

# Challenges and Progress: Automating Static Analysis Alert Handling with Machine Learning

Lori Flynn, PhD

Software Security Researcher

Software Engineering Institute of Carnegie Mellon University

Copyright 2018 Carnegie Mellon University. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

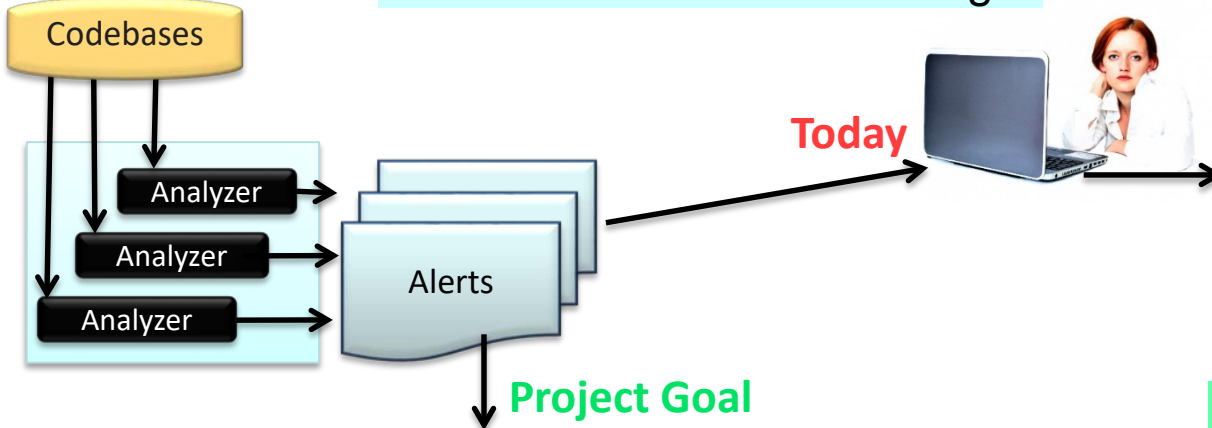
This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM18-0547

# Overview

**Problem:** too many alerts  
**Solution:** automate handling



## Project Goal

Classification algorithm development using “pre-audited” and manually-audited data, that accurately classifies most of the diagnostics as:

**Expected True Positive (e-TP)** or  
**Expected False Positive (e-FP),**  
and  
the rest as Indeterminate (I)

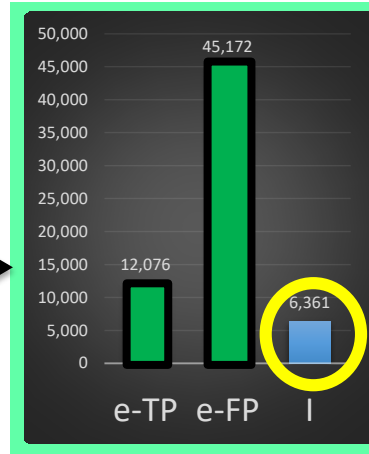
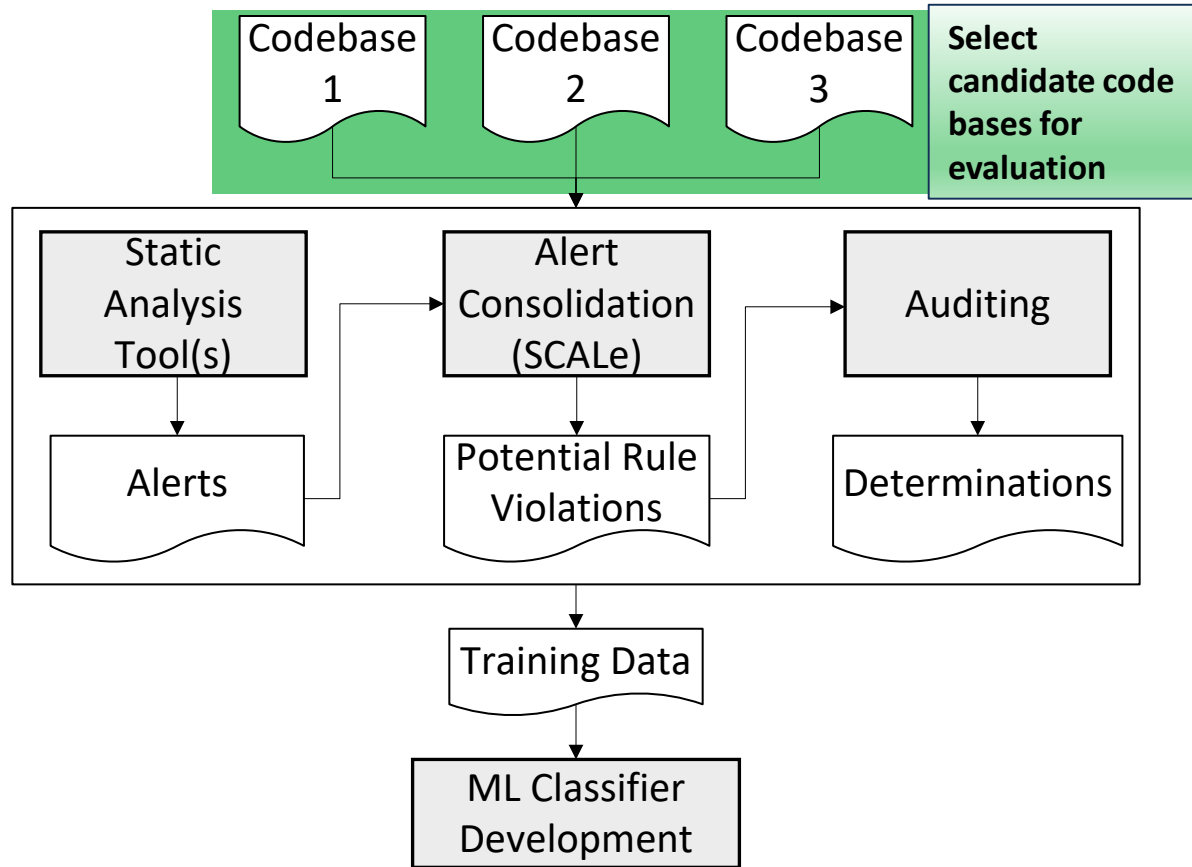
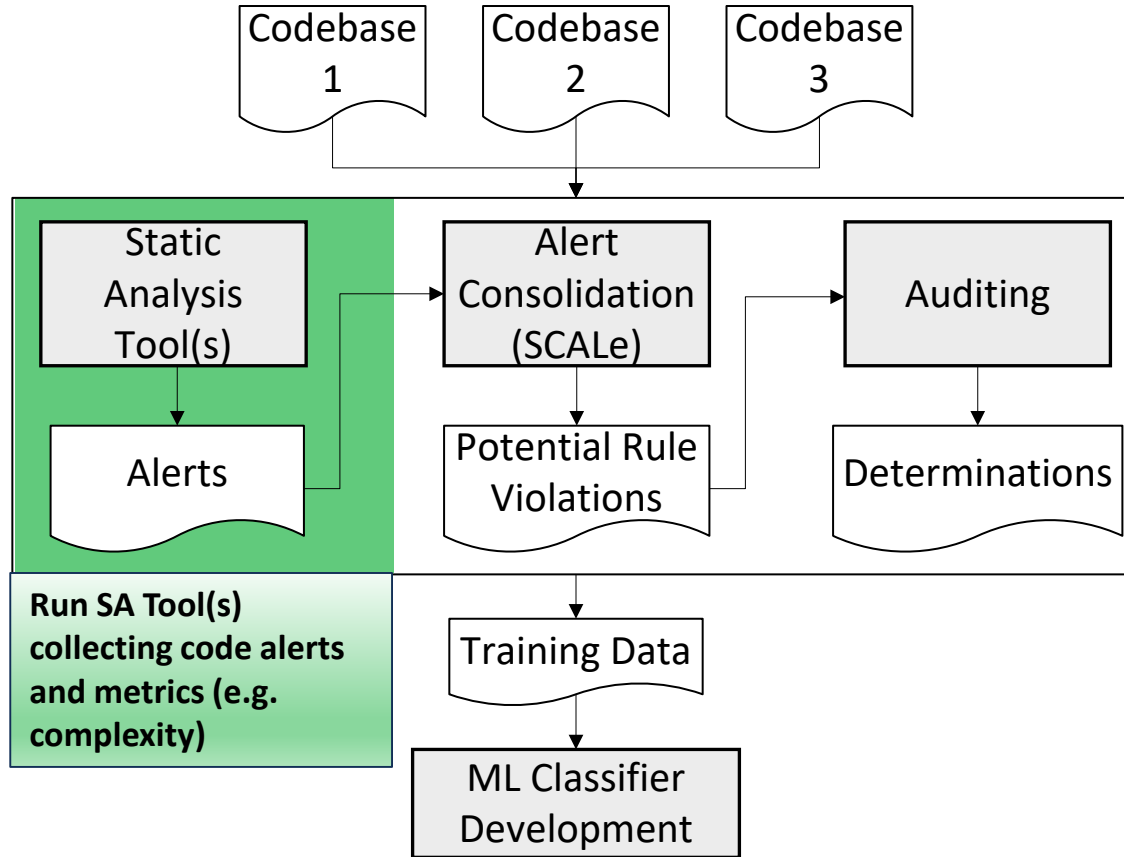


