Research Review 2018

Scaling Software Testing & Evaluation

SEI CERT Division
Cyber Security Foundations Directorate

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

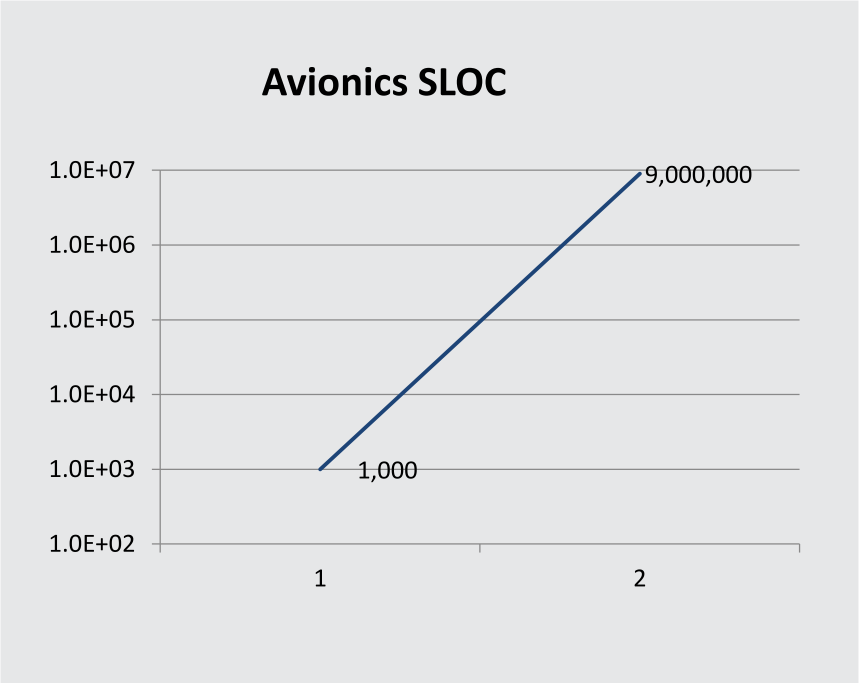
This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

