.

In the following sections, the findings from the literature review and the empirical evaluation of the tools performed are summarized. This will be used to present which detection mechanisms are already in use, which auto fix tools exist both in the field of software security but also in general, and to show how existing software has been tested and which metrics have been used. We will also argue why there is a need for continued research into auto fixing of software vulnerabilities, and why a common test bed for testing how well tools detect software vulnerabilities and how well tools perform auto fixes is necessary.

B.1 Implementation of the Literature Review

The methodology used to conduct the literature review will be summarized in this section. See [89] for more details. The questions which were answered through the prestudy by performing the literature review were as follows. Please note that the research questions for the prestudy have been renamed to Q1 and Q2 instead of RQ1 and RQ2 to avoid confusion with the research questions for this master thesis:

Q1: What is the state of the art in web application security auto fixing?

Q1.1: What are the existing tools and approaches in the market today?

Q1.2: What are the strengths and weaknesses of different software security tools and approaches?
Q2: What is the state of the art in evaluating auto fixing tools and methods?

The literature review was implemented as a structured literature review based on the steps presented by Oates in Chapter 6 titled *Reviewing the Literature* [99, Chapter 6]. The main goal of the literature review was to evaluate the existing literature and to consolidate the information in order to answer the questions posed. An overview of the steps taken can be seen in Figure B.1.

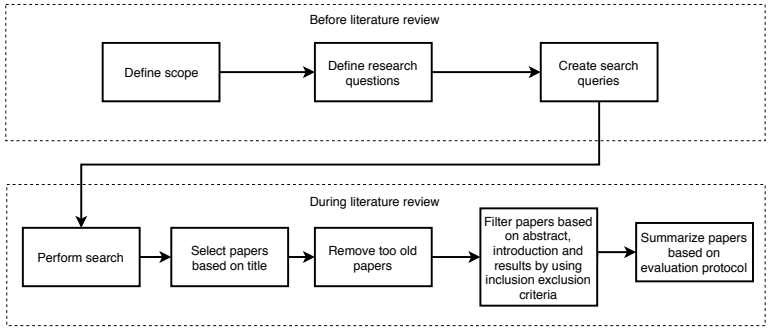


Figure B.1: Overview of the structured literature review process [89]

As can be seen, the scope was first defined to be identifying the current state of the art within software security auto fixing, general bugs, as well as what security vulnerabilities tools are capable of automatically fixing, and how well the tools have been evaluated and the test bed used. Automatic fixing of other bugs was also included since general approaches not specific to software security could be used to fix software security vulnerabilities. The research questions shown above were then defined based on the scope. The literature review itself was conducted by performing a search in prestigious research databases by using the Oria.no search engine using the search queries shown in Figure B.2. After completing the search, the papers were assessed using the inclusion and exclusion criteria shown in Table B.1. After filtering, a total of 49 papers were kept. Both forward and backward snowballing was done on these papers using Oria.no and Google Scholar, after which 61 total papers were kept.

Information from the 61 papers was extracted and summarized based on the evaluation protocol shown in Table B.2. This also shows which research question for the prestudy was answered by which point. Of interest was the approach and scope to identify which approach worked well for which kind of vulnerabilities, the test bed used to identify strengths and weaknesses of these, as well as what metrics different authors used to evaluate their tool and how these results could be compared. The different author's evaluations were also of interest to find the most promising tools, and to identify how the tool could be improved.

- Static code analysis security
- Software security auto* fix
- IDE plugin software security
- Eclipse vulnerabilit*
- Static security analysis tools
- Software vulnerabilit* machine learning
- Auto* fix vulnerabilit* machine learning
- Software security tool*
- Data flow analysis auto* fix

Figure B.2: List of search queries used to obtain papers [89]

Inclusion Criteria	Exclusion Criteria
Paper is published between 2004 and 2019 inclusive (last 15 years)	Duplicate papers
Paper is peer reviewed	Papers that do not contain the information necessary to answer the research questions
Paper is written in English	Inaccessible papers
	White papers

Table B.1: List of the inclusion and exclusion criteria used for filtering the papers [89]

	Title:	Relevant Research Question
1.	Approach & Scope	Q1.1
2.	Test bed	Q2
	Test result	Q1.2 and Q2
3.	Strengths	Q1.2
	Weaknesses	Q1.2

Table B.2: Evaluation criteria for papers used to assess the papers identified in the literature review [89]. Please note that RQs have been replaced with Qs to avoid confusion with RQs for the master thesis.

B.2 Implementation of Empirical Evaluation of Existing Auto Fix Tools

To get a better understanding of the existing tools, and to better answer Q1.1 as shown in section B.1, the tools that were identified through the literature review for Java were

downloaded and evaluated. This was done using the Juliet Test Suite [95]. The tools were empirically evaluated according to the following criteria:

1. Does the tool support the most recent version of Eclipse (Eclipse 2019-09 at the time of writing)?
2. Is there an ongoing active development of the tool?
3. Are the quick fixes implemented in the tool relevant for the identified vulnerability?

These criteria were chosen to identify which of the identified tools seemed the most promising to be extended as part of this master thesis. Results from [71, 107, 117] were also used in this evaluation since these studies performed a qualitative analysis of plugins for detecting security vulnerabilities in Java code. A more in-depth justification of why these criteria were chosen can be found in [89].

B.3 Results of Evaluating Tools

A variety of different tools for detecting security vulnerabilities were found through the literature review based on both static and dynamic analysis approaches. To answer Q1, an empirical evaluation was performed on the tools identified for Java in order to identify the tool that seemed the most promising to be extended with auto fix functionality. A summary of the findings from this evaluation is presented in the next subsection. For a more detailed description of the tools identified and the evaluation results see the prestudy [89].

B.3.1 Q1.1: What are the Existing Tools in the Market Today?

The main tools identified for identifying security vulnerabilities for Java was found to be ASIDE [136], ESVD [117], FindSecBugs [5], JoanAudit [130], LAPSE+ [104], and Snyk [119]. All of these are based on static analysis. All of these tools except Snyk focus on injections, with FindSecBugs focusing on a vast array of different vulnerabilities as well. Snyk focuses primarily on identifying outdated dependencies. The result of empirically evaluating the existing auto fix tools is shown in Table B.3. We wish to remind that this is not a qualitative analysis of the tools, but rather an empiric evaluation in order to identify which tools seem the most promising to extend with auto fix functionality and to learn from the existing techniques employed by these tools. More in-depth evaluations were found to have been done by others, for instance [71] and [107]. As can be seen most of the tools found were outdated, except for Snyk and FindSecBugs which have been updated within the last year. Out of the tools identified, JoanAudit and LAPSE+ were the only ones found to not support the latest version of Eclipse. Both ASIDE and ESVD were found to implement quick fixes, but these were found to be too general to be useful. ESVD crashed a lot in our testing making it difficult to run and evaluate the tool. JoanAudit was found to not be open source making it difficult to evaluate their implementation. Out of the tools identified, FindSecBugs were found to have been most recently updated, detected most vulnerabilities, installed successfully in Eclipse, and thus seemed most promising to extend with automatic fixes. Additionally, an extension for SpotBugs, which FindSecBugs

is based on, called fb-contrib quickfixes was found which provided quick fixes for the classical bugs in SpotBugs. This seemed the most promising to make use of to extend FindSecBugs with auto fix functionality.

Tool	Last Updated	Newest Eclipse Version	Quick Fixes
ASIDE	2012	✓	✓
ESVD	2016	✓	✓
FindSecBugs	2019	✓	✗
JoanAudit	2017	✗	-
LAPSE+	2011	✗	✗
Snyk	2019	✓	✗

Table B.3: Overview of evaluated plugins [89]

B.4 Results of the Literature Review

In this section, the main results from the literature review performed in the prestudy will be summarized. To make it easier to identify which part corresponds to which part of the literature review the content has been structured after the research questions answered in the specialization project. These research questions have been reiterated in section B.1.

B.4.1 Q1.2: What are the Strengths and Weaknesses of Different Software Security Tools?

The strengths and weaknesses of different software security tools were identified through the literature review. The table summarizing these findings have been placed in Table A.1. The following paragraphs are a summary of the key findings in the prestudy.

Most existing tools were found to use static analysis which focused on scanning the bytecode of the program code. The main benefits of using static code analysis were found to be allowing the code to be scanned without executing it, performing the analysis without needing to provide runtime parameters, and early detection instead of late detection. Tools focusing on analyzing the bytecode are also able to leverage the optimizations performed by the compiler to make the analysis easier. Additionally, tools were identified that used bytecode analysis to create automatic fixes by manipulating the abstract syntax tree representation of the code.

Tools using dynamic analysis were also identified. The benefits of these tools were that they can evaluate the program without knowing how it was implemented. Compared to static analysis, dynamic analysis was also found to be more precise. However, it may be difficult to identify the source code responsible for a vulnerability due to dynamic analysis evaluating the program at runtime.

Of the specific techniques used, pattern matching was found to be the most common one. A drawback of using pattern matching is that it leads to a lot of false positives. Data flow analysis was identified as another approach used to help mitigate this. Tools using

data flow analysis were able to take the context of the data flow into account when analyzing the program and helped increase the number of vulnerabilities that can be detected. The main weakness of data flow analysis was found to be the complexity of the analysis. Some authors were also found to use machine learning; however, a key drawback of these methods was found to be the lack of labeled data sets that could be used for training.

Classical tools implementing bug fixes were also examined to identify existing approaches that can be re-used to fix security related vulnerabilities. The main techniques identified were: Code search, pattern matching based approaches, genetic programming, and modeling the bug as satisfiability modulo theory problems. Code search was found to perform well if similar code that could be used to find the patches was present in the code database. However, it performed poorly if the code was not present in the database. Pattern matching approaches were found to be able to modify the Abstract Syntax Tree and performed well if all the necessary patterns were identified to perform the fixes. Bugs that did not fit any of the patterns could not be fixed. Genetic programming approaches searched over a space of replacements from other expressions in the program which helped the solvers evolve patches with greater expressiveness. However, the technique was found to be slow and had a low success rate. Satisfiability modulo theory problems model if-else statements and use a test suite to generate a patch that satisfies the buggy program. However, this approach was found to be limited in its expressiveness.

B.4.2 Q2: The state of the Art In Evaluating Auto Fixing Tools and Methods

Many different test beds were found to be used in evaluating the different tools identified through the literature review. Some used open source software, others used publicly available test beds, whereas others used custom made vulnerable test beds. None of the test beds identified allowed for verifying automatic fixes for software security vulnerabilities. A common test bed with checks for these security fixes is necessary to be able to evaluate and compare different software security auto fix tools. An overview of the different test beds used are shown in Table A.1.

The metrics collected by different researchers were found to vary drastically. Some presented the recall and the precision, whereas others only listed the number of bugs found, or collected insufficient data to be able to calculate any metrics outside of this. A large number of papers were also found to not explain what vulnerabilities they fixed, nor how many vulnerabilities were missed.

The different publicly available test beds used for evaluating the tools identified are shown in Table 3.2. Of these, Juliet and WebGoat are for evaluating software security vulnerabilities, whereas Defects4J and Bugs.jar are for evaluating bugs not necessarily related to software security. Juliet contains vulnerable test cases classified by the CWE code making it the largest test bed for evaluating static analysis tools. WebGoat is a vulnerable web application as opposed to vulnerable methods only. Defects4J contains a collection of vulnerable functions from real-world applications. Bugs.jar contains bugs from open source Java programs.

Many of the tools, as can be seen in Table A.1, were found to be evaluated on proprietary test sets, or on open source projects. The main drawbacks of these were that the test

sets are often not made public making it difficult to replicate the results. This means that one must rely on the metrics published by the authors, which was found to be, in many cases, lacking. Results from evaluating on open source application were also found to not specify which test sets that was used to generate the test cases. Open source application also does not include a ground truth making calculating metrics such as true positives and false positives difficult or impossible.

B.5 Strengths and Weaknesses of Existing Approaches

Pattern matching was found to be the most commonly used technique for detecting vulnerabilities. It is easy to implement and can be used to implement both detection and auto fixes. It was, however, found to be limited in capability due to not taking the information flow into account. Another weakness is that tools based on pattern matching cannot detect vulnerabilities outside of the patterns identified.

Data flow analysis is an approach that was found to perform better than pattern matching in terms of precision and recall. This was confirmed across multiple studies. It lets the analysis take the information flow within the program into account. Many tools used both pattern matching and data flow analysis. It is, however, limited by the resources available. Approaches based on dynamic analysis were identified, but the main challenge with these was that there is no easy way to link the source code where the vulnerability occurred to the runtime analysis results. This led to the conclusion that dynamic analysis is not as popular for automatic fixing as it is for detection.

Classical auto fixing approaches were also identified. Many tools use code search to create patches from a database of similar code to the bug identified. This lets the knowledge of common bugs be stored in a knowledge database. These patches were made on the abstract syntax tree representation of the program. This performed well if the necessary code construct was present in the database. This is also the main drawback since the approach is limited to only the knowledge stored in the database. Many existing tools also use genetic programming to search over the space of replacements to generate a possible patch. This was found to adapt well to unseen situations, but necessitated a good test set to perform evaluations, and was found to be slow in practice. Machine learning techniques were also identified and performed well in the limited tests identified, but there does not currently exist any good data set for training models

While analyzing the strengths and weaknesses of different approaches it became evident that some vulnerabilities are covered far better than others by auto fix tools. SQL injections and XSS injections were found to have been studied in great detail, and a need for focusing on other vulnerabilities was identified. MITRE [86] released a list of their 25 most dangerous vulnerabilities of 2019, which can be seen in Table B.4. The irrelevant vulnerabilities for Java are marked in grey. As can be seen, the remaining vulnerabilities are injection related, or related to information exposure. Vulnerabilities that do not fall under the injection category have not been studied in great detail and are not covered by existing tools, and are of interest to look into.

CWE-ID	Description
119	ClassImproper Restriction of Operations within the Bounds of a Memory Buffer
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
20	Improper Input Validation
200	Information Exposure
125	Out-of-bounds Read
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
416	Use After Free
190	Integer Overflow or Wraparound
352	Cross-Site Request Forgery (CSRF)
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
787	Out-of-bounds Write
287	Improper Authentication
476	NULL Pointer Dereference
732	Incorrect Permission Assignment for Critical Resource
434	Unrestricted Upload of File with Dangerous Type
611	Improper Restriction of XML External Entity Reference
94	Improper Control of Generation of Code ('Code Injection')
798	Use of Hard-coded Credentials
400	Uncontrolled Resource Consumption
772	Missing Release of Resource after Effective Lifetime
426	Untrusted Search Path
502	Deserialization of Untrusted Data
269	Improper Privilege Management
295	Improper Certificate Validation

Table B.4: Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors [86]

B.6 Limitations of Existing Tools

In this section, a summary of the limitations found with the existing tools and approaches identified through the literature review will be presented. First, an overview of the tools that were found to be missing will be given. Then, the main results of empirically evaluating the existing tools will be presented. Finally, this will be used to motivate the necessity for extending the detection capabilities of these tools and the importance of implementing quick fixes for software security vulnerabilities.