Image of woman and laptop from <http://www.publicdomainpictures.net/view-image.php?image=47526&picture=woman-and-laptop> “Woman And Laptop”

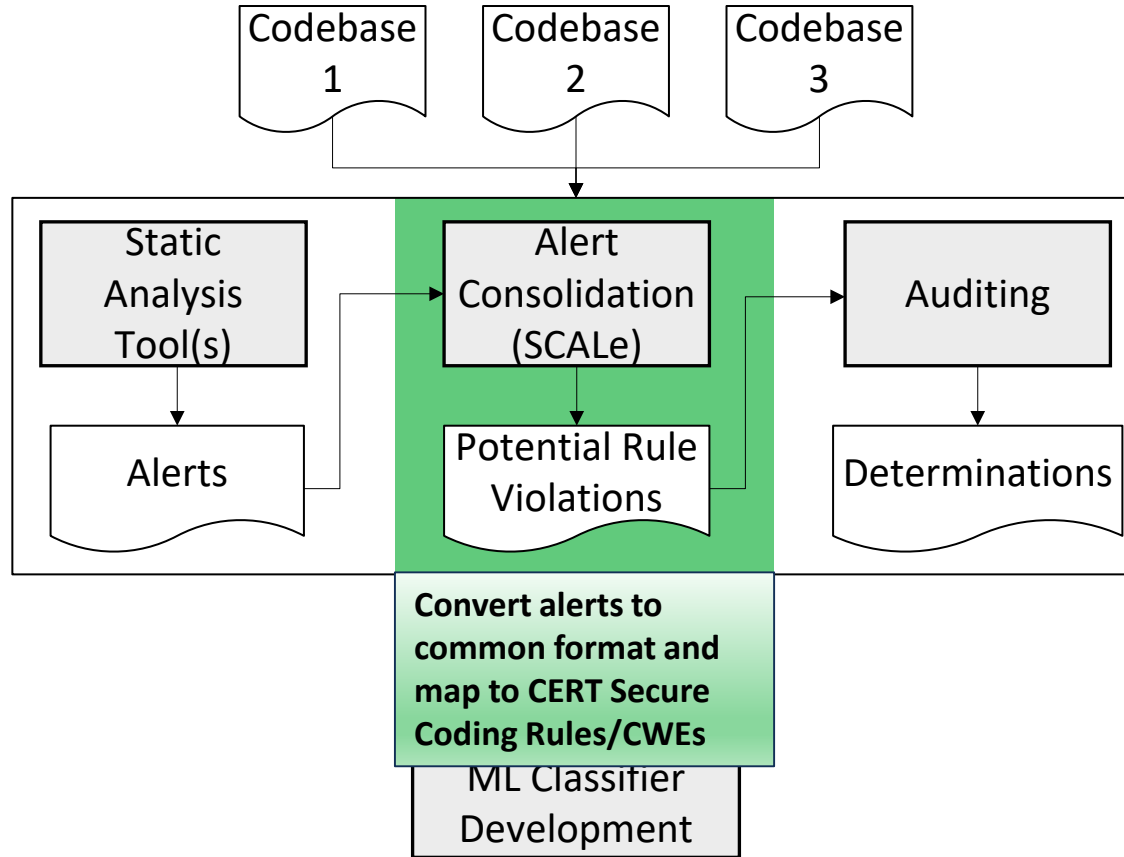
# Background: Automatic Alert Classification