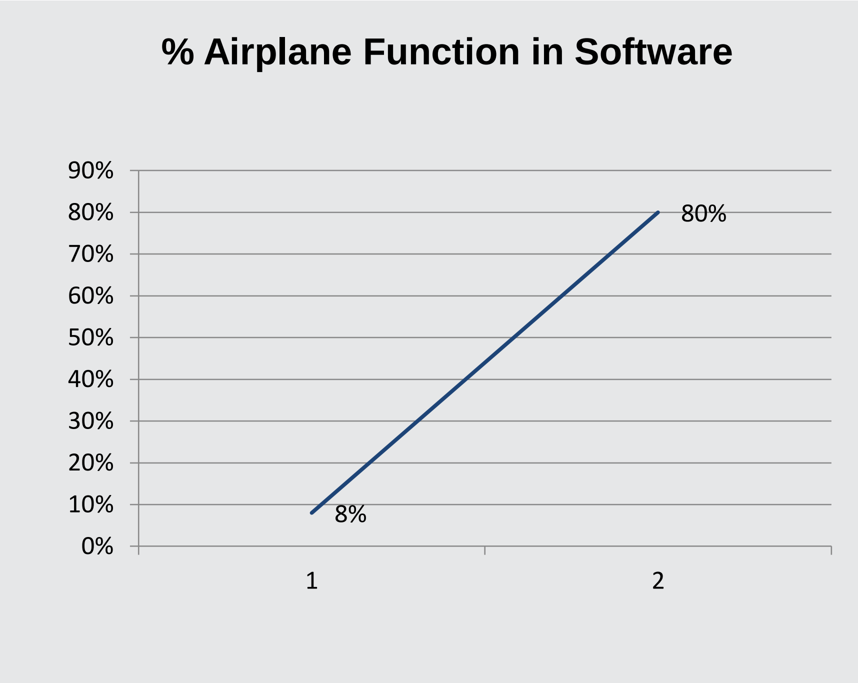
DM18-1194

The Problem

Complex Software Is Business and Mission Critical

Evolution of avionics size and function from F-4A (1960) to F-35 (2000):*

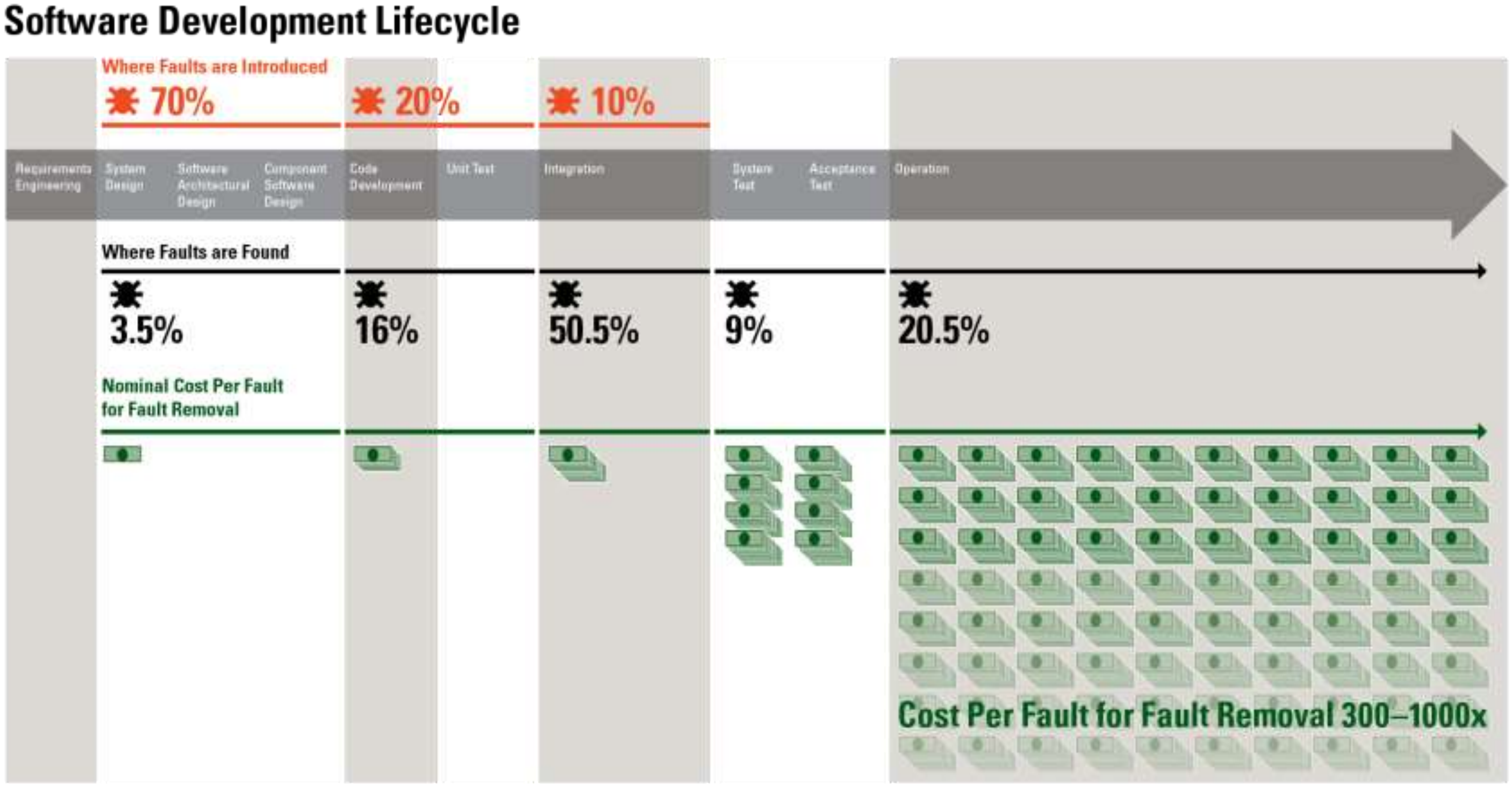




*Final Report, NASA Study on Flight Software Complexity, Mar. 2009; Mel Conway, “Tower of Babel and the Fighter Plane,” Oct. 9, 2013.

Catching Software Faults Early Saves Money

Faults account for 30%–50% of total software project costs.*



**Critical Code*; NIST, NASA, INCOSE, and Aircraft Industry Studies.

Enduring Software Challenges: Scaling Software Testing and Evaluation

Affordable
Be Affordable such that the cost of acquisition and operations, despite increased capability, is reduced and predictable

Trustworthy
Be Trustworthy in construction, correct in implementation, and resilient in the face of operational uncertainties

Capable
Bring Capabilities that make new missions possible or improve the likelihood of success of existing ones

Timely
Be Timely so that the cadence of fielding is responsive to and anticipatory of the operational tempo of the warfighter

SOTA SOTP

Testing

Testing Methods

Method	Strengths	Weaknesses
Architectural/Design Analysis	Early identification of costly defects	Conceptual and early
Inspection & Reviews	Effective at identifying nuanced defects that require developer context	Manual (expensive, slow)
Static Analysis	More thorough coverage	High false-positive rates Generally requires buildable source
Dynamic Analysis	Very low false-positive rate	Difficult to get good coverage
Formal Methods	Proves software attributes	Requires significant time and space resources, plus model validation is challenging; significant manual effort
Simulation	Useful for gaining validation confidence	Testbed setup can be costly