The existing tools identified for Java were ASIDE, ESVD, FindSecBugs, JoanAudit, LAPSE+, and Snyk. All were found to use pattern matching in their analysis, with ESVD, JoanAudit, and FindSecBugs using data flow analysis as well making these more capable. FindSecBugs covered the most vulnerabilities of the identified tools with 134 different bug patterns being detected, with others focusing primarily on injection related vulnerabilities, except for Snyk which focuses on outdated dependencies. ASIDE, ESVD, and JoanAudit provided automatic fixes for security vulnerabilities, but with JoanAudit being closed source and not being available as an Eclipse plugin, we decided not to focus any further on JoanAudit leaving only ASIDE and ESVD. The fixes provided by these tools were found to be too generic to be useful. They also focused primarily on sanitization methods which can be used to mitigate injection attacks.

We also empirically evaluated the tools to find the tool that performs the best to get a good starting point for implementing security fixes as part of the master thesis. The questions to be answered are shown in section B.2. The following paragraph will briefly summarize these results. A more exhaustive discussion of these results can be found in [89].

ASIDE, ESVD, Snyk, and FindSecBugs supported the latest version of Eclipse, with ASIDE and ESVD being more complicated to set up compared to FindSecBugs and Snyk. This led to the conclusion that FindSecBugs and Snyk are the plugins that best support the newest version of Eclipse. These were also the tools found to have ongoing active development. ASIDE, JoanAudit, and ESVD were the plugins found that supported automatic fixes. However, ASIDE and ESVD's fixes were not contextually aware showing all fixes for each vulnerability. JoanAudit is closed source making it impossible to inspect the quick fixes. There were no quick fixes made for FindSecBugs, however, SpotBugs which FindSecBugs is an extension to has an auto fix framework built-in making it possible to add quick fixes for FindSecBugs.

Based on these findings, we found that there is a need for a tool that can provide specific fixes for identified vulnerabilities focusing not only on injection vulnerabilities. FindSecBugs was found to be the most promising candidate, making this the tool we chose to extend for this master thesis. This is further detailed in [89, Chapter 8].

B.7 Limitations of Existing Test Beds

In this section, the results from analyzing the state of the art in evaluating auto fix tools and methods are summarized. Some of the limitations of the test beds that have been used will be highlighted.

First and foremost, a big difference in the way different tools have been evaluated was found. Many evaluations were found to be done using poorly explained techniques making them hard to reproduce and making it difficult to compare the results with other evaluations. The evaluation done by the author themselves were also found to claim better performance compared to external evaluations. Different evaluation methods were discovered to use different metrics, and there is a need for a standardization of these using e.g. precision and recall. Many tools were found to use poor metrics as shown in Table A.1, making comparing the tools difficult. In order to get enough metrics a proper and well-established test bed is necessary due to the difficulty of obtaining good metrics from

real-life applications due to not knowing the total number of vulnerabilities and similar. Most testbeds were found to be implemented as a collection of vulnerable functions or classes, simplifying the process of extending the test bed, and making it easier to classify the results due to the numbers being clustered after for instance CWE code or other nomenclature schemes.

ManyBugs [78], which is a test bed designed for benchmarking automatic fixing of errors in C code was found. It uses an associated test suite that checks the functionality of the test bed [70]. However, this test suite cannot be used for evaluating the auto fix performance of security vulnerabilities due to focusing on classical bugs. Of all the different test beds we identified, No test bed that was able to validate fixes related to security vulnerabilities was found. This led to the conclusion that it is currently difficult for researchers to know if their quick fixes are effective and if they preserve the functionality of the system without any side effects. For more information on this see [89].

In general, existing test beds were found to focus on testing the detection of software security tools, but not how well they are able to fix the vulnerable code. This is likely due to the focus of many tools being on detection leading to a test suite for detection having been more important. Furthermore, although well-established test suites for testing the detection of tools are available, many researchers were still found to create their own test suite or testing on real-life applications making comparisons between different tools difficult or impossible. As we see a shift towards focusing on automatic fixes a test suite for verifying and validating these will become necessary.

Appendix C

Test Bed Use Cases

The use cases shown in Table C.1 show the intended use of the test bed for evaluation of auto fixes, for researchers and developers. These use cases do not explain the functionality of the vulnerable code, but rather how evaluating tools using the test bed should work.

ID and Name	UC-1 Validate fixes
Created by: Andreas Berger	**Date Created:** 2019-11-08
Primary Actor: Developer	**Secondary Actor: N/A**
Description	A developer checks the correctness of automatic fixes created by a tool he is evaluating. The system checks that the system has the exact same functionality as before the fix.
Trigger	Developer runs the test suite.
Preconditions	PRE-1: Auto fix tool used to fix vulnerabilities in test bed.
Postconditions	POST-1: The terminal displays information about correctness of code after auto fixing.
Normal Flow	1. Developer runs an auto fix tool on the code in the test bed. 2. Developer runs the test suite created for the test bed. 3. The tests complete and all tests pass. 4. A message explaining that all tests pass is shown in the terminal.
Alternative Flow	1. Developer runs an auto fix tool on the code in the test bed. 2. Developer runs the test suite created for the test bed. 3. The tests complete and some tests fail. 4. A message listing failed test cases is shown in the terminal.
Exceptions	E1 No change has been made to the code base.
Goal(s)	Test that an auto fix tool preserves functionality and displays the results to the tester.

Table C.1: Use case for validating fixes using test bed

Appendix D

Test Case Flow Variants

In this chapter the flow variants used to create the test cases are detailed. First, the instance based flow variants are shown. Then the Juliet style flow variants are presented.

D.1 Instance Based Flow Variants

The instance based flow variants are shown in Table D.1. These test various ways of initializing an object and invoking methods on the object instance.

Test Case	**CWE-ID**	**OWASP Category**	**Flow Type**	**Description**
Case 1	CWE-611, CWE- 776	A4	Data	Factory initialized into local variable. Parser initialized into class field using the insecure factory. Then the insecure parser is used.
Case 2	CWE-611, CWE- 776	A4	Data	Factory initialized into class field. Parser initialized into local variable using factory. Then the insecure parser is used.
Case 3	CWE-611, CWE- 776	A4	Data	Factory initialized into class field. Parser initialized into class field using factory. Then the insecure parser is used.
Case 4	CWE-611, CWE- 776	A4	Data	Parser initialized into class field by using factory directly. Then the insecure parser is used.
Case 5	CWE-611, CWE- 776	A4	Data	Factory initialized into local variable. Parser initialized into local variable using factory. Then the insecure parser is used.

Case 6	CWE-611, CWE- 776	A4	Data	Parser initialized into local variable by using factory directly. Then the insecure parser is used.
Case 7	CWE-611, CWE- 776	A4	Flow	Factory initialized into a local variable. Then a parser is initialized into a local variable without making the factory secure. Then this insecure parser is used. Then another parser initialized into local variable after making factory secure. Then this secure parser is used.
Case 8	CWE-611, CWE- 776	A4	Flow	Factory initialized into local variable. Then parser initialized into local variable without making factory secure. Then this insecure parser is used. Then another parser initialized after making factory secure. Then this secure parser is used. Then another parser initialized after making factory insecure. Then this insecure parser is used.
Case 9	CWE-611, CWE- 776	A4	Flow	Factory initialized into local variable. Then parser initialized into local variable. The insecure parser is then used twice. Then another factory initialized into local variable and made secure. Then new parser initialized using this factory. The secure parser is then used twice.
Case 10	CWE-611, CWE- 776	A4	Flow	Factory initialized into local variable. Then parser initialized into local variable without making factory secure. The insecure parser is used twice. Then factory made secure and new parser initialized. Then old insecure parser is used, and new secure parser is used. Then factory made insecure and a third parser initialized. Then third insecure parser used, and second secure parser used.

Case 11	CWE-611, CWE- 776	A4	Flow	A new instance of the class *Bar* with a method titled the same as the secure method of the class instance being tested e.g. *setFeature* for *DocumentBuilder* and *SAXParser*, and *setProperty* for *XMLStreamReader*, *XMLEventReader*, and *FilteredReader*, taking the same parameters as the secure method for the factory of the parser, e.g. a string and a boolean for *setFeature* and a string and an object for *setProperty*. Then this method is invoked on *Bar*. Then a factory is initialized into a local variable, and a parser is initialized into a local variable. The insecure parser is then used.

Table D.1: Control and data flow test cases focusing on different ways of initializing an object and different ways of invoking methods on an object instance

D.2 Flow Variants in Juliet Test Suite

Various flow variants are tested by the Juliet Test Suite as described in [93]. These different flow variants have been reiterated in Table D.2.

Flow Variant	**Flow Type**	**Description**
01	None	Baseline – Simplest form of the flaw
02	Control	if(true) and if(false)
03	Control	if(5==5) and if(5!=5)
04	Control	if(PRIVATE_STATIC_FINAL_TRUE) and if(PRIVATE_STATIC_FINAL_FALSE)
05	Control	if(privateTrue) and if(privateFalse)
06	Control	if(PRIVATE_STATIC_FINAL_FIVE==5) and if(PRIVATE_STATIC_FINAL_FIVE!=5)
07	Control	if(privateFive==5) and if(privateFive!=5)
08	Control	if(privateReturnsTrue()) and if(privateReturnsFalse())
09	Control	if(IO.STATIC_FINAL_TRUE) and if(IO.STATIC_FINAL_FALSE)
10	Control	if(IO.staticTrue) and if(IO.staticFalse)
11	Control	if(IO.staticReturnsTrue()) and if(IO.staticReturnsFalse())

12	Control	if(IO.staticReturnsTrueOrFalse())
13	Control	if(IO.STATIC_FINAL_FIVE==5) and if(IO.STATIC_FINAL_FIVE!=5)
14	Control	if(IO.staticFive==5) and if(IO.staticFive!=5)
15	Control	switch(6) and switch(7)
16	Control	while(true)
17	Control	for loops
21	Control	Flow controlled by value of a private variable. All methods contained in one file.
22	Control	Flow controlled by value of a public static variable. Sink methods are in a separate file from sources.
31	Data	Data flow using a copy of data within the same method
41	Data	Data passed as an argument from one method to another in the same class
42	Data	Data returned from one method to another in the same class
45	Data	Data passed as a private class member variable from one method to another in the same class
51	Data	Data passed as an argument from one method to another in different classes in the same package
52	Data	Data passed as an argument from one method to another to another in three different classes in the same package
53	Data	Data passed as an argument from one method through two others to a fourth; all four methods are in different classes in the same package
54	Data	Data passed as an argument from one method through three others to a fifth; all five methods are in different classes in the same package
61	Data	Data returned from one method to another in different classes in the same package
66	Data	Data passed in an array from one method to another in different classes in the same package
67	Data	Data passed in a class from one method to another in different classes in the same package

68	Data	Data passed as a member variable in the "a" class from one method to another in different classes in the same package
71	Data	Data passed as an Object reference argument from one method to another in different classes in the same package
72	Data	Data passed in a Vector from one method to another in different classes in the same package
73	Data	Data passed in a Linked List from one method to another in different classes in the same package
74	Data	Data passed in a HashMap from one method to another in different classes in the same package
75	Data	Data passed in a serialized object from one method to another in different classes in the same package
81	Data	Data passed in an argument to an abstract method called via a reference

Table D.2: The different test case flow variants from Juliet Test Suite [93]

Appendix E

Implementation Details of Existing XML Vulnerability detectors in FindSecBugs

By using the bug patterns described in [6] and comparing these with the entries in *findbugs.xml* the XML detectors were found to be located in the main FindSecBugs module in a package called *xml*. The relevant excerpt from *findbugs.xml* is shown in Listing 20. Each detector is represented by the *Detector* tag which in turn specifies the class implementing the detector, and the bug patterns reported by this detector. As can be seen, the class *com.h3xstream.findsecbugs.xml.XxeDetector* is responsible for reporting vulnerabilities for the parsers *SAXParser*, *XMLReader*, and *DocumentBuilder*, the class *com.h3xstream.findsecbugs.xml.TransformerFactoryDetector* is responsible for reporting vulnerabilities for the *Transformer* parser, and the class *com.h3xstream.findsecbugs.xml.XmlStreamReaderDetector* is responsible for reporting vulnerabilities for the *XMLStreamReader* parser. As a side note, the class *com.h3xstream.findsecbugs.xml.XxeDetector* also reports bugs of the pattern *XXE_XPATH*, however, as shown in Listing 21, there already exists another xPath injection detector which reports the *XPATH_INJECTION bug pattern*. It is unclear why the class *com.h3xstream.findsecbugs.xpath.XPathInjectionDetector* is not used to handle the *XXE_XPATH* bug pattern as well. The focus of this analysis is on the detectors for XML parsers.

```
<Detector class="com.h3xstream.findsecbugs.xml.XxeDetector"
reports="XXE_SAXPARSER,XXE_XMLREADER,XXE_DOCUMENT,XXE_XPATH"/>
<Detector
class="com.h3xstream.findsecbugs.xml.TransformerFactoryDetector"
reports="XXE_DTD_TRANSFORM_FACTORY,XXE_XSLT_TRANSFORM_FACTORY"/>
<Detector
class="com.h3xstream.findsecbugs.xml.XmlStreamReaderDetector"
reports="XXE_XMLSTREAMREADER"/>
```

Listing 20: Excerpt from findbugs.xml showing the different detector classes for identifying XML related vulnerabilities and the bug patterns they report

```
<Detector
class="com.h3xstream.findsecbugs.xpath.XPathInjectionDetector"
reports="XPATH_INJECTION"/>
```

Listing 21: Excerpt from findbugs.xml showing the taint analysis based injection detector for detecting xPath injection vulnerabilities

The XxeDetector class

The *XxeDetector* class extends the *OpcodeStackDetector* base detector class from SpotBugs. This is a base class for detectors that want to scan the bytecode of a method and use a stack of opcodes [55]. The detector uses the *sawOpcode* method from the *OpcodeStackDetector* class to look for the opcodes *invokevirtual* and *invokeinterface*. If either of these opcodes are found, the detector then gets the fullClassName of the operand, which corresponds to the package the parser class resides in. The different fullClassNames and the corresponding parser are shown in Table E.1. The detector also obtains a string representation of the method name of the opcode, and the signature of the operand used by the opcode. Next, the detector checks if the fullClassname matches any of the fullClassNames for the parsers *SAXParser*, *XMLReader*, or *DocumentBuilder*. If it does, the detector checks if the method invoked on the parser is called *parse*, indicating that the parser has been used to parse XML.

