The Flavour of Real World Vulnerability Detection and Intelligent Configuration

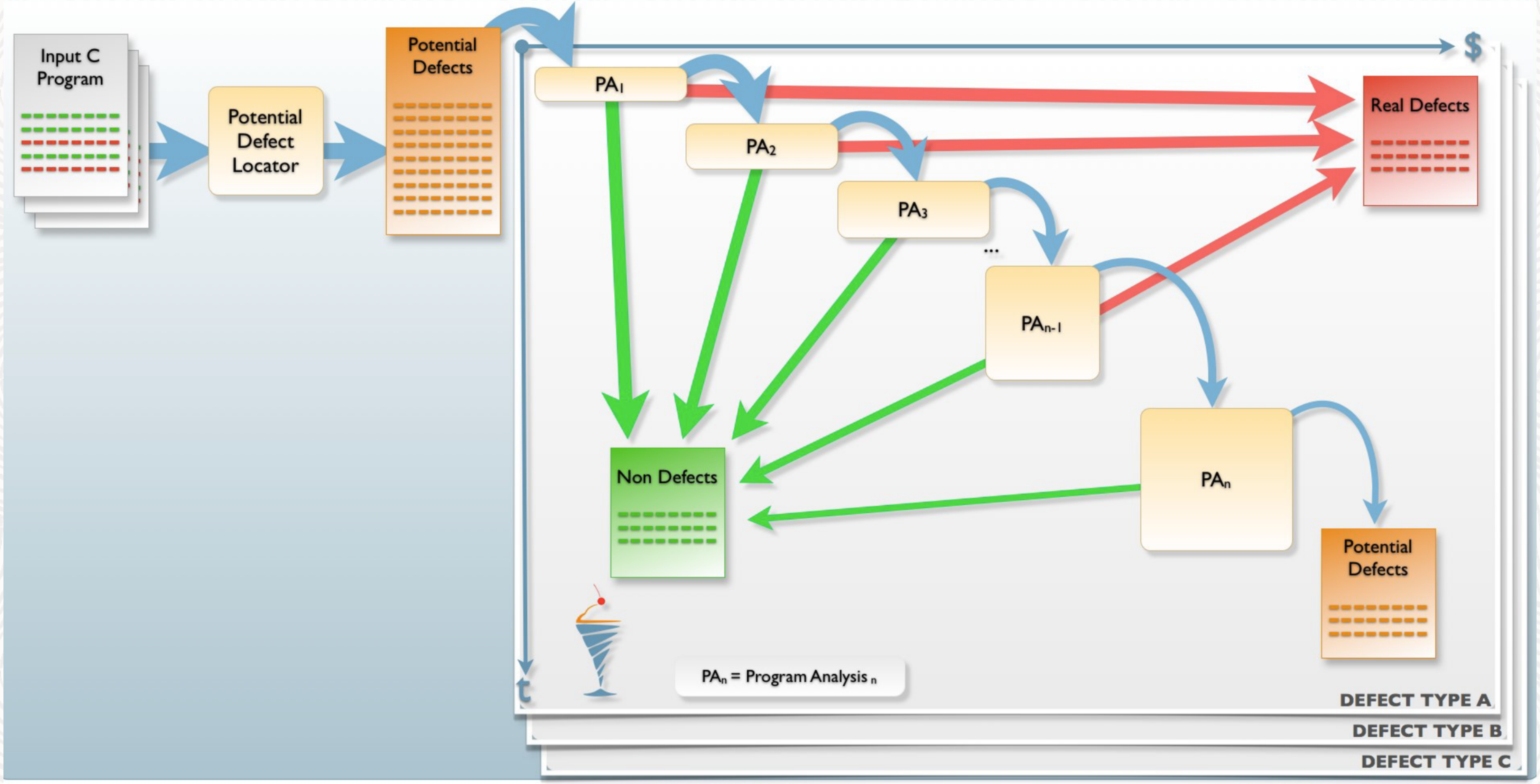
Cristina Cifuentes
Oracle Labs
May 2021

The Oracle Parfait static code analysis tool is used by thousands of developers worldwide on a day-to-day basis over commercial and open source codebases of multi-million lines of code.

The Parfait Design and Implementation

2007 design

2007-2018 implementation


Input C Program
Potential Defect Locator
Potential Defects
$
PA1
PA2
PA3
...
PAn-1
PAn
Real Defects
Non Defects
Potential Defects
t
PAn = Program Analysis n
DEFECT TYPE A
DEFECT TYPE B
DEFECT TYPE C

Key Features of the Parfait Design

Scalability achieved by

- Layered approach
- Demand-driven analyses
  - Process subsets of the code; not whole program at a time
- Multiple ways to parallelise framework
  - Per bug-type, per analysis, per “executable”-file

Key Features of the Parfait Design

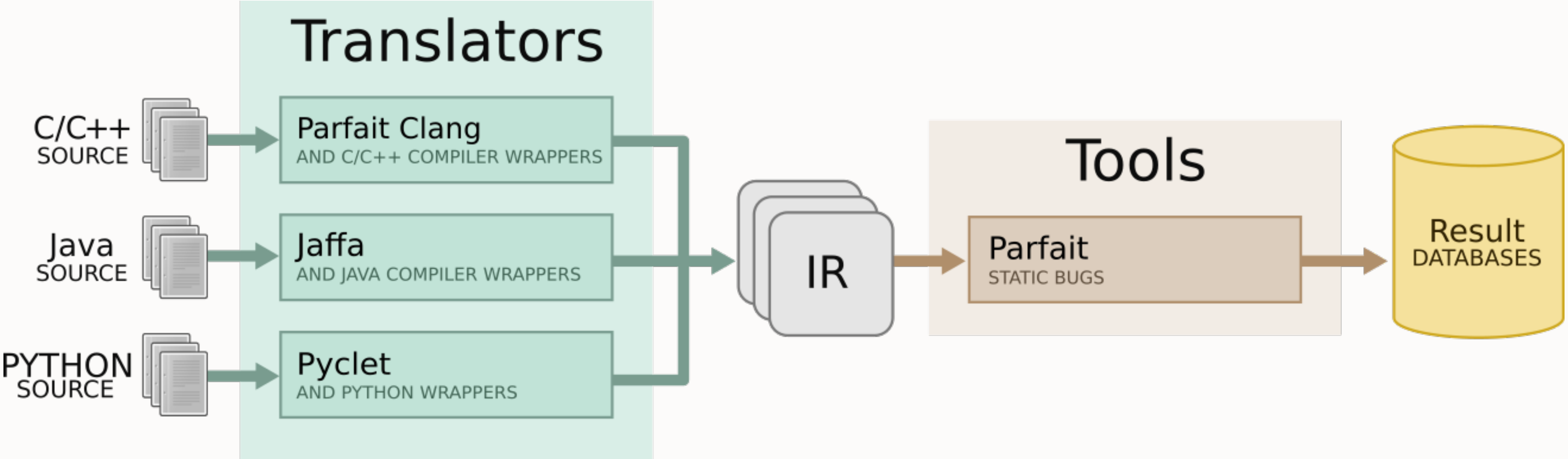
Scalability achieved by

- Layered approach
- Demand-driven analyses
  - Process subsets of the code; not whole program at a time
- Multiple ways to parallelise framework
  - Per bug-type, per analysis, per “executable”-file

Precision achieved by

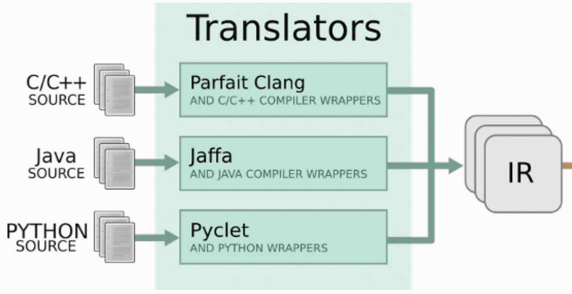
- Multiple lists of bugs (NoBugs, PotentialBugs, RealBugs)
- Bugs moved from PotentialBugs to RealBugs list conservatively