Testing Purposes and Evolution

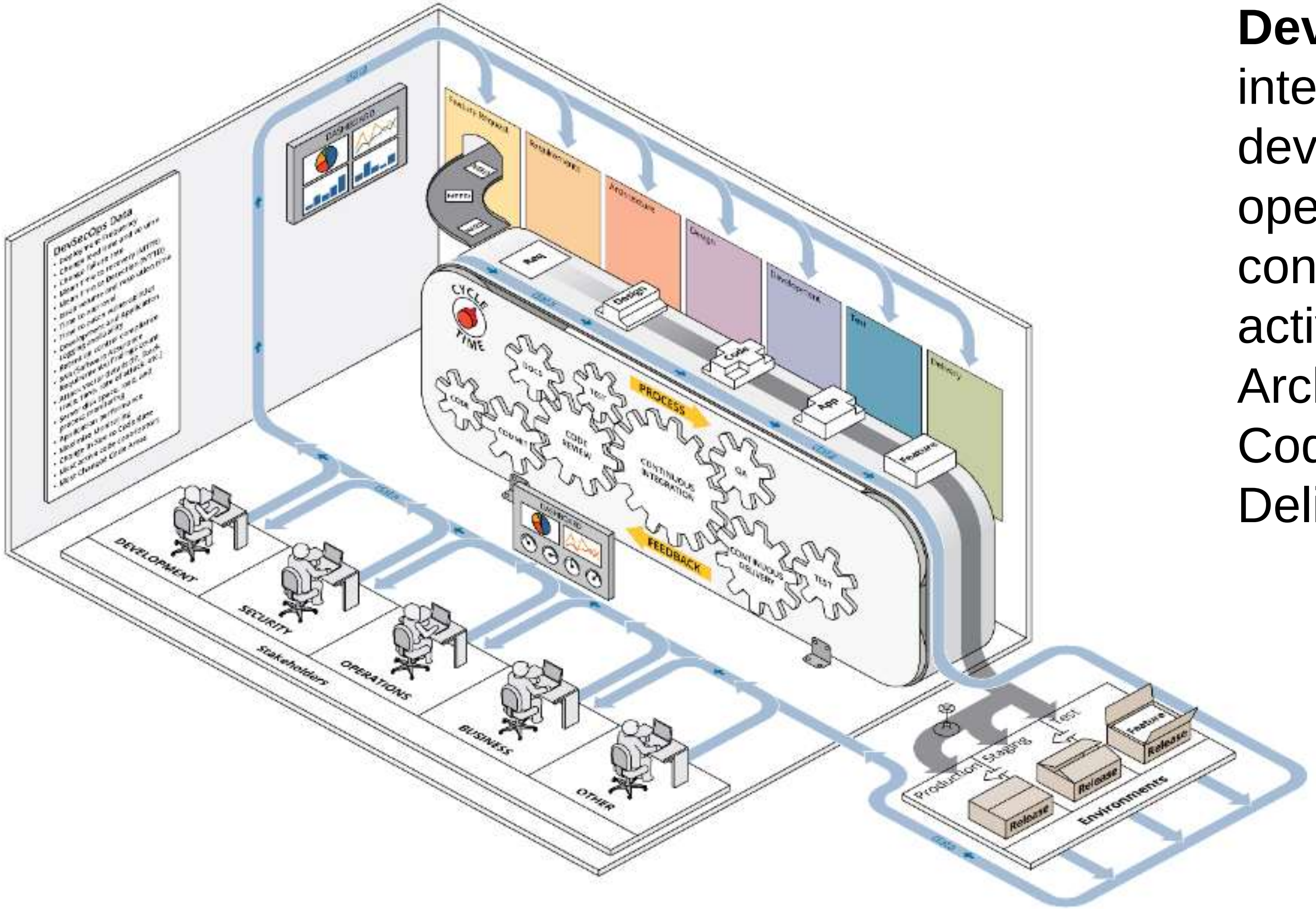
Testing Purposes

- Unit testing
- Integration/system testing
- Regression testing
- Acceptance testing

Evolution (as possible)

- Manual inspection
- Tool-supported
- Integrated
- Automated testing
- Automated repair

Secure DevOps



DevSecOps is a model integrating the software development and operational process considering security activities: Requirements, Architecture, Design, Coding, Testing, and Delivering.

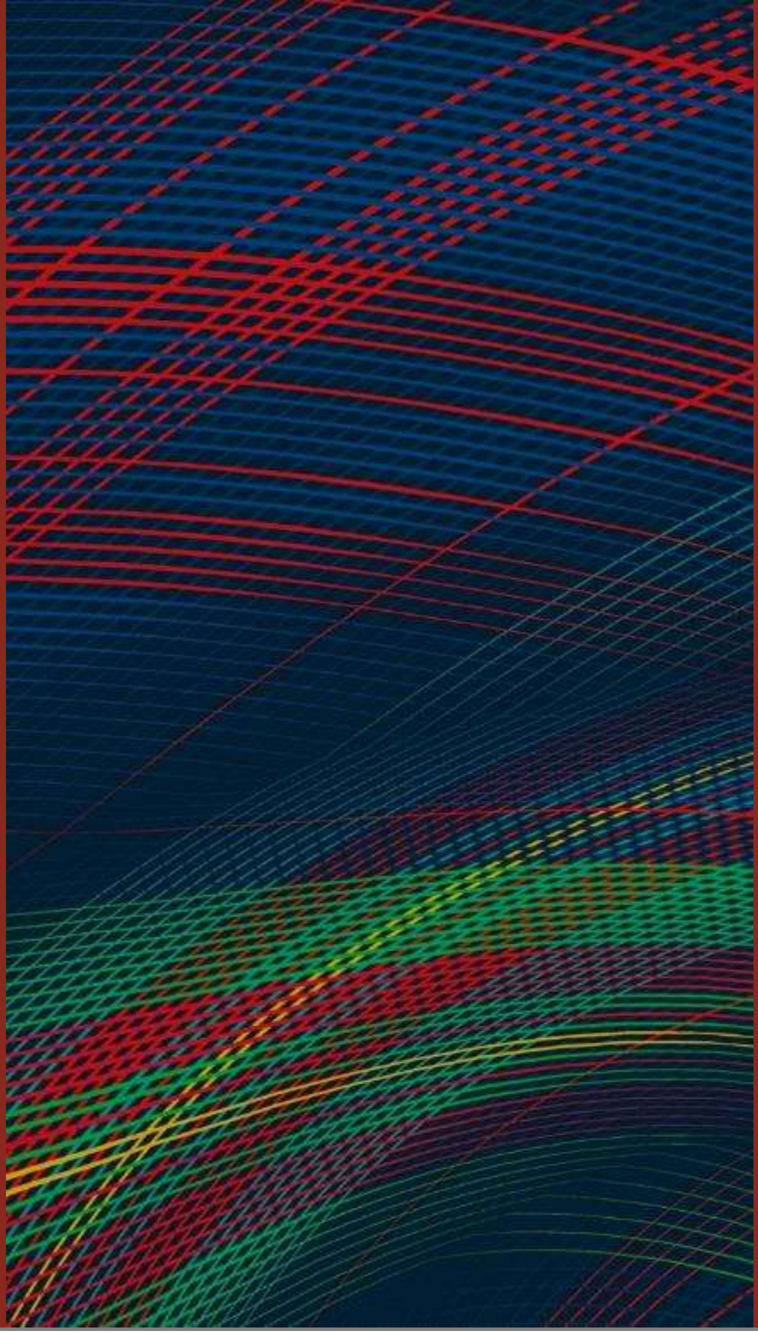
Scaling Testing

SEI is researching how to make testing (where possible)

- less expensive
- more precise
- automatable

SEI also researches scalable automated repairs, following testing.