Parser/Factory	fullClassName
SAXParser	javax/xml/parsers/SAXParser
XMLReader	org/xml/sax/XMLReader
DocumentBuilder	javax/xml/parsers/DocumentBuilder
TransformerFactory	javax/xml/transform/TransformerFactory
TransformerFactory	javax/xml/transform/sax/SAXTransformerFactory
XMLInputFactory	javax/xml/stream/XMLInputFactory

Table E.1: The different parsers and their corresponding fullClassName as identified by FindSecBugs

Then, the detector checks if the class subclasses the *java.security.PrivilegedExceptionAction* interface. If it does, the detector assumes that there are no vulnerabilities. Then the detector obtains a control flow graph for the method being analyzed. The detector checks whether the methods *setEntityResolver*, *setFeature*, *setXIncludeAware*, or *setExpandEntityReferences* have been called. If *setEntityResolver* is called the detector assumes there are no XML vulnerabilities. If *setFeature* is called, the detector checks if it has been called with different attributes. If these attributes are *http://apache.org/xml/features/disallow-doctype-decl* or *http://javax.xml.XMLConstants/feature/secure-processing* the detector immediately assumes there are no vulnerabilities. If the attributes are either *http://xml.org/sax/features/external-general-entities* or *http://xml.org/sax/features/external-parameter-entities*,

the detector stores that this was found as the boolean values *hasFeatureGeneralEntities* and *hasFeatureExternalEntities*. The detector only checks the first of the attributes for the *setFeature* method, not checking the second attribute of this method to check if the attribute has been set correctly. For many of the parameters in subsection 2.7.1 the second parameter indicates whether to enable or disable the feature. Thus, not checking the second parameter means that the vulnerable feature may be enabled as opposed to disabled. Similarly, if the *setXIncludeAware* method has been called with the value *false* this is stored as the boolean *hasSetXIncludeAware*, and if the *setXIncludeAware* method has been called with the value *false* this is stored as the boolean *hasExpandEntityReferences*. The detector then checks if the booleans *hasFeatureExternalEntities*, *hasFeatureGeneralEntities*, *hasSetXIncludeAware*, and *hasExpandEntityReferences* are all true. Since the *setExpandEntityReferences* method is only valid in the case that *DocumentBuilder* is used, the *hasExpandEntityReferences* is only set if DocumentBuilder is used. Therefore, this is initialized to true initially if *DocumentBuilder* is not the current parser being looked at. If all these booleans are true, the detector assumes there are no vulnerabilities present. Otherwise, the detector uses the fullClassName of the parser being looked at to report the vulnerability for the parser using the parser specific bug pattern.

The TransformerFactoryDetector class

The *TransformerFactoryDetector* class is implemented in a similar fashion to the *XxeDetector* class. It extends the *OpcodeStackDetector* base class from SpotBugs, and uses the *sawOpcode* method from the *OpcodeStackDetector* class to look for the opcode *invokestatic*. If this is found, the fullClassName of the operand and a string representation of the method name of the operand is obtained. The fullClassName is compared to the one for the *TransformerFactory* as shown in Table E.1. Note that there are two fullClassNames for the *TransformerFactory* since this can be initialized either by using a *TransformerFactory* or a *SAXTransformerFactory* [33]. The detector also checks if the method invoked is named *newInstance*. This means that for the *Transformer* parser the existing detector reports the initialization of the parser itself as vulnerable even if the parser has not been used by using the *transform* method of the parser.

The detector then obtains a control flow graph for the method being analyzed and checks whether the *setAttribute* method or the *setFeature* method has been called somewhere in the method. If *setAttribute* has been called, the detector gets the previous bytecode instruction that matches the *LDC* bytecode instruction, and the second last bytecode instruction which matches the *LDC* bytecode instruction, and obtains these constant value. If the 2nd last instruction loaded the constant value *http://javax.xml.XMLConstants/property/accessExternalDTD*, the detector checks if the last instruction before the method call equals *""* (an empty string) and sets the boolean value of *hasFeatureDTD* to true meaning the *Transformer* parser is considered to not be vulnerable to DTD attacks. If the second last instruction loaded the constant value *http://javax.xml.XMLConstants/property/accessExternalStylesheet* and the last instruction before the method call equals *""* (an empty string), the detector sets the boolean value of *hasFeatureStylesheet* to true meaning the *Transformer* parser is considered to not be vulnerable to XSLT attacks. Otherwise, if the *setFeature* method is called, and the last bytecode instruction before the method invocation matches the ICONST bytecode instruction, and the second last bytecode in-

struction matches the LDC bytecode instruction, the detector obtains the value of the integer constant and checks if the value is *1*, and that the value of the constant equals *http://javax.xml.XMLConstants/feature/secure-processing*. If they do, then the detector sets the boolean value of *hasSecureProcessing* to true meaning that the *Transformer* parser is considered to not be vulnerable to either XSLT or DTD attacks. If *hasSecureProcessing* was set to true, the detector declares the use of the Transformer parser to be secure. Otherwise, if hasFeatureDTD is false, the detector reports the *XXE_DTD_TRANSFORM_FACTORY_TYPE* bug pattern. If *hasFeatureStylesheet* is false, the detector reports the *XXE_XSLT_TRANSFORM_FACTORY_TYPE* bug pattern. This means that if the *Transformer* parser is vulnerable to both XSLT and DTD attacks, the detector will report two bugs for the parser, which agrees with the testing performed in subsection 6.2.2.

The XMLStreamReaderDetector class

The *XMLStreamReaderDetector* class is implemented similarly to both the *TransformerFactoryDetector* class and the *XxeDetector* class. This detector also extends the *OpcodeStackDetector* class and uses the *sawOpcode* method to look for the opcode *invokevirtual*. The fullClassname and a string representation of the method name of the operand is obtained. The detector then checks if the fullClassName matches the one for XMLInputFactory as shown in Table E.1 and if the method invoked is either *createXMLStreamReader*, *createXMLEventReader*, or *createFilteredReader*. This means that the detector is capable of detecting bugs for the *XMLEventReader* parser and the *FilteredReader* parser in addition to the *XMLStreamReader* parser which is not explicitly stated in [6]. Upon discovering this, test cases for *FilteredReader* and *XMLEventReader* were created using the instance based test cases as well as test cases based on the test cases from Juliet Test Suite as described in subsection 6.2.2.

The detector then obtains a control flow graph for the method and checks if the *setProperty* method has been called. If it has been called, the detector gets the value of the previous LDC bytecode instruction. If this is equal to either *javax.xml.stream.supportDTD* or *javax.xml.stream.isSupportingExternalEntities*, the detector obtains the second last bytecode instruction and checks if it is a wrapped boolean invocation instruction of the class *java.lang.boolean* calling the method *valueOf*. If it is, the detector gets the last bytecode instruction and checks if it is an *iconst* instruction, and if it is, gets the value of this and checks if it is 0. If it is, then the detector declares the parser secure. The detector also checks if the second last bytecode instruction is a *getstatic* call with the class type *java.lang.boolean* and the field name is *FALSE*. If it is, the detector declares the parser secure. Otherwise, the parser use is reported as vulnerable. This special consideration for different ways of specifying a boolean value is necessary because the *setProperty* method takes as its second argument a parameter of the type *Object* instead of a boolean primitive [35].

Appendix F

Implementation of Instruction based Data Flow Analysis

The data flow analysis framework in SpotBugs, as explained in section 2.10, was used to implement an analysis method based on instruction based data flow analysis by extending the *AbstractDataflowAnalysis* class from SpotBugs. The data flow analysis was implemented in the SpotBugs data flow framework according to the description in [57]. In the following paragraphs, an in-depth description of this implementation will be detailed.

First, an *EngineRegistrar* class implementing the *IAnalysisEngineRegistrar* interface from SpotBugs was created and the method *registerAnalysisEngines* was created as specified by the interface. Then, a new factory class called *CallListDataflowFactory* extending the *AnalysisFactory* from SpotBugs was created. The *analyze* method from the superclass was overridden to create a new *CallListDataflow* analysis object and to perform the data flow analysis. The *analyze* method obtained the data flow analysis result from the existing *ConstantDataFlow* class preexisting in SpotBugs. Then the *CallListAnalysis* class was created which extended the *AbstractDataflowAnalysis* class from SpotBugs which was used to perform the data flow analysis.

The *CallListAnalysis* class first builds a map from InstructionHandles to a *Call* class by using the control flow graph of the method to be analyzed. The *Call* class was initialized with the class name, name of the instruction, and the signature of the instruction. Then the necessary methods initEntryFact, isForwards, getBlockOrder, makeFactTop, isTop, createFact, same, meetInto, copy, transferInstruction, and isFactValid methods were implemented.

Since the data flow analysis framework in SpotBugs requires a custom class representing the data flow values to be implemented, a *CallList* class representing these was created, keeping track of the data flow values. The *initEntryFact* method called the clear method in the *CallList* class setting the boolean values *isTop* and *isBottom* to false and clearing the list of calls for this data flow value. Similarly, the *setTop* method set the *isTop* to true, *isBottom* to false, and cleared the *CallList*. The *createFact* method created a new *CallList* instance. The *same* method used the *equals* method in Java to check if the two CallLists

were equal. The *meetInto* method merged the results from two different data flow values. If both callists were bottom, then the *isBottom* boolean of the result was set to true. If one of the *CallLists* to be merged had *isTop* true, then the result was copied from the other *CallList* into this one. Otherwise, the values from the first CallList was added to the result as long as they differed from the values in the other CallList. This was then copied to the other result as the result of the *meetInto* function. The *copy* function simply copied the values from the source to the destination *CallList*. The *transferInstruction* method first used the *callMap* to get the *Call* corresponding to the instruction to be looked at. Next, if the opcode of this instruction was either invokevirtual or invokeinterface, the constant-DataFlow result was used to obtain the constant values of the method invocation. The *Call* was then added to the *CallList* data flow value. Finally, the *isFactValid* method simply checked if the *CallList* had either *isTop* or *isBottom* set to true.

Appendix G

Implementation of XML Vulnerability Detectors using Instance Tracking

In this chapter, the implementation details of the instance tracking approach will be given. First, the generalized instance tracker class will be explained. Then each of the detectors implemented to identify XXE vulnerabilities using the generalized instance tracker class will be explained. A more top-level description can be found in subsection 6.3.3.

G.1 Implementation of Instance Tracking Approach

The *OpcodeStackDetector* base class, which is explained in more detail in section 2.10, was extended to create a new base detector class in FindSecBugs which was called *InstanceTrackDetector*. This class first takes a list of *TrackedObject* class instances which contains information about which initialization instruction the detector is looking for, which calls the detector is tracking, and what return values the detector is tracking. The *TrackedCall* and *TrackedObject* objects as well as the top-level project structure are based on a closed pull request to the FindSecBugs GitHub repository from 2016 [79] targeting detection of insecure cookies. The *TrackedCall* class stores information about what values the call should have been invoked with, the stack parameter indexes of these calls, the instruction this call corresponds to, and the bug that occurs when this call is not found. The *MultipleTrackedCalls* class is a wrapper around *TrackedCall* objects which is used to track multiple calls needed to make an instance secure. Additionally, a *MultipleTrackedCall* has a list for each *TrackedCall* which is used to mark a call as vulnerable again. Both *MultipleTrackedCall* and *TrackedCall* instances are added to the *TrackedObject* instance. The *TrackedReturnValue* class is used to store the results of method invocations that the detector is looking for, as well as which return value the bug should be reported on allowing the bug to be reported on either of the return values. Instances of this class is also added to

the *TrackedObject* instance. The detector then overrides the *sawOpcode* method from the *OpcodeStackDetector* to perform the analysis.

First, the detector checks if the opcode corresponds to putfield or putstatic, which means that a field was initialized using this operand. Using the *getXFieldOperand* method from the *OpcodeStackDetector* class allows one to obtain this field. After checking that the field pushed to the stack is indeed going to be stored in a registry, the field is then put into a map between fields and the line they were stored on named *fieldUse*. Next, the detector looks for the opcodes invokevirtual, invokeinterface, invokestatic, and invokespecial since it only needs to consider invocations. If the class is subclassing the *java.security.PrivilegedExceptionAction* interface, it assumes that the correct properties have been applied to the sandbox and declares the class secure.

Then, the detector obtains the class name of, and the method invoked on the operand which will be denoted fullOperand for the invocation opcode. If this is one of the objects the detector is tracking, the source line of the creation of the object is stored, alongside the method it occurred in and the class it occurred in by initializing a new *TrackedObjectInstance* and storing it on the *TrackedObject* instance that is looking for the particular initialization instruction. The different bugs stored on the *TrackedCall* instances and on the *MultipleTrackedCalls* instances stored in the *TrackedObject* instance is added to the *TrackedObjectInstance* instance. Then the source line of the object initialized is added to a map between source line locations, and a list of the bugs at this location in the *TrackedObjectInstance* instance.

The detector then continues looking for the opcodes invokevirtual, invokeinterface, invokestatic, and invokespecial adding new *TrackedObjectInstance* instances to the *TrackedObject* instances the detector is looking for. If an object initialization call is not found, the detector then first checks if the opcode is either invokevirtual or invokeinterface since it does not make sense to track instances of static invocations since they are not instances. Next, the instance the opcode was invoked on is obtained from the stack alongside source line where the instance was created, the XMethod [55] of the return value of the instance, the XField [55] value of the instance, and the signature. The initialization location of the current return value of the invocation opcode being called is also obtained. If the return value of the instance is null, the source line of the creation of the object is obtained from the *fieldUse* map instead. The detector then compares the return value, the signature, and the fullOperand of the return value to the values of the *TrackedReturnValue* instances stored on the *TrackedObject* instance under consideration. If these match, then the initialization location of the current opcode is stored on the instance since it is a return value that is being tracked. Additionally, if the bug should be reported on this line, the current bugs present on this line are stored alongside the source line.

The detector then checks if any of the *TrackedObjectInstances* correspond to the object creation location of the instance obtained from the stack. If it does, then all the *TrackedCalls* calls that match the invocation instruction of the fullOperand is checked to see if the call has been found. This is done by obtaining the list of expected values and the stack parameter indexes these should have been present on from the *TrackedCall* instance, and comparing these to the values on the operand stack. *Boolean.valueOf*, *Boolean.FALSE*, and *Boolean.TRUE* values are converted to the integer representation to allow these to be specified as their integer representation in the *TrackedCall* class, which means that only

one *TrackedCall* instance has to be created for all of these. Other stack values, for instance, stack values that are subclassing an interface, are also considered. If all of the stack values necessary to make this call secure have been found, a check is made to see if this call is vulnerable if called, or secure if called. If it is vulnerable, then the vulnerabilities of this call is added to the instance. Otherwise, the vulnerabilities this call is mitigating is removed from the instance. This means that any additional *TrackedReturn* values of this instance will either be declared secure from the bugs this call was necessary to mitigate or vulnerable from the bugs this call added.

Similarly, all the *TrackedCall* calls of the *MultipleTrackedCalls* instances are gone through. First, the *TrackedCall* is checked. If the stack parameters are equal to the values the *TrackedCall* is looking for, then this call is added to the *TrackedObjectInstance* instance. If they aren't, then the list of *TrackedCalls* that mark the *TrackedCall* as vulnerable is checked. If one such *TrackedCall* is found, then the *TrackedCall* this call makes vulnerable again is removed from the list of found tracked calls stored on the *TrackedObjectInstance* and the bug is added to the instance again. Then all the *TrackedCalls* stored on the *TrackedObjectInstance* are compared to all the *TrackedCall* instances the *MultipleTrackedCall* is looking for. If all the calls have been found, then the bug the *MultipleTrackedCall* is reporting is removed from the instance.

Lastly, the *report* method from *OpcodeStackDetector* is used to report the bugs. All the *TrackedObjectInstance* instances stored in the *TrackedObject* instance is gone through to obtain the vulnerable lines for the *TrackedObjectInstance* as well as all the vulnerabilities for this line which the found bugs are reported on. If no *TrackedReturnValues* are tracked, then any vulnerabilities found are reported directly on the instance, instead of subsequent calls done on the instance.