The Parfait Implementation



Built on top of LLVM

Build Integration for Make (C, Java) and Python



Drop-in replacement for C compiler

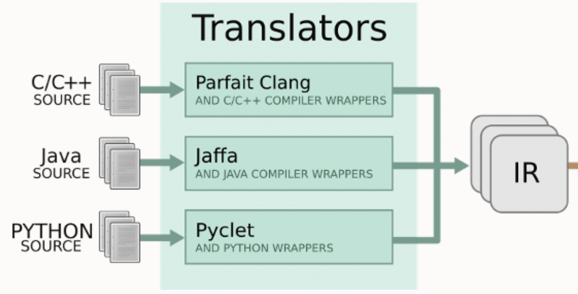
```
parfait-gcc -o test test.c
parfait test.bc
```

Drop-in replacement for Java compiler

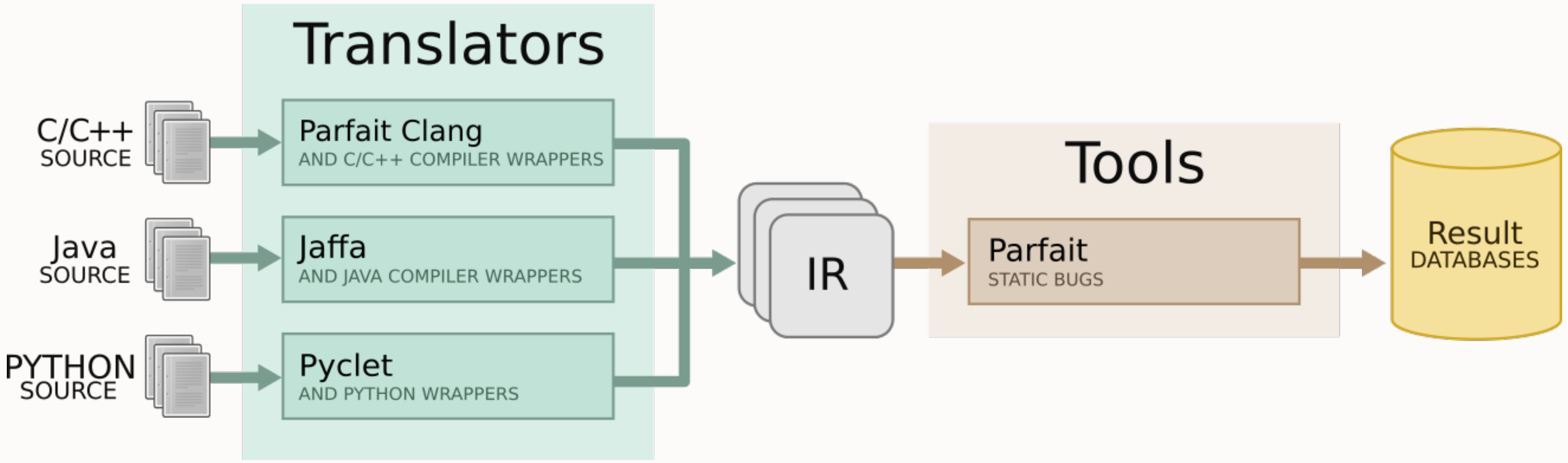
```
parfait-javac -o test test.java
parfait test.bc
```

Drop-in replacement for Python bytecode compiler

```
parfait-python -p test-dir -o test.bc test-dir
parfait test.bc
```

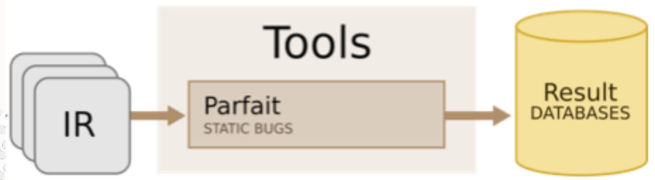


Build Integration with Maven and Gradle Plugins (Java)



Orchestrated by the plugin

```
buildscript {
    repositories {
        maven { url 'https://<artifactory-parfait-release>'} }
    dependencies {
        classpath 'oracle.parfait:gradle.plugins:1.0.5'}
}
apply plugin: 'oracle.parfait'
```



Sample Analyses

Data flow analysis

- Keeps track of data values at each point in the program

Partial evaluation

- Executes partially-evaluated slice of a potential bug

Symbolic analysis

- Symbolically tracks values of a program slice of interest

Control flow analysis

- Keeps track of flow of control through the program

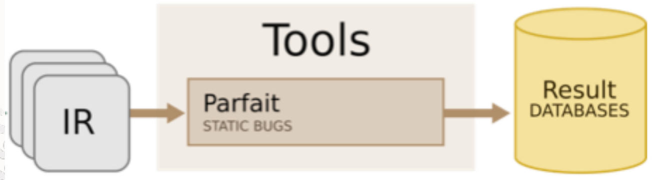
Taint analysis

- Keeps track of data that is user controllable

Leak analysis

- Keeps track of sensitive data that reaches lower privileged parts of the application

Bugs and Vulnerabilities that Matter



C, C++

- Buffer overflows
- Memory/pointer bugs
  - NULL pointer dereference, use after free, double free, memory leak, …
- Integer overflow

Java EE

- SQL injection, cross-site scripting (XSS), LDAP injection, OS injection, …
- XXE/XEE
- Insecure crypto
- Insecure deserialization

Java Platform

- Unguarded caller-sensitive method calls
- Unsafe use of doPrivileged
- Call to overridable method during deserialization

Python

- SQL injection, command injection
- Insecure deserialization
- Unsafe eval

Bugs and Vulnerabilities that Matter

C, C++

- Buffer overflows
- Memory/pointer bugs
  - NULL pointer dereference, use after free, double free, memory leak, …
- Integer overflow

Java EE

- SQL injection, cross- … LDAP injection, OS in…
- XXE/XEE
- Insecure crypto
- Insecure deserialization

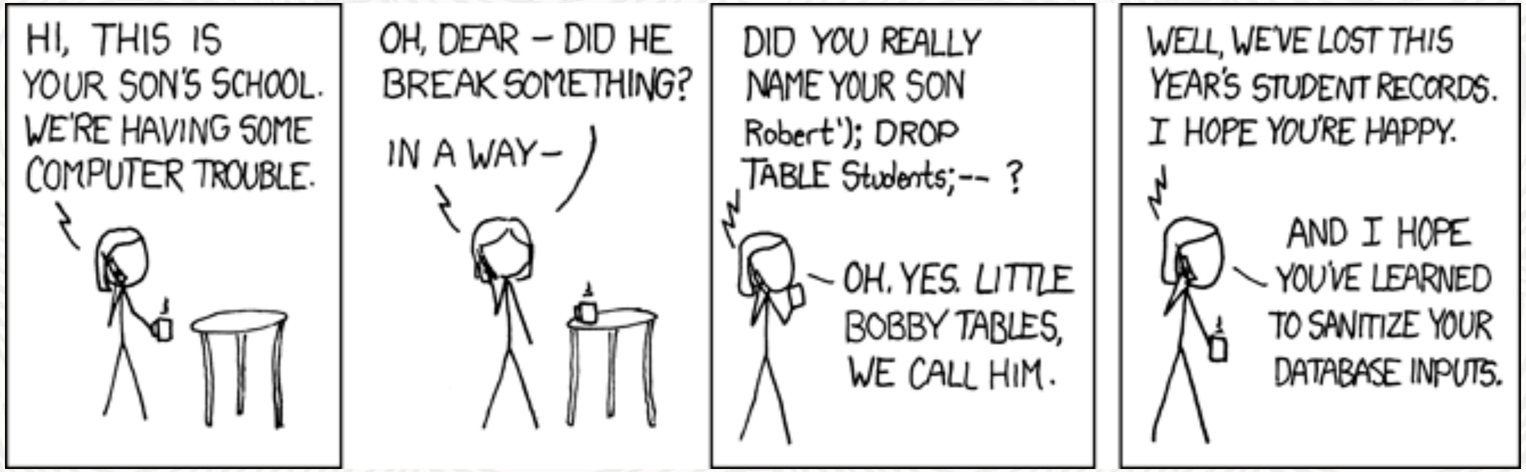
Java Platform

- Unguard… …er-sensitive method calls
- U… …Privileged
- … method during

- SQL injection, command injection
- Insecure deserialization
- Unsafe eval

Cross-language vulnerabilities (Java - C)
- Buffer overflows
- Dereference of untrusted pointer
- SQL injection
- All injection vulnerabilities