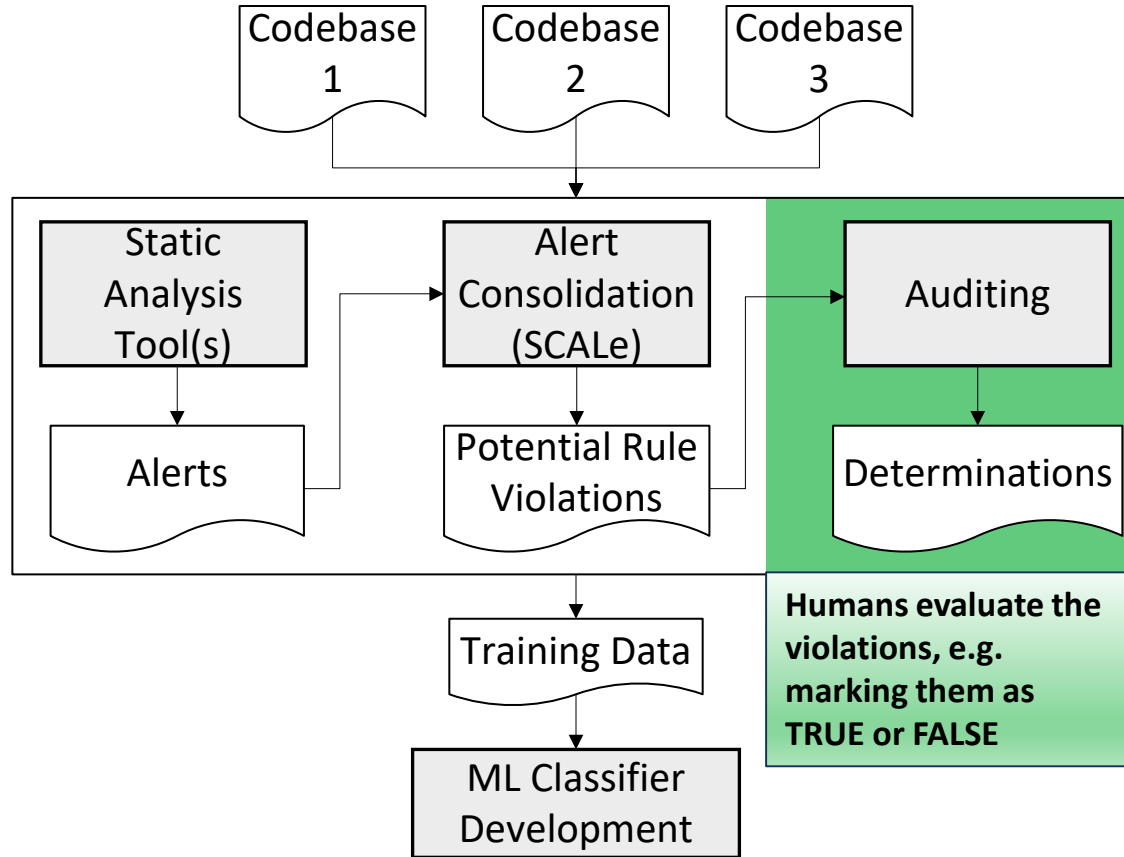
# Background: Automatic Alert Classification



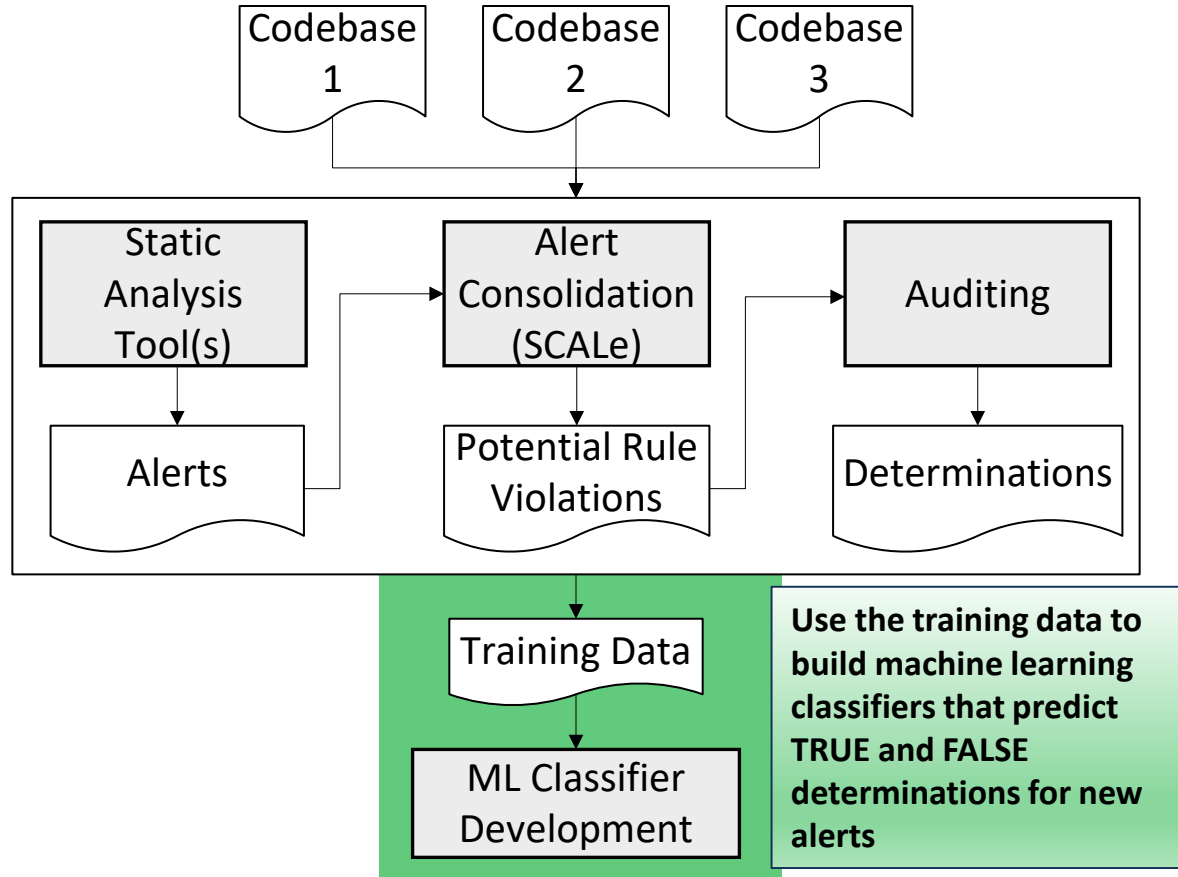
# Background: Automatic Alert Classification