Predicting Security Flaws through Architectural Flaws



Enduring Software Challenges: Predicting Security Flaws through Architectural Flaws

Affordable
Be Affordable such that the cost of acquisition and operations, despite increased capability, is reduced and predictable

Trustworthy
Be Trustworthy in construction, correct in implementation, and resilient in the face of operational uncertainties

Problem

Software security defects → risk exposure and $$$.

Existing analysis methods have limitations:

- Some security flaws influenced by code structure and module relationships.
- Not easily found or fixed locally.*

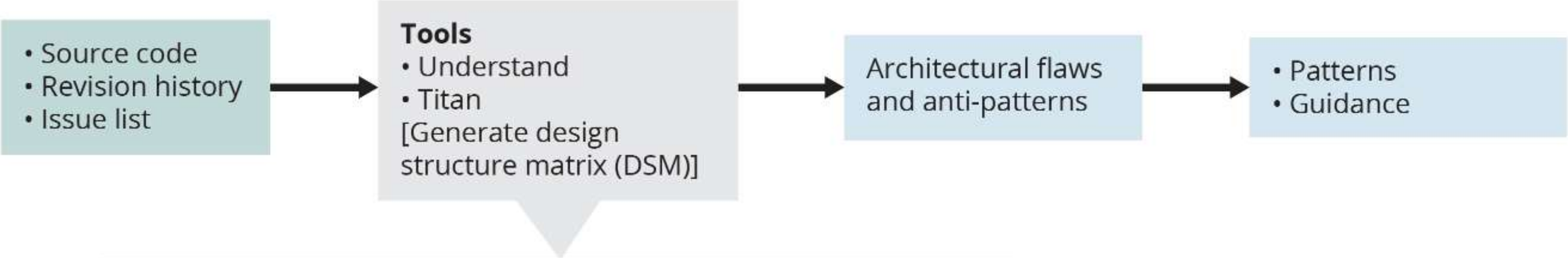
*“Analyzing Security Bugs from an Architectural Perspective,” Kazman et al., 2017.

Impact of Issues Involving >10 Files (Chromium)

Potential Impact: ~50% of the total effort (LoC) to fix security issues came from fixing <10% of the security issues.*

*"Analyzing Security Bugs from an Architectural Perspective," Kazman et al., 2017.

