Automating Static Analysis Alert Handling with Machine Learning: 2016-2018

Lori Flynn, PhD

Software Security Researcher

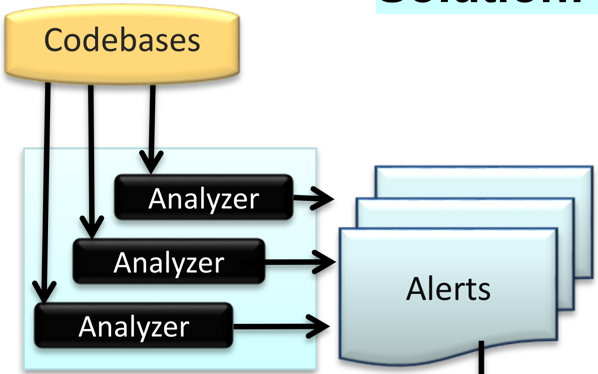
Software Engineering Institute of Carnegie Mellon University

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.

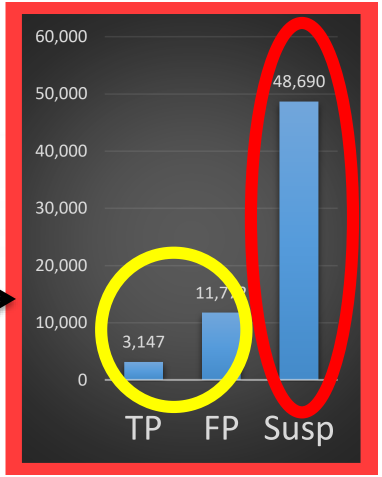
DM18-1078

Overview

Problem: too many alerts
Solution: automate handling



Today



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)

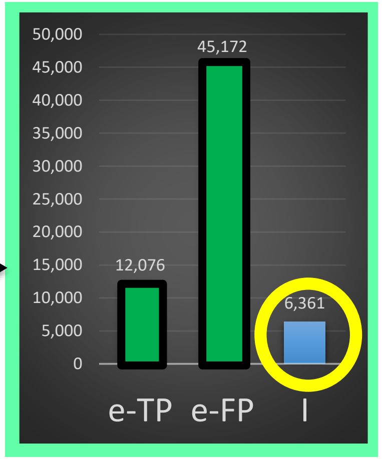
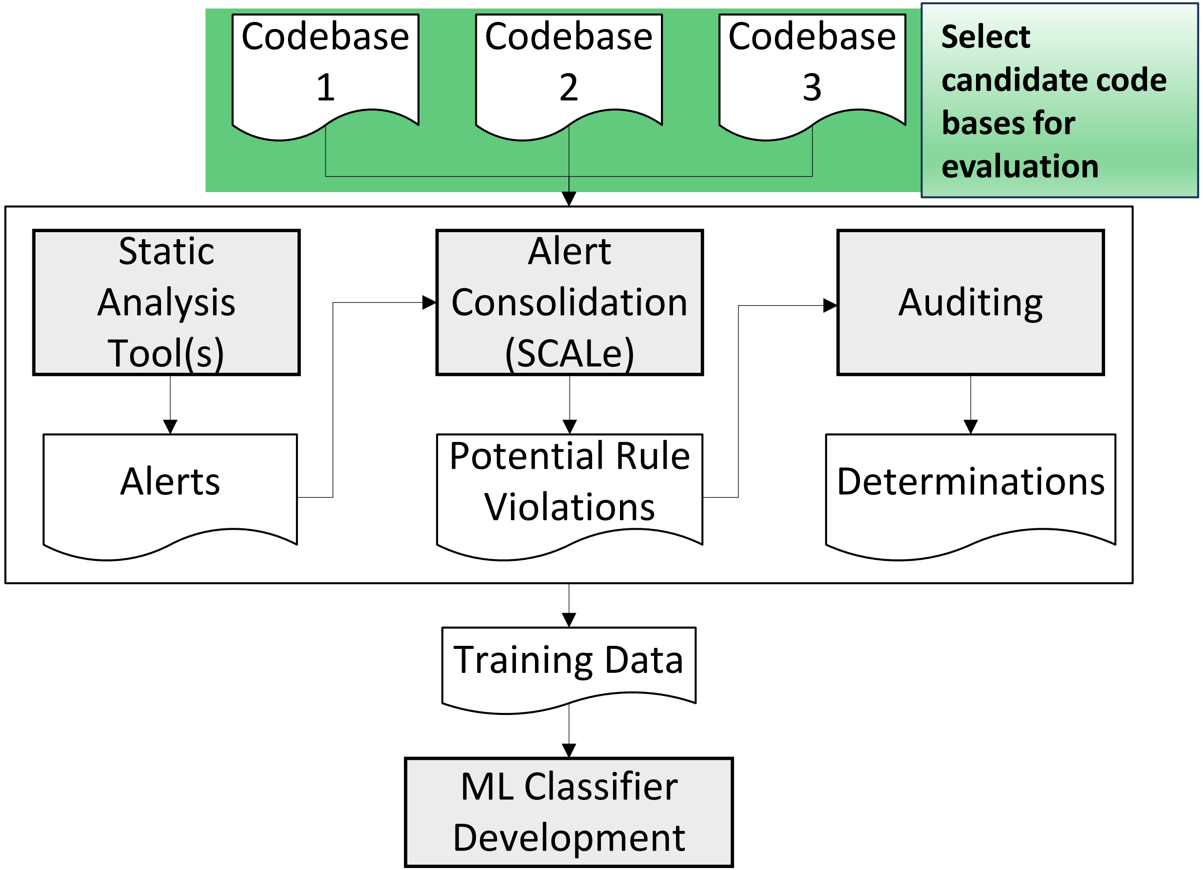
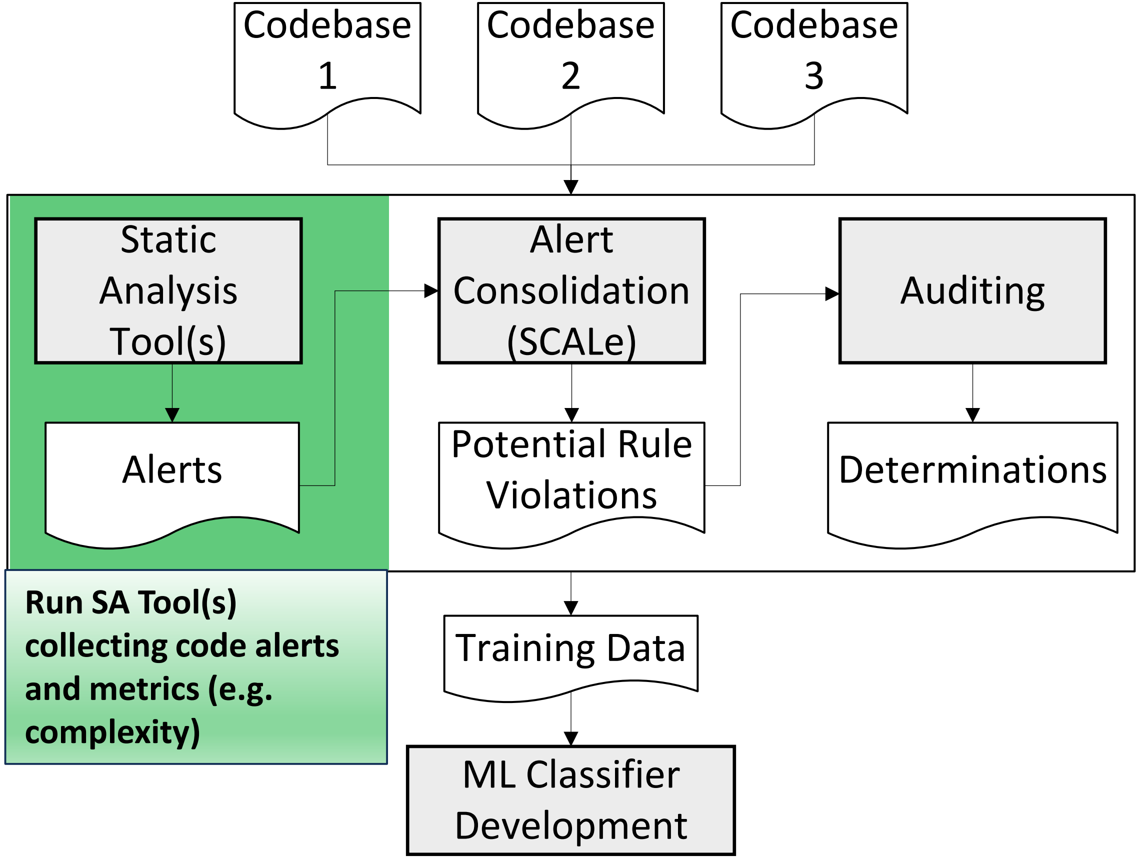