# Background: Automatic Alert Classification

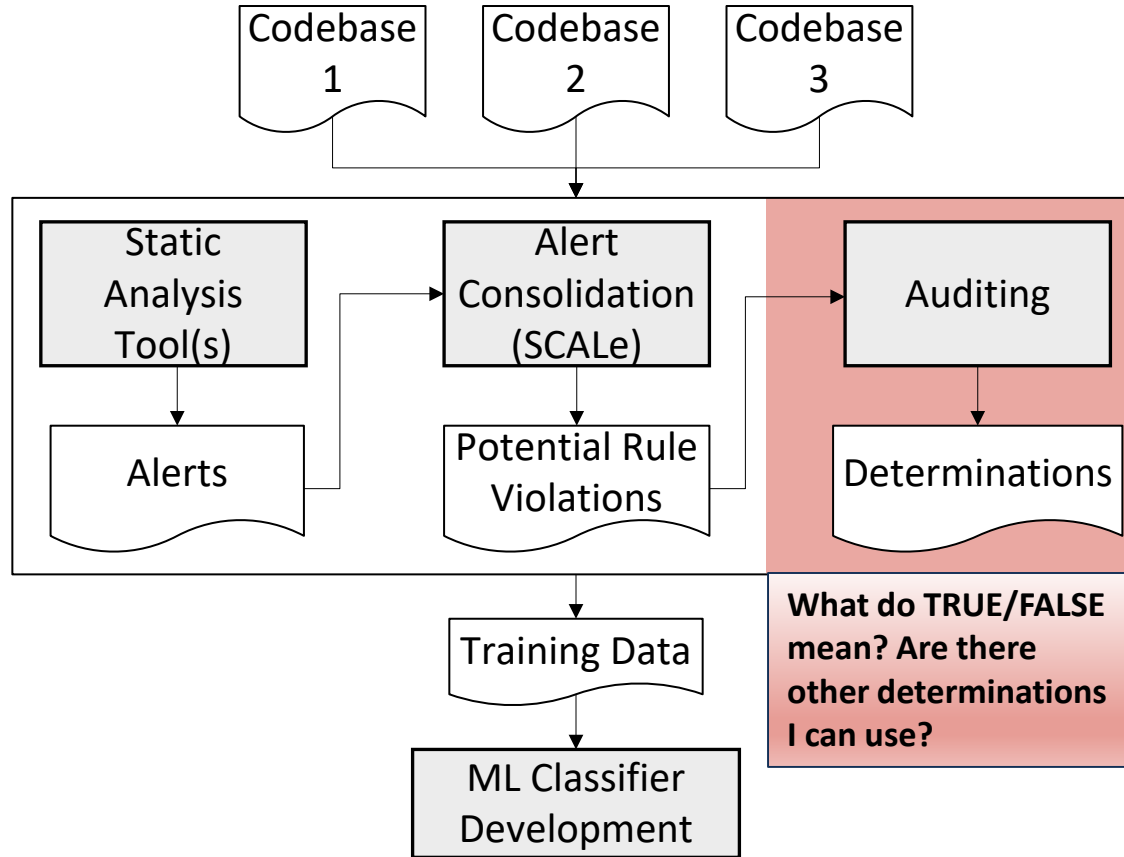


# Background: Automatic Alert Classification





# Background: Automatic Alert Classification

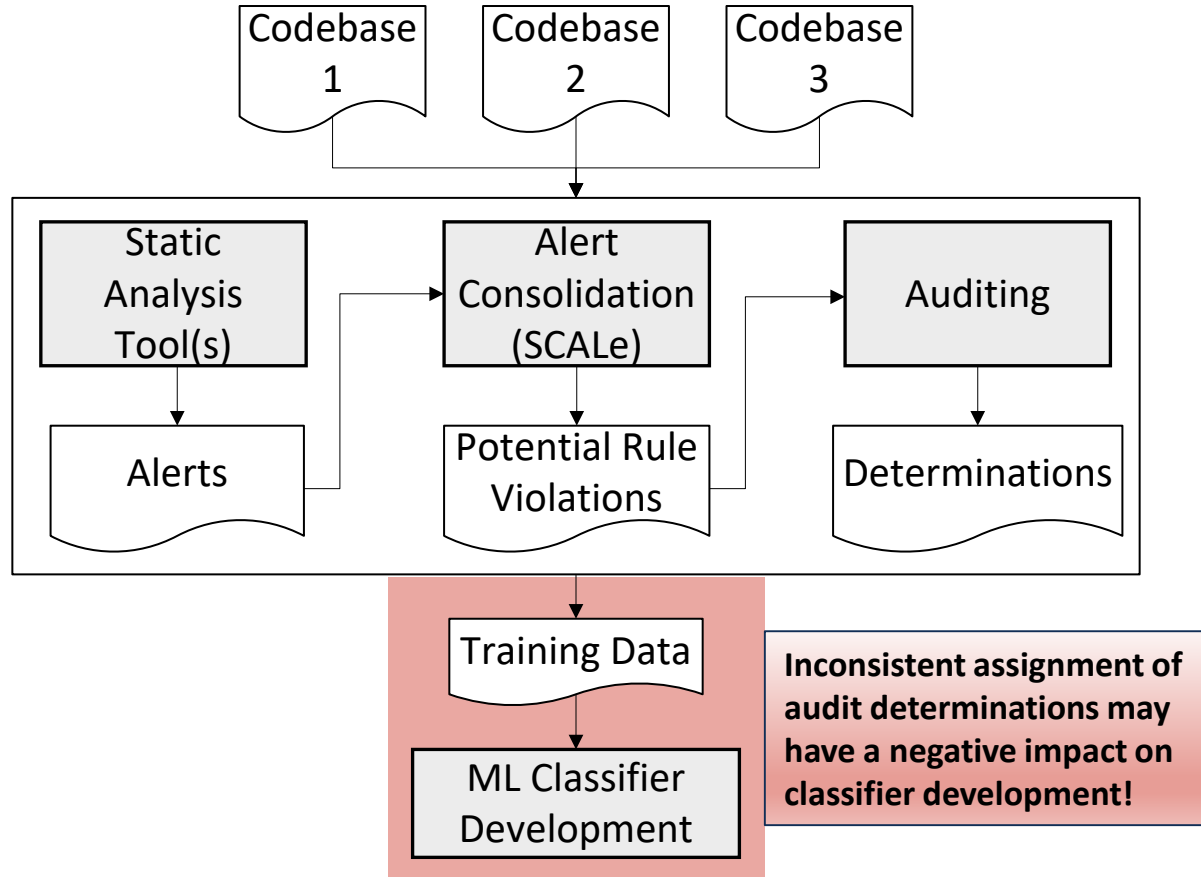


# What is truth?

One collaborator reported using the determination **True** to indicate that the issue reported by the alert was a real problem in the code.

Another collaborator used **True** to indicate that *something* was wrong with the diagnosed code, even if the specific issue reported by the alert was a **false positive!**

# Background: Automatic Alert Classification



# Solution: Lexicon And Rules

- We developed a **lexicon** and auditing **rule set** for our collaborators
- Includes a standard set of well-defined **determinations** for static analysis alerts
- Includes a set of **auditing rules** to help auditors make consistent decisions in commonly-encountered situations

**Different auditors** should make the **same determination** for a given alert!

Improve the **quality and consistency** of audit data for the purpose of building **machine learning classifiers**

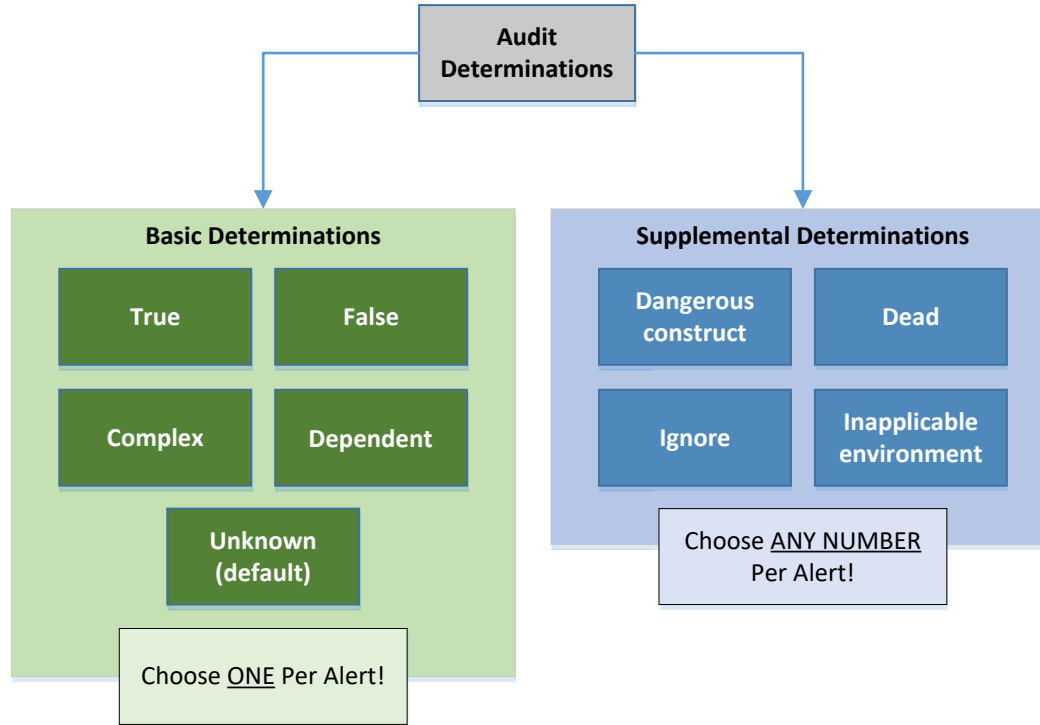
Help organizations make **better-informed** decisions about **bug-fixes, development, and future audits.**