Detecting SQL Injection in C, Java and Python Code



Taint Using Dataflow Analysis for SQL Injection

```
protected Element createContent(WebSession s)
{
    ...
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                          connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                     ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

https://github.com/WebGoat/WebGoat

Taint Using Dataflow Analysis for SQL Injection

```
protected Element createContent(WebSession s)
{
    ...
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                       connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                  ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

Taint Using Dataflow Analysis for SQL Injection

```
protected Element createContent(WebSession s)
{
    ...
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                           connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                      ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

Taint Using Dataflow Analysis for SQL Injection

```
protected Element createContent(WebSession s)
{
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                   connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                              ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

Taint Using Dataflow Analysis for SQL Injection

```
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + "not found");
    else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + "was empty");
    }

    return (values[0]);
}
```

Taint Using Dataflow Analysis for SQL Injection

A source of tainted data

```
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + “not found”);
    else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + “was empty”);
    }

    return (values[0]);
}
```

Taint Using Dataflow Analysis for SQL Injection

```
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + "not found");
    else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + "was empty");
    }

    return (values[0]);
}
```

No sanitisation of String values

Taint Using Dataflow Analysis for SQL Injection

```
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + “not found”);
    else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + “was empty”);
    }

    return (values[0]);
}
```

Returns tainted String values[0]

Taint Using Dataflow Analysis for SQL Injection

tainted String

```
protected Element createContent(WebSession s)
{
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                          connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                     ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

Taint Using Dataflow Analysis for SQL Injection

```
protected Element createContent(WebSession s)
{
    ...
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                          connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                     ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

A sink of tainted data

org.owasp.webgoat.session.ParameterParser.getRawParameter

org/owasp/webgoat/session/ParameterParser.java

c1c5914ec99978da1a99c7ca7a04dd0e13a7e3a1-nightly

```
     * @throws org.owasp.webgoat.session.ParameterNotFoundException
     */
    public String getRawParameter(String name) throws ParameterNotF
        String[] values = request.getParameterValues(name);
```

#1: 'values[...]' is tainted by user-provided input returned by method javax.servlet.ServletRequest.getParameterValues

```

        if (values == null) {
            throw new ParameterNotFoundException(name + " not found
        } else if (values[0].length() == 0) {
            throw new ParameterNotFoundException(name + " was empty
        }

        return (values[0]);
```

#2: '' is tainted (load of tainted 'values[...]')

#3: Tainted '' returned

```
    }

    /**
     * Gets the named parameter value as a short
```

org.owasp.webgoat.plugin.DOS_Login.createContent

org/owasp/webgoat/plugin/DOS_Login.java

Line # | File | Expand | Collapse

c1c5914ec99978da1a99c7ca7a04dd0e13a7e3a1-nightly

```
            String password = "";
            username = s.getParser().getRawParameter(USERNAME);
            password = s.getParser().getRawParameter(PASSWORD);
```

#4: 'password' is tainted by return value of method org.owasp.webgoat.session.ParameterParser.getRawParameter

```

            // don;t allow user name from other lessons. it would be too simple.
```

```
            Connection connection = DatabaseUtilities.getConnection(s);

            String query = "SELECT * FROM user_system_data WHERE user_name = '" +
username + "' and password = '"
```

#5: 'java.lang.StringBuilder.append()' is assumed to be tainted by the return value of method java.lang.StringBuilder.append, due to tainted argument 'password'

#6: 'query' is assumed to be tainted by the return value of method java.lang.StringBuilder.toString, due to tainted method receiver 'java.lang.StringBuilder.append()'

```
                    + password + "'";
            ec.addElement(new StringElement(query));

            try
            {
                Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,

ResultSet.CONCUR_READ_ONLY);
                ResultSet results = statement.executeQuery(query);
```

SQL INJECTION
Run SQL query with tainted input 'query'

A source to sink trace for SQL injection example

Finding Unguarded Caller-Sensitive Method Call Vulnerabilities in the Java Platform

CVE 2012-4681, August 2012

The Java Security Model

The Java Security model is access control based on inspecting current call stack

- The `SecurityManager` checks all frames on the stack
- E.g., if to execute a method, the method needs permission q, then all frames on the stack need to have permission q

A Caller-Sensitive Method (CSM) is a Java platform method that bypasses the standard stack inspection

- The check is determined based on the immediate caller's `ClassLoader`
- E.g., `Class.forName("Foo")` is a CSM that returns the `Class` object associated with the "`Foo`" class

Gondvv in a Nutshell

a Java platform restricted package

```
private Class GetClass(String paramString) throws Throwable
{
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new
        Expression(Class.class, "forName", arrayOfObject);
    localExpression.execute();
    return (Class)localExpression.getValue();
}
```

localExpression Ξ Expression{ Class.forName("sun.awt.SunToolkit") }

2 Gondvv.GetClass(String)

1 Gondvv.SetField(Class, String, Object, Object)

Gondvv in a Nutshell

a Java platform restricted package

```
private Class GetClass(String paramString) throws Throwable
{
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new
        Expression(Class.class, "forName", arrayOfObject);
    localExpression.execute();
    return (Class)localExpression.getValue();
}
```

Expression.execute() is a JDK method (and therefore trusted)

3 Expression.execute()

2 Gondvv.GetClass(String)

1 Gondvv.SetField(Class, String, Object, Object)

The Exploit’s Stack Trace

12 Class.forName(String)
11 ClassFinder.findClass(String)
10 ClassFinder.findClass(String, ClassLoader)
9 ClassFinder.resolveClass(String, ClassLoader)
8 Expression(Statement).invokeInternal()
7 Statement.access$000(Statement)
6 Statement$2.run()
5 AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlContext)
4 Expression(Statement).invoke()
3 Expression.execute()
2 Gondvv.GetClass(String)
1 Gondvv.SetField(Class, String, Object, Object)

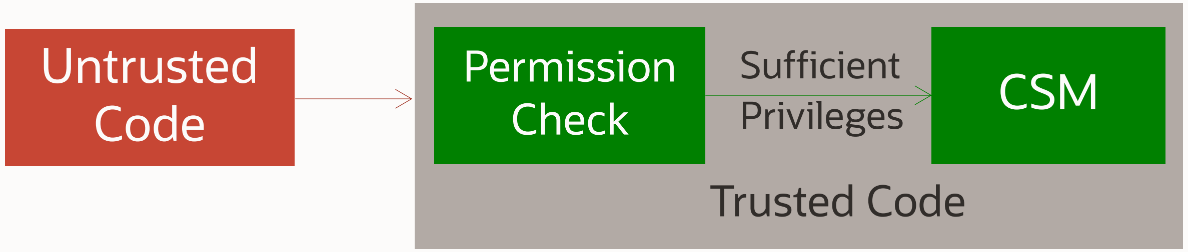
Rules to Detect Unguarded Caller-Sensitive Method Call

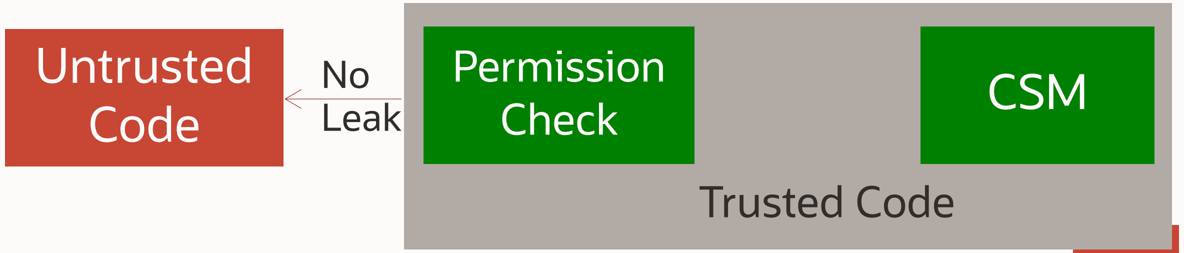
CSM is reachable from untrusted code

CSM is unprotected

One of the following holds based on CSM used

a) Taint CSM: the arguments to the CSM are tainted and not sanitised
b) Escape CSM: the CSM returns an object that is leaked to untrusted code
c) Taint-or-escape CSM: a) or b) applies
d) Taint-and-escape CSM: a) and b) applies.