Image of woman and laptop from http://www.publicdomainpictures.net/view-image.php?image=47526&picture=woman-and-laptop "Woman And Laptop"

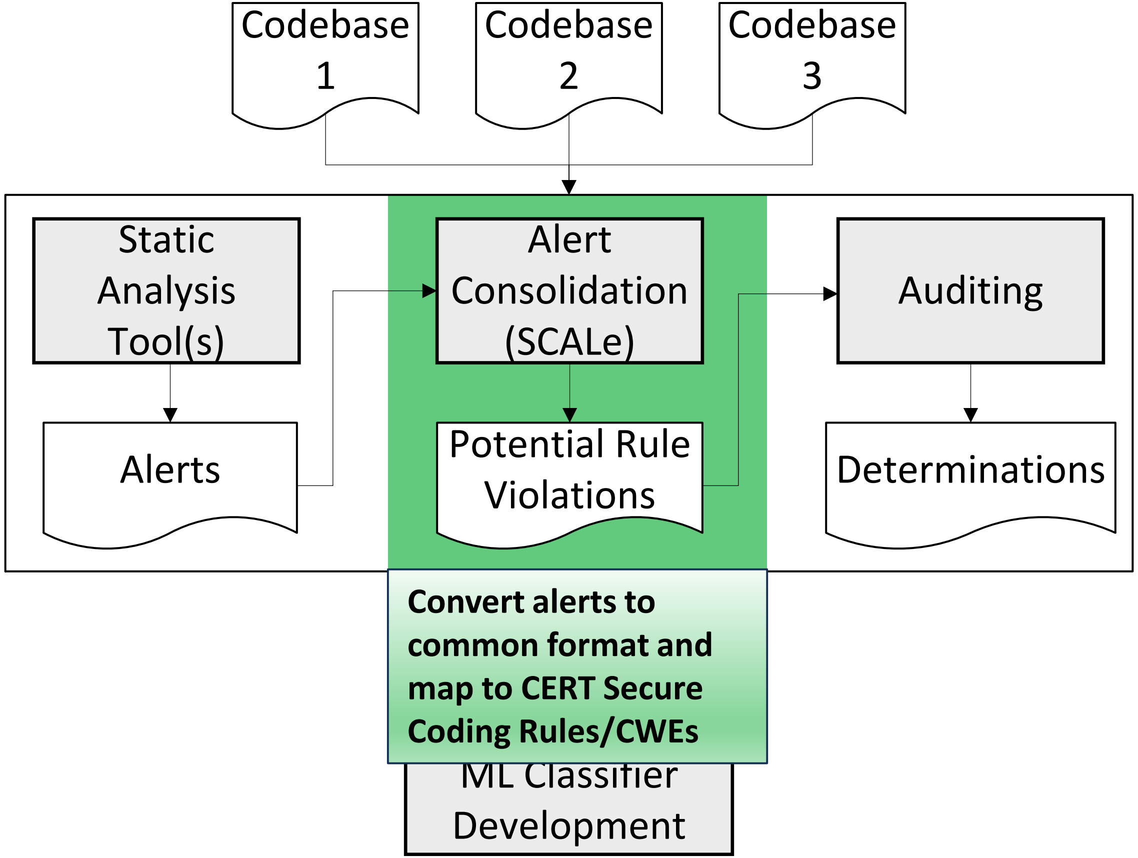
Background: Automatic Alert Classification