Approach – Today

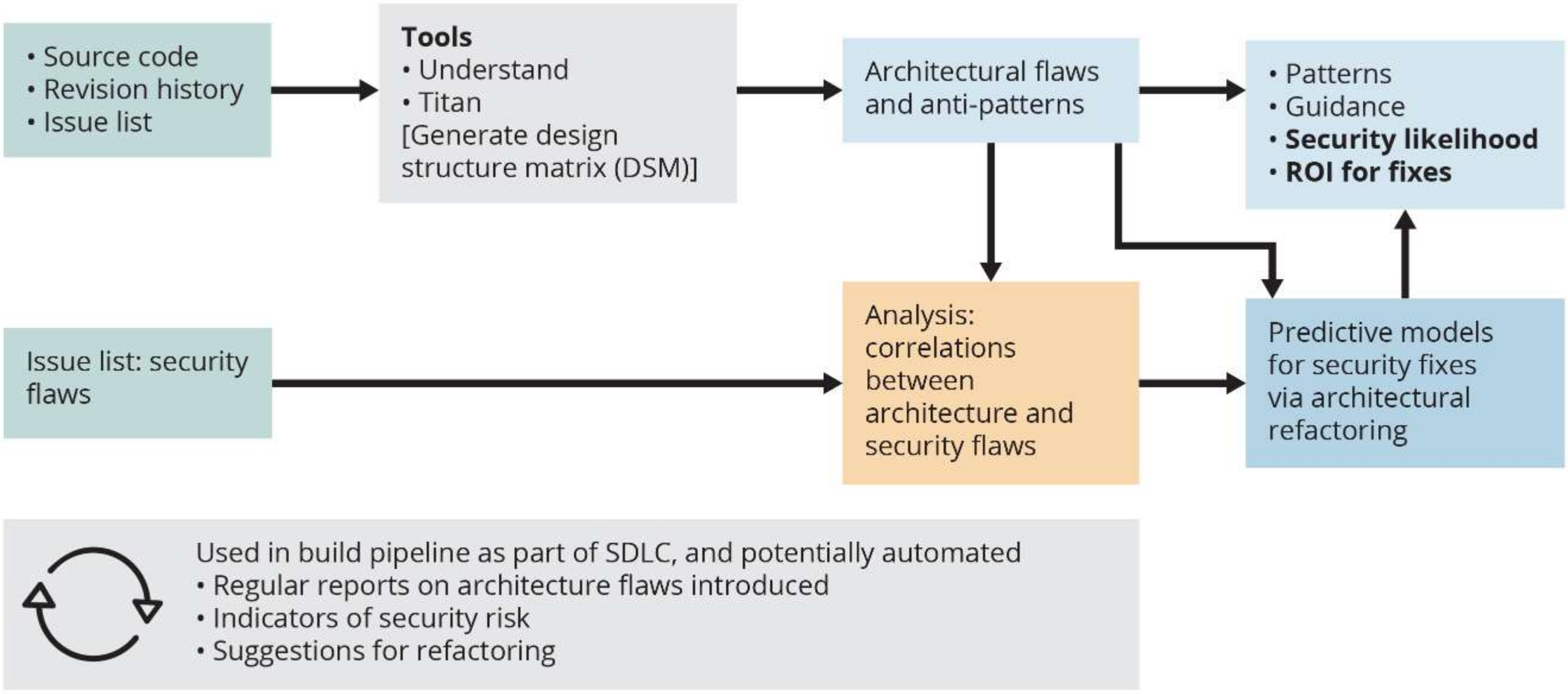


Improper Interface

		1	2	3	4	5	6	7	8	9	10
1	122654.content.browser.ssl.ssl_error_handler.h	(1)									
2	122654.content.browser.ssl.ssl_manager.h	I,6	(2)								
3	122654.content.browser.renderer_host.resource_dispatcher_host_impl.h	I,Pu,2	,2	(3)							
4	122654.content.browser.ssl.ssl_cert_error_handler.h	I,Pu,5	,12	,2	(4)						
5	122654.content.browser.ssl.ssl_error_handler.cc	U,I,6	,6	,2	C,I,4	(5)					
6	122654.content.browser.ssl.ssl_manager.cc	,4	C,I,11	I,2	C,I,10	C,4	(6)				
7	122654.content.browser.ssl.ssl_cert_error_handler.h	,4	C,U,I,23	I,2	I,10	,12	C,10	(7)			
8	122654.content.browser.rendererHost.socket_stream_dispatcher_host.h	I,Pu	,2		,2				(8)		
9	122654.content.browser.rendererHost.socket_stream_dispatcher_host.cc		I					C	U,I,9	(9)	
10	122654.content.browser.rendererHost.resource_dispatcher_host_impl.cc	,2	I,3	U,I,18	,2	,2	,2	C,3			(10)

C = Call; U = Use; I = Include; T = Type; S = Set; O = Override;
Pu = Public Inherit; ,# = # concurrent check-ins

Approach – Research and Vision



Progress and Results

Chromium, OpenSSL, and Mozilla analyzed

- 6000 Chromium issues analyzed after commits (50% security / 50% non-security)
- 1600 Chromium issues analyzed before/after commits
- Analyzing entire Chromium project over entire per-file history
- Analyzing Chromium issue chains for common files

Tools

- Scripts for data extraction of code repository and issue logs
- Scripts for filtering, categorizing, and analyzing data

Findings (anecdotal; still analyzing for statistical significance)

- Architectural flaws have been strongly correlated with security flaws at a project-wide level
- Still iterating on precision of defining architectural flaws for correlative analysis

Rapid Construction of Accurate Automatic Alert Handling System

Enduring Software Challenges: Rapid Construction of Accurate Automatic Alert Handling System

Affordable
Be Affordable such that the cost of acquisition and operations, despite increased capability, is reduced and predictable

Trustworthy
Be Trustworthy in construction, correct in implementation, and resilient in the face of operational uncertainties

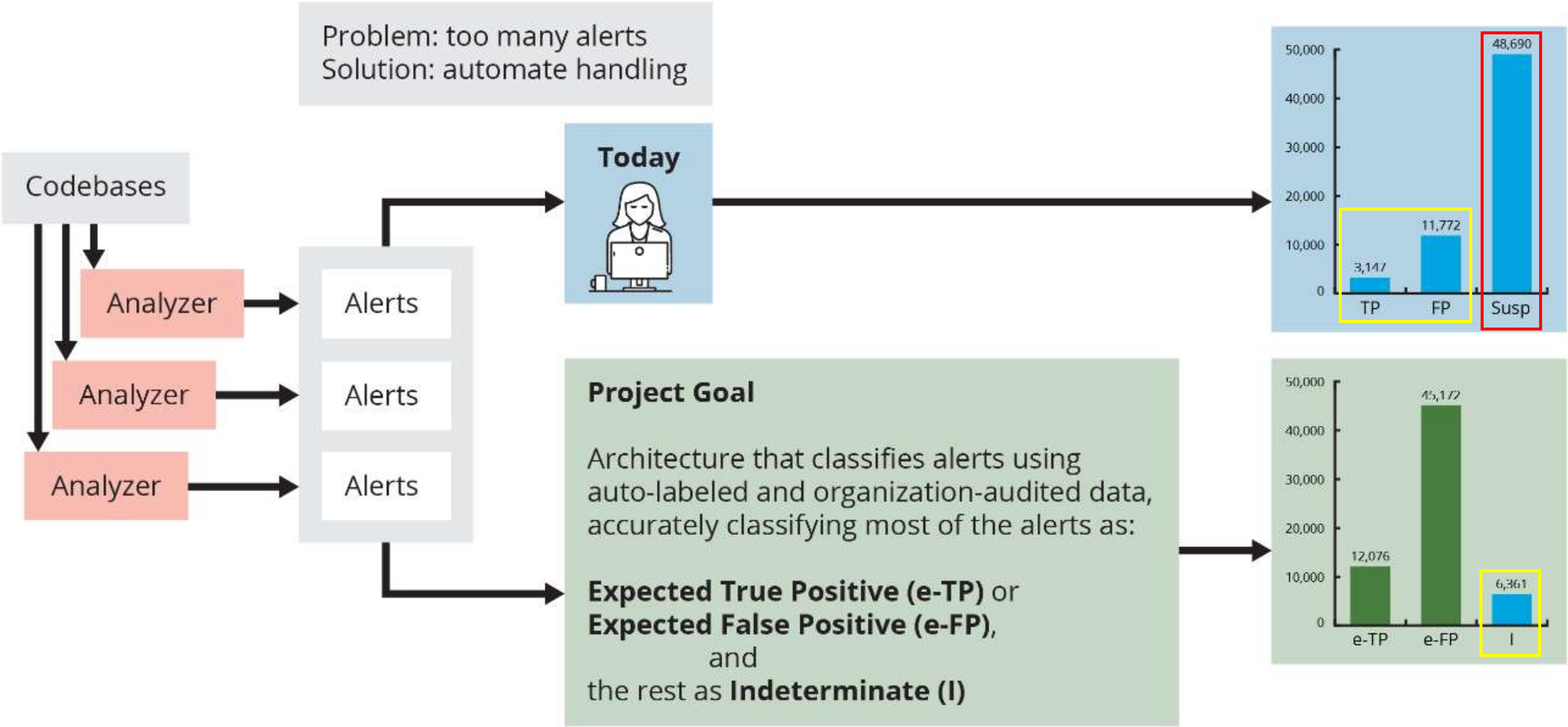
Capable
Bring Capabilities that make new missions possible or improve the likelihood of success of existing ones