Finding Spectre Variant 1 Vulnerabilities in C, C++ Code

CVE-2017-5753

Spectre (v1)
CVE-2017-5753

“Systems with microprocessors utilizing speculative execution and branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis.”

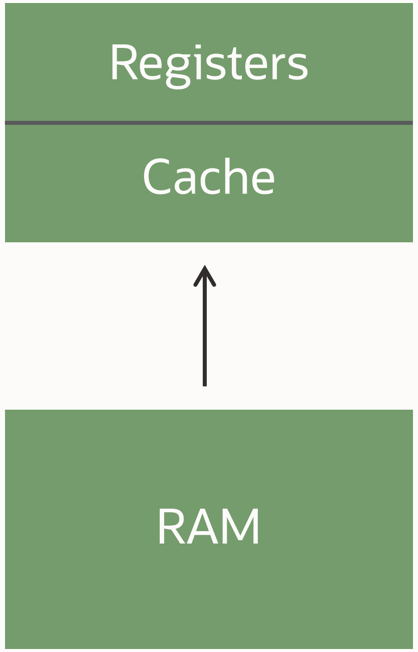
Meltdown
CVE-2017-5754

“Systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis of the data cache.”

Spectre v1 in a Nutshell

x is user controlled

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```



x in registers or cache
k= array1[x] is cached

array1_size in RAM
array2 in RAM

P. Kocher et al, Spectre Attacks: Exploiting Speculative Execution: https://spectreattack.com/spectre.pdf

Spectre v1 Pattern

x is user controlled

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

Branch-load-load

- Branch is a bounds check on first load
- Offset to second load based on first load
- No LFENCE/MEMBAR/array_index_nospec() in the pattern
- Heuristics to determine whether array2 cannot be held in one cache line

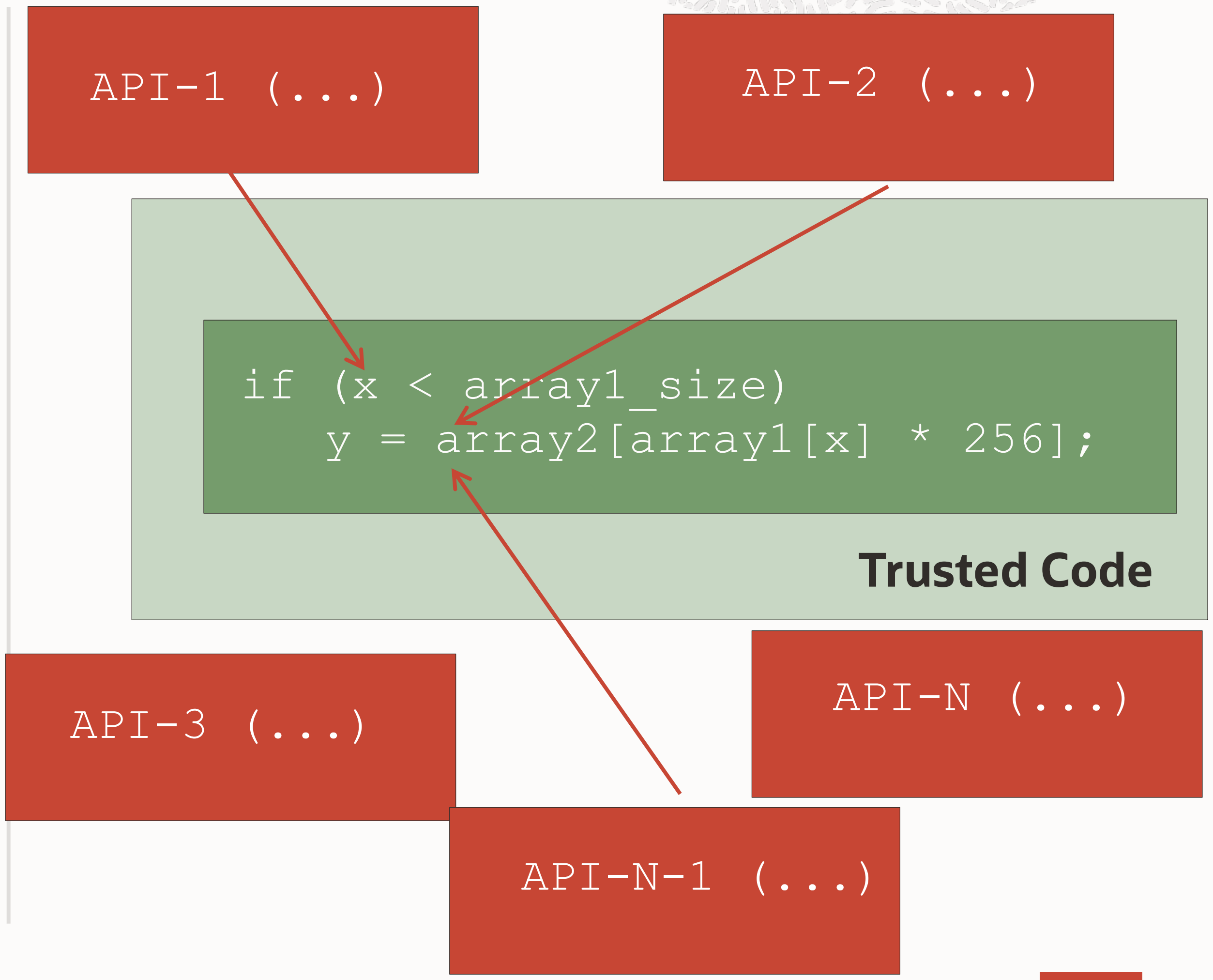
User-controllable offset to first load

Load-load is reachable from less privileged code

P. Kocher et al, Spectre Attacks: Exploiting Speculative Execution: https://spectreattack.com/spectre.pdf

Rules to Detect Spectre Variant 1

1. Identify branch-load-load pattern
2. Identify the API boundary between more and less privileged code, e.g., syscalls
3. Check (interprocedurally) for reachability and taint from the API entry points to the potential defect



Security Issues Can Arise at Any Level of Abstraction

App

Library

VM

Processor

µArch

Each level *provides* a service under certain *assumptions*

Each level *consumes* a service with certain *expectations*

Mismatch between assumptions and expectations can be exploited

Examples

- App and Library: Sanitisation not performed: SQL Injection
- Library and VM: Isolation not guaranteed: information leakage
- Processor and $\mu Arch$: Spectre/Meltdown