Audit Lexicon And Rules

# Lexicon



# Lexicon: Audit Determinations



# Lexicon: Basic Determinations


## True

- The code in question violates the **condition** indicated by the alert.
- A **condition** is a constraint or property of validity.
  - E.g. A valid program should not dereference NULL pointers.
- The condition can be determined from the definition of the alert itself, or from the **coding taxonomy** the alert corresponds to.
  - CERT Secure Coding Rules
  - CWEs

# Lexicon: Basic Determinations

## True Example

```
char *build_array(size_t size, char first) {  
    if(size == 0) {  
        return NULL;  
    }  
  
    char *array = malloc(size * sizeof(char));  
    array[0] = first;  
    return array;  
}
```



**ALERT:** Do not  
dereference  
NULL  
pointers!

Determination  
:  
**TRUE**



# Lexicon: Basic Determinations

## False

- The code in question does **not** violate the **condition** indicated by the alert.

```
char *build_array(int size, char first) {  
    if(size == 0) {  
        return NULL;  
    }  
  
    char *array = malloc(size * sizeof(char));  
    if(array == NULL) {  
        abort();  
    }  
    array[0] = first;  
    return array;  
}
```

Determination  
:  
**FALSE**

**ALERT: Do not dereference NULL pointers!**

# Lexicon: Basic Determinations

## Complex

- The alert is **too difficult** to judge in a **reasonable amount of time and effort**
- “Reasonable” is defined by the individual organization.

## Dependent

- The alert is related to a **True** alert that occurs earlier in the code.
- Intuition: fixing the first alert would implicitly fix the second one.

## Unknown

- None of the above. This is the default determination.

# Lexicon: Basic Determinations

## Dependent Example

```
char *build_array(size_t size, char first, char last) {  
    if(size == 0) {  
        return NULL;  
    }  
  
    char *array = malloc(size * sizeof(char));  
    array[0] = first;  
    array[size - 1] = last;  
    return array;  
}
```

**ALERT: Do not dereference NULL pointers!**

**Determination: TRUE**

**ALERT: Do not dereference NULL pointers!**

**Determination: DEPENDENT**

# Lexicon: Supplemental Determinations

## Dangerous Construct

- The alert refers to a piece of code that poses **risk** if it is not modified.
- Risk level is specified as **High**, **Medium**, or **Low**
- Independent of whether the alert is true or false!

## Dead

- The code in question **not reachable at runtime**.

## Inapplicable Environment

- The alert does not apply to the current environments where the software runs (OS, CPU, etc.)
- If a new environment were added in the future, the alert may apply.

## Ignore

- The code in question does not require mitigation.

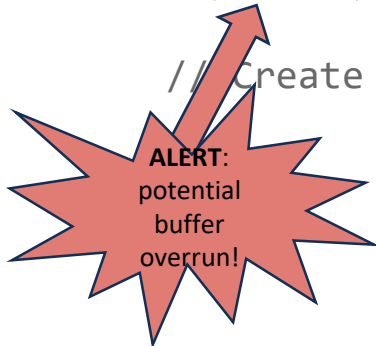
# Lexicon: Supplemental Determinations

## Dangerous Construct Example

```
#define BUF_MAX 128

void create_file(const char *base_name) {
    // Add the .txt extension!
    char filename[BUF_MAX];
    snprintf(filename, 128, "%s.txt", base_name);

    // Create the file, etc...
}
```



Seems ok...but  
why not use  
**BUF\_MAX**  
instead of 128?

Determination:  
**False**  
+  
**Dangerous  
Construct**

Audit Lexicon And Rules

# Rules



# Audit Rules

## Goals

- Clarify **ambiguous or complex** auditing scenarios
- Establish **assumptions** auditors can make
- Overall: help make audit determinations **more consistent**

## We developed **12 rules**

- Drew on our own experiences auditing code bases at CERT
- Trained 3 groups of engineers on the rules, and incorporated their feedback
- In the following slides, we will inspect three of the rules in more detail.

## Example Rule: Assume external inputs to the program are malicious

An auditor should assume that **inputs to a program module** (e.g. function parameters, command line arguments, etc.) may have arbitrary, **potentially malicious**, values.

- Unless they have a strong guarantee to the contrary

### Example from recent history: **Java Deserialization**

- Suppose an alert is raised for a call to `readObject`, citing a violation of the CERT Secure Coding Rule **SER12-J, Prevent deserialization of untrusted data**
- An auditor can assume that external data passed to the `readObject` method may be malicious, and mark this alert as **True**
  - Assuming there are no other mitigations in place in the code

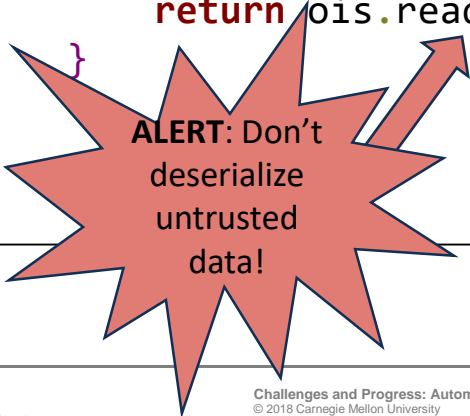


# Audit Rules

## External Inputs Example

```
import java.io.*;

class DeserializeExample {
    public static Object deserialize(byte[] buffer)
        throws Exception {
        ByteArrayInputStream bais;
        ObjectInputStream ois;
        bais = new ByteArrayInputStream(buffer);
        ois = new ObjectInputStream(bais);
        return ois.readObject();
    }
}
```



**ALERT:** Don't  
deserialize  
untrusted  
data!

Without strong  
evidence to the  
contrary, assume  
the buffer could be  
malicious!

Determination:  
**TRUE**

## Example Rule: Unless instructed otherwise, assume code must be portable.

When auditing alerts for a code base where the target platform is **not specified**, the auditor should **err on the side of portability**.

If a diagnosed segment of code **malfunctions on certain platforms**, and in doing so violates a condition, this is suitable justification for marking the alert **True**.