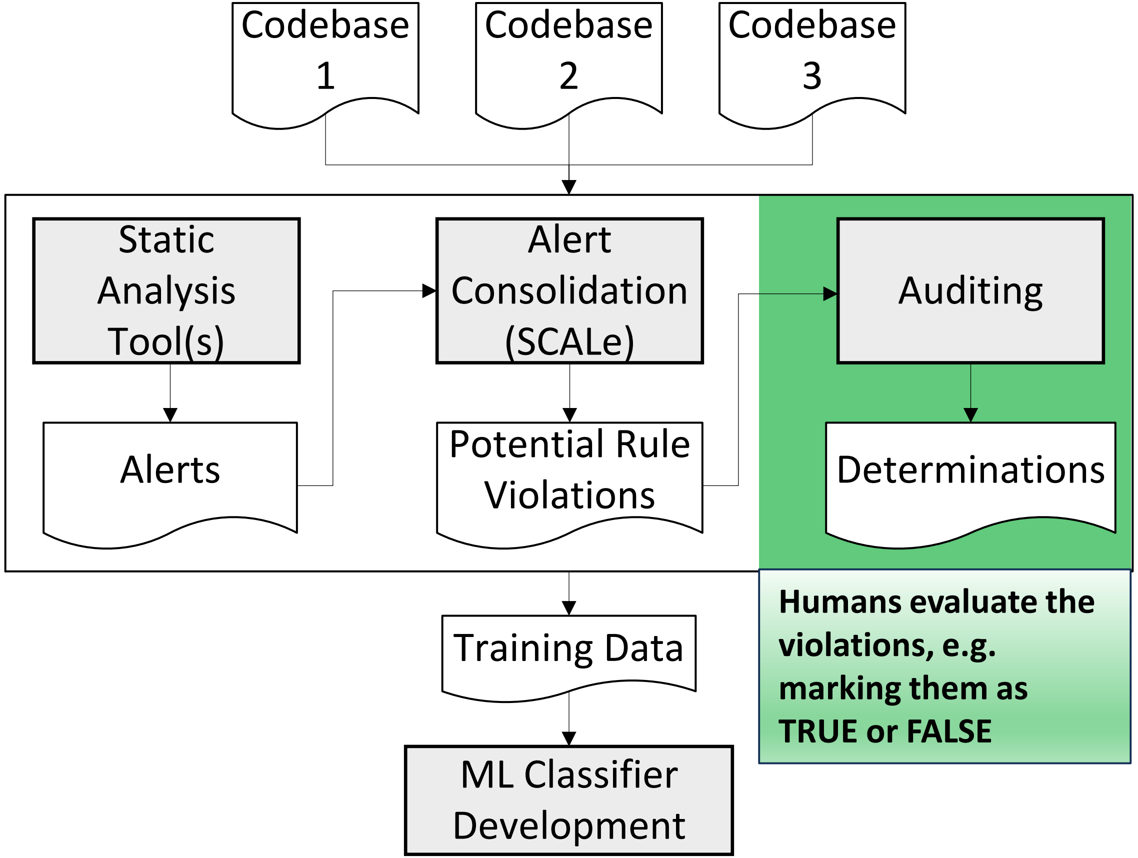
Background: Automatic Alert Classification



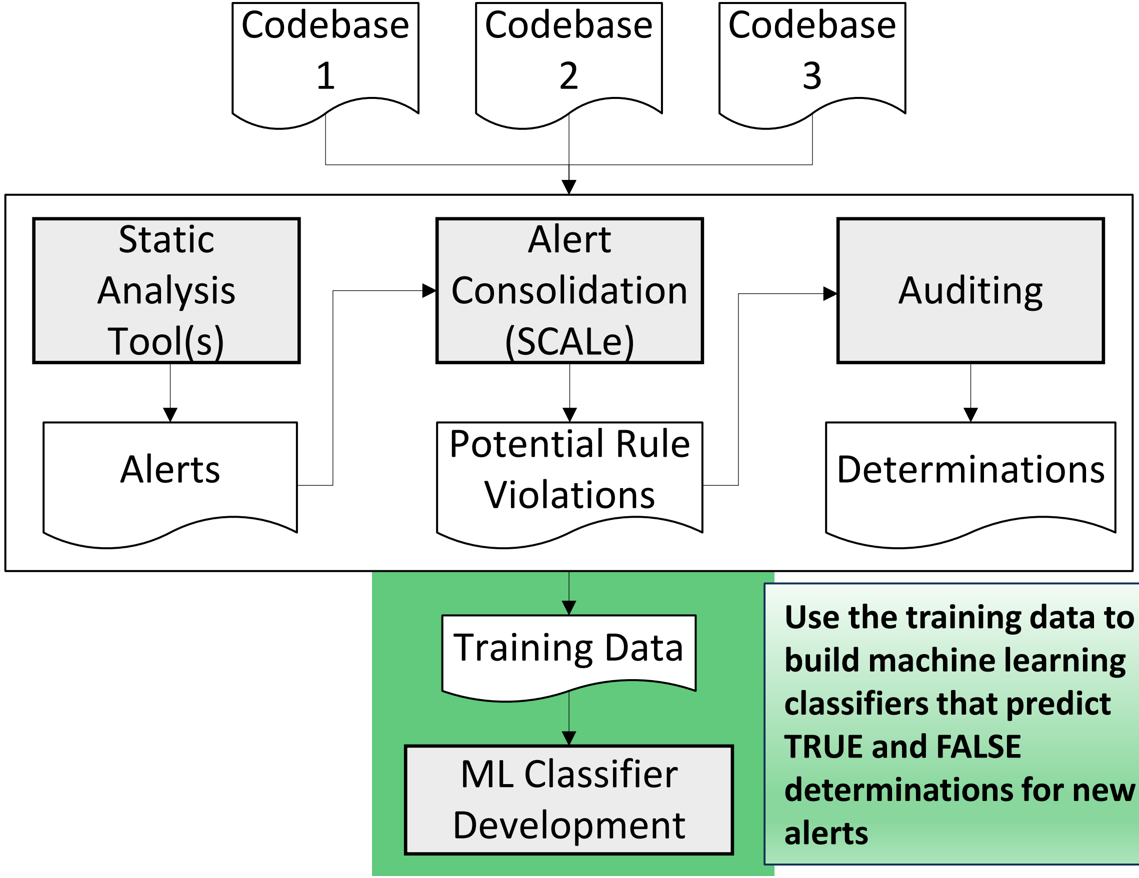
Background: Automatic Alert Classification