G.2 The BetterDocumentBuilderDetector class

To implement the new detector for the *DocumentBuilder* parser using the *InstanceTrackDetector* class described above, the bytecode of the different test cases shown in Table D.1 as well as the test cases present in FindSecBugs as explained in section 2.10 was inspected. This was done in order to find the initialization calls necessary to supply the *InstanceTrackDetector* with to track the instances, the return value of and the fullOperand values of subsequent calls necessary to create a parser and parse using the factory, as well as the invocation instruction for the tracked calls, the parameters pushed to the operand stack and the index of these. The invocation instruction that needs to be tracked for *DocumentBuilderFactory* is *javax/xml/parsers/DocumentBuilderFactory.newInstance*.

The different return values of performing the invocation calls necessary to create a new *DocumentBuilder* parser and to parse the XML that needed to be tracked are shown in Table G.1. The signature of calling the method the return value looks for was found by inspecting the constant pool of the bytecode and comparing it with the official documentation [29]. These are the necessary calls that need to be made to go from creating a new instance of the *DocumentBuilderFactory* factory to parsing using the *DocumentBuilder* parser as described in [29].

Class name	signature	Full Operand	Report bug on this line
javax/xml/parsers/ DocumentBuilder Factory	Ljavax/xml/ parsers/ DocumentBuilder Factory;	javax/xml/parsers/ DocumentBuilder Factory .newDocumentBuilder	✗
javax/xml/parsers/ DocumentBuilder Factory	Ljavax/xml/ parsers/ DocumentBuilder;	javax/xml/parsers/ DocumentBuilder .parse	✓

Table G.1: Result of inspecting the bytecode and the documentation for which return values to track for the *DocumentBuilder* parser

Both Oracle and OWASP specify different attributes that can be used to make the parser secure as detailed in subsection 2.7.1. These attributes have been split up into attributes where one attribute is necessary to consider the parser secure and attributes where multiple attributes are necessary to consider the parser secure. The first are shown in Table G.2, while the latter is shown in Table G.3. For each attribute specified by OWASP and Oracle, the corresponding bytecode instructions were inspected to find the invocation instructions, stack parameters, and stack indexes. Additionally, the bug patterns reported for the calls are shown as well as whether the call is considered secure if found, or insecure if not found. For the multiple tracked calls, the insecure call that makes the secure call vulnerable is shown on the row under each secure call.

Invocation instruction	Parameters	Stack indexes	Bug pattern	When to report bug
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *1*	*1*, *0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://apache.org/ xml/features/ disallow-doctype-decl*, *1*	*1*, *0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder .setEntityResolver	*org.xml.sax. EntityResolver*	*0*	XXE _DOCUMENT	Report bug when not called

javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *0*	*1*, *0*	XXE _DOCUMENT	Report bug when called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://apache.org/ xml/features/ disallow-doctype-decl*, *0*	*1*, *0*	XXE _DOCUMENT	Report bug when called

Table G.2: Different calls considered vulnerable if not called as well as calls considered vulnerable if called for the *DocumentBuilder* as detailed in subsection 2.7.1. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug was found by inspecting the bytecode and consulting the documentation for the parser

Invocation instruction	**Parameters**	**Stack indexes**	**Bug pattern**	**When to report bug**
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://xml.org/sax/ features/external-general-entities*, *0*	*1*, *0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://xml.org/sax/ features/external-general-entities*, *1*	*1*, *0*	XXE _DOCUMENT	Report bug when called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://xml.org/sax/ features/external-parameter-entities*, 0	*1*, *0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://xml.org/sax/ features/external-parameter-entities*, 1	*1*, *0*	XXE _DOCUMENT	Report bug when called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://apache.org/ xml/features/ nonvalidating/ load-external-dtd*, 0	*1*, *0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder Factory .setFeature	*http://apache.org/ xml/features/ nonvalidating/ load-external-dtd*, 1	*1*, *0*	XXE _DOCUMENT	Report bug when called
javax/xml/ parsers/ DocumentBuilder Factory .setXIncludeAware	*0*	*0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder Factory .setXIncludeAware	*1*	*0*	XXE _DOCUMENT	Report bug when called

javax/xml/ parsers/ DocumentBuilder Factory .setExpandEntity References	*0*	*0*	XXE _DOCUMENT	Report bug when not called
javax/xml/ parsers/ DocumentBuilder Factory .setExpandEntity References	*1*	*0*	XXE _DOCUMENT	Report bug when called

Table G.3: Different calls that all must be called to mitigate the XXE vulnerability for the *DocumentBuilder* parser as detailed in subsection 2.7.1. The insecure calls are shown under the secure one they make vulnerable again if called. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug found by inspecting the bytecode and consulting the documentation for the parser

G.3 The BetterSAXParserDetector class

The new detector for the *SAXParser* parser was implemented in a similar fashion to the new detector for the *DocumentBuilder* parser described above. The bytecode of the test cases described in Table D.1 and from FindSecBugs as detailed in section 2.10 was inspected. The initialization call necessary to supply the *InstanceTrackDetector* with to track the instances was *javax/xml/parsers/SAXParserFactory.newInstance*.

The different return values of performing the invocation calls necessary to create a new *SAXParser* parser and parse the XML that needed to be tracked are shown in Table G.4. These are the necessary calls that need to be made to go from creating a new instance of the *SAXParserFactory* factory to using the *SAXParser* parser as described in [31]. The two last rows describe the return values from obtaining the underlying *XMLReader* parser of the *SAXParser* parser, which allows one to set a custom entity resolver which is considered secure according to Oracle and OWASP as shown in subsection 2.7.1.

Class name	**signature**	**Full Operand**	**Report bug on this line**
javax/xml/parsers/ SAXParserFactory	Ljavax/xml/ parsers/ SAXParserFactory;	javax/xml/parsers/ SAXParserFactory .newSAXParser	✗
javax/xml/parsers/ SAXParserFactory	Ljavax/xml/ parsers/ SAXParser;	javax/xml/parsers/ SAXParser.parse	✓

javax/xml/parsers/ SAXParserFactory	Ljavax/xml/ parsers/ SAXParser;	javax/xml/parsers/ SAXParser .getXMLReader	✗
javax/xml/parsers/ SAXParser	Lorg/xml/sax/ XMLReader;	org/xml/sax/ XMLReader.parse	✓

Table G.4: Result of inspecting the bytecode and the documentation for which return values to track for the *SAXParser* parser

Similar to the description for the *DocumentBuilder* parser, both Oracle and OWASP specify different attributes that can be used to make the parser secure as detailed in subsection 2.7.1. The attributes where one attribute is necessary is shown in Table G.5. These attributes can also be used to make the parser insecure, which is also shown in the table. The attributes where multiple attributes are necessary to make the parser secure are shown in Table G.6. For the multiple tracked calls, the insecure call that makes the secure call vulnerable is shown on the row under each secure call.

Invocation instruction	**Parameters**	**Stack indexes**	**Bug pattern**	**When to report bug**
javax/xml/ parsers/ SAXParserFactory .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *1*	*1*, *0*	XXE _SAXPARSER	Report bug when not called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://apache.org/ xml/features/ disallow-doctype-decl*, *1*	*1*, *0*	XXE _SAXPARSER	Report bug when not called
org/xml/sax/ XMLReader .setEntityResolver	*org.xml.sax .EntityResolver*	*0*	XXE _SAXPARSER	Report bug when not called
javax/xml/ parsers/ SAXParserFactory .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *0*	*1*, *0*	XXE _SAXPARSER	Report bug when called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://apache.org/ xml/features/ disallow-doctype-decl*, *0*	*1*, *0*	XXE _SAXPARSER	Report bug when called

Table G.5: Different calls considered vulnerable if not called as well as calls considered vulnerable if called for the *SAXParser* as detailed in subsection 2.7.1. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug was found by inspecting the bytecode and consulting the documentation for the parser

Invocation instruction	**Parameters**	**Stack indexes**	**Bug pattern**	**When to report bug**
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://xml.org/sax/ features/external-general-entities*, *0*	*1*, *0*	XXE _SAXPARSER	Report bug when not called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://xml.org/sax/ features/external-general-entities*, *1*	*1*, *0*	XXE _SAXPARSER	Report bug when called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://xml.org/sax/ features/external-parameter-entities*, *0*	*1*, *0*	XXE _SAXPARSER	Report bug when not called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://xml.org/sax/ features/external-parameter-entities*, *1*	*1*, *0*	XXE _SAXPARSER	Report bug when called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://apache.org/ xml/features/ nonvalidating/ load-external-dtd*, *0*	*1*, *0*	XXE _SAXPARSER	Report bug when no called
javax/xml/ parsers/ SAXParserFactory .setFeature	*http://apache.org/ xml/features/ nonvalidating/ load-external-dtd*, *1*	*1*, *0*	XXE _SAXPARSER	Report bug when called
javax/xml/ parsers/ SAXParserFactory .setXIncludeAware	*0*	*0*	XXE _SAXPARSER	Report bug when not called
javax/xml/ parsers/ SAXParserFactory .setXIncludeAware	*1*	*0*	XXE _SAXPARSER	Report bug when called

Table G.6: Different calls that all must be called to mitigate the XXE vulnerability for the *SAXParser* parser as detailed in subsection 2.7.1. The insecure calls are shown under the secure one they make vulnerable again if called. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug found by inspecting the bytecode and consulting the documentation for the parser

G.4 The BetterXmlStreamReaderDetector class

The new detector for the *XMLStreamReader* parser was implemented similarly to the *DocumentBuilder* parser described above. The bytecode of the test cases described in Table D.1 and from FindSecBugs as detailed in section 2.10 was inspected. The initialization call necessary to supply the *InstanceTrackDetector* with to track the instances were *javax/xml/stream/XMLInputFactory.newFactory* and *javax/xml/stream/XMLInputFactory.newInstance*.

The different return values of performing the invocation calls necessary to create a new *XMLStreamReader*, *XMLEventReader*, and *FilteredReader* are shown in Table G.7. According to [35], the instance obtained from creating the respective parser using the *XMLInputFactory* is used directly to perform the parsing, hence there only being one tracked return value for each parser.

Class name	**signature**	**Full Operand**	**Report bug on this line**
javax/xml/stream/ XMLInputFactory	Ljavax/ xml/stream/ XMLInputFactory;	javax/xml/stream/ XMLInputFactory .createXMLStream Reader	✓
javax/xml/stream/ XMLInputFactory	Ljavax/ xml/stream/ XMLInputFactory;	javax/xml/stream/ XMLInputFactory .createXMLEvent Reader	✓
javax/xml/stream/ XMLInputFactory	Ljavax/ xml/stream/ XMLInputFactory;	javax/xml/stream/ XMLInputFactory .createFiltered Reader	✓

Table G.7: Result of inspecting the bytecode and the documentation for which return values to track for the *XMLStreamReader*, the *XMLEventReader* parser, and the *FilteredReader* parser

Both Oracle and OWASP describe different attributes that can be used to make the parser secure as shown in subsection 2.7.1. For *XMLStreamReader*, *XMLEventReader*, and *FilteredReader*, only singular calls are required for the parser to be considered secure. These calls are shown in Table G.8. These attributes can also be used to make the parser insecure, which is also shown in the table.

Invocation instruction	**Parameters**	**Stack indexes**	**Bug pattern**	**When to report bug**
javax/xml/ stream/ XMLInputFactory .setProperty	*XMLInputFactory .IS_SUPPORTING _EXTERNAL _ENTITIES*, *0*	*1*, *0*	XXE _XML STREAM READER	Report bug when not called

javax/xml/ stream/ XMLInputFactory .setProperty	*XMLInputFactory .SUPPORT _DTD*, *0*	*1*, *0*	XXE _XML STREAM READER	Report bug when not called
javax/xml/ stream/ XMLInputFactory .setProperty	*XMLInputFactory .IS_SUPPORTING _EXTERNAL _ENTITIES*, *1*	*1*, *0*	XXE _XML STREAM READER	Report bug when called
javax/xml/ stream/ XMLInputFactory .setProperty	*XMLInputFactory .SUPPORT _DTD*, *1*	*1*, *0*	XXE _XML STREAM READER	Report bug when called

Table G.8: Different calls considered vulnerable if not called as well as calls considered vulnerable if called for the *XMLStreamReader*, *XMLEventReader*, and *FilteredReader* parsers as detailed in subsection 2.7.1. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug was found by inspecting the bytecode and consulting the documentation for the parser

G.5 The BetterTransformerFactoryDetector class

The new detector for the *Transformer* parser was implemented similarly to the new detector for the *DocumentBuilder* parser. The bytecode of the test cases described in Table D.1 and from FindSecBugs as detailed in section 2.10 was inspected. According to [33], both *TransformerFactory* factory and *SAXTransformerFactory* factory can be used to initialize a *Transformer* parser. The corresponding initialization calls needed to be tracked are *javax/xml/transform/TransformerFactory.newInstance* and *javax/xml/transform/sax/SAXTransformerFactory.newInstance*.

The different return values of performing the invocation calls necessary to create a new *transformer* parser and parse the XML that needed to be tracked are shown in Table G.9. These are the necessary calls that need to be made to go from creating a new instance of the *TransformerFactory* factory to using the *Transformer* parser as described in [33].

Class name	**signature**	**Full Operand**	**Report bug on this line**
javax/xml/transform/ TransformerFactory	Ljavax/xml/ transform/ Transformer Factory;	javax/xml/transform/ TransformerFactory .newTransformer	✗
javax/xml/transform/ TransformerFactory	Ljavax/xml/ transform/ Transformer;	javax/xml/transform/ Transformer.transform	✓

Table G.9: Result of inspecting the bytecode and the documentation for which return values to track for the *transformer* parser

Both Oracle and OWASP specify different attributes that can be used to make the parser secure as shown in subsection 2.7.1. Only singular attributes are necessary to make the parser secure, which are shown in Table G.10. These attributes can also be used to make the parser insecure, which is also shown in the table.

Invocation instruction	**Parameters**	**Stack indexes**	**Bug pattern**	**When to report bug**
javax/xml/ transform/ TransformerFactory .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *1*	*1*, *0*	XXE_DTD _TRANSFORM _FACTORY, XXE_XSLT _TRANSFORM _FACTORY	Report bug when not called
javax/xml/ transform/ TransformerFactory .setAttribute	*XMLConstants .ACCESS_EXTERNAL _DTD*, ""	*1*, *0*	XXE_DTD _TRANSFORM _FACTORY	Report bug when not called
javax/xml/ transform/ TransformerFactory .setAttribute	*XMLConstants .ACCESS_EXTERNAL _STYLESHEET*, ""	*1*, *0*	XXE_XSLT _TRANSFORM _FACTORY	Report bug when not called
javax/xml/ transform/ TransformerFactory .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *0*	*1*, *0*	XXE_DTD _TRANSFORM _FACTORY, XXE_XSLT _TRANSFORM _FACTORY	Report bug when called
javax/xml/ transform/ TransformerFactory .setAttribute	*XMLConstants .ACCESS_EXTERNAL _DTD*, all	*1*, *0*	XXE_DTD _TRANSFORM _FACTORY	Report bug when called
javax/xml/ transform/ TransformerFactory .setAttribute	*XMLConstants .ACCESS_EXTERNAL _STYLESHEET*, all	*1*, *0*	XXE_XSLT _TRANSFORM _FACTORY	Report bug when called

Table G.10: Different calls considered vulnerable if not called as well as calls considered vulnerable if called for the *transformer* as detailed in subsection 2.7.1. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug was found by inspecting the bytecode and consulting the documentation for the parser

G.6 The BetterXMLReaderDetector class

The new detector for the *XMLReader* parser was implemented similarly to the *DocumentBuilder* parser described above. The bytecode of the test cases described in Table D.1 and from FindSecBugs as detailed in section 2.10 was inspected. The initialization call necessary to supply the *InstanceTrackDetector* with to track the instances was *org/xml/sax/helpers/XMLReaderFactory.createXMLReader*.