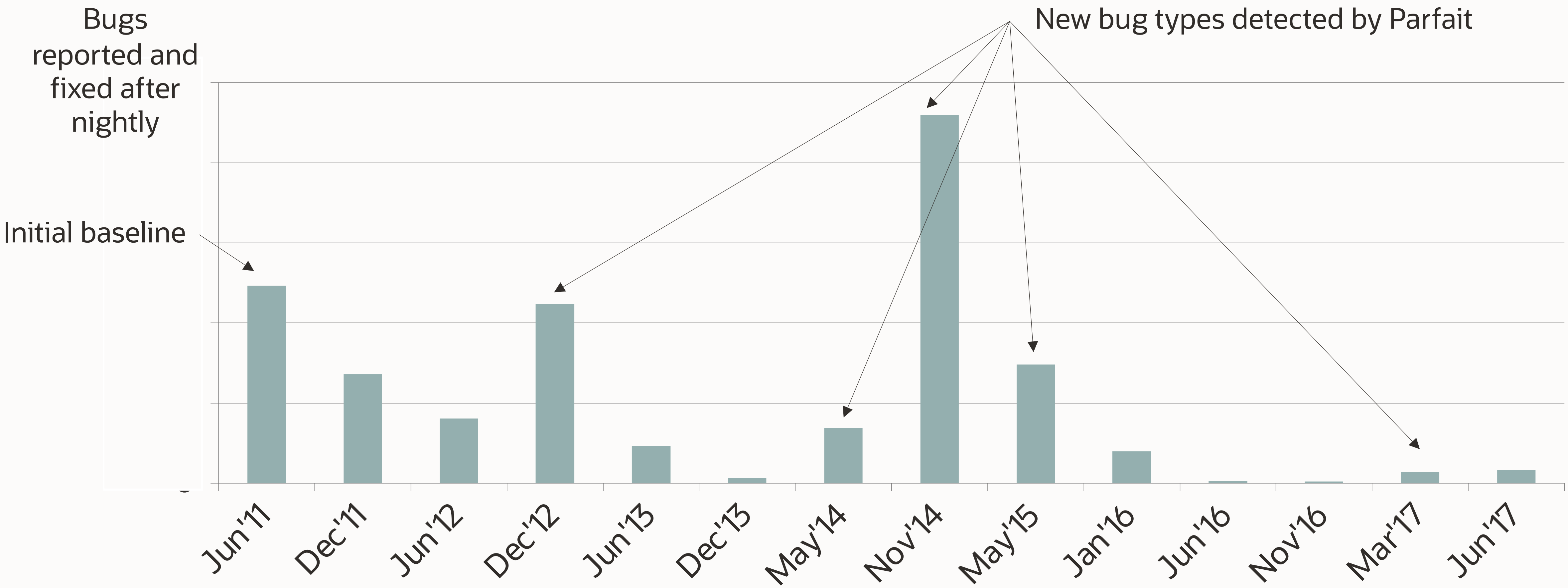
A Sample of Results

Parfait – Scalable, Deep Static Code Analysis

Codebase	Non Commented Lines of Code	Number of Bug Types	Analysis runtime	Runtime in KLOC/min
Oracle Linux Kernel 5	16,586,325 C	34	19m 20s	858 KLOC/min
Cloud service	1,216,168 Java	5	7m 2s	173 KLOC/min
Cloud service	229,000 Python	4	5m 15s	43 KLOC/min

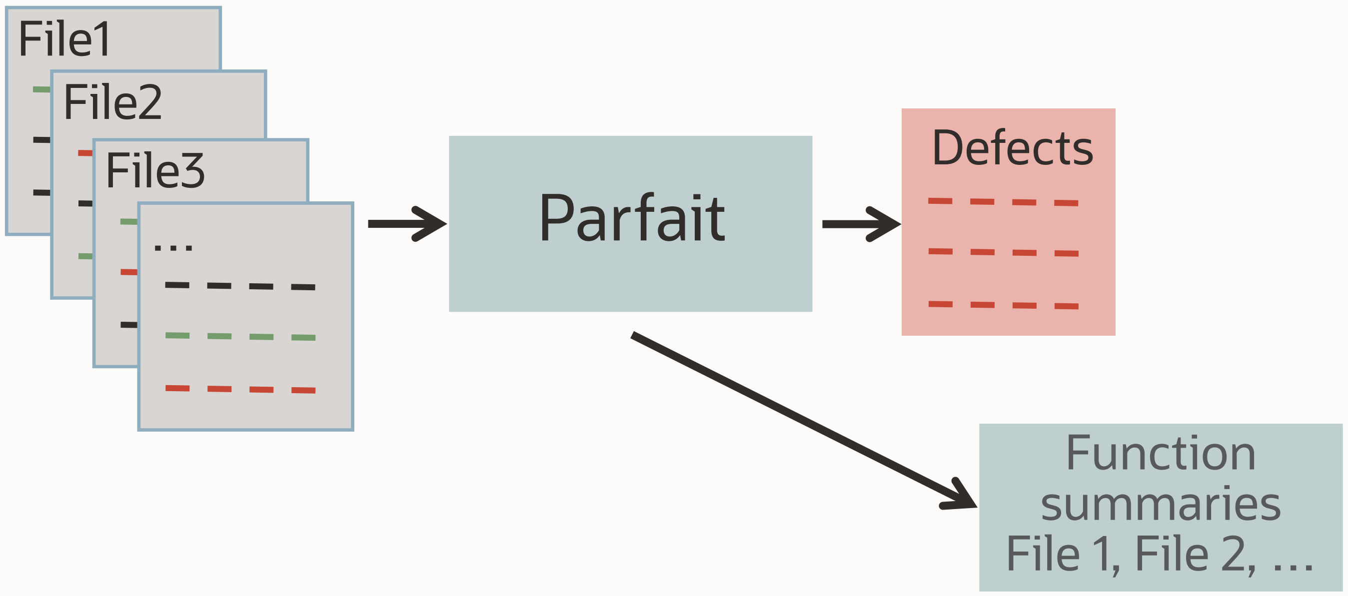
Parfait – Precise, Deep Static Code Analysis