Timely
Be Timely so that the cadence of fielding is responsive to and anticipatory of the operational tempo of the warfighter

SOTA	SOTP

- Affects state-of-the-art and state-of-the- practice for static analysis
- Novel use of test suites for classification
- Effect: more secure code at same cost

Overview


Problem: too many alerts
Solution: automate handling
Codebases
Analyzer
Analyzer
Analyzer
Alerts
Alerts
Alerts
Today
50,000
40,000
30,000
20,000
10,000
0
3,147
11,772
48,690
TP
FP
Susp
Project Goal
Architecture that classifies alerts using auto-labeled and organization-audited data, accurately classifying most of the alerts as:
Expected True Positive (e-TP) or
Expected False Positive (e-FP),
and
the rest as Indeterminate (I)
50,000
40,000
30,000
20,000
10,000
0
12,076
45,172
6,361
e-TP
e-FP
I

FY16-18 Static Analysis Alert Classification Research

FY16

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

FY17

- 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

FY18

- 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 FY18

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

Architecture API definition and new SCALe features

- Special Report: “Integration of Automated Static Analysis Alert Classification and Prioritization with Auditing Tools” (Aug. 2018)
  - Technical Report: public version (Sep. or Oct. 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 and results

- Paper “Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models,” SQUADE (ICSE workshop)
- 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

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

- CERT manifest for Juliet (created to test CWEs) to test CERT rule coverage
- Per-rule precise CWE mapping

For code flaws you care about, understand your tool coverage

Analysis of Juliet Test Suite: Initial CWE Results

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

Lots of new data for creating classifiers

(37,853 labeled alerts)

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*).

- 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).

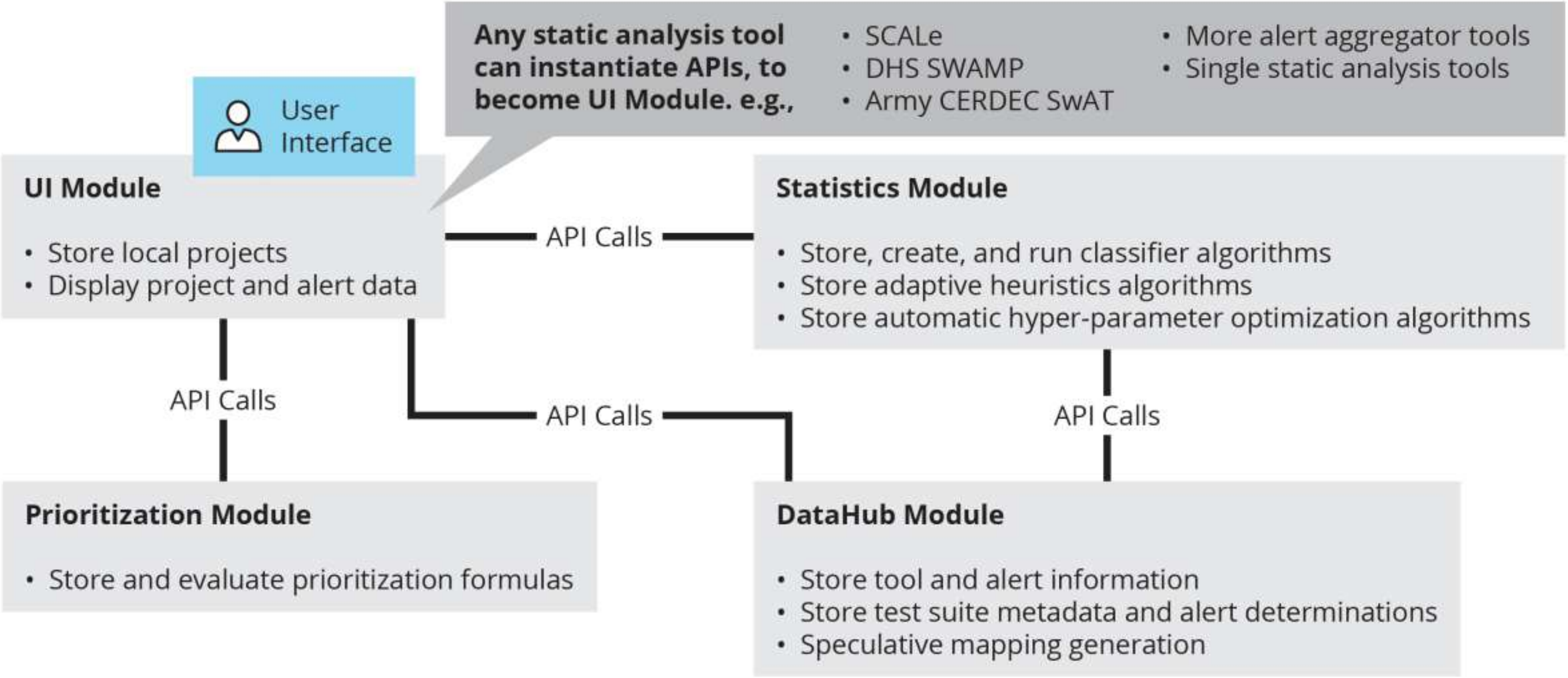
*Nathaniel Ayewah and William Pugh, "The Google FindBugs Fixit," Proceedings of the 19th International Symposium on Software Testing and Analysis, ACM, 2010.

