Secure Coding Practices Enforcement via Static Analysis

Enforcing Secure Coding Techniques In Java Using Error Prone

Prashant Rajput
M.S. Student At UCLA
prashanthrajput@ucla.edu

Rachel Chu
M.S. Student At UCLA
rchu8@ucla.edu

Therese Horey
M.S. Student At UCLA
thorey@ucla.edu

JungBeom Lee
M.S. Student At UCLA
jungbeol@ucla.edu

Lana Ramjit
PhD Student At UCLA
lana@cs.ucla.edu

ABSTRACT

Security has become a growing problem in the Software as a Service (SaaS) industry as the Internet of Things (IoT) movement grows. This project aims to analyze infringement of common security practices that possibly occur in Open Source Projects using the custom checks functionality of the tool : Error Prone.

KEYWORDS

Error Prone, Security Practices, Java

1 INTRODUCTION

One of the most pressing concerns for a company today is the security of its software products. Often attackers exploit a zero-day vulnerability to attack the company in the hope of stealing some valuable data, which can later be sold or used to further damage the company. For instance, a zero-day exploit was detected for Adobe Flash Player on May 8, 2016. Attackers used a Flash exploit inside a Microsoft Office document and, upon opening the document, the exploit would download and execute the payload from a web server hosted by the attackers. Adobe then released a patch in just four days to fix this vulnerability. [1]

Even though such attacks are increasing in occurrence, software developers do not have adequate security training for incorporating basic security techniques in their code. According to the Breach Level Index, there were 1,792 breach incidents alone in 2016, where more than 1 Billion records were breached. [5]

Breaches due to attacks such as Buffer Overflow, SQL Injection, Cross Site Scripting, etc. can be avoided if developers have sufficient knowledge about coding securely. However, due to the pressures of delivering code on time and the high cost of security training, many developers do not pay attention to their code from a security perspective. Often, they assume APIs come with built-in or guarantee security that will be caught by the compiler . Unfortunately, this is not the case.

As the cloud computing and machine learning world grows, Java has become the predominant language in the software industry. According to the Tiobe Index [11] Java is used 8% more than C which is 2nd place. As such, security holes in Java or in Java API use may be common in the software world.

Error Prone [2], developed by Google, is a static Java analysis tool that can run on a variety of build environments such as Gradle, Bazel, and Maven, such that when the developer compiles the code, the tool's checks are run before stating it was a successful build. The tool, also, already comes built in with several security practice checks such as Return Value Ignored, Constant Overflow, and Using Strong Cryptography and allows easy custom checks to be coded and executed during the build. Since some of the built-in checks only cover a portion of the security practices we are interested in, we have enhanced the tool to better cover our security checks listed in Section 3.1. The research questions this paper seeks to answer are:

RQ1: How effective are idioms at identifying potential security vulnerabilities?

RQ2: How closely do open source Java projects obey security guidelines?

Before we go into detail of what we are checking for, first we will cover how our list of checks and approach were influenced by related works in Section 2. After covering our approach, we'll reveal our results, limitations to our work, and future work.

2 RELATED WORK

As the need for security in today's applications grows, there are many works, both in industry and academia, that try different techniques to address this growing need. One such technique is outlined in Secure Coding: Building Security into the Software Development Life Cycle [6]. Russel Jones and Abhinav Rastogi emphasize on the importance of consideration of security at every step of the development life cycle for software. They pinpoint that a major flaw as to why most software developed today have security holes is because security is built after the product is developed and released. Despite the growing number of organizations trying to enforce security while development, code generally gets committed with no security consideration built in. To fix this, they suggest having a delegated security expert per team to work with the major key stakeholders of the project. At every phase, the security expert is consulted for pinpointing possible security holes and errors till the product is released. Jones and Rastogi also provide a checklist which can be used by a security expert as a reference of what to look for, including the development checklist summarized in Table 1. They believe, by having security as a high priority at each step such that security is "baked" into the life cycle, the product will be secure.