Bugs fixed by developers once **baseline** had been established



Analysis of Full Codebase vs Analysis of Commit/Push/Pull/Merge Request

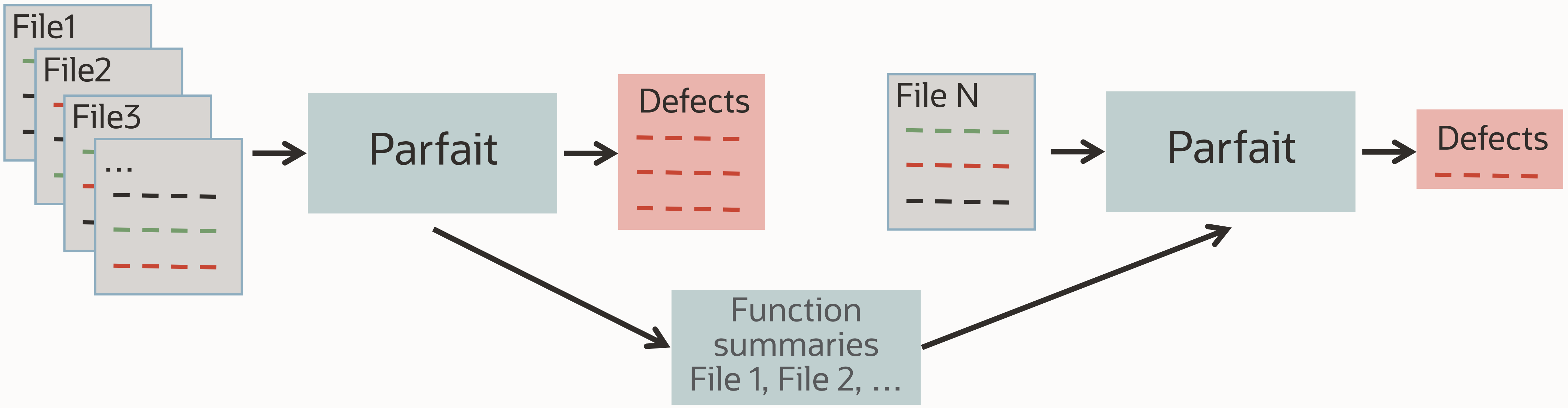
Analysis of full codebase



Analysis of Full Codebase vs Analysis of Commit/Push/Pull/Merge Request

Analysis of full codebase

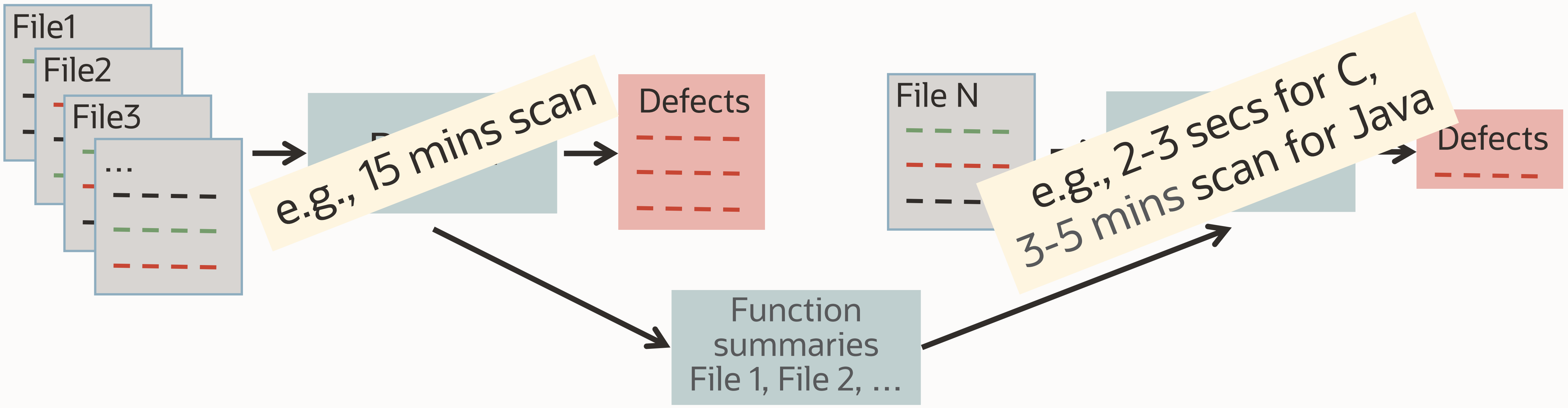
Analysis of changeset



Analysis of Full Codebase vs Analysis of Commit/Push/Pull/Merge Request

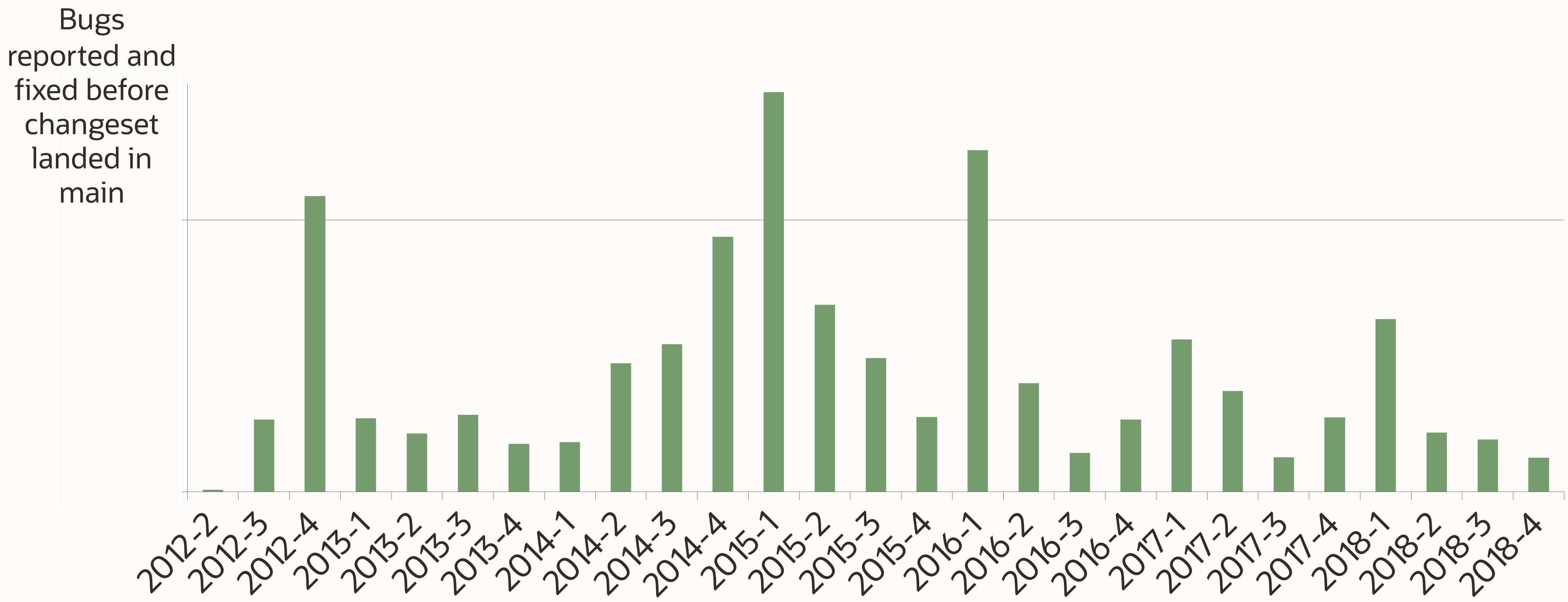
Analysis of full codebase

Analysis of changeset



Bugs Prevented from Being Introduced into the Codebase

Changeset analysis prevents **80%** of new bugs (compared to baseline)



Innovations During the Past 14 Years

Code ← Build ← Test

- Efficient analysis of full codebase
  - Used to be nightly runs
  - Now part of Continuous Integration

- Efficient analysis of changeset
  - Prevent bugs from being introduced into the codebase
  - Can be hooked into the commit, push, pull request or merge request

Parfait innovations

- Precise results
- Scalable, can integrate early in the development cycle

What About Configuring the Tool?

Using machine learning to determine sanitisers, validators and taint sinks

Configuring Parfait for Taint Analysis Information

- Taint sources, sanitisers, validators and taint sinks need to be **configured**

- Pre-made configurations for JDK, Java EE and commonly-used libraries are available in Parfait

  - Fasterxml Jackson
  - Google API Client
  - Google Guava (partial!)
  - Jsch
  - Jmustache
  - Micronaut
  - OkHttp3
  - Java Http Server
  - gRPC
  - Netty
  - Eclipse Jetty
  - Helidon
  - Eclipse Vert.x
  - Commons FileUpload
  - Commons IO
  - Commons Lang
  - Spring Framework
  - Apache Spark (partial!)
  - Apache HttpComponents
  - Apache Xerces
  - Simple Java Mail
  - Berkeley DB Java Edition API

Configuring a static analysis tool is a manual and time consuming process

Taint Using Dataflow Analysis for SQL Injection

A source of tainted data

```
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + "not found");
    else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + "was empty");
    }

    return (values[0]);
}
```

Taint Using Dataflow Analysis for SQL Injection

```
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + "not found");
    else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + "was empty");
    }

    return (values[0]);
}
```

No sanitisation of String values

Taint Using Dataflow Analysis for SQL Injection

```
protected Element createContent(WebSession s)
{
    ...
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                  "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                           connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                      ResultSet.CONCUR_READ_ONLY);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

A sink of tainted data

Taint Using Dataflow Analysis for SQL Injection

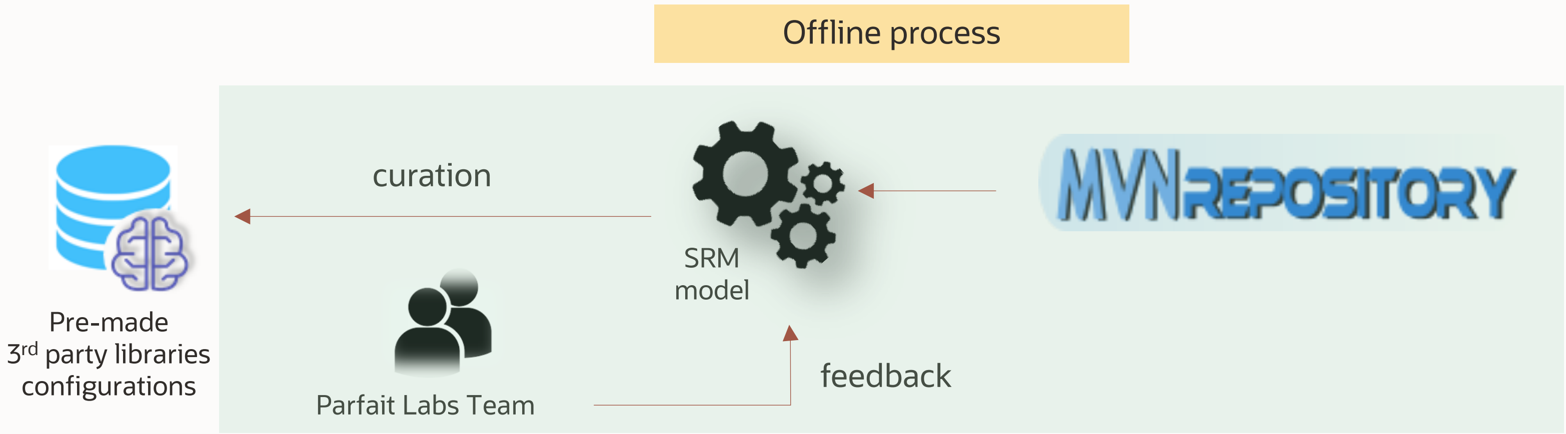
Example sanitization of the query String using the Enterprise Security API for Java