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

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

		Actual condition			
	Total population	Condition true	Condition false	Accuracy =	$\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$
Predicted condition	Predicted condition true	True positive	False positive	Precision =	$\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition true}}$
	Predicted condition false	False negative	True negative		
		True positive rate, recall, sensitivity = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ (Condition true)}}$	False positive rate = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ (Condition false)}}$		

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.

Continue FY19: Classifier Research and Development

Using test suite data for classifiers, research:

- Adaptive heuristics:
  - How classifiers incorporate new data
  - Test suite vs. non-test-suite data
  - Weighting recent data
- Semantic features for cross-project prediction
  - Test suites as different projects

FY19 Next Steps

Collaborator API implementation

More collaborator audit archive data sharing

Metrics of success:

- Compare classifier precision on DoD datasets (cross-validation on test set):
  - Test with semantic features
  - Variations of adaptive heuristics
- Test fault detection rates by tracking true positives detected versus number of manual alert inspections
- Goal: minimum 60% classified e-TP or e-FP with 95% accuracy against collaborator data
- Test architecture generality using varied plug-ins to API

Can Deep Learning Predict Security Defects in Synthetic Code?

Enduring Software Challenges: Can Deep Learning Predict Security Defects in Synthetic Code?

Trustworthy
Be Trustworthy in construction, correct in implementation, and resilient in the face of operational uncertainties

Capable
Bring Capabilities that make new missions possible or improve the likelihood of success of existing ones

- Developed, sa-bAbI, a new software assurance benchmark and training set to be included in NIST Software Assurance Reference Dataset (SARD)
- Identified next steps for AI in software assurance: better representations of code and different learning strategies

Can Deep Learning Predict Security Defects in Synthetic Code?

Problem: Predicting security defects in source code is of significant national security interest (e.g., NIST SAMATE), but existing static analysis tools have unacceptable performance.*

- Artificial intelligence approaches may improve performance, but
- Existing software assurance datasets have limited variability in examples of defects (e.g., Juliet, SARD, IARPA Stone Soup, LAVA)

Our Approach:

- Develop a new software assurance dataset: sa-bAbI
- Benchmark state-of-the-art artificial intelligence system and existing static analysis tools on sa-bAbI

*Oliveira et al., 2017.

sa-bAbI: “Baby AI” Software Assurance Tasks

Conditional Reasoning Example

```
char entity_7[27];
entity_1 = 45; entity_8 = 74;
if(entity_8 > entity_1){
    entity_8 = 64;
} else {
    entity_8 = 17;
}
entity_7[entity_8] = ‘i’;
```

Is the last access safe? No.

Modeled after bAbI*

Code generator for detecting buffer overflow errors

Intentionally very simple

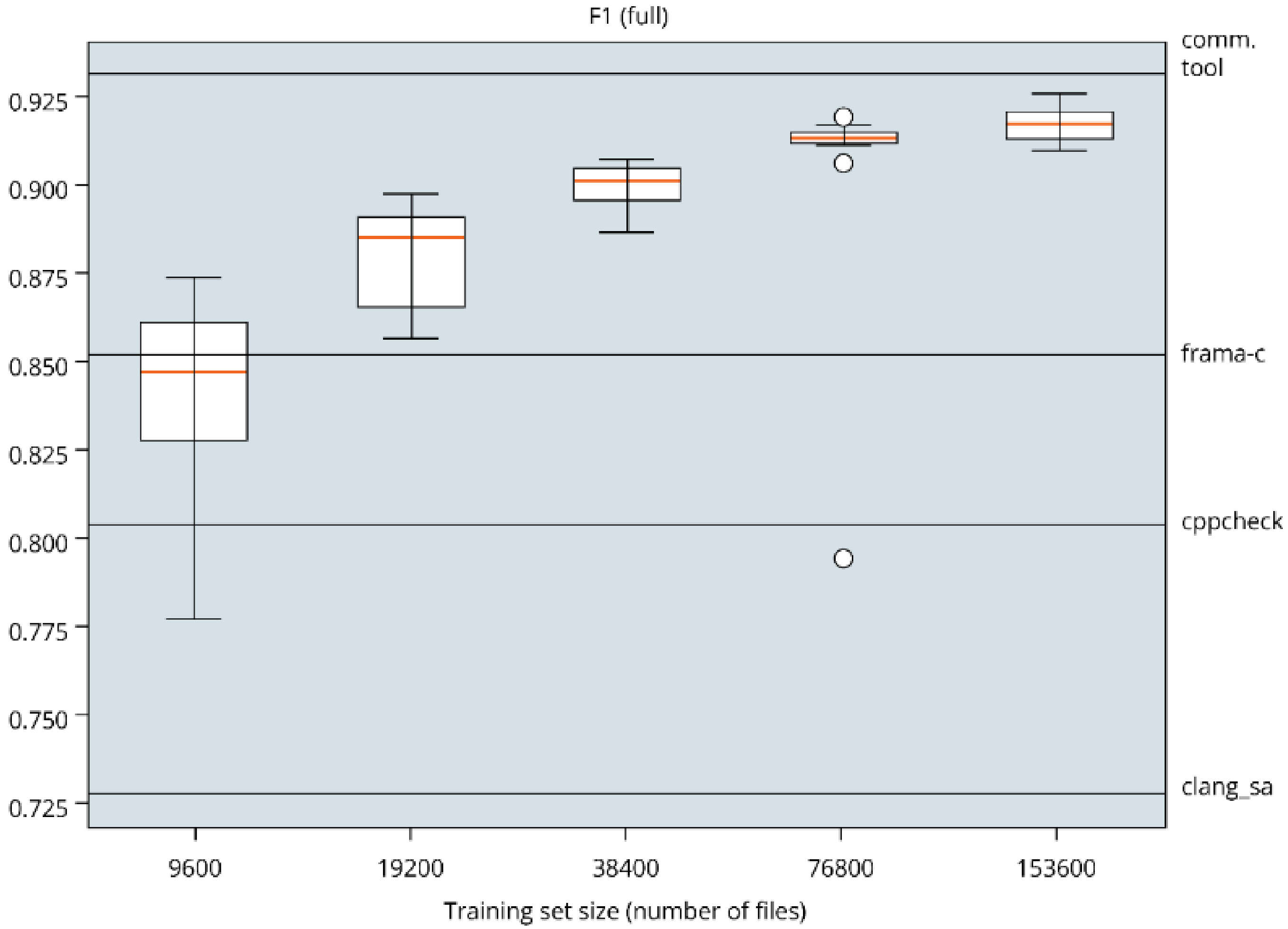
- Valid C code
- Conditionals
- Loops
- Unknown values such as `rand()`