The different return values of performing the invocation calls necessary to create a new *XMLReader* parser are shown in Table G.11. According to [36], the instance obtained from creating the respective parser using the *XMLReaderFactory* is used directly to perform the parsing, hence there only being one tracked return value for the parser.

Class name	**signature**	**Full Operand**	**Report bug on this line**
org/xml/sax/helpers/ XMLReaderFactory	Lorg/xml/sax/ XMLReader;	org/xml/sax/ XMLReader.parse	✓

Table G.11: Result of inspecting the bytecode and the documentation for which return values to track for the *XMLReader* parser

Both Oracle and OWASP describe different attributes that can be used to make the parser secure as shown in subsection 2.7.1. For *XMLReader* only singular calls are required for the parser to be considered secure. These calls are shown in Table G.12. These attributes can also be used to make the parser insecure, which is also shown in the table.

Invocation instruction	**Parameters**	**Stack indexes**	**Bug pattern**	**When to report bug**
org/xml/sax/ XMLReader .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *1*	*1*, *0*	XXE _XMLREADER	Report bug when not called
org/xml/sax/ XMLReader .setFeature	*http://apache.org/ xml/features/ disallow-doctype-decl*, *1*	*1*, *0*	XXE _XMLREADER	Report bug when not called
org/xml/sax/ XMLReader .setEntityResolver	*org.xml.sax. EntityResolver*	*0*	XXE _XMLREADER	Report bug when not called
org/xml/sax/ XMLReader .setFeature	*XMLConstants .FEATURE_SECURE _PROCESSING*, *0*	*1*, *0*	XXE _XMLREADER	Report bug when called

org/xml/sax/ XMLReader .setFeature	*http://apache.org/ xml/features/ disallow-doctype-decl*, *0*	*1*, *0*	XXE _XMLREADER	Report bug when called

Table G.12: Different calls considered vulnerable if not called, and calls considered vulnerable if called for the *XMLReader* parser as detailed in subsection 2.7.1. The parameters, the stack indexes for these parameters, the bug pattern to report, and when to report the bug was found by inspecting the bytecode and consulting the documentation for the parser

Appendix H

Implementation of Detection of Insecure Cookies

The base instance tracker detector described in subsection 6.3.3 is a generalized implementation of a detector for tracking method invocations on instances and denoting these instances as secure and insecure at different points in the method under consideration. The instance tracker supports tracking singular calls that either make the instance vulnerable or make the instance secure, as well as tracking multiple calls that together make an instance secure or insecure. The sections section G.2, section G.3, section G.4, section G.5, and section G.6 are similar precisely because the main detection mechanism has been extracted into a generalized base class. The only differences between these detectors are the initialization calls they're tracking, the calls they are tracking, both singular tracked calls and multiple tracked calls, the bug patterns they are reporting, and the return values they are tracking.

Since the main instance tracking mechanism has been generalized and extracted into a base class, the overhead of implementing additional detectors using the instance tracker is quite low. A new detector would only need to extend the base class and supply it with information about which initialization instructions to look for, which calls to look for, what bug patterns to report, and the return values to track if any. For instance, for cookies, it is recommended to set the *setSecure* attribute to true as per the recommendations by OWASP [126]. An example of how a cookie is initialized and the *setSecure* attribute is set is shown in Listing 22 with the corresponding bytecode shown in Listing 23. To detect when the *setSecure* call has not been called, the base instance tracking detector is supplied with the initialization call *javax/servlet/http/Cookie.<init>*, which looks for the call *javax/servlet/http/Cookie.setSecure* which should have been called with the stack parameter *1* residing on stack index *0*. Additional entries for the detector to describe the bug pattern the detector class detects, as well as a description of the bug pattern is placed in *findbugs.xml* and *messages.xml* as described in section 2.10. If the call is not found, a new bug pattern, e.g. *insecure_cookie* can be reported. The code that needs to be written is shown in Listing 24. Running this detector on the 17 test cases within the CWE614_

Sensitive_Cookie_Without_Secure package in Juliet Test Suite presents the results shown in Figure H.1. Through manual verification, this detector was found to correctly identify all the missing calls to the *setSecure* method in these test cases.

```
Cookie cookie = new Cookie("myCookieName", "myCookieValue");
cookie.setSecure(true);
```

Listing 22: Example code for initializing a new cookie and setting the secure attribute by calling *setSecure*

```
 0 new #23 <javax/servlet/http/Cookie>
 3 dup
 4 ldc #24 <myCookieName>
 6 ldc #25 <myCookieValue>
 8 invokespecial #26 <javax/servlet/http/Cookie.<init>>
11 astore_2
12 aload_2
13 iconst_1
14 invokevirtual #27 <javax/servlet/http/Cookie.setSecure>
17 return
```

Listing 23: Corresponding bytecode for the cookie example in Listing 22

```
addTrackedObject(new TrackedObject("javax/servlet/http/Cookie.<init>")
.addTrackedCallForObject(
new TrackedCall("javax/servlet/http/Cookie.setSecure",
    Arrays.asList(1),
    Arrays.asList(0),
"INSECURE_COOKIE")
.reportBugWhenNotCalled(true))
);
```

Listing 24: Code required to detect cookies missing the *setSecure* call

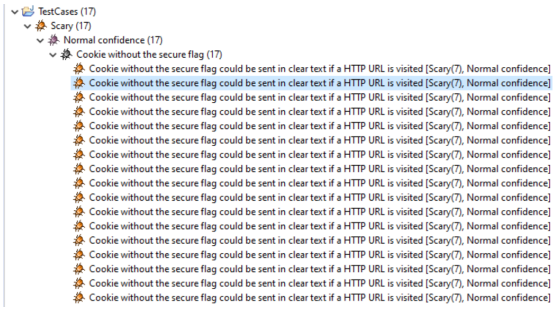


Figure H.1: Detection result of evaluating the insecure cookie detector shown in Listing 24 on the 17 test cases in Juliet Test Suite [95]

This also shows that the detectors for the additional XML parsers mentioned in subsection 2.7.1 can easily be implemented using this base instance tracker detection class.

This was not done as part of this thesis since the main focus was on improving the existing detectors present in FindSecBugs. However, as shown by the cookie detector example above the effort required to implement these new detectors is low.

Appendix I

Implementation of the Auto Fix Approach

As mentioned in Appendix B the plugin project *fb-contrib-eclipse-quick-fixes*, which is an extension to SpotBugs, seemed the most promising to extend with auto fixes for security vulnerabilities. This plugin is an Eclipse plugin project set up for integrating new auto fixes with SpotBugs. Each auto fix requires an entry in the *plugin.xml* file and a corresponding class to perform the auto fix. An excerpt from *plugin.xml* is shown in Listing 25. This shows the entry for one of the quick fixes for the bug pattern XXE_XMLREADER. The auto fix classes extend the *BugResolution* class from SpotBugs, which was found to not be documented in [55]. The *BugResolution* class uses an AST representation of the source code created by the Eclipse JDT and handles writing the auto fixes back to the AST of the class being auto fixed, as well as updating the bug markers in the IDE. Each auto fix extends this class and implements the *repairBug* method which queries the AST for the necessary information to create the auto fix. Using and modifying the AST for creating auto fixes was one of the approaches found to be used by classical auto fixes in the prestudy [89].

```
<quickfix
   class="xml.XMLReaderResolution"
   label="Replace with safe function"
   pattern="XXE_XMLREADER">
</quickfix>
```

Listing 25: Excerpt from plugin.xml showing the quick fix entry for the bug pattern XXE_XMLREADER

I.1 Auto Fixing using Instance Tracking Resolution

The *BugResolution* class from SpotBugs was extended to create a generalized abstract instance track resolution class for creating auto fixes for bugs identified by the *Instance-*

TrackDetector class described in section G.1. This class was named *AbstractInstanceTrackResolution*. Classes extending this class provide the abstract class with a list of *NodeToFind* classes which each store a string representation of the type of nodes to find, a string representation of the method invoked on this type, and a boolean value to denote whether the quick fix should be inserted after that node. This *NodeToFind* list is used to find the AST nodes that need to be traversed in order to find the insertion location for the auto fix. For some auto fixes, as will be more explicitly stated in the specific subsections below for each parser specific auto fix, the auto fix is inserted directly on the instance that is used to parse XML, whereas for others the quick fix is inserted on a different instance than the one the bug is reported on meaning that the AST needs to be traversed to find the correct instance to insert the fix on. Classes extending *AbstractInstanceTrackResolution* also provide a list of the imports that need to be added for this quick fix, and a list of *QuickFix* classes. Each *QuickFix* class stores the name of the method that needs to be invoked, and which parameters it needs to be invoked with.

The *AbstractInstanceTrackResolution* begins by overriding the *repairBug* method from the *BugResolution* class in SpotBugs. The helper method *getASTNode* from *edu.umd.cs.findbugs.plugin.eclipse.quickfix.util.ASTUtil* is used to get the AST node that corresponds to the source line of the bug marker from the *compilationUnit*. A *compilationUnit* is the type of the root node of an AST in Eclipse [25]. Next, a new visitor named *InvocationVisitor* extending *ASTVisitor* from Eclipse is created and used to visit all the *MethodInvocation* nodes of this AST node. This visitor obtains the *IVariableBinding* binding of the *FieldAccess* or the *SimpleName* which the *MethodInvocation* was called on and stores it in a variable called *nameOfInvocedInstance*.

The enclosing method block of the AST node is obtained by traversing the parents of the AST node. Then the *nameOfInvocedInstance* is checked to see if it is *null*. If it is not null, then the method called on the AST node corresponding to the bug marker was called on a variable, which means that the auto fix might need to be inserted on a different variable than the one corresponding to this node. To identify if the auto fix should be inserted on a different AST node, a new visitor named *InstanceTrackVisitor* extending *ASTVisitor* from Eclipse is created by providing a list of *NodeToFind* classes. The AST node of the enclosing method block is then visited using this visitor. The visitor visits all *VariableDeclarationStatement* nodes and checks if the type of this node corresponds to one of the types from the nodes to find list. If it does, the *Initializer* of the *Fragment* of this *VariableDeclarationStatement* is obtained. If this initializer is a *MethodInvocation* and the name of this method corresponds to the name of the method of the node to find, the visitor then obtains the *IVariableBinding* of the right-hand side of the *VariableDeclarationStatement* and maps the *IVariableBinding* of the left-hand side to the *IVariableBinding* of the right-hand side and stores it in a map called *assignmentMap*. E.g. for the code line shown in Listing 26, the corresponding *IVariableBinding* for *dBuilder* would be mapped to the *IVariableBinding* for *dbFactory*. Additionally, the *MethodInvocation* invocations for this *VariableDeclarationStatement* are compared to the invocations in the nodes to find list, and if they match, a mapping is created between the *IVariableBinding* on the left side of the *VariableDeclarationStatement* and the *MethodInvocation* invocation on the right side. This is stored in a map called *invocationMap*.

```
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
```

Listing 26: A new DocumentBuilder called dBuilder is created by calling *newDocumentBuilder()* on dbFactory

Similarly, the *InstanceTrackVisitor* visitor visits the corresponding *ExpressionStatement* nodes to find assignments where a variable that has already been declared is assigned to a new value. If the type of the left-hand side corresponds to the type of one of the nodes to find, a mapping is created between the *IVariableBinding* of the left-hand side and the *IVariableBinding* of the right-hand side. A mapping between the *IVariableBinding* and the *MethodInvocation* invocations for this node is also created for the assignments and stored in the *assignmentMap*.

The assignment map is then used to traverse the *IVariableBinding* bindings until the desired *IVariableBinding* is found. Then the list of invocations corresponding to this *IVariableBinding* is obtained. If the number of invocations is greater than one, the auto fix cannot be applied because the same variable was initialized more than once, and the *AbstractInstanceTrackResolution* class cannot determine where to put the auto fix. Otherwise, a check is done to determine if the auto fix can be called directly on the *IVariableBinding* that was found, or if there are nested invocations that need to be split up such that a reference can be obtained by making use of an auxiliary variable to apply the secure properties before continuing to invoke the remaining methods. If there are no nested invocations, then the list of *QuickFixes* is obtained and each auto fix is inserted right after the *IVariableBinding* found. Otherwise, if there are nested invocations, then these are traversed recursively until the desired node corresponding to the *MethodInvocation* is found. These chained invocations are then split up by making use of an auxiliary variable which the quick fixes are inserted on, before the remaining method invocations are called on this auxiliary variable

If the *nameOfInvocedInstance* variable is null, then the method called in the AST node corresponding to the bug marker was not called on a variable, which means that the auto fix does not have to traverse the AST to find the correct AST node to insert the auto fix on. A visitor to obtain the first *MethodInvocation* node for this AST node or the first *ClassInstanceCreation* node together with the variable this *ClassInstanceCreation* instance is stored in is created. If the *MethodInvocation* is not null, then the auto fix is inserted by splitting up the nested invocations as described above. Otherwise, the auto fix is inserted on the variable the new object was initialized to by using the variable the *ClassInstanceCreation* expression was called on. Finally, the AST nodes returned by the *importsToAdd* method provided by the subclasses are added.

I.1.1 The DocumentBuilderResolution class

Oracle specifies that setting *XMLConstants.FEATURE_SECURE_PROCESSING* explicitly to true is sufficient to disable external entities as well as DTD processing for the *DocumentBuilder* parser as mentioned in subsection 2.7.1. Therefore, the list of *QuickFix* classes consisted of one quick fix which is to set this attribute to true as shown in Table I.1. Additionally, when performing the auto fix *javax.xml.XMLConstants* is imported.

Function to call	Parameter	Value
setFeature	XMLConstants .FEATURE_SECURE _PROCESSING	true

Table I.1: The function to call, as well as which parameters to call the function with and which value that is necessary to make the *DocumentBuilder* parser safe

To instruct the *AbstractInstanceTrackResolution* class on where to insert the quick fix two *NodeToFind* classes were created. This is necessary because the attributes need to be set on the *DocumentBuilderFactory* that creates the *DocumentBuilder*, and not directly on the *DocumentBuilder* itself. The values of these nodes are shown in Table I.2.

Type	Method invocation	Insert quick fix here
DocumentBuilderFactory	newInstance	✓
DocumentBuilder	newDocumentBuilder	✗

Table I.2: The different AST nodes that need to be traversed to find the correct node to apply the quick fix to for the *DocumentBuilder* parser

I.1.2 The SAXParserResolution class

Similar to the quick fix for the *DocumentBuilder*, Oracle also specifies that setting *XMLConstants.FEATURE_SECURE_PROCESSING* explicitly to true is sufficient to disable external entities and DTD processing for the *SAXParser* parser as mentioned in subsection 2.7.1. Therefore, the list of *QuickFix* classes consisted of one quick fix which is to set this attribute to true as shown in Table I.3. Additionally, when performing the auto fix *javax.xml.XMLConstants* is imported.