```
protected Element createContent(WebSession s)
{
    ...
    password = s.getParser().getRawParameter(PASSWORD);
    ...
    String query = "SELECT * FROM user_system_data WHERE user_name = '" + username +
                   "' and password = '" + password + "'";
    ...
    try {
        Statement statement =
                         connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                                    ResultSet.CONCUR_READ_ONLY);
        query = ESAPI.encoder().encodeForSQL (MYSQL_CODEC, query);
        ResultSet results = statement.executeQuery(query);
        ...
    }
    ...
}
```

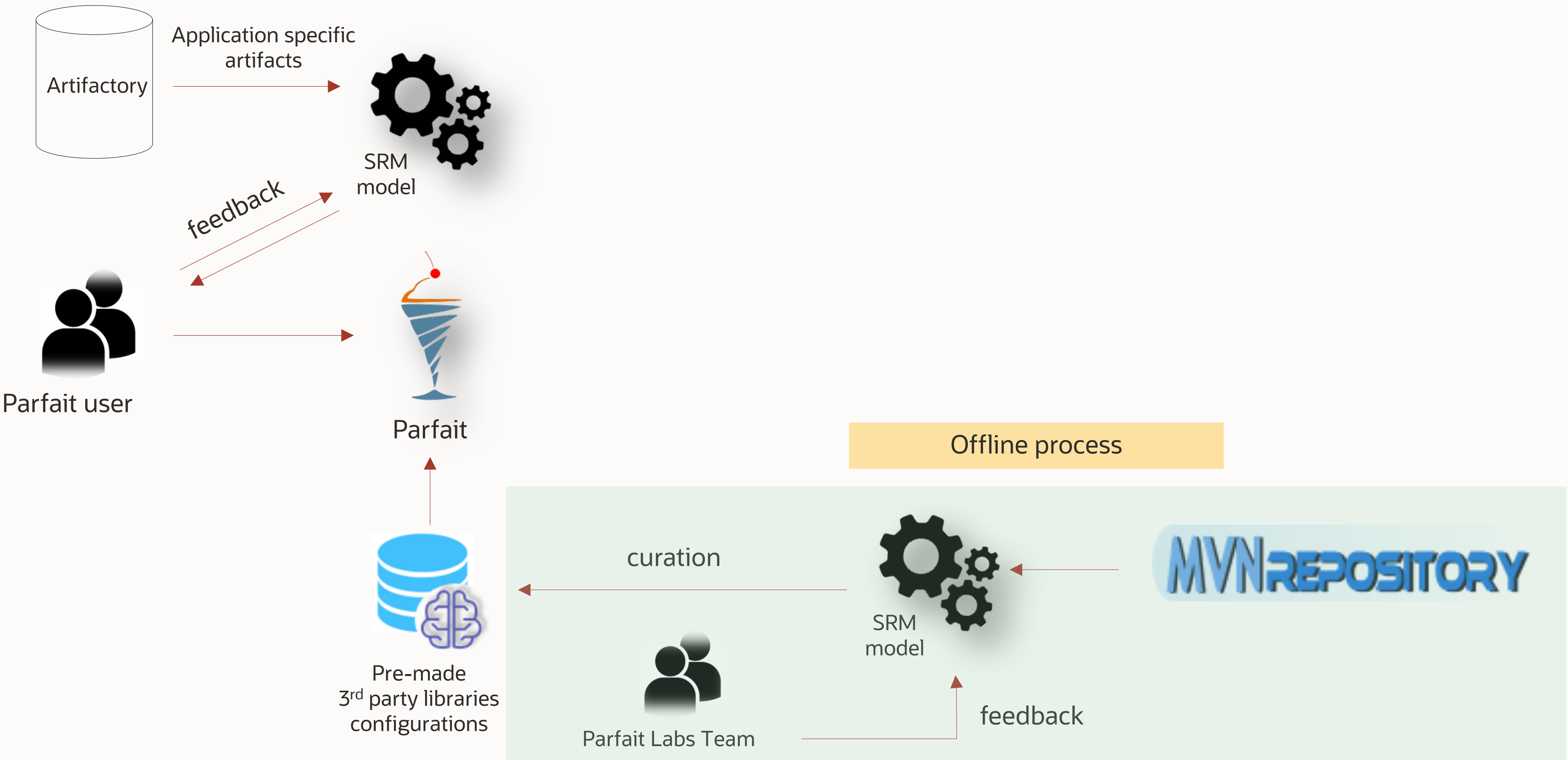
A sanitisation method for MySQL queries

https://github.com/ESAPI/esapi-java

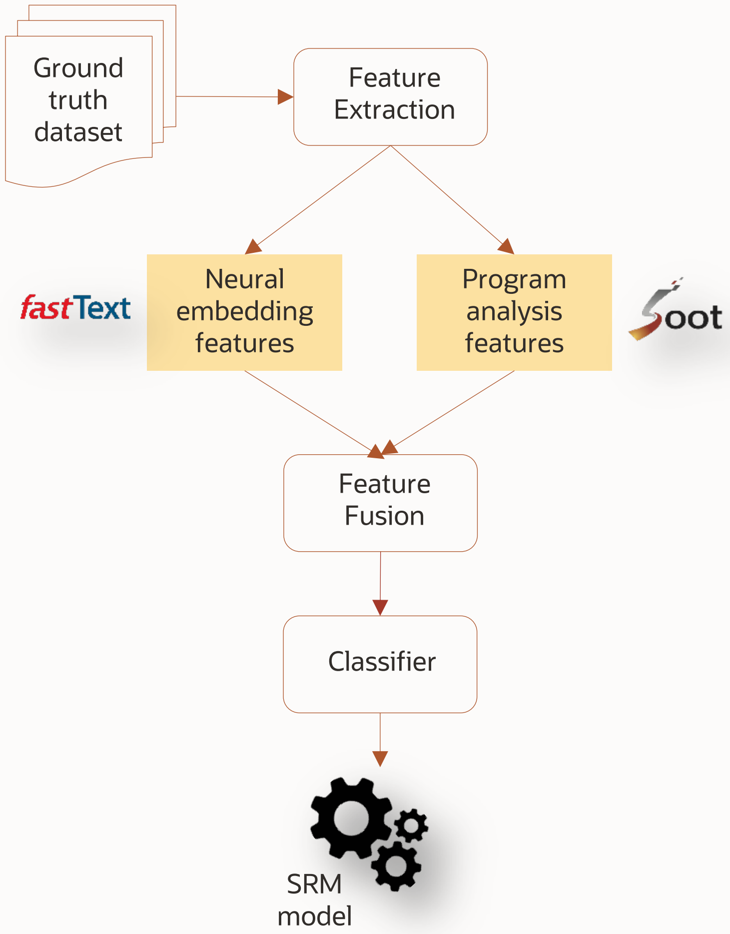
Semi-automation of Configuration Generation using Machine Learning



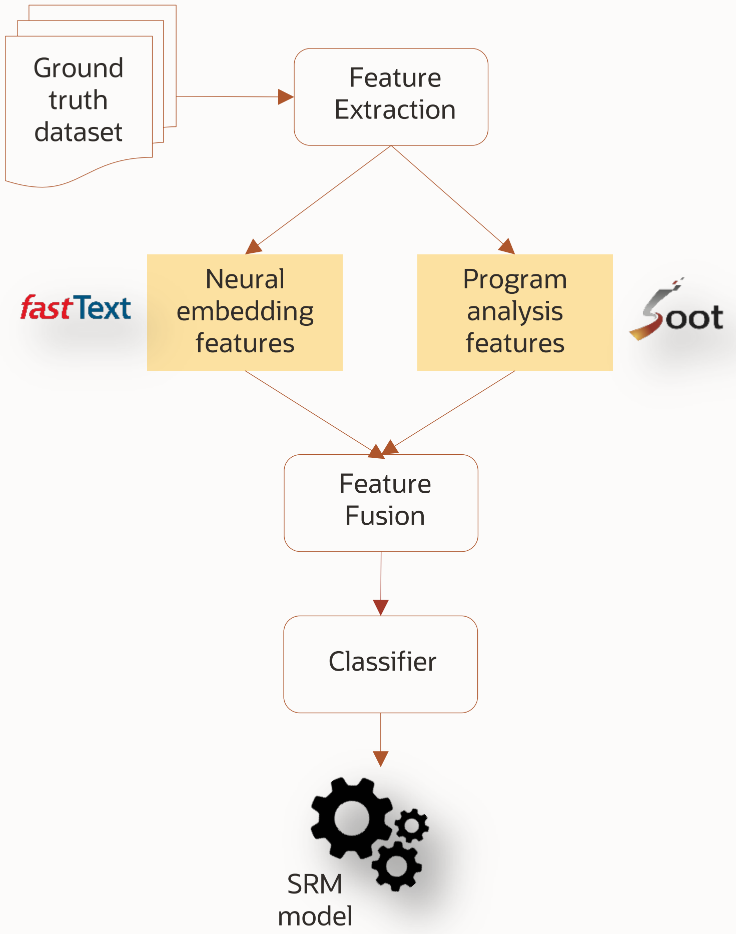
Semi-automation of Configuration Generation using Machine Learning



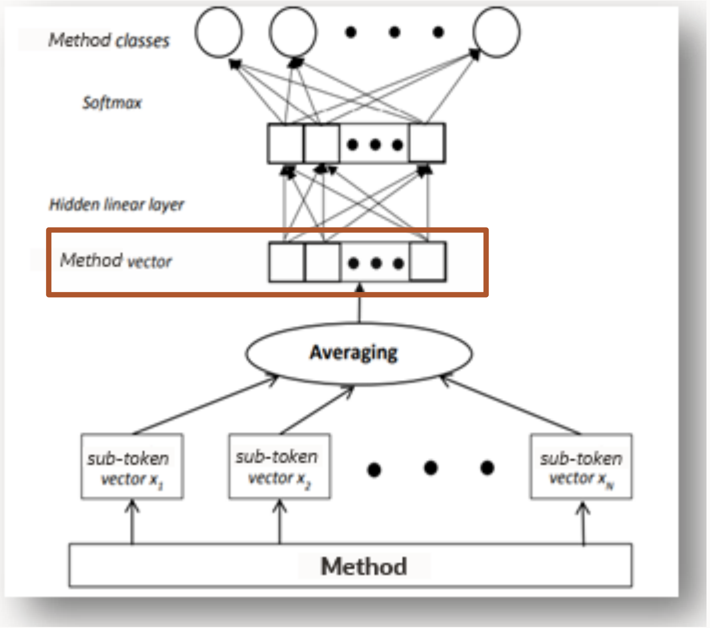
High-level Architecture



High-level Architecture



Neural embedding features



Program analysis features

- Soot based light-weight program analysis
- Intra-procedure analysis
- 83 features in total:
  - HasParam
  - HasRetType
  - RetConstant
  - ParamFlowsToRet
  - ParamFlowsToCondCheck
  - ClassHasKnownSrc
  - ClassHasKnownSink
  - …

Results – Detecting Sanitisers/Validators

Library	No. of Libraries Analyzed	Total Classes	Total Methods	Sanitisers/ Validators	False Positives	FP Rate
OCI Common Libraries	4	394	2,212	8	**4**	**50%**
Third-party Libraries	6	883	11,288	76	**27**	**36%**

Processing time: ~2mins per JAR

Results – Detecting Sinks