# Audit Rules

## Portability Example

```
int strcmp(const char *str1, const char *str2) {  
    while(*str1 == *str2) {  
        if(*str1 == '\0')  
            return 0;  
        str1++;  
        str2++;  
    }  
    if(*str1 < *str2)  
        return -1;  
    else {  
        return 1;  
    }  
}
```

ALERT: Cast to unsigned char before comparing!

This code would be safe on a platform where chars are unsigned, but that hasn't been guaranteed!

Determination:  
**TRUE**

## Example Rule: Handle an alert in unreachable code depending on whether it is exportable.

Certain code segments may be **unreachable** at runtime. Also called **dead code**.

A static analysis tool might not be able to realize this, and **still mark alerts** in code that **cannot be executed**.

The **Dead** supplementary determination can be applied to these alerts.

However, an auditor should **take care** when deciding if a piece of code is truly dead.

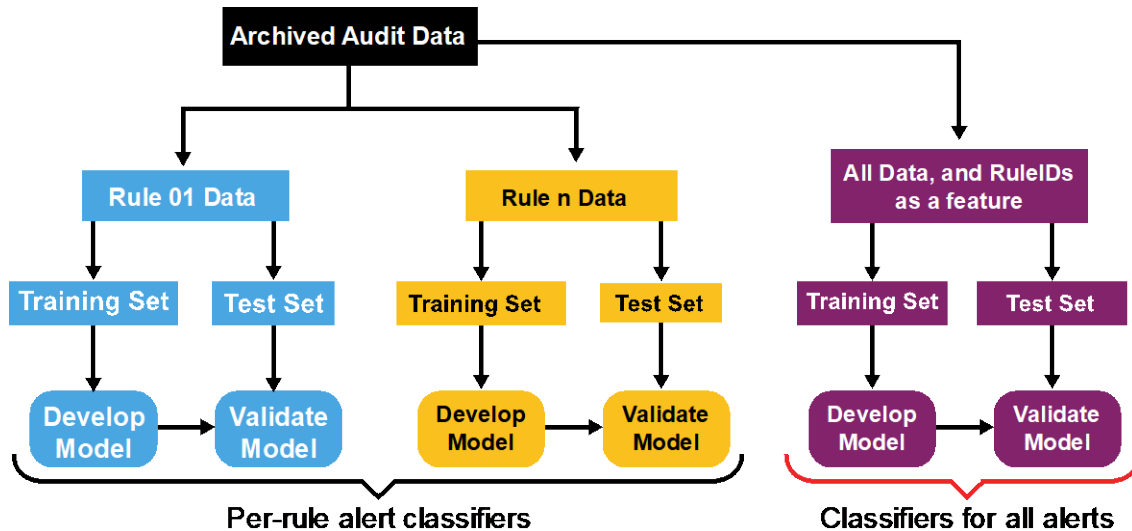
In particular: just because a given program module (function, class) is not used does **not** mean it is dead. The module might be exported as a **public interface**, for use by another application.

This rule was developed as a result of a scenario encountered by one of our collaborators!

# Scientific Approach

Build on novel (in FY16) combined use of:

- 1) multiple analyzers, 2) variety of features,
- 3) competing classification techniques



**Problem:** too many alerts  
**Solution:** automate handling

## Competing Classifiers to Test

Lasso Logistic Regression

CART (Classification and Regression Trees)

Random Forest

Extreme Gradient Boosting (XGBoost)

## Some of the features used (many more)

Analysis tools used

Significant LOC

Complexity

Coupling

Cohesion

SEI coding rule

# Rapid Expansion of Alert Classification

**Problem 1:** too many alerts  
**Solution 1:** automate handling

## Problem 2

Too few manually audited alerts to make classifiers (i.e., to automate!)

**Problems 1 & 2:** Security-related code flaws detected by static analysis require too much manual effort to triage, plus it **takes too long to audit enough alerts to develop classifiers to automate the triage accurately for many types of flaws.**

Extension of our previous alert classification work to address challenges:

1. Too few audited alerts for accurate classifiers for many flaw types
2. Manually auditing alerts is expensive

## Solution 2

Automate auditing alerts, using test suites

**Solution for 1 & 2:** Rapid expansion of number of classification models by using “pre-audited” code, plus collaborator audits of DoD code.

## Approach

1. Automated analysis of “pre-audited” (not by SEI) tests to gather sufficient code & alert feature info for classifiers
2. Collaboration with MITRE: Systematically map CERT rules to CWE IDs in subsets of “pre-audited” test code (known true or false for CWE)
3. Modify SCALE research tool to integrate CWE (MITRE’s Common Weakness Enumeration)
4. Test classifiers on alerts from real-world code: DoD data

# Overview: Method, Approach, Validity

**Problem 2:** too few manually audited alerts to make accurate classifiers for many flaw types

**Solution 2:** automate auditing alerts, using test suites

Rapidly create **many** coding-rule-level classifiers for static analysis alerts, then use DoD-audited data to validate the classifiers.

## Technical methods:

- Use test suites' CWE flaw metadata, to quickly and automatically generate many “audited” alerts.
  - o Juliet (NSA CAS) 61,387 C/C++ tests
  - o IARPA's STONESOUP: 4,582 C tests
  - o Refine test sets for rules: use **mappings, metadata, static analyses**
- Metrics analyses of test suite code, to get feature data
- Use DoD-collaborator enhanced-SCALe audits of their own codebases, to validate classifiers. **Real codebases with more complex structure than most pre-audited code.**

# Make Mappings Precise

**Problem 2:** too few manually audited alerts to make classifiers

**Solution 2:** automate auditing alerts, using test suites

**Problem 3:** Test suites in different taxonomies (most use CWEs)

**Solution 3:** Precisely map between taxonomies, then partition tests using precise mappings

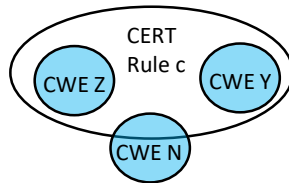
**Precise mappings:** Defines *what kind* of non-null relationship, and if overlapping, *how*. Enhanced-precision added to “imprecise” mappings.

Imprecise mappings  
(“some relationship”)



Precise mappings  
(set notation, often more)

2 CWEs subset of CERT rule,  
AND partial overlap



Mappings	
Precise	248
Imprecise TODO	364
<b>Total</b>	<b>612</b>

Now: all CERT C rules  
mappings to CWE precise

If a **condition** of a program violates a CERT rule  $R$  and also exhibits a CWE weakness  $W$ , that **condition** is in the overlap.



# Test Suite Cross-Taxonomy Use

Partition sets of thousands of tests relatively quickly.

Examine together:

- Precise mapping
- Test suite metadata (structured filenames)
- Rarely examine small bit of code (variable type)

## CWE test programs useful to test CERT rules

STONESOUP: **2,608** tests

Juliet: **80,158** tests

- Test set partitioning incomplete (32% left)

Some types of CERT rule violations not tested, in partitioned test suites (“**0**”s).