However, the dependency on an expert to be consulted at each progression may be unnecessary overhead considering that security's importance peaks right before the code is committed. Many works address this aspect through the use of checkers that can be run before commit. Benjamin Livshits and Monica Lam from Stanford University tested this by creating their own static checker

Table 1: Jones & Rastogi Secure Coding Checklist

Sr. No.	Item
1.	Always validate all input to the system
2.	Always assume a hostile environment
3.	Use Open Standards
4.	Minimize and protect the trusted components in the system
5.	Have safeguards to protect the data wherever it exists
6.	Always authenticate
7.	Do not subvert the in-house security
8.	Ensure that if an application fails, it does not become less secure
9.	Always log and monitor what's going on in the system
10.	Use accurate date and time
11.	Provide least privilege and limit access to need to know
12.	Handle exceptions at the appropriate level
13.	Industry Standard Strong Ciphers and Algorithms should be used, not home-grown ones!
14.	Random numbers used should be nondeterministic

in Finding Security Vulnerabilities in Java Applications with Static Analysis [7]. Before conducting their study, they found code reviews were expensive to the code process and may be skipped over due to the incurred cost. Having a developer trained in security is also difficult because to have a security audit at every phase requires a third party security consultant which incurs an even higher cost. Instead, Livshits and Lam propose having tools for the specific language which can find security holes. In their opinion, this may be a more economical and effective solution. They decided to analyze Java since Java was created with security in mind (e.g. no direct access to memory or preventative buffer overflow). Also, Java, at the time of the paper, was growing in popularity for web based applications. To develop their tool, Livshits and Lam created a list of security holes as shown in Table 2 that can be caught through static analysis including unchecked input, data tampering, and query injection. They then created patterns for each security vulnerability to be searched for via use of PQL. PQL runs the pattern as a query against the code and is based on an earlier work that used Datalog and Binary Decision Diagrams to look for features that indicate a security vulnerability. Teams can add their own patterns to look for by defining a new query.

The tool developed by Livshits and Lam is only one of many tools developed to address code before commit. John Viega, Gary McGraw, and others joined together to develop Jslint in their work Statically Scanning Java Code: Finding Security Vulnerabilities [12]. They considered twelve rules as shown in Table 3 for Java code that should be adhered to for secure code and built their tool, JsLint, to directly deal with eight of the twelve. The tool's inner workings are much simpler in comparison to the work done by Livshits and Lam. Instead of using PQL to scan the language, the tool developed by Viega, McGraw, and others is a Java based program that takes the target program as input and pinpoints the violating lines by building a tree of Visitor nodes where a Visitor is a possible security violation. They also state it is easy to extend their tool in both

Table 2: Security Flags in Livshits & Lam 2005

Sr. No.	Flags
1.	Unchecked Parameter Passed
2.	Unchecked URL manipulation
3.	Unchecked Hidden field manipulation
4.	HTTP header tampering
5.	Cookie Poisoning
6.	SQL Injection
7.	Cross Site Scripting
8.	HTTP Response splitting
9.	Exploited Path Traversal
10.	Shell Command Injection

Table 3: JsLint Rules

Sr. No.	Rule
1.	Don't depend on initialization
2.	Limit Access to your classes, methods, and variables
3.	Make everything final by default, unless there's good reason not to do so
4.	Do not ignore values returned by methods
5.	Don't depend on package scope
6.	Do not use inner classes
7.	Avoid signing your code
8.	Make your classes uncloneable
9.	Make your classes unserializable
10.	Make your classes undeserializable
11.	Don't compare classes by Name
12.	Secrets stored in your code won't protect you

language and number of vulnerabilities via a class extension or an UI extension, similar to how Error Prone currently operates.

3 METHODOLOGY

3.1 Approach

Based on the checks in the related works covered, we found common elements among them. One such example is "Always validate all input to the system" from Jones & Rastogi, "Unchecked Parameter Passed", "Unchecked URL manipulation", "Unchecked Hidden field manipulation", and "SQL Injection" from Livshits & Lam. This set of rules deals with unchecked parameter passing, so we mapped them to the closest match from the book, The CERT Oracle Secure Coding Standard for Java [8]. We also decided to add several rules resulted in major bugs in real-world practices (e.g. Return Value Ignored).

Each rule chosen indicates a symptom of code that is not following security best practices based on the related work. We have attempted to translate these "best practice violation" rules into Java AST patterns, then extended ErrorProne via custom checks to detect those AST patterns. The rules we are checking can be found

Table 4: Secure Coding Techniques

Sr. No.	Secure Coding Technique
1.	Sanitize untrusted data passed across a trust boundary (SQL & XML)
2.	Do not pass untrusted unsanitized data to the Runtime.exec() method
3.	Do not ignore values returned by methods
4.	Prevent integer overflow
5.	Do not catch NullPointerException
6.	Use an int to capture the return value of methods that read a character or byte
7.	Fail securely: Perform proper cleanup at program termination
8.	Use strong cryptography

in Table 4. In the following section, we explain each rule and supplement them with code examples and a description of the AST pattern which we use to approximate the rule.