Background: Automatic Alert Classification

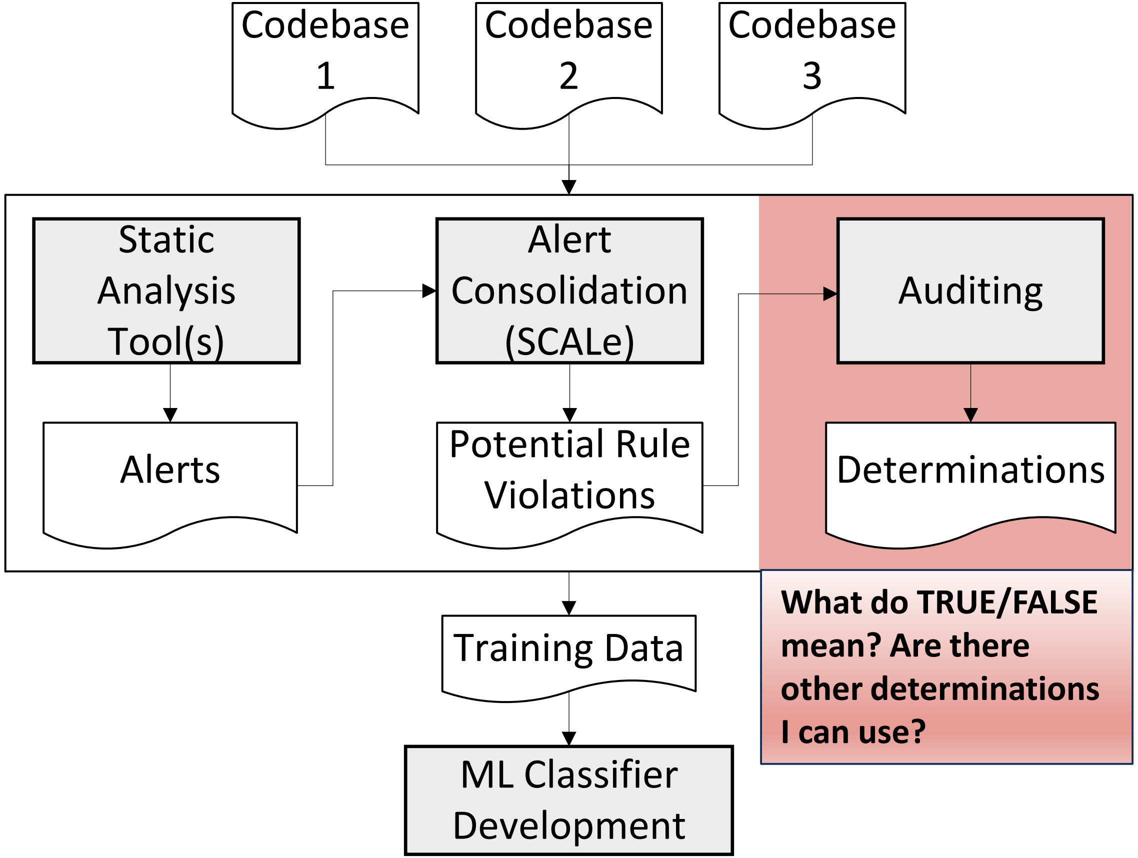


Background: Automatic Alert Classification



Use the training data to build machine learning classifiers that predict TRUE and FALSE determinations for new alerts