- Possible coverage in other suites

**Problem 3:** Test suites in different taxonomies (most use CWEs)

**Solution 3:** Precisely map between taxonomies, then partition tests with precise mappings

CERT rule	CWE	Count files that match
ARR38-C	CWE-119	0
ARR38-C	CWE-121	6,258
ARR38-C	CWE-122	2,624
ARR38-C	CWE-123	0
ARR38-C	CWE-125	0
ARR38-C	CWE-805	2,624
INT30-C	CWE-190	1,548
INT30-C	CWE-191	1,548
INT30-C	CWE-680	984
INT32-C	CWE-119	0
INT32-C	CWE-125	0
INT32-C	CWE-129	0
INT32-C	CWE-131	0
INT32-C	CWE-190	3,875
INT32-C	CWE-191	3,875
INT32-C	CWE-20	0
INT32-C	CWE-606	0
INT32-C	CWE-680	984

# Process

Generate data for Juliet

Generate data for STONESOUP

Write classifier development and testing scripts

Build classifiers

- Directly for CWEs
- Using partitioned test suite data for CERT rules

Test classifiers

**Problem 1:** too many alerts

**Solution 1:** automate handling

**Problem 2:** too few manually audited alerts to make classifiers accurate for some flaws

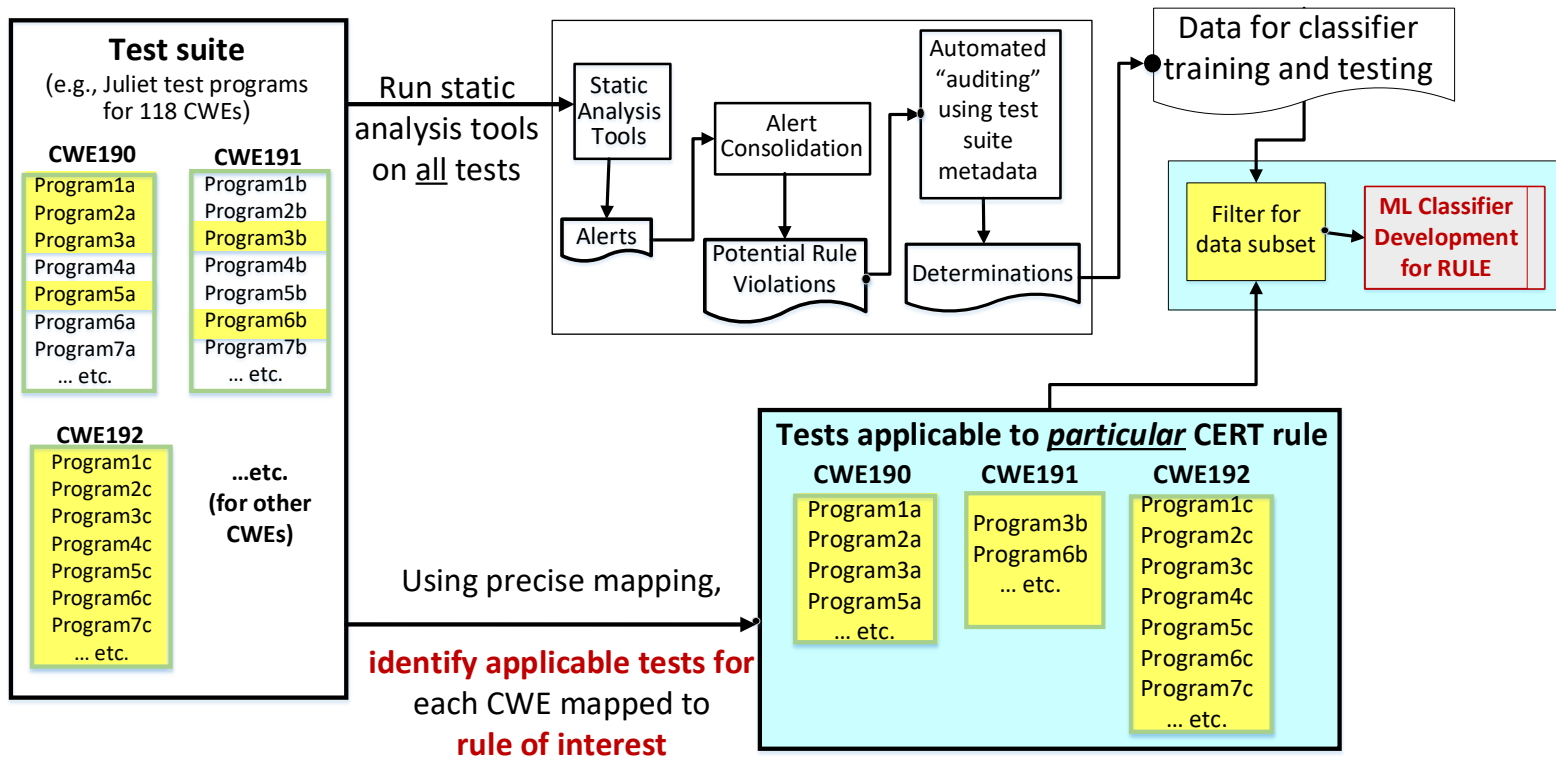
**Solution 2:** automate auditing alerts, using test suites

**Problem 3:** Test suites in different taxonomies (most use CWEs)

**Solution 3:** Precisely map between taxonomies, then partition tests using precise mappings

# Using CWE Test Suites for Multi-Taxonomy Classifiers

One time, develop data for classifiers. Per rule or CWE classifier, filter data.



# Analysis of Juliet Test Suite: Initial CWE Results

## - We automated defect identification of Juliet flaws with location 2 ways

- A Juliet program tells about only one type of CWE
- Bad functions definitely have that flaw
- Good functions definitely don't have that flaw
- Function line spans, for FPs
- Exact line defect metadata, for TPs

Number of "Bad" Functions	103,376
Number of "Good" Functions	231,476

## - Used static analysis tools on Juliet programs

## - We automated alert-to-defect matching

- Ignore unrelated alerts (other CWEs) for program
- Alerts give line number

	Tool A	Tool B	Tool C	Tool D	Total
"Pre-audited" TRUE	1,655	162	7,225	16,958	<b>26,000</b>
"Pre-audited" FALSE	8,539	3,279	2,394	23,475	<b>37,687</b>

## - We automated alert-to-alert matching (alerts fused: same line & CWE)

**Lots of new data for creating classifiers!**

Alert Type	Equivalence Classes: (EC counts a fused alert once)	Number of Alerts Fused (from different tools)
TRUE	<b>22,885</b>	3,115
FALSE	<b>29,507</b>	8,180