3.2 Our Checks & Error Prone

Error Prone is built off of OpenJDK [10] which allows programmers to use APIs once sealed away to be exclusive to Sun Microsystems including the compiler trees which reveal the grammar of Java. As such, it relies heavily on turning our checks into Abstract Syntax Tree Patterns. We will go over in this section our approach to each check and the check in detail.

Sanitize untrusted data passed across a trust boundary. This check is SQL and XML injection prevention. When programmers start a connection to an external server, which may use SQL or XML, they should only pass sanitized strings to the server:

```
public void SQLViolation(Connection conn)
{
   String sqlString = "Select * from db where username =" +
        username + " and password = " + pwd;
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery(sqlString);
}
```

For example, if the user were to set username to "badname or 1 == 1", SQL will evaluate the 1==1 to true and return the entire database to the user!

A better approach is to either sanitize the string so that it contains no executable code using a custom sanitization method, or to enforce constraints on any user-provided strings, such as, "No usernames longer than 8 characters." The perfect solution varies depending on context and language; as much as we wish to enforce all these solutions, our check was only supposed to fire a warning when a parameter is being passed into sensitive methods like Statement.executeQuery() (for SQL) and BufferedOutputStream.write() (for XML). This is one of the checks that we failed to implement properly, in part because a robust solution was extremely difficult due to the limitations of Error Prone, addressed in section 5.1.

Do not pass untrusted unsanitized data Runtime.exec() Runtime.exec() executes commands for the respective computer's terminal or command prompt. As such, it is liable to the same issue as XML and SQL where if it allows the user to input commands into the terminal, the user could execute a malicious script if the variable is not checked.

```
public void runtimeExecViolation()
{
   String dir = System.getProperty("dir");
   Runtime rt = Runtime.getRuntime();
   Process proc = rt.exec("cmd.exe /C dir " + dir);
}
```

In this example, the variable dir is not verified to be an actual directory. The user can make dir include another command which will get executed and possibly reveal sensitive information about the device. The check for this is covered by the same check as the one before.

Do not ignore values returned by methods. When a Java API returns a value, it generally is important to check the value that is returned. One such example is the following:

```
public void ignored(File file)
{
  file.delete();
}
```

The problem here is that delete() returns a boolean but, as shown, is ignored. If the deletion was unsuccessful and the file is never properly closed, attackers are able to read the contents of the file even when the code is executing something else. Error Prone already has a built in check for most of the return value ignored cases, including string replacement, in their ReturnValueIgnored check. However, it does not have a check on File methods such as the one in the example. We have extended their AbstractReturnValueIgnored API to also check for File methods that perform an operation on the file and return a boolean value indicating success or failure.

Prevent integer overflow. When a numeric overflow occurs, unknown results can occur since it does not throw an exception. Though the name of the check only states integer and overflow, the security practice is best applied to all numeric types and applies to underflow as well. Attackers exploit this weakness to trigger alternate control flows or cause a system crash. The HeartBleed attack is one of the best known instance of BufferOverflow Attack. Bug was in the OpenSSL's implementation of the TLS/DTLS (transport layer security protocols) heartbeat extension (RFC6520). When it was exploited, it lead to the leak of memory contents from the server to the client and from the client to the server. A code snippet introduced by the book:

```
public int multAccum(int oldVal, int newVal, int scale)
{
```

```
  //this will return wrong answer when overflow occurs
  return oldVal + (newVal * scale);
}
```

Suggested solution is encapsulating statements which contain potential overflow/underflow operations with check and throw exceptions. Error Prone already has a check on Integer Overflow via Constant Overflow but it only accounts for variables that are declared as constants that already exceed the limits. Due to time constraints and the complexity of the check, we were unable to create a new check for this.