Background: Automatic Alert Classification

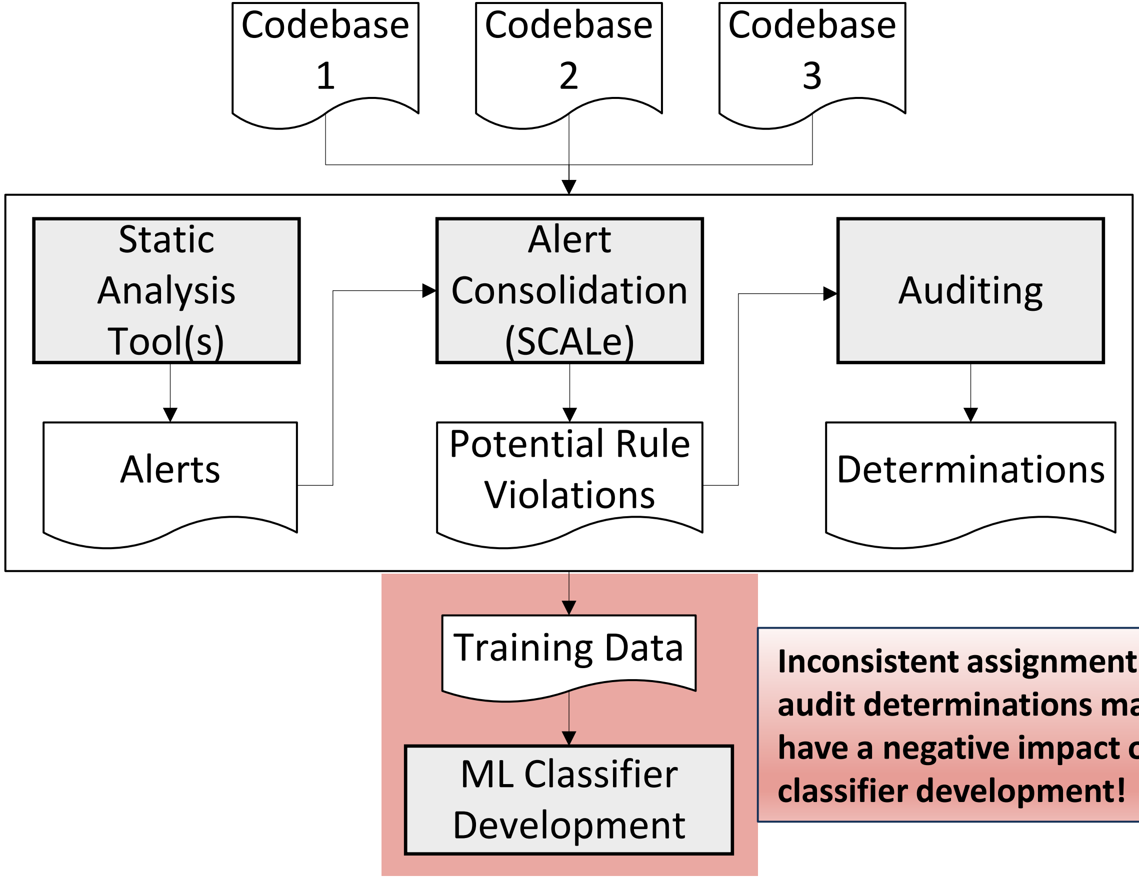


What is truth?

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



Inconsistent assignment of audit determinations may have a negative impact on classifier development!

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

Audit Determinations

Basic Determinations

- True
- False
- Complex
- Dependent
- Unknown (default)

Choose <u>ONE</u> per alert!

Supplemental Determinations

- Inapplicable environment
- Dangerous construct
- Dead
- Ignore

Choose <u>ANY NUMBER</u> per alert!

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 deference 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;
}
```

ALERT: Do not dereference NULL pointers!

Determination: **FALSE**

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
        }

        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