## - These are initial metrics (more EC as use more tools, STONESOUP)

# Juliet: Data from 4 Tools, per CWE

35 CWEs with **at least 5 HCFPs** and 45 HCTPs

More data to be added

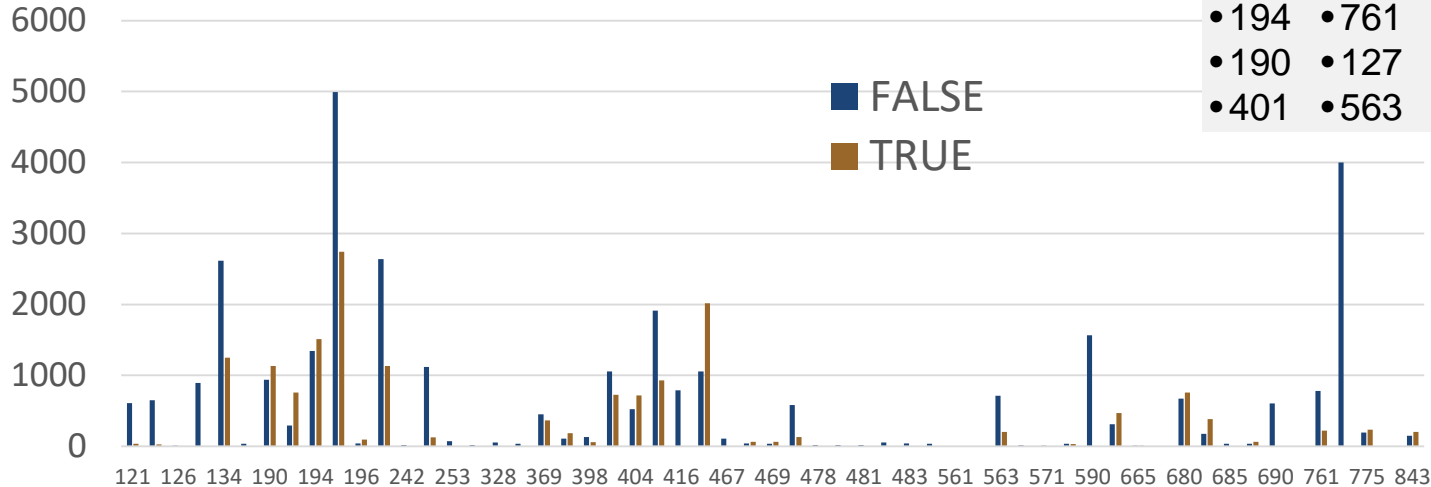
- Tools
- STONESOUP

Classifier development  
requires  
True and False

Successfully generated lots of data for classifiers

## The 35 CWEs

- 457 •680 •252 •843 •483
- 195 •404 •369 •377 •126
- 197 •415 •606 •398 •835
- 134 •665 •122 •196
- 758 •191 •121 •468
- 194 •761 •681 •469
- 190 •127 •476 •688
- 401 •563 •775 •587



# Classifiers: Accuracy, #Alerts, AUROC

Major improvement: 67 per-rule classifiers (and more coming!) vs. only 3 in FY16!

Rule	Accuracy	# Alerts	AUROC
ARR30-C	96.9%	483	99.8%
ARR32-C	100.0%	947	100.0%
ARR36-C	63.3%	30	50.0%
ARR37-C	74.0%	77	83.6%
ARR38-C	94.0%	397	98.0%
ARR39-C	67.7%	31	50.0%
CON33-C	100.0%	88	100.0%
ERR33-C	91.2%	376	94.9%
ERR34-C	100.0%	947	100.0%
EXP30-C	100.0%	947	100.0%
EXP33-C	89.5%	5214	96.3%
EXP34-C	91.8%	546	95.4%
EXP39-C	70.7%	116	83.1%
EXP46-C	82.5%	143	87.8%
FIO30-C	86.5%	1065	95.1%
FIO34-C	72.5%	1132	78.5%
FIO42-C	83.9%	933	93.2%
FIO46-C	100.0%	947	100.0%

Rule	Accuracy	# Alerts	AUROC
FIO47-C	86.4%	1070	95.4%
FLP32-C	100.0%	947	100.0%
FLP34-C	70.5%	3619	78.0%
INT30-C	63.7%	1365	66.4%
INT31-C	68.7%	5139	77.5%
INT32-C	69.9%	1599	75.7%
INT33-C	79.8%	228	86.3%
INT34-C	100.0%	947	100.0%
INT35-C	64.3%	622	72.2%
INT36-C	100.0%	967	100.0%
MEM30-C	94.5%	1461	99.3%
MEM31-C	83.9%	933	93.2%
MEM35-C	66.7%	2514	76.0%
MSC37-C	100.0%	947	100.0%
POS54-C	90.0%	239	94.5%
PRE31-C	97.8%	46	99.1%
STR31-C	94.0%	397	98.0%
WIN30-C	95.6%	1465	97.8%

Model	Accuracy	AUROC
lightgbm	83.7%	93.8%
xgboost	82.4%	92.5%
rf	78.6%	86.3%
lasso	82.5%	92.5%

← All-data CWE classifiers

← Lasso per-CERT-rule classifiers (36)

Avg. accuracy	Count accuracy 95+%	Count accuracy 85-94.9%	Count accuracy 0-84.9%
85.8%	12	9	15
	99.2%	90.9%	72.1%

Similar for other classifier methods

Lasso per-CWE-ID classifiers (31)

Avg. accuracy	Count accuracy 95+%	Count accuracy 85-94.9%	Count accuracy 0-84.9%
81.8%	7	10	14
	98.4%	89.6%	67.9%

Similar for other classifier methods

# Summary and Future

FY17 Line “Rapid Classifiers” built on the FY16 LENS “Prioritizing vulnerabilities”.

- Developed widely useful general method to use test suites across taxonomies
- Developed large archive of “pre-audited” alerts
  - Overcame challenge to classifier development
  - For CWEs and CERT rules
- Developed code infrastructure (extensible)
- In-progress:
  - Classifier development and testing in process
  - Continue to gather data
  - Enhanced SCALe audit tool for collaborator testing: distribute to collaborators soon
- **FY18-19 plan:** architecture for rapid deployment of classifiers in varied systems
- **Goal: improve automation of static alert auditing** (and other code analysis and repair)

## Publications:

- New mappings (CWE/CERT rule): MITRE and CERT websites
- IEEE SecDev 2017 “Hands-on Tutorial: Alert Auditing with Lexicon & Rules”
- SEI blogposts on classifier development
- Research papers (SQUADE’18), others in progress

# Ideas for collaboration welcome

Collaborative work topics might include:

- Continuous integration:
  - Optimizing alert analysis of developing project over time
  - Modifications to previously-developed techniques
- Enhancements to algorithms/architecture, to enable more widespread use
- ??



# Contact Information

## Presenter / Point(s) of Contact

Lori Flynn (Principal Investigator)  
Software Security Researcher

Email: [lflynn@cert.org](mailto:lflynn@cert.org)

Telephone: +1 412.268.7886

## Contributors

### SEI Staff

William Snavelly  
David Svoboda  
Zach Kurtz

### SEI Student Interns

Lucas Bengtson (CMU)  
Charisse Haruta (CMU)  
Baptiste Vauthey (CMU)  
Michael Spece (Pitt)  
Christine Baek (CMU)

# Future Work

- Gather feedback on our lexicon and rules from surveys, focus groups, experts, etc.
- Continue to refine the lexicon/rules.
- Further develop CERT's SCALe auditing framework to fully incorporate these concepts.
- Work with more collaborators to test the rules/lexicon in practice.
  - We have some initial feedback from two collaborators, who used our rules to audit several hundred alerts from C and Java codebases