Do not catch NullPointerException. This check is not limited to NullPointerException but is also meant to cover its parents RuntimeException and Exception. NullPointerException, RuntimeException and Exception are generic exceptions and the attacker can utilize the fact that one of these three exceptions is caught and trigger a different child exception to access the catch control block. The attacker can use this in combination with other security holes, like failing insecurely for triggering a state where objects are not closed properly and the attacker can read the data from the connection before the program exits. This is an example which throws NullPointerException but needs to be avoided:

```
public Boolean ThrowingNPE(String s)
{
  try {
  //doing something with s
  ...}
  catch(NullPointerException e)
  {return false;}
}
```

NullPointerException is treated on a different level than its two parents because it seems to be a common practice where instead of trying "variable == null", programmers depend on the NullPointerException to be thrown and caught to simulate an if-else statement. NullPointerException should really only be thrown if the object that is expected to be created is not created and the Java Compiler handles that.

The custom check for this runs on every catch block it finds and looks at the Exception parameter being passed to it. If the type matches Exception or RuntimeException, the check will fire. For NullPointerException, we match the name because the symbol table for Java only has Runtime Exception and Exception as explicit types.

Use an int to capture the return value of methods that read a character or byte.
Java objects of InputStream and Reader have read() methods that return the read character or byte as an int. When the end of file is reached, they both return the integer form of -1. However, since the programmer wants the character or byte, she may first typecast the output of the read as a byte or character. This causes the compare against -1 to never work or be incorrect since the typecast causes an inaccurate compare. An example of this is shown below:

```
public void badTypeCast()
{
  FileInputStream in;
  // initialize stream byte data;
  while ((data = (byte) in.read()) != -1) {
  // ...
  }
}
```

A correct solution for this issue is to first store the result in an integer variable. The programmer should use the integer variable for the compare and, only when the compare is successful, typecast it into the byte or char.

Our custom checker implemented in Error Prone first searches for a call to the method "read" by instances of either "FileInputStream" or "FileReader" class. If there is a match, it then checks for the data type of the variable that receives the return value from the method read. If the variable receiving the value is not of the type "int", throw an error. If there is no match, then return NO_MATCH.

Fail securely: perform proper cleanup at program termination. When an exception gets triggered, the programmer only thinks about exiting the program quickly or performing an alternate path. An example which does not consider about the issue:

```
public static void InsecureFail(String s) throws IOException
{
  BufferedReader reader = new BufferedReader(new
       FileReader(s));

  String line = null;
  while ((line = reader.readLine()) != null) {
       //process the line...
     }
  reader.close();
}
```

Doing so may result in resources that can leak potentially sensitive information, like an open file reader or connection. The attacker can use this open object to then obtain the information. This has also lead to Denial of Service attacks where hackers intentionally open connections and trip errors, causing the system to hang on the malicious threads.

Error Prone has a built in check called MustBeClosed but the limitation of this check is that it runs only if code is annotated with a @MustBeClosed and uses a child of AutoClosable. We have also implemented a simple check for catching instances where a Java program does not have a finally block for freeing all of it's resources.

Use strong cryptography. Encryption schemes like DSA have been broken where as well-known and widely used schemes such as RSA have not been broken yet. Hence using DSA encryption scheme in an application can lead to an attack.

Security also relies heavily on the quality of randomness in the pseudorandom numbers that are generated. This is because, these pseudorandom numbers are then used in various tasks such as encrypting a communication, hashing a value, etc. For instance generating random numbers with Random.nextInt() leads to generation of pseudorandomnumbers with poor randomness.

Error Prone already has a built in check for encryption scheme

called InsecureCryptoUsage (also called InsecureCypherMode). It will check where the Java's cryptography API is called and verify if the keys and encryption use an approved scheme. We've also implemented an additional check that searches for method calls to nextInt by the instances of class Random. If there is match, an error is thrown. This is because Random's pattern of generating integers is known and predictable.

3.3 Evaluation Plan

We are interested in finding out the extent to which well-known security best practices are followed by development communities. To answer this, we write custom security-checking rules for Error Prone and compile a suite of open-source projects with the modified checker, reporting the number of violations detected for each of the practices listed in Table 4.

Additionally, we are also interested in finding out whether adhering to these guidelines would effectively prevent security flaws in real projects. To answer this, we will use a subset of these repositories to investigate known and reported security problems by manually inspecting bug reports and code check-ins. We will then see if the bug was predicted by our bug, and if not, a qualitative description of why.