```c
#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...
}
```

ALERT: potential buffer overrun!

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.

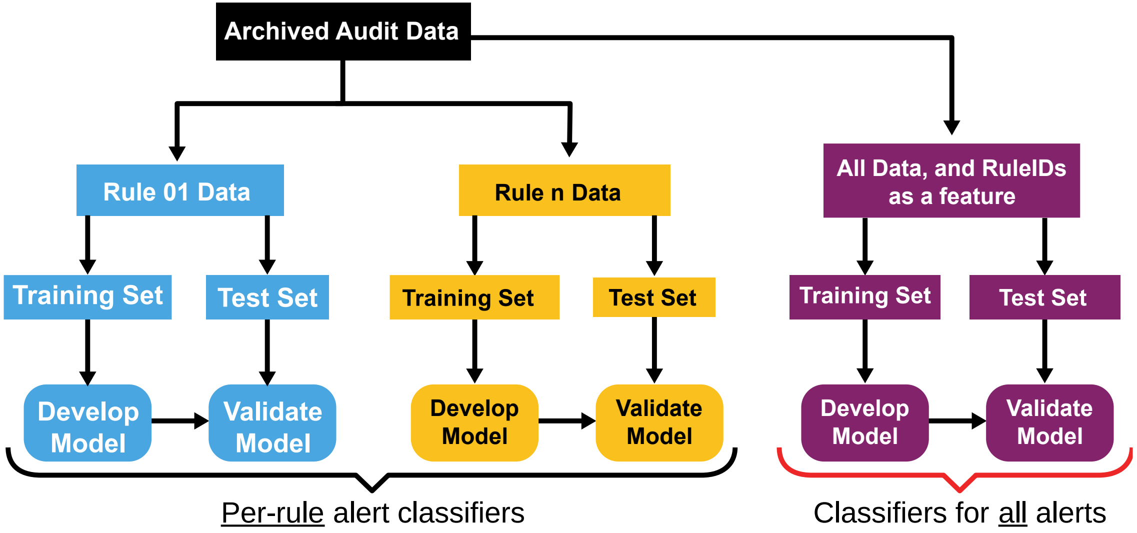
In particular: just because a given program module (function, class) is not used does **not** mean it is dead. The module might be exported as a **public interface**, for use by another application.

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

Classifier Development and Testing

Machine Learning with Static Analysis Audit Archives

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

Data Used for Classifiers

Data used to create and validate classifiers:

- CERT-audited alerts:
  - ~7,500 audited alerts
- 3 collaborators audit their own codebases with our auditing research prototype tool “enhanced SCALe”

We pooled data (CERT + collaborators) and segmented it:

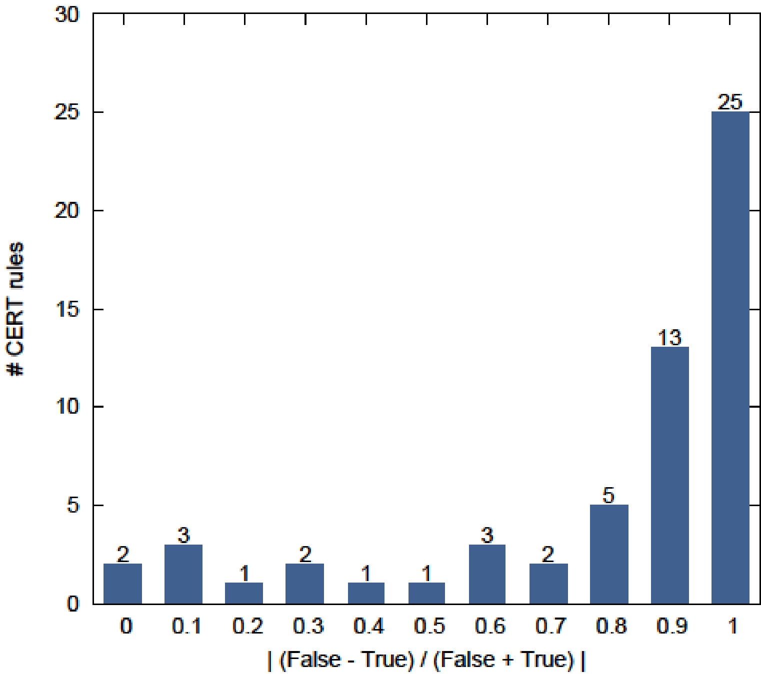
- Segment 1 (70% of data): train model
- Segment 2 (30% of data): testing

Added classifier variations on dataset:

- Per-rule
- Per-language
- With/without tools
- Others

CERT- Audited Archives Characterization

- 58 CERT coding rules with 20 or more audited (labeled) alerts
- 25 rules all (or nearly all) determined one way (True or False)
- Other 324 CERT rules have little or no labeled data
- Labeled data for 158 of 382 CERT rules
- 2,487 True and 4,980 False



Archive sanitizer: enabled collaborator data use

Added data sanitizer to “enhanced SCALe”

- Anonymizes sensitive fields
- SHA-256 hash with salt
- Enables analysis of features correlated with alert confidence

Audit archive for project is in a database

- DB fields may contain sensitive information
- Sanitizing script anonymizes or discards fields
  - Diagnostic message
  - Path, including directories and filename
  - Function name
  - Class name
  - Namespace/package
  - Project filename

Classifier Result Highlights: Data All Sources

Classifiers made from all data, pooled:

All-rules (158) classifier accuracy:

- Lasso Logistic Regression: 88%
- Random Forest: 91%
- CART: 89%
- XGBoost: 91%

Single-rule classifier accuracy:

Rule ID	Lasso LR	Random Forest	CART	XGBoost
INT31-C	98%	97%	98%	97%
EXP01-J	74%	74%	81%	74%
OBJ03-J	73%	86%	86%	83%
FIO04-J*	80%	80%	90%	80%
EXP33-C*	83%	87%	83%	83%
EXP34-C*	67%	72%	79%	72%
DCL36-C*	100%	100%	100%	100%
ERR08-J*	99%	100%	100%	100%
IDS00-J*	96%	96%	96%	96%
ERR01-J*	100%	100%	100%	100%
ERR09-J*	100%	88%	88%	88%

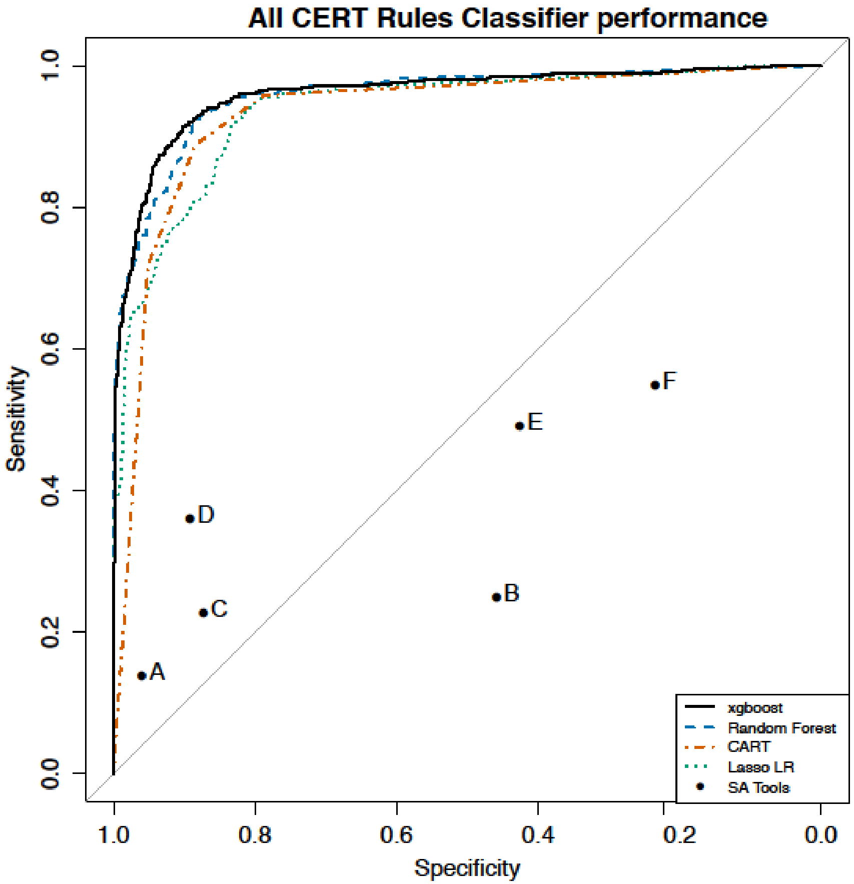
Also, 15 one-way “classifiers”.

General results (not true for every test)

- Classifier accuracy rankings for all-pooled test data:
  XGBoost ≈ RF > CART ≈ LR
- Classifier accuracy rankings for collaborator test data:
  LR ≈ RF > XGBoost > CART
- Per-rule classifiers generally not useful (lack data), but 3 rules (INT31-C best) are exceptions.
- With-tools-as-feature classifiers better than without.
- Accuracy of single language vs. all-languages data:
  C > all-combined > Java

* Small quantity of data, results suspect

Tool as Feature Helped



Using toolname as a feature improved classifier performance

Dots show performance of tool alone

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 conditions with labeled alerts by using test suites, plus collaborator audits of DoD code.

Approach

1. Automated analysis of test suite programs to label data for many conditions for classifiers

2. Collaboration with MITRE: Systematically map CERT rules to CWE IDs

3. Test classifiers on alerts from real-world code: DoD data

Overview: Method, Approach, Validity

Problem 2: too <u>few</u> manually audited alerts to make accurate classifiers for many flaw types