Function to call	Parameter	Value
setFeature	XMLConstants .FEATURE_SECURE _PROCESSING	true

Table I.3: Function to call, as well as which parameter and which value necessary to make the *SAXParser* parser safe

The *SAXParser* parser is initialized similarly to the *DocumentBuilder* parser. To instruct the *AbstractInstanceTrackResolution* class on where to insert the quick fix two *NodeToFind* classes were created. This is necessary because the attributes need to be set on the *SAXParserFactory* creating the *SAXParser*, and not directly on the *SAXParser* itself. The values of these nodes are shown in Table I.4.

Type	Method invocation	Insert quick fix here
SAXParserFactory	newInstance	✓
SAXParser	newSAXParser	✗

Table I.4: The different AST nodes that need to be traversed to find the correct node to apply the quick fix to for the *SAXParser* parser

I.1.3 The XMLStreamReaderResolution class

Since the *XMLStreamReader*, *XMLEventReader* the *FilteredReader* parsers do not support *XMLConstants.FEATURE_SECURE_PROCESSING* according to Oracle as mentioned in subsection 2.7.1, the recommendations from OWASP was applied instead to explicitly disable external entities and DTD processing. The list of *QuickFix* classes, therefore, consisted of two quick fixes. The two attributes alongside their value that needs to be set are shown in Table I.5.

Function to call	Parameter	Value
setProperty	XMLInputFactory .SUPPORT_DTD	false
setProperty	XMLInputFactory.IS _SUPPORTING _EXTERNAL _ENTITIES	false

Table I.5: Function to call, as well as which parameter and which value necessary to make the *XMLStreamReader* parser, the *XMLEventReader* parser and the *FilteredReader* parser safe

The *XMLInputFactory* which is used to create the *XMLStreamReader* parser can also be used to create an *XMLEventReader* parser, and a *FilteredReader* [35]. In both cases the quick fix should be applied to the factory, as shown in Table I.6.

Type	Method invocation	Insert quick fix here
XMLInputFactory	newInstance	✓
XMLInputFactory	newFactory	✓

Table I.6: The different AST nodes that need to be traversed to find the correct node to apply the quick fix to for the *XMLStreamReader* parser, the *XMLEventReader* parser and the *FilteredReader* parser

I.1.4 The TransformerResolution class

Similar to the quick fixes for the *DocumentBuilder* parser, and the *SAXParser* parser, Oracle also specifies that setting *XMLConstants.FEATURE_SECURE_PROCESSING* explicitly to true is sufficient to disable external entities and DTD processing for the *Transformer* parser as mentioned in subsection 2.7.1. Therefore, the list of *QuickFix* classes consisted of one quick fix which is to set this attribute to true as shown in Table I.7. Additionally, when performing the auto fix *javax.xml.XMLConstants* is imported.

Function to call	Parameter	Value
setFeature	XMLConstants .FEATURE_SECURE _PROCESSING	true

Table I.7: Function to call, as well as which parameter and which value necessary to make the *Transformer* parser safe

The Transformer parser is initialized similarly to the *DocumentBuilder* parser and the *SAXParser* parser. To instruct the *AbstractInstanceTrackResolution* class on where to insert the quick fix two *NodeToFind* classes were created. This is necessary because the attributes need to be set on the *TransformerFactory* creating the *Transformer*, and not directly on the *Transformer* itself. The values of these nodes are shown in Table I.8.

Type	Method invocation	Insert quick fix here
TransformerFactory	newInstance	✓
Transformer	newTransformer	✗

Table I.8: The different AST nodes that need to be traversed to find the correct node to apply the quick fix to for the *Transformer* parser

I.1.5 The XMLReaderResolution class

Oracle specifies that setting *XMLConstants.FEATURE_SECURE_PROCESSING* explicitly to true is sufficient to disable external entities and DTD processing for the *XMLReader* parser. However, evaluating the auto fixes on the test suite described in section 6.1 showed that the parser was still vulnerable to external entities after setting *XMLConstants.FEATURE_SECURE_PROCESSING* explicitly to true. Therefore, the recommendations from OWASP as detailed in subsection 2.7.1 was applied instead. The list of *QuickFix* classes consisted of four quick fixes whose attributes and values are shown in Table I.9.

Function to call	Parameter	Value
setFeature	http://apache.org/xml/features/disallow-doctype-decl	true
setFeature	http://apache.org/xml/features/nonvalidating/load-external-dtd	false
setFeature	http://xml.org/sax/features/external-general-entities	false
setFeature	http://xml.org/sax/features/external-parameter-entities	false

Table I.9: Function to call, as well as which parameter and which value necessary to make the *XMLReader* parser safe

An XMLReader is created directly by invoking the *createXMLReader* method on the *XMLReaderFactory* [36]. Therefore the resulting quick fix should be applied directly on the resulting *XMLReader* instance, hence why the *NodeToFind* only includes one entry as shown in Table I.10.

Type	Method invocation	Insert quick fix here
XMLReader	createXMLReader	✓

Table I.10: The different AST nodes that need to be traversed to find the correct node to apply the quick fix to for the *XMLReader* parser

I.2 Benefits of Extending FindSecBugs

As mentioned in section I.1, the *BugResolution* class from SpotBugs is used to initialize the auto fix mechanism. The main benefit of using the *BugResolution* class from SpotBug as a starting point is that it sets up the link between SpotBugs and Eclipse JDT which is used to perform the auto fix. This class handles setting up the mapping between the bug marker supplied by the detectors implemented in FindSecBugs and the corresponding AST representation within Eclipse. Using this base class for integrating with Eclipse also allows the bugs to make use of the bulk auto fix functionality in Eclipse. This means that instead of manually having to go through each bug and fixing them, the bulk auto fix feature within Eclipse can be used to fix all the bugs of a similar type.

Appendix J

Implementation of AST based Auto Fixes of Insecure Cookies

A base auto fix class for detectors using the base instance tracker detector class was implemented as described in section I.1. This base auto fix class gathers the necessary information to perform the auto fix and traverses from the node the vulnerability occurred on to the node where the auto fix should have been inserted. It also includes a generalized mechanism for inserting multiple fixes as well as inserting multiple imports where necessary. The sections subsection I.1.1, subsection I.1.2, subsection I.1.3, subsection I.1.4, subsection I.1.5 are all similarly written precisely because of this base auto fix class having been implemented. The only differences between these different auto fixes are the fixes to apply, which nodes to traverse between, and which imports to add. Therefore, the overhead of implementing additional auto fixes is quite low. Additionally, since each auto fix is defined by an entry in *plugin.xml* and a class to perform the auto fix as described in Appendix I, multiple auto fixes for the same vulnerability can also be added by creating multiple auto fix classes extending the base auto fix class and adding multiple entries in the *plugin.xml* for the same bug pattern.

An auto fix for the example mentioned in Appendix H, where a detector for finding insecure cookies is created, can easily be created by extending the base instance tracker auto fix class. Since no nodes need to be traversed for this quick fix, an empty node to find is created stating that the auto fix should be inserted on the line where the bug is reported. Similarly, the list of auto fixes only requires one entry denoting the call to *setSecure* which should be called with the Boolean true. The necessary code lines to create an auto fix for insecure cookies are shown in Listing 27. As can be seen, keeping the generalized auto fix mechanism in a base class, together with a generalized instance tracking detection mechanism means that the overhead of developing new detectors and auto fixes based on instance tracking is quite low.

```
@Override
public List<NodeToFind> addNodesToFind() {
    return Arrays.asList(new NodeToFind("", "", true));
}
@Override
public List<QuickFix> getQuickFixes(AST ast) {
    QuickFix quickFix1 = new QuickFix(ast.newSimpleName("setSecure"),
        Arrays.asList((Expression)ast.newBooleanLiteral(true)));
    return Arrays.asList(quickFix1);
}
@Override
public String[] importsToAdd() {
    return new String[0];
}
```

Listing 27: Code required to auto fix cookies missing the *setSecure* call

Appendix K

Research Paper

The research paper created from the results of this master's thesis can be found below. This paper will be submitted to The Asia-Pacific Software Engineering Conference (APSEC) by the 10th of July 2020.

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

Torstein Molland
Norwegian University of Science and Technology
Trondheim, Norway
torstmol@alumni.ntnu.no

Andreas Nesbakken Berger
Norwegian University of Science and Technology
Trondheim, Norway
aberger@alumni.ntnu.no

Jingyue Li
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
jingyue.li@ntnu.no

***Abstract*—Web security is an important part of any web-based software system. XML External Entity attacks are one of the biggest security risks for web applications, both according to OWASP and MITRE. A successful XML External Entity attack can have severe consequences like denial of service, remote code execution, and information extraction. To better advance the field of automatic vulnerability fixing, we focused on finding out how XXE vulnerability detection can be improved, and how automatic fixing of these vulnerabilities can be done. We have also studied how a test bed can be designed for evaluating auto fixing tools. First, we created a test bed containing typical code vulnerable for XXE. Then, we used the test bed to evaluate one state of the art XXE vulnerability detector, i.e., FindSecBug. The results of the evaluation show the weakness of FindSecBugs. We improve the XXE detectors in FindSecBug by using a novel instance tracking approach. Our improved detection provides 100% precision and recall of detecting XXE vulnerability. We have also implemented auto fixes for XXE vulnerabilities in FindSecBugs. Our research contributes a novel instance tracking method to detect XXE vulnerabilities for the tool FindSecBugs, a novel extension for the tool adding auto fixes for the detected vulnerabilities, and a novel test bed for the evaluation of the detection and the auto fixes of XXE.**

***Index Terms*—software security, instance tracking, auto fix, XXE, AST**

I. Introduction

According to OWASP [1], web vulnerabilities can be classified into the categories injection, broken authentication, sensitive data exposure, XML External Entities, broken access control, security misconfiguration, Cross-Site Scripting, insecure deserialization, using components with known vulnerabilities, and insufficient logging and monitoring in descending order of security risk. XML External Entity attacks are ranked as the fourth most critical security risk to web applications. This is due to the severe consequences of such an attack being successfully carried out. XXE can be used for information extraction, Server Side Request Forgery (SSRF), denial of service attacks, and remote code execution. The Common Weakness Enumeration [2] classifies XXE as part of the top 25 most dangerous software errors in their list from 2019 [3]. Two kinds of XML vulnerabilities related to the parsing of XML Documents are recognized. CWE-611 denotes the vulnerability that occurs when an XML document which contains external entities outside of the sphere of control, is processed which leads to these documents becoming part of the output [4]. CWE-776 denotes the improper restriction of recursive entity references in Document Type Definitions [5].

The code below shows an XML parser being instantiated with default parameters. This parser will be vulnerable to XXE.

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

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

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

By adding the third line shown in the code listing shown below, the SAXParser is made secure and is not vulnerable to XXE.

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

The same vulnerable parser shown above is also vulnerable to denial of service, remote code execution, and information extraction. These attacks are performed by inputting different XML into the parser. The fixed parser is not vulnerable to any of these attacks.

In a study done by Späth et al. [6], all the XML parsers in Java were found to be vulnerable to XXE by default. This requires the developer to manually add lines of code to make the parsers secure every time the parser is used to mitigate the vulnerability. This means that a developer who uses an XML parser without changing the default settings will be vulnerable

to XXE without knowing it. Jan et al. [7] studied the presence of the Billion Laughs attack and XML External Entities attack in 13 popular parsers. These parsers were chosen due to being included with the programming languages Java, Python, PHP, Perl, and C#. They found that the parsers together had been used over half a million times. Evaluating the parsers on the billion laughs attack Jan et al. observed ram usage of above 8Gb and CPU usage of up to 51 minutes depending on the parser. When checking if open source systems that use the vulnerable XML parsers DocumentBuilder and SAXParser remember to apply mitigation strategies to prevent these attacks, Jan et al. found that 98.13% of open source projects had vulnerable parsers.

Oliveira et al. [8] implemented a tool based on WS-Attacker for testing the security of web service frameworks by dynamically evaluating them. The tool contains the nine attack types, i.e., coercive parsing, malformed XML, malicious attachment, oversized XML, soap array attack, XML bomb, XML document size, repetitive entity expansion, and XXE. After evaluating Apache Axis 1 and Apache Axis 2, Oliveira et al. found that both were vulnerable to numerous of the vulnerabilities tested including oversized XML and XML document sizer vulnerabilities. An extension to the dynamic testing tool WS-attacker for testing DOS attacks against XML parsers was also created by Falkenberg et al. [9]. In their evaluation, all the parsers were vulnerable to XXE attacks.

There currently exist present-day tools that can identify software vulnerabilities using means such as static code analysis [10, 11], byte code analysis [10, 12, 13, 14], dynamic analysis [15, 16, 17], pattern matching [13, 14], data flow analysis [10, 18, 19] and machine learning approaches [20]. Some tools are available as plugins that integrate directly into the IDE [10, 13, 14] giving developers feedback on possible vulnerabilities in the code as code is written. However, none of these tools were found to provide auto fix functionality for XML vulnerabilities.

To further advance the field of software security by improving the detection of XML vulnerabilities and auto fixing of XML vulnerabilities, three research questions are proposed:
RQ1: How can a test suite for evaluating web sec auto fixes be designed?
RQ2: How can detection of XXE be improved?
RQ3: How can auto fixing of XXE be implemented using an IDE plugin?

Our contributions are:

- A novel test bed for evaluating auto fixes of software vulnerabilities
- A novel detection mechanism for XXE based on instance tracking.
- A novel auto fix mechanism for XXE based on modifying the Abstract Syntax Tree (AST).

The rest of the paper is organized as follows. In section II the related work is presented. In section III the research method is detailed. In section IV the result and evaluation of the test bed, the novel instance tracker approach for XXE, and the AST based auto fix approach for XXE are shown. In section V the strengths and weaknesses of each of these are discussed. Finally, in section VI the conclusion and future work are presented.

II. Related Work

Vulnerability detection and auto fixing tools are currently evaluated on different test beds. The Juliet Test Suite [21] is a collection of vulnerable functions classified by CWE code. WebGoat [22] is a test bed designed as a complete application with many web vulnerabilities. ManyBugs [23] is the only test bed discovered tailored for evaluation of automatic bug fixes, but the test bed is created for non-security programming bugs in C.

FindSecBugs [10] is an extension to SpotBugs [24] which supports 134 different bug patterns. It supports the detection of XXE vulnerabilities using pattern matching. FindSecBugs uses data flow analysis and taint analysis for the detection of other vulnerabilities, like SQL Injections and Cross-Site Scripting.

LAPSE+ [14] is a vulnerability detection tool created by OWASP. The tool supports detection of XML injection vulnerabilities. LAPSE+ identifies sources and sinks using pattern matching and uses a manual provenance tracker which developers can use to manually check if a source is reachable from a sink using backward propagation. The tool is no longer supported by OWASP and has not been updated since 2011 and requires Eclipse Indigo from 2011 to run.

OWASP ASIDE [25] and ESVD [12] do not support detecting XXE vulnerabilities, however, they provide auto fixes for vulnerabilities using OWASP's ESAPI [26] for other vulnerabilities. These fixes are inserted by modifying the AST. ESAPI sanitizes inputs, but the API is deprecated, and it is not recommended to use. Both ASIDE and ESVD propose all possible auto fixes for each detected vulnerability, e.g., they propose fixes for SQL Injections for XXE vulnerabilities. Kim et al. [27] created a tool for automatically fixing non-software security related bugs. They used Eclipse JDT's AST API to apply their fixes to the source code.