Our final list of projects was narrowed specifically by how easily we could hook our custom ErrorProne into each project's compilation process. Ant, Maven, and Bazel provide known hooks for using custom Error Prone plug-ins. Of these, Ant and Maven are the ones most commonly used by open source Java programs. The build tool Gradle also provides a plug-in mechanism for working with Error Prone, but it was non-standard and ineffective for reasons discussed in section 5.2. Consequently, our evaluation benchmark favors programs that use either Ant or Maven as their build tool. It was also necessary that the subcomponents of each project do not use custom compilation rules that were incompatible with ErrorProne.

Table 5: Benchmark projects and descriptions.

Project Name	Description
Apache Ivy	An agile tracking tool for managing dependencies
Apache Tomcat	A widely used open source Java webserver
AspectJ	An aspect-oriented programming extension for the Java language
Closure compiler	A compiler for the popular Closure langauge
Oracle Soda	NOSQL document storage for Oracle database

4 EVALUATION RESULTS

4.1 Tool Results

To verify that the tool did in fact match the ASTs we intended them to match, we wrote unit tests for each check and then ran them using Ant and Maven integrations to ensure that all integrations were using the custom plug-ins. We then applied the plug-ins that passed their tests to real world repositories to see if the custom checks picked up anything.

The Error Prone checks that we looked for as signs of insecure practice are as follows, with a star indicating that it is a custom check.

1. CaptureReturnValueofRead*
2. ConstantOverFlowCheck
3. ExtendedReturnValueIgnoredCheck*
4. FailSecurelyCheck*
5. InsecureCipherModeCheck
6. InsecureRandomCheck*
7. MustBeClosed 8. NPEAndParentsCaughtCheck*
9. ReturnValueIgnoredCheck

Note that several checks, such as UnsanitizedDataPassed, were not included, because they did not pass their tests cases.

For each project, we verified that the project compiled without errors as-is. Then we modified the build files so that all references to the standard Java compiler were replaced with the custom Java compiler that included all of the regular Error Prone checks as well as ours.

Error Prone works by applying the checks at compile time. Any matches found against the AST manifest as either a compiler warning or a compiler error. Since compiler errors force compilation to terminate while warnings do not, we forced the level of any matches found by Error Prone to "warning" level. This is safe because we tested each project to make sure it compiled without errors usually the regular Java compiler, so any errors that would have shown up must have been introduced by ErrorProne.

Additionally, the compiler stops generating warnings at 100 warnings by default. To work around this, we set the -Xmaxwarns flag on the compiler to 1000; if, after running the compiler we found exactly 1000 warnings generated, we re-ran it with a higher warning flag.

Table 6 on the next page contains counts of each check and the number of instances found per repository.

4.2 Accuracy

We manually examined 22 security bugs that had been disclosed in Apache Tomcat between versions 8 and 9. Tomcat was chosen in particular because it is a server application, therefore it has the type of security requirements that our checks are most likely to find. For each bug, we checked whether or not it was caused by one of the types of security violation that our checks aim to prevent. Then we manually examined the patches that were used to fix them, cross-referencing it with the checks picked up by our custom ErrorProne to determine whether our checks would have correctly predicted a security vulnerability.

Of the 22 released security bugs between the two versions, 8 of them belonged to the one of the categories we identified. 4 of them were untrusted data/data injection attacks, which is the security idiom we did not have a working check for at the time of evaluation. Of the remaining four, 3 of them involved failure to close resources, and the last was a failure to check the return value of a read operation.

Table 6: Incidence of ErrorProne warnings of selected checks

Name of Check	Apache Tomcat v9	Apache Tomcat v8	Apache Ivy	Oracle SODA	Clojure Compiler	Totals
		# of instances found				
CaptureReadReturnValue	23	0	0	0	0	23
ConstantOverFlow	0	0	0	0	0	0
ExtendedReturnValue	0	22	21	0	2	45
FailSecurely	2610	2689	213	45	61	5618
InsecureCipherMode	0	0	0	0	0	0
InsecureRandom	2	2	0	0	0	4
MustBeClosed	0	0	0	0	0	0
NPEAndParentsCaught	666	663	53	5	21	1408
ReturnValueIgnored	0	0	0	0	0	0
Totals	3299	3352	287	50	82	7094

Of the four remaining our tool predicted bugs in the three out of the four files that were changed in the patch. For the three bugs related to closing resources, CVE-2017-5651, CVE-2017-5650, and CVE-2017-5647, our checks predicted bugs in the first two instances. We determined that one of our predictions was unrelated. For the bug related to check read return value, CVE-2016-6817, we did find that our tool picked up an error in the abstract class at that line number, but for a different reason.