Solution 2: automate auditing alerts, <u>using test suites</u>

Create alert classifiers trained on many conditions, 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.
  - Juliet (NSA CAS) 61,387 C/C++ tests
  - IARPA's STONESOUP: 4,582 C tests
  - Refine test sets for rules: use **mappings, metadata, static analyses**
- Metrics analyses of test suite code, to get feature data
- Use DoD-collaborator SCALe **<u>audits</u>** 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

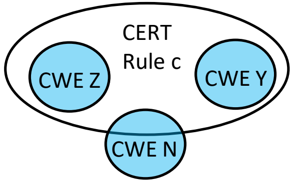
Problem 3: Test suites in different taxonomies (most use CWEs)
Solution 3: Precisely map between taxonomies, then partition tests using precise mappings

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
Total	**612**

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

Problem 3: Test suites in different taxonomies (most use CWEs)
Solution 3: Precisely map between taxonomies, **then partition tests with precise mappings**

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

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

- Possible coverage in other suites

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

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
  - Exact line defect metadata, for TPs
  - Function line spans, for FPs

Number of “**Bad**” Functions	103,376
Number of “**Good**” Functions	231,476

- Used 8 static analysis tools on Juliet programs
- Automated alert-to-defect matching
- 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)**
TRUE	**13,330**
FALSE	**24,523**

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

Analysis of Juliet Test Suite: Initial CWE Results

Lots of new data for creating classifiers (37,853 labeled alerts)

Alert Type	Labeled fused alerts (counts a fused alert once)
TRUE	**13,330**
FALSE	**24,523**

- Big savings: manual audit of 37,853 alerts from non-test-suite programs would take an **unrealistic minimum of 1,230 hours** (117 seconds per alert audit [1]).
  - First 37,853 alert audits wouldn't cover many conditions (and sub-conditions) covered by the Juliet test suite!
  - Need true and false labels for classifiers.
  - **Realistically: enormous amount of manual auditing time** to develop that much data.
- These are initial metrics (more data as we use more tools and test suites)

[1] Nathaniel Ayewah and William Pugh. "The Google FindBugs fixit." *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, 2010.

Juliet Test Suite Classifiers: Initial Results (Hold-out Data)

Classifier	Accuracy	Precision	Recall	AUROC
rf	0.938	0.893	0.875	0.991
lightgbm	0.942	0.902	0.882	0.992
xgboost	0.932	0.941	0.798	0.987
lasso	0.925	0.886	0.831	0.985

2016-2018 Static Analysis Alert Classification Research

2016 2017 2018

2016

- Issue addressed: classifier accuracy
- Novel approach: **multiple static analysis tools as features**
- Result: increased accuracy

2017

- Issue addressed: **too little labeled data** for accurate classifiers for some conditions (CWEs, coding rules)
- Novel approach: **use test suites to automate production of labeled** (True/False) **alert archives for many conditions**
- Result: high accuracy for more conditions

2018

- Issue addressed: **little use of automated alert classifier technology** (requires $$, data, experts)
- Novel approach: **develop extensible architecture with novel test-suite data method**
- Result: extensible architecture, API definition, software to instantiate architecture, adaptive heuristic research

Code

- API definition (swagger) and code development
- SCALe v2.1.3.0 static analysis alert auditing tool