III. Method

A. Research method to answer RQ1

RQ1 was answered by designing and creating a new test bed for automatically evaluating the tool to detect and auto fix XXE vulnerabilities. The test bed is designed based on flow variants from the Juliet Test Suite. For each test case, a set of tests for evaluation of the effectiveness and correctness of detection and auto fix were created. We discovered that the flow variants from Juliet Test Suite could not sufficiently detect XML vulnerabilities. Therefore we implemented 11 additional test cases with more complex data flows. These test cases are created to identify the limits of intraprocedural analysis in FindSecBugs, i.e. the test cases were limited to one method, which introduced multiple instances of parsers and other objects.

B. Research design to answer RQ2

To be able to improve the detection of XXE, we started by identifying the shortcomings of existing tools. FindSecBugs was discovered to be the best performing tool [28]. Therefore it was chosen as a basis for our research. The existing XXE vulnerability detectors in FindSecBugs was evaluated for XXE using the test bed proposed in answering RQ1.

After the evaluation, we have implemented several new detectors to address the limitations discovered in FindSecBugs. The most promising technique was implemented within FindSecBugs and evaluated on all the tests in the test bed proposed in answering RQ1. The detectors were also evaluated on the entirety of Juliet Test Suite to see if the new detection would bring any negative side effects, e.g., increased false positive rate or decreased time performance. Some function and variable names used are based on a closed pull request to the FindSecBugs GitHub repository [29].

C. Research design to answer RQ3

Many existing classical bug fix tools and auto fix tools for software security were found to use AST to perform their auto fixes and were found to perform well [27, 30]. Additionally, FindSecBugs was found to be the most promising tool to extend with auto fix functionality, since it was discovered to be an extensible tool minimizing the groundwork needed to be done. RQ3 was answered by designing and creating a new auto fix mechanism for XXE vulnerabilities based on modifying the AST. The auto fixes were implemented using parts of [31] which contains fixes for classical Java bugs detected by SpotBugs.

IV. Research Results

The source code for the test bed, the instance tracking based detectors, and the AST based auto fix tool detailed below have been made available in [32].

A. Results of RQ1

1) Test Bed Design: To ensure the test bed had a representative enough number of test cases for the different flow variants tested, it was designed based on the flow variants from the Juliet Test Suite. These test cases were created for the seven XML parsers included in the test bed, i.e., SAXParser, DocumentBuilder, EventReader, FilteredReader, TransformerFactory, XMLReader, and XMLStreamReader. The first 17 variants from Juliet [33] were chosen since these are the ones that are applicable to XXE and only include intraprocedural data flows.

Since Juliet only covers test cases with one parser wrapped in different control flows, e.g., for-loops and if-statements, we created additional test cases with more complex data flows. This included different ways of initializing an object as well as different ways of invoking methods on an object instance which may affect the detection and auto fix performance of a tool. 11 additional test cases were enumerated to better cover these cases:

- Six test cases with variations of class field and method variable were added. *SAXParser p* and *SAXParserFactory f* are either a class field or a method variable, for example:

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

- Four test cases with multiple parsers being made secure and insecure in the same method were added. In the code shown below, *SAXParser p1* is vulnerable. *SAXParser p2* is secure since the factory f has been made secure prior to initializing the parser:

```
// Test case  7
InputStream is = new FileInputStream(filePath);
SAXParserFactory f = SAXParserFactory.newInstance();

SAXParser p1 = f.newSAXParser();
PrintHandler h1 = new PrintHandler();
p1.parse(is, h1); // Insecure

f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,
                    true);

SAXParser p2 = f.newSAXParser();
PrintHandler h2 = new PrintHandler();
p2.parse(is, h2); // Secure
```

- One test case where an object with the same secure method as an XML parser, and an XML parser, was added. If instances are not tracked, it is impossible to know if the *.setFeature()* method has been called on the factory or on a different object. An example of this is shown below:

```
// Test case 11
Bar b = new Bar();
// Calls the setFeature method with correct parameters
// but on the wrong object
b.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,
                    true);

InputStream is = new FileInputStream(filePath);
SAXParserFactory f = SAXParserFactory.newInstance();
SAXParser p = f.newSAXParser();
PrintHandler h = new PrintHandler();
p.parse(is, h); // Insecure
```

2) Evaluation Process: The automatic evaluation of the auto fixes is performed by running a JUnit test suite for the test cases in the test bed. The JUnit tests check that the auto fixes preserved the intended functionality of the function and that it successfully mitigated the vulnerability in the test case. The evaluation outputs a list showing the broken and the vulnerable functions after the auto fix is applied. This data can then be used to calculate the number of successful fixes, missed fixes, and incorrect fixes. The unit test for checking if a test case is vulnerable is shown below.

```
@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();
  }
  assertFalse(vulnerable,
    "Parser should not be vulnerable to XXE");
}
```

3) Automatic Validation of Fixes: The effectiveness of auto fixes is evaluated by a unit test attempting to parse an XML file with external entities. If the content of the external entity is retrieved, then the parser is insecure. If the parser throws a specific exception, e.g., SAXParserException, for the unit test shown in the code listing above, then the parser is secure.

The functionality of the test case after the fix is evaluated by a unit test that parses an XML file without external entities. The unit test is similar to the unit test evaluating the effectiveness. The XML document should be parsed correctly and without any exceptions. If the contents of the XML file is returned, the test case's functionality is preserved after the auto fix.

B. Results of RQ2

1) Results of evaluating existing XXE vulnerability detectors: FindSecBugs already supports detecting vulnerabilities related to the parsers XMLStreamReader, DocumentBuilder, SAXParser, XMLReader, and Transformer for Java. This detection was evaluated on XML test cases based on the Juliet Test Suite, as well as the 11 additional test cases we created for evaluating instance based vulnerabilities.

The results of evaluating the existing detectors on the flow variants from the Juliet Test Suite is shown in Table I. Since the test cases based on the relevant flow variants test different control flows, but not how these different control flows affect the methods called on the class instances, it can be expected that the precision and recall of the existing detectors for the different parsers are high.

Parser	TP	FP	FN	Precision	Recall
DocumentBuilder	17	0	0	100%	100%
XMLStreamReader	17	0	0	100%	100%
XMLEventReader	17	0	0	100%	100%
FilteredReader	17	0	0	100%	100%
SAXParser	17	0	0	100%	100%
XMLReader	17	0	0	100%	100%
Transformer	34	0	0	100%	100%

TABLE I
SUMMARY OF THE TRUE POSITIVES, FALSE POSITIVES, AND FALSE NEGATIVES AFTER EVALUATING THE EXISTING DETECTION ON TEST CASES BASED ON THE FLOW VARIANT FROM JULIET TEST SUITE FOR XML VULNERABILITIES

The result of evaluating on the 11 additional test cases created for evaluating instance based vulnerabilities is shown in Table II. There is a high number of true positives and a low number of false positives. However, the high number of false negatives shows that the detectors miss a lot of vulnerabilities. All of the detectors were able to handle test case one through six, which tests different ways of initializing the parser, but fell through on test case seven through eleven. Test case seven through ten tests the use of multiple parsers within the same method. Test case 11 tests the use of a parser and a separate instance with the same secure method as the parser in the same method. The reason for the false negatives in test case seven through 11 is that the existing detectors do not keep track of which instance the secure calls have been called on.

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%

TABLE II
SUMMARY OF THE TRUE POSITIVES, FALSE POSITIVES, AND FALSE NEGATIVES AFTER EVALUATING ON TEST CASES CREATED FOR EVALUATING INSTANCE BASED VULNERABILITIES

The result of evaluating the performance of the existing detectors for XML vulnerabilities has been summarized in Table III. The execution time was measured on a PC with 16G of memory and a 3.9GHz CPU using Windows 10 pro. The test bed detailed in subsection IV-A and the existing Juliet Test Suite was evaluated separately to show the execution time for test cases without XML vulnerabilities and test cases with. FindSecBugs allows enabling and disabling specific detectors. Therefore, only the XML vulnerability detectors were enabled. This is useful for comparing the detection performance of the new detectors presented later. A cold run denotes running the detection after restarting Eclipse and clearing all the bug markers. A hot run denotes running the detectors after only clearing the bug markers. The difference in execution time for a cold run and a hot run is due to caching performed by FindSecBugs.

Test suite	LOC	Execution time cold run	Execution time hot run
Test cases with XML vulnerabilities	24,087	4.6s	1.5s
Juliet Test Suite	5,143,930	84.1s	79.1s

TABLE III
EXECUTION TIME FOR THE EXISTING DETECTORS

2) Instance Tracking: Evaluating the existing detectors in FindSecBugs showed that the main weakness of the existing analysis is that it is not capable of knowing which instance the secure or vulnerable methods have been called on. Therefore, a detector that can track the instance and the methods called on this instance is needed.

The instance tracking approach we implemented is explained in the flow chart shown in Figure 1.

The approach can be broken down into four steps:

1) Find the instances to track: In this step, the opcodes corresponding to object initialization are used to find which instances to track the calls of. These instances are identified by their initialization instruction. The pseudocode for this step is shown below

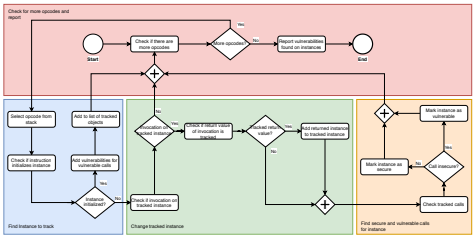


Fig. 1. Flow chart for the instance tracking approach

```
for each opcode in method
  if opcode is in list of initialization instructions
    add instance to list of tracked instances
    add vulnerabilities to the instance
  end if
end for
```

2) Combine instances that should be treated as the same instance: In this step, the opcodes corresponding to methods called on an instance being tracked are analyzed to find instances that should be tracked as part of the same instance with the same vulnerabilities. The pseudocode for this step is shown below

```
for each opcode in method
    if opcode is invocation instruction
        if invocation invoked on tracked instance
          add the return value to the tracked instance
        end if
    end if
end for
```

3) Identify the secure and vulnerable calls called on the tracked instances: In this step, the methods called on an instance being tracked are analyzed to identify methods that make the instance secure, and methods that make the instance vulnerable. The pseudocode for checking the calls is shown below

```
if invocation is singular tracked call
  if parameters of invocation corresponds to tracked call
    mark call as found
    if call is secure:
        remove the vulnerability reported by it
    end if
    if call is vulnerable:
        add the vulnerability reported by it
    end if
  end if
end if
if invocation is multiple tracked call
  for each singular tracked call
    if parameters of invocation corresponds\
        to tracked call
      if singular call is secure
        mark call as found
      end if
      if singular call is vulnerable
        mark call as not found
        add the vulnerability reported by
            the multiple tracked call
      end if
    end if
  end for
  if all singular calls for multiple tracked call found
    remove the vulnerability reported by it
end if
```

4) Report the vulnerabilities found for each instance: In this step, all the vulnerabilities found for each tracked instance are reported. The pseudocode for this step is shown below

```
for each tracked instance
    for each vulnerability
        report vulnerability
    end for
end for
```

A summary of the evaluation on the 11 test cases created for evaluating instance based vulnerabilities is shown in Table IV. There is a high number of true positives and a low number of false positives. Additionally, there is a low number of false negatives compared to the results shown in Table II. Compared to the existing detectors in FindSecBugs, a higher number of vulnerabilities were identified.

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%

TABLE IV
SUMMARY OF THE TRUE POSITIVES, FALSE POSITIVES, AND FALSE NEGATIVES AFTER EVALUATION OF THE PARSERS

The new detectors were also evaluated on the test cases based on the flow variants in Juliet. The results are shown in Table V. As can be seen, the new detectors are still able to identify all the test cases based on the control and data flow variants from the tests based on Juliet. The new detectors were also run on the entirety of the existing Juliet Test Suite which does not include any test cases for XXE. The results show that the changes do not lead to any negative side-effects.

Parser	TP	FP	FN	Precision	Recall
DocumentBuilder	17	0	0	100%	100%
XMLStreamReader	17	0	0	100%	100%
SAXParser	17	0	0	100%	100%
XMLReader	17	0	0	100%	100%
Transformer	34	0	0	100%	100%

TABLE V
SUMMARY OF THE TRUE POSITIVES, FALSE POSITIVES, AND FALSE NEGATIVES AFTER THE PRELIMINARY EVALUATION OF THE PARSERS ON THE JULIET STYLE TEST CASES

The result of evaluating the performance of the new detectors for XML vulnerabilities have been summarized in Table VI. The execution time was measured on a PC with 16G of memory and a 3.9GHz CPU using Windows 10 pro. The test bed detailed in subsection IV-A and the existing Juliet Test Suite was evaluated separately to show the execution time for test case without XML vulnerabilities and test cases with. A cold run denotes running the detection after restarting Eclipse and clearing all the bug markers. A hot run denotes running the detectors after only clearing the bug markers. The difference in execution time for a cold run and a hot run is due to caching performed by FindSecBugs.

C. Results of RQ3

In this section, a novel auto fix approach for instance related vulnerabilities based on traversing and modifying the AST is

Test suite	LOC	Execution time cold run	Execution time hot run
Test cases with XML vulnerabilities	24,087	4.21s	1.75s
Juliet Test Suite	5,143,930	110s	98.9s

TABLE VI
EXECUTION TIME FOR INSTANCE TRACKING DETECTORS

presented. This is useful for developers who are not domain experts to help them mitigate XML vulnerabilities by inserting the fixes at the correct location in the code. Given a detection mechanism, it is desirable with an auto fix mechanism to make mitigating the vulnerabilities easier. There are many different APIs and features that need to be set for different parsers to make them secure. Having an auto fix mechanism will help reduce the complexity, time, and effort spent identifying the correct fixes for the different parsers.

The AST based auto fix approach is summarized in Figure 2.