Overall, our checks did not predict any known bugs with absolute accuracy; though it found errors in three related files out of the four that were patched, this is most likely a result of our tool being over general.

In an analysis of security bugs from Jetty, we found that out of 40 that were examined, 12 of them were in the category we predicted. Our 8 checks are on the right path to common security vulnerabilites, but our limited analysis of our translation into ASTs is not specific enough to accurately predict real bugs.

```
-       while (payloadSize > 0) {
-           int toRead =
    Math.min(headerReadBuffer.remaining(), payloadSize);
+       int remaining = payloadSize;
+
+       while (remaining > 0) {
```

4.3 Precision

We manually examined a number of bugs for each project to see if our custom Error Prone can actually detect real security vulnerabilities. For Clojure, we examined 13 warnings, consisting of ExtendedReturnValueIgnoredCheck, NPEAndParentsCaughtCheck, and FailSecurelyCheck. Of these, 9 we considered real bugs, and 3 were unclear while 1 we did not consider a bug. For Tomcat, we examined 20 warnings, consisting of the checks mentioned previously plus InsecureRandomCheck. In this case, 10 we considered real bugs, and 2 were unclear while 7 we did not consider a bug. For Ivy, we examined the same checks as Clojure, discovering 7 bugs while the rest were unclear NPEAndParentsCaughtCheck and FailSecurelyCheck warnings. For Soda, we only had warnings for NPEAndParentsCaughtCheck and FailSecurelyCheck. We looked through a representative sample of these warnings and did not find any bugs we considered to be a real threat. NPEAndParentsCaughtCheck and FailSecurelyCheck accounted for most of our warnings, in some cases accounting for real bugs, but most were not. As an example of what we considered a real bug, in Clojure, a NPEAndParentsCaughtCheck found a code block for dealing with an exception that was empty. Likewise, in Tomcat a FailSecurelyCheck warning found a try statement that had no matching catch.

4.4 Analysis

It seems apparent that our FailSecurely check was over general; it generated over 2000 warnings. Forcing every instance of a try-catch block to have a finally block is indeed a lofty requirement and unlikely for developers to follow in practice. If we could perform dataflow analysis to tell when a method had resources it needed to be released, our check would have been more precise.

It also seems that idioms like "Don't catch NullPointerException or overgeneral exceptions" are so broad as to be ignored in practice by developers.

RQ1. How effective are idioms at identifying potential security vulnerabilities? This was a particularly difficult question to answer, since we needed very specific project data in order to answer it, and many projects were lacking a requisite feature. However, it seems that the more specific the idiom, the more likely it is to be followed. High level idioms such as NullPointerException and "fail securely" are difficult to follow, but low-level ones like "use an int for a return value" were directly correlated with some patches. Unfortunately, our dataset is too limited to provide a definitive guess. **How closely do open source Java projects obey security guidelines?** For the most part, well. Many of our checks yielded few errors, particularly the ones that are already included in ErrorProne. The high rate of incidence for our custom check most likely indicates that our AST patterns were, as discussed above, over general, and not a fault of developers.

5 LIMITATIONS

5.1 Error Prone

Google's Error Prone has several limitations which we continuously encountered as we worked on this project. The first of the limitations is the amount of documentation about using Error Prone and

developing a check for it was limited. It is highly likely that most of the documentation for Error Prone is kept internal at Google but it greatly affected the usability of the tool. We had to spend a significant amount of time to figure out how to set up our environment to run our custom plug in checks and navigating various documentation sites [9] [3] to find adequate descriptions of what are the available APIs and their functions.

Apart from that hurdle, as we created some of our checks, we realized Error Prone is limited to a small local scope. If we wished for a more global scope for the check, like the unsanitized data checks, it was very difficult to code. We originally started coding a more specific check for the Unsanitized Data checks, but, when it reached 300 lines, we decided to settle with a more simple but bound to cause more false positives check. The main constraint on why the tool has to adhere to the local scope of the method call is that it heavily depends on the Oracle Compiler Tree API. Oracle's Compiler Tree API specifies their tree scanners can only return one node at a time. Unsanitized data checks, to be effective, requires the API to return multiple nodes at a time since the syntax, not just the data type, of the node mattered.