Library	Total methods	Sinks manually identified by Parfait team	New sinks identified by SRM
Apache Commons IO	3,933	31	11
Netty	79,539	34	16
Apache Commons Lang3	7,301	11	1
Google Guava	48,122	22	7
Total	**-**	**87**	**35**

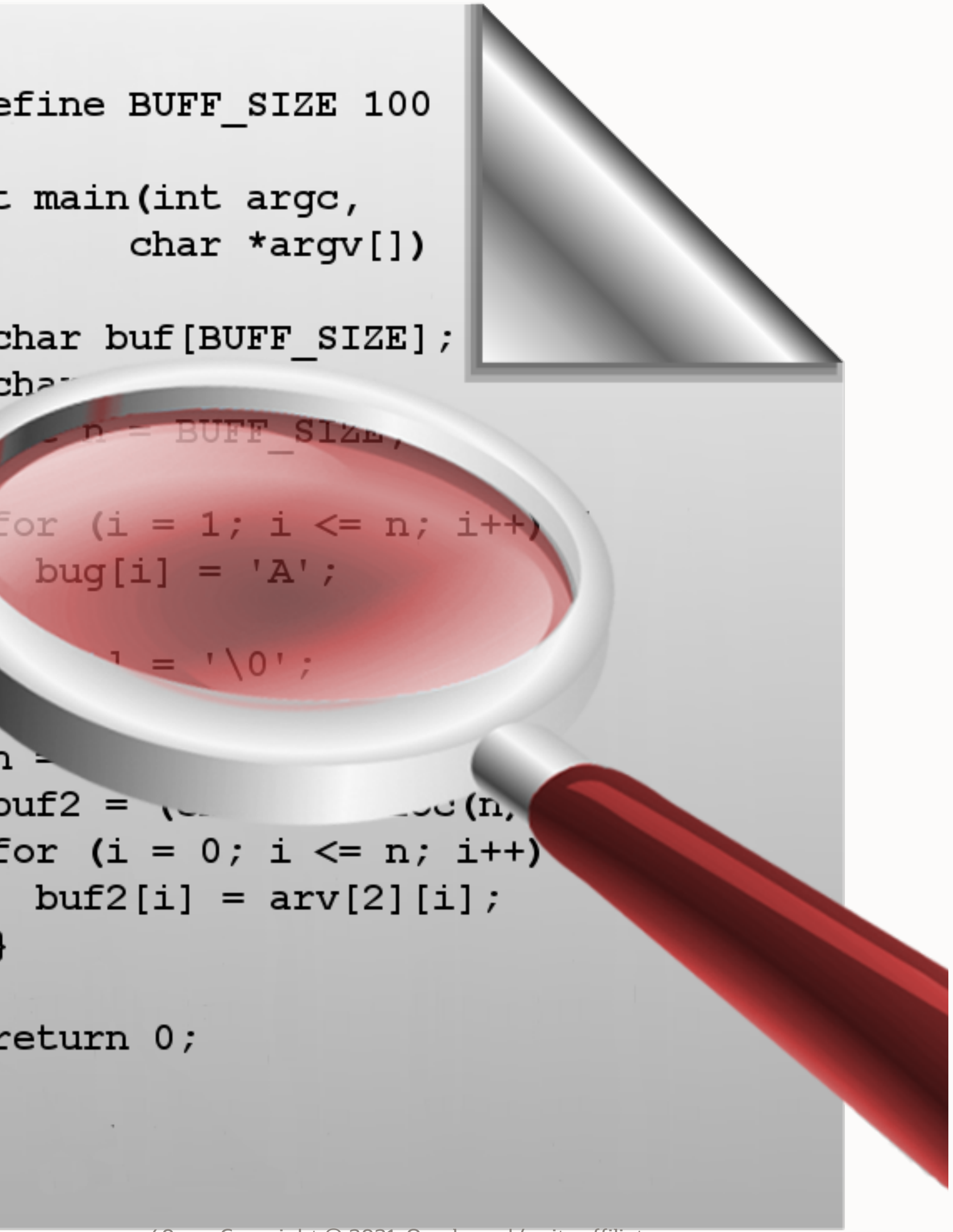
Processing time: ~2mins per JAR

2.5 hours vs
63 person days

- Semi-automated: 42 mins analysis + 108 mins manual curation for 21 libraries
- Manual: 63 person days manual curation for 21 libraries

Lessons Learnt

Analyses need to be **precise**, **scalable** and **incremental** in order to be useful to developers and practical for CI/CD integration.



Fine tuning of the analysis is best done with a team who owns their codebase and understands the vulnerability at hand.

Analyses need to easily integrate into existing build processes.

Makefile

Analyses need to explain why the tool reports a bug at a given line; i.e., provide a trace/witness.

Auto-configuration of the tool aids deployment and adoption.

Lessons Learnt
Requirements To Successfully Deploy A Static Code Analysis Tool

✓High precision (i.e., few incorrect issues)

✓Fast runtime (i.e., seconds and minutes, not hours)

✓Integration into build system

✓Explanation of results of the analysis

✓Auto-configuration

Success Metric – Large Number of Bugs Fixed By Development

Parfait – precise, scalable and incremental static analysis for C, Java and Python.

cristina.cifuentes@oracle.com

http://labs.oracle.com/locations/australia

Twitter: @criscifuentes

LinkedIn: drcristinacifuentes

Our mission is to help people see data in new ways, discover insights, unlock endless possibilities.

ORACLE