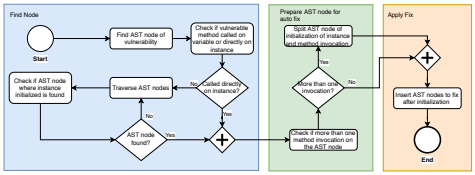


Fig. 2. Flow chart for the auto fixing approach

The approach can be broken down into three steps:

1) Find node to insert auto fix on: In this step, the location of the vulnerability reported by a vulnerability detector is used to find the AST node to insert the auto fix on. First, the AST node of the vulnerable source code line is obtained. Then a check is made to identify if the vulnerable method is called on a variable, e.g. for *p.parse()* the vulnerable method parse is called on the p variable, or directly on an instance, e.g. for *SAXParserFactory.newInstance(). newSAXParser().parse()* the vulnerable method parse is called directly on the instance created. If the vulnerable method is called on a variable, then the auto fix approach attempts to traverse the predecessors of the AST nodes of this variable until it finds the AST node of the variable where the instance to be auto fixed was initialized to. It does this by matching the names of the variables with each other.
   For the vulnerable parser example shown in section I, the AST node of p.parse() is found to be initialized by *f.newSAXParser()* so a mapping between p and f is made. Then f is found to have been initialized by *SAXParserFactory.newInstance()* so a mapping between f and *SAXParserFactory.newInstance()* is made. The type of this node is SAXParserFactory, which is used to determine that an instance where the auto fix should be inserted has been found. The pseudocode for this step is shown below

```
vulnerableNode = null
nodeToFix = null
visit each node n in class:
    if n corresponds to vulnerability location:
        vulnerableNode = n
visit each node n1 in vulnerableNode:
    if vulnerable method called on variable:
        visit each node n2 in parent method:
            if variable initialized:
                nodeToFix = n2
    if vulnerable method called on instance:
        nodeToFix = n1
```

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. E.g. for *SAXParserFactory. newInstance().newSAXParser().parse()*, the fix should be inserted on the AST node between *SAXParserFactory. newInstance()* and *.newSAXParser().parse()*. Hence why *SAXParserFactory.newInstance()* first needs to be stored in a variable e.g. f, and then the remaining calls are called on this variable e.g. *f.newSAXParser().parse()*. The pseudocode for this step is shown below

```
nodeToFix = node
numberOfMethodCalls = 0
visit each node n in nodeToFix:
    if node is method call:
        numberOfMethodCalls++
if numberOfMethodCalls > 1:
    instance = nodeToFix.instance
    remainingCalls = nodeToFix.calls
else:
    instance = nodeToFix
```

3) Apply auto fix: In this step, the AST nodes corresponding to the missing secure method calls are inserted after the initialization of the instance by modifying the AST. The result of this is equivalent to inserting line three in the secure parser example shown in section I. The necessary imports are also added. The pseudocode for this step is shown below

```
instance = node
fixes = list of fixes to apply
imports = list of imports to add
for each fix in fixes:
    instance.insertCall(fix)
end for
for each import in imports:
    instance.parentClass.add(import)
end for
```

A summary of the evaluation of the auto fix on the 11 test cases created for evaluating vulnerable instances is shown in Table VII. There is a high number of successful fixes for all the parsers. The number of successful fixes, missed fixes, and incorrect fixes for DocumentBuilder, SAXParser, and Transformer are identical. 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 minimal example of such a case is shown below. The fix is inserted on the second line making the factory secure which in turn makes parser p1 secure. However, line four makes the factory insecure again making parser p2 insecure. If line four is manually removed by a developer, then the fix inserted on

line two makes both parser p1 and p2 secure.

```
SAXParserFactory f = SAXParserFactory.newInstance();
f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
SAXParser p1 = f.newSAXParser();
f.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);
SAXParser p2 = f.newSAXParser();
```

The new detectors described in subsubsection IV-B2 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. These fixes were still regarded as incorrect fixes since a future improvement would be to give the developer the option to automatically remove the vulnerable code as part of the auto fix.

Parser	Successful fixes	Missed fixes	Incorrect fixes
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	14	0	2

TABLE VII
SUMMARY OF THE SUCCESSFUL FIXES, MISSED FIXES, AND INCORRECT FIXES AFTER EVALUATING THE AUTO FIXES

The auto fixes were also evaluated on the test cases based on Juliet. The result of this evaluation is shown in Table VIII. As can be seen, the auto fixes were successfully able to fix all the vulnerabilities in these test cases with no missed or incorrect fixes.

Parser	Successful fixes	Missed fixes	Incorrect fixes
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	17	0	0

TABLE VIII
SUMMARY OF THE SUCCESSFUL FIXES, MISSED FIXES, AND INCORRECT FIXES AFTER EVALUATING THE AUTO FIX MECHANISM ON THE JULIET STYLE TEST CASES

The result of evaluating the performance of the auto fixes for XML vulnerabilities is shown in Table IX. The execution time was measured on a PC with 16G of memory and a 3.9GHz CPU using Windows 10 pro. These numbers were obtained by fixing all vulnerabilities for each parser using the test bed detailed in subsection IV-A. As can be seen, auto fixing close to 30 vulnerabilities takes less than a second for each parser.

Parser	Number of vulnerabilities	Execution time
DocumentBuilder	33	668ms
XMLStreamReader	27	604ms
XMLEventReader	27	764ms
FilteredReader	27	684ms
SAXParser	34	861ms
XMLReader	29	799ms
Transformer	33	807ms

TABLE IX
EXECUTION TIME FOR AUTO FIXES

V. DISCUSSION

Comparing to existing test beds, e.g., Juliet Test Suite [21], WebGoat [22], and ManyBugs [23], the automatic evaluation of auto fixes of web vulnerabilities for Java is the main contribution of the test bed. No such feature exists for security auto fixing tools. The automatic evaluation makes checking the performance of auto fixing tools much simpler than it would be without this feature. This test bed allows researchers to quickly and easily do a thorough evaluation of their auto fixing tool. The test bed also has more robust testing of intraprocedural data flows than Juliet. The test bed also includes cases for XXE, which is not found in other test beds like Juliet.

The main strength of the new detectors based on the instance tracking approach is that they are able to handle more complex control and data flow variants with no false positives, compared to FindSecBugs. All of the Juliet style test cases and all the instance based test cases in our test bed are handled by instance tracking.

The instance tracking approach handles a general number of parameters, secure and insecure calls, and singular and multiple calls needed to determine if a parser is secure. The sequence of the calls called on the instance is kept track of, which means that the approach is able to identify when an instance is vulnerable and when it is secure. Due to keeping track of the calls performed on different instances, it is also capable of knowing which parsers within a method that is vulnerable and which is secure.

The main strength of the new detector implementations based on the instance tracking approach is that they are able to handle more complex control and data flow variants in addition to the simplest forms of XML vulnerabilities with no false positives, compared to FindSecBugs. All of the Juliet style test cases and all the instance based test cases are handled by this approach. 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 [34]. 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 missing the setSecure call, which can be handled by the instance tracking approach presented in this paper.

The main weakness of the instance tracking we implemented so far is that it performs only intraprocedural analysis and not interprocedural analysis. However, it was designed to perform only intraprocedural analysis to focus on supporting the auto fixing tool.

Another weakness of the instance tracker is that it takes longer to run compared to the existing detectors. However, our evaluation shows that is only 31% slower than FindSecBugs' XXE detectors on 5 million lines of code and the performance is comparable on 24K lines of code. Additionally, this is only a one-time penalty due to FindSecBugs only scanning files that have changed after the initial run.

The main strength of the auto fixes we implemented is that they provide specific auto fix suggestions for XXE. This allows

bulk auto fixes to be performed. ASIDE [25] and ESVD [12] suggests all auto fixes for each detected vulnerability.

When applying fixes to source code using ASTs, developers are assured that the code change will not break the semantics of the code. 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. The core algorithm has been extracted to a superclass allowing subclasses to supply the algorithm with which nodes to look for, and which AST nodes to insert to perform the auto fix. The approach can be used to auto fix other vulnerabilities with insecure instances, such as insecure cookies.

The main weakness of our AST based auto fix approach is that it cannot fix parsers that have been made explicitly vulnerable through calls to insecure methods as explained in subsection IV-C. The insecure calls need to be manually removed by a developer for the fixes to be effective. Another weakness of our implementation at the moment is that it only supports intraprocedural fixes and not interprocedural fixes because our detectors implemented currently only support intraprocedural analysis.

Our auto fixes are backward compatible with the existing detectors in FindSecBugs which means that they can be applied using the existing detectors and the instance tracking based detectors we have implemented. FindSecBugs reports a vulnerability where it can be exploited not where the fix should be inserted. These two locations may differ as well as the instance the fix should be inserted on. For the secure parser example shown in section I, the vulnerability is reported on line six on the SAXParser instance, but the fix is inserted on line three on the SAXParserFactory instance. Extending FindSecBugs means that the auto fix needs to traverse the AST to find the location of and which instance the fix should be inserted on. This makes the auto fixes more complicated. If instead the vulnerability was reported where the fix should be inserted, the location of and the instance to fix could be directly obtained.

A. Threats to Validity

1) Threats to Internal Validity: The soundness of our findings depends on the soundness of Eclipses APIs and FindSecBugs' underlying detection framework. If these do not perform as intended, the auto fixing tool proposed will not perform as demonstrated. This risk is the reason that improving the detection was made to be a part of this research. To reduce the risk of incorrect detection by FindSecBugs the tool was evaluated thoroughly, and changes were made to ensure a better and more reliable detection mechanism.

The auto fixes may be tailored for the test bed. This would result in better evaluation scores when evaluating the auto fixes and detection on this test bed than on other test beds. This was mitigated by examining known detection and fixing techniques and evaluating these as objectively as possible.

2) Threats to External Validity: One threat to the external validity of our research is the limited test bed used for evaluation. The results of this evaluation cannot simply be generalized and compared to results from evaluations on other test beds. This is the result of possible selection bias when creating test cases for the test bed. To mitigate selection bias, the test bed was made to be as similar as possible to the Juliet Test Suite, the most common test bed for evaluation of security analysis tools. This should help ensure our results are comparable to evaluations on other test beds. The evaluations done in this study also only focused on intraprocedural data flows. The results can therefore not be directly compared to results of tools evaluated on test beds including interprocedural data flows.

Another threat is that the study was based on FindSecBugs and the Eclipse IDE and its APIs. FindSecBugs might be implemented differently than other analysis tools, and Eclipse could have APIs that are not found in other IDEs. 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. To avoid this, generalized approaches such as modifying an AST and using data flow analysis were used.

VI. Conclusion and Future Work

A test bed for evaluation of detection and auto fixes of XXE has been proposed. Using this test bed the state of the art in the detection of XXE vulnerabilities has been evaluated. A novel instance tracking approach has been proposed and evaluated using the test bed. A novel auto fix approach based on AST for fixing XXE vulnerabilities is also proposed. Our results show that the test bed is effective for evaluating XXE detectors and auto fixes. Instance tracking was found to perform significantly better than the existing state of the art detectors for XXE vulnerabilities with high numbers of precision and recall. The auto fixes were able to fix a high number of vulnerabilities with a high number of successful fixes, and a low number of missed or incorrect fixes. The performance impact of instance tracking is only 31% slower on 5 million lines of code and comparable on 24k lines of code. The performance impact of auto fixing is negligible.

Future work is to add interprocedural support to the test bed, the instance tracking approach, and the auto fix approach. Additional support for more parsers can be added to the test bed, and detectors and auto fixes can be created for these. The generation of test cases can be automated. A study can be conducted to figure out where developers want the bug to be reported by the detectors. The auto fixes should let the developer remove vulnerable code as part of adding the secure calls. Additionally, if parts of the secure calls are present the auto fixes should only add the remaining ones.

References

[1] The OWASP Foundation, "Owasp top 10 - 2017," 2017, accessed January 21st, 2019. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.pdf

[2] MITRE, “Common vulnerabilities and exposures (cve),” 2019, accessed September 22, 2019. [Online]. Available: https://cwe.mitre.org/

[3] ——, “Cwe view: Weaknesses in the 2019 cwe top 25 most dangerous software errors,” 2019, accessed November 23, 2019. [Online]. Available: https://cwe.mitre.org/data/definitions/1200.html

[4] ——, “Cwe-611: Improper restriction of xml external entity reference,” 2019, accessed Jun 01, 2020. [Online]. Available: https://cwe.mitre.org/data/definitions/611.html

[5] ——, “Cwe-776: Improper restriction of recursive entity references in dtds (’xml entity expansion’),” 2019, accessed Jun 01, 2020. [Online]. Available: https://cwe.mitre.org/data/definitions/776.html

[6] C. Späth, C. Mainka, V. Mladenov, and J. Schwenk, “Sok: Xml parser vulnerabilities,” in *WOOT*, 2016.

[7] S. Jan, C. D. Nguyen, and L. Briand, “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*, Aug. 2015, pp. 233–241.

[8] R. A. Oliveira, N. Laranjeiro, and M. Vieira, “Wsfaggressor: An extensible web service framework attacking tool,” in *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, ser. MIDDLEWARE ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2405146.2405148

[9] A. Falkenberg, C. Mainka, J. Somorovsky, and J. Schwenk, “A new approach towards dos penetration testing on web services,” in *2013 IEEE 20th International Conference on Web Services*, 2013, pp. 491–498.

[10] P. Arteau, “Find Security Bugs,” 2019, accessed September 22, 2019. [Online]. Available: https://find-sec-bugs.github.io/

[11] F. Yan and T. Qiao, “Study on the detection of cross-site scripting vulnerabilities based on reverse code audit,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9937 LNCS. Springer Verlag, 2016, pp. 154–163.

[12] L. Sampaio and A. Garcia, “Exploring context-sensitive data flow analysis for early vulnerability detection,” *Journal of Systems and Software*, vol. 113, pp. 337 – 361, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215002873

[13] OWASP, “OWASP ASIDE Project,” 2016, accessed September 22, 2019. [Online]. Available: https://www.owasp.org/index.php/OWASP_ASIDE_Project

[14] The OWASP Foundation, “OWASP LAPSE Project,” 2017, accessed September 22, 2019. [Online]. Available: https://www.owasp.org/index.php/OWASP_LAPSE_Project

[15] C. Artho and A. Biere, “Combined static and dynamic analysis,” in *Electronic Notes in Theoretical Computer Science*, vol. 131, May 2005, pp. 3–14.

[16] T. Wang, T. Wei, G. Gu, and W. Zou, “Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection,” in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 497–512.

[17] G. Novark, E. D. Berger, and B. G. Zorn, “Exterminator: Automatically correcting memory errors with high probability,” *Commun. ACM*, vol. 51, no. 12, pp. 87–95, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1409360.1409382

[18] L. Sampaio, “Early Security Vulnerability Detector - ESVD,” 2019, accessed September 22, 2019. [Online]. Available: https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/

[19] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand, “Joanaudit: A tool for auditing common injection vulnerabilities,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 1004–1008. [Online]. Available: http://doi.acm.org/10.1145/3106237.3122822

[20] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, “Automated software vulnerability detection with machine learning,” Feb. 2018. [Online]. Available: http://arxiv.org/abs/1803.04497

[21] NIST, “Test suites,” 2017, accessed October 29, 2019. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[22] OWASP, “Webgoat project,” 2019, accessed December 4, 2019. [Online]. Available: https://www2.owasp.org/www-project-webgoat/

[23] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[24] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1052883.1052895

[25] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, “ASIDE: IDE support for web application security,” in *ACM International Conference Proceeding Series*, 2011, pp. 267–276.

[26] OWASP, “Owasp enterprise security api,” accessed November 21, 2019. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

[27] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 2013, pp. 802–811.

[28] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. Soares Cruzes, “Myths and facts about static application security testing tools: An action research at telenor digital,” in *Lecture Notes in Business Information Processing*, vol. 314. Springer Verlag, 2018, pp. 86–103.

[29] MaxNad, “Implement a base detector to track specific calls (#211),” 2016, accessed Jun 19, 2020. [Online]. Available: https://github.com/find-sec-bugs/find-sec-bugs/pull/220

[30] V. Balachandran, “Fix-it: An extensible code auto-fix component in review bot,” in *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*. IEEE Computer Society, 2013, pp. 167–172.

[31] kjlubick, “fb-contrib eclipse quick fix plugin,” accessed November 19, 2019. [Online]. Available: https://github.com/kjlubick/fb-contrib-eclipse-quick-fixes

[32] A. Berger and T. Molland, “xxe-autofix-tool,” 2020, accessed Jun 21, 2020. [Online]. Available: https://github.com/Berger-and-Molland/xxe-autofix-tool

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

[34] The OWASP Foundation, “Secure Cookie Flag,” 2020, accessed Jun 01, 2020. [Online]. Available: https://owasp.org/www-community/controls/SecureFlag

Andreas Nesbakken Berger and Torstein Molland

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

NTNU
Norwegian University of Science and Technology