5.2 Evaluation

We were limited in our evaluation by having very specific requirements for the open-source tools we use. We needed repositories that track security vulnerabilities in past versions, release patch information, release source code to previous versions, and that had build requirements that were compatible with ErrorProne plug-ins. We were unable to analyze earlier versions of Tomcat, for example, because ErrorProne is incompatible with Java versions before 1.6. We could not analyze Jetty because the build files for each had similar compiler arguments that were incompatible with ErrorProne. Consequently, our sample size for accuracy is very limited.

5.3 Threats to Validity

5.3.1 Internal Validity. Because most approaches were introduced around 2000s developers already know the secure approaches. It could affect our result to be biased. Possible solution to avoid this is choosing recently introduced security problems and implement checkers for those.

5.3.2 External Validity. Since we are limited to using primarily open-source projects, our evaluation risks being biased towards projects that may have lower security standards, posing a threat to external validity. However, we tried to include projects that specifically focus on interfacing with Oracle APIs or servers, so that the projects we were using would be prone to security vulnerabilities like the ones we were looking for.

5.3.3 Construct Validity. Threats to construct validity in our approach include the fact that we are limited to security flaws that are detectable by Error Prone. Our translation of the idioms into ASTs that were detectable by ErrorProne was very high level and as a consequence, often over-general. Our evaluation can only be as good as the precision of the ASTs.

A GITHUB USERNAME TO STUDENT MAPPINGS

When analyzing our GitHub repository [4] for task delegation assignments and coding changes, please refer to Table 7 to map our GitHub username to the contributor's name.

Table 7: GitHub User Name to Student Mapping

GitHub Username	Student
starlordphr	Prashant Rajput
NinjaLlama	Therese Horey
KHikami	Rachel Chu
nanaya07	JungBeom Lee
alanamramjit	Lana Ramjit

B CONTRIBUTIONS PER INDIVIDUAL

Individual	Contributions
Prashant Rajput	Idea Contributor; Coding: Return int for read(), Secure Random, Fail Securely; Paper Sections: Introduction, Approach
Therese Horey	Evaluation: open source Java project research - version logs and security vulnerabilities, result log parsing, precision analysis; Paper Sections: Introduction, Precision, presentation slides
Rachel Chu	Coding: Environment Setup, NPE & Parents Check, Simple Unsanitized Data, Extended Return Value Ignored, and Error Prone help; Paper Sections: Related Works, Error Prone Limitations, Checks, Introduction
JungBeom Lee	Coding: Capturing Checkers' Documentation, MustBeClosed(Fail securely); Paper Sections: Approach code examples internal validity
Lana Ramjit	Evaluation: Benchmark selection, custom ErrorProne integration into benchmark build files, unit test build tool integration, result log generation and parsing, accuracy analysis; Paper Sections: Evaluation Plan, Tool Results, Accuracy, Analysis, research questions, Threats to Validity, presentation slides

REFERENCES

[1] FireEye. White Paper: Survey of Zero-Day Attacks. https://www.fireeye.com/blog/threat-research/2016/05/cve-2016-4117-flash-zero-day.html. ().
[2] Google. Error Prone. http://errorprone.info/index. (). Accessed: 2017-04-24.
[3] Inc. Google. Error Prone API. http://errorprone.info/api/latest/. ().
[4] Therese Horey, Prashant Rajput, JungBeom Lee, and Rachel Chu. CS230 Secure Code Analysis. https://github.com/starlordphr/CS230-Secure_Code_Analysis. ().
[5] Breach Level Index. Breach Level Index Report 2016. http://breachlevelindex.com/assets/Breach-Level-Index-Report-2016-Gemalto.pdf. ().

[6] Russell L Jones and Abhinav Rastogi. 2004. Secure coding: building security into the software development life cycle. *Information Systems Security* 13, 5 (2004), 29–39.
[7] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *Usenix Security*, Vol. 2013.
[8] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. 2011. *The CERT Oracle Secure Coding Standard for Java.* Addison-Wesley Professional.
[9] Oracle. Compiler Tree API. https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/overview-summary.html. ().
[10] Oracle. OpenJDK. http://openjdk.java.net/. ().
[11] Tiobe. Tiobe Index. https://www.tiobe.com/tiobe-index/. ().
[12] John Viega, Gary McGraw, Tom Mutdosch, and Edward W Felten. 2000. Statically scanning java code: Finding security vulnerabilities. *IEEE software* 17, 5 (2000), 68–74.