Complements existing software assurance datasets for training AI

Will be included in NIST SARD

*Weston et al., 2015

Results: Deep Learning Cannot Do This Yet



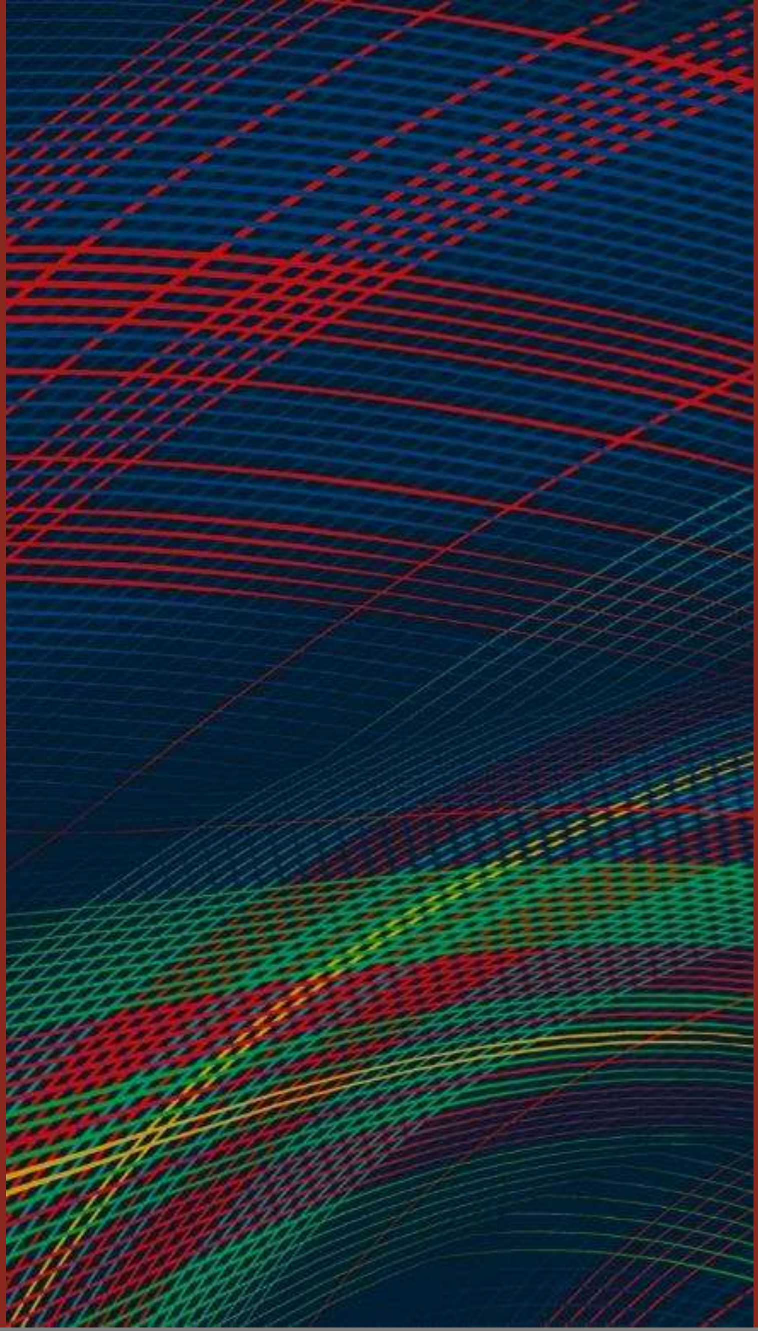
The state-of-the-art AI system can be **competitive** with existing static analysis engines, but it **fails to generalize.**

sa-bAbI illuminated why:
We need better

- representations of code and
- neural integer computation

See arXiv.org for more details.

Status of Available Technology



Enduring Software Challenges: Status of Available