  - New features for prioritization and classification
    - Fused alerts, CWEs, new determinations (etc.) for collaborators to generate data
  - Released to collaborators Dec. 2017–Feb. 2018
  - GitHub publication Aug. 2018 ← First public SCALe release (2.1.4)
- SCALe v3.0.0.0 released Aug. 2018 to collaborators
- Develop and test classifiers. Novel work includes
  - enabling cross-taxonomy test suite classifiers (using precise mappings)
  - enabling "speculative mappings" for tools (e.g., GCC)

Non-code Publications & Papers 2018

- **Architecture API definition and new SCALe features**

For collabs, others to implement API calls or use new SCALe

  - Special Report: "Integration of Automated Static Analysis Alert Classification and Prioritization with Auditing Tools" (Aug. 2018)
    - Technical Report: public version (Sep. 2018)
  - SEI blog post: "SCALe: A Tool for Managing Output from Static Code Analyzers" (Sep. 2018)
- **Classifier development research methods and results:**

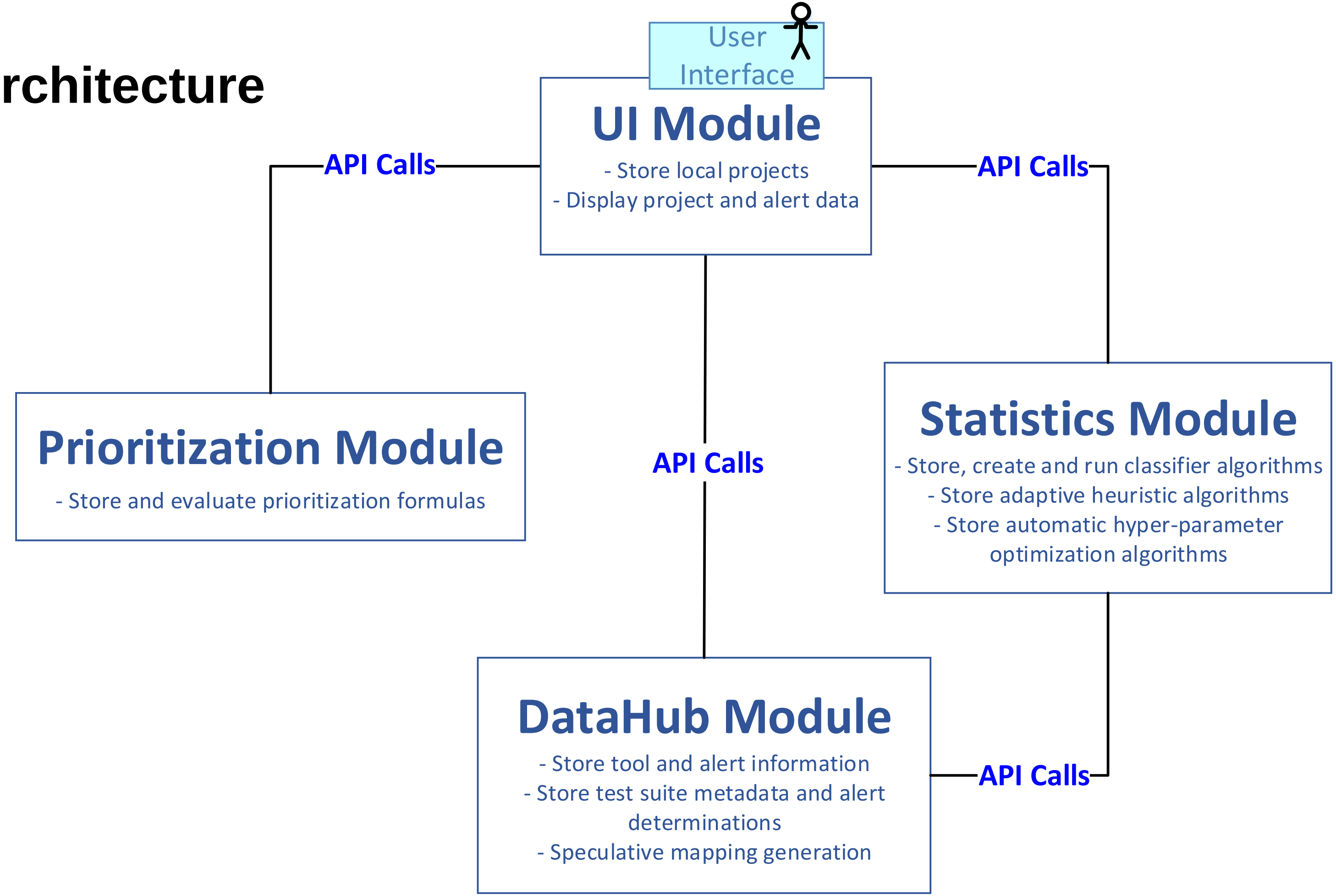
Explain research methods & results

  - Paper "Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models," SQUADE (ICSE workshop) (June 2018)
  - SEI blog post: "Test Suites as a Source of Training Data for Static Analysis Alert Classifiers" (Apr. 2018)
  - SEI Podcast (video): "Static Analysis Alert Classification with Test Suites" (Sep. 2018)
  - In-progress conference papers (4): precise mapping, architecture for rapid alert classification, test suites for classifier training data, API development
- **Precise mappings** on CERT C Standard wiki
  1. Metadata for `Juliet` (created to test CWEs) to test CERT rule coverage
  2. Per-rule precise CWE mapping

For code flaws you care about, understand your tool coverage

Static analysis tool developers can automatically test for CERT rule coverage (some rules)

Architecture



Architecture Development

Representational State Transfer (REST)

- Architectural style that defines a set of constraints and properties based on HTTP
- RESTful web services provide interoperability between systems
- Client-server

We chose to develop a RESTful API

- Swagger/OpenAPI open-source development toolset
  - Develop APIs
  - Auto-generate code for server stubs and clients
  - Test server controllers with GUI
  - Wide use (10,000 downloads/day)

SCALe Development for Architecture Integration

SCALe will make UI Module API calls in prototype system.

- Other alert auditing tools (e.g., DHS SWAMP) also can instantiate UI Module API.

Next Steps and Collaboration Opportunities

Goal: increase automation of static alert auditing, using machine learning

- Work in progress through 2019:
  - Using test suite data for classifiers, research adaptive heuristics
    - How classifiers incorporate new data
    - Test suite vs. non-test-suite data
    - Weighting recent data
  - Code development to complete 4-server system instantiation with SCALe as UI Module
- Collaboration opportunities:
  - Implementation of API by collaborators to extend their own alert auditing tools
    - Feedback on API, code system, and adaptive heuristics
  - Alert audit data needed (sanitized fine)
  - Precise mapping to more code flaw taxonomies
  - **Additional ideas welcome!**

Contact Information

Presenter / Point(s) of Contact

Lori Flynn (Principal Investigator)

Software Security Researcher

Email: lflynn@cert.org

Office: +1 412.268.7886

Additional Contributors

SEI Staff

Ebonie McNeil
William Snavely
Zach Kurtz
David Svoboda
Derek Leung

SEI Student Interns

Jiyeon Lee (CMU)

Lucas Bengtson (CMU)

Charisse Haruta (CMU)

Baptiste Vauthey (CMU)

Christine Baek (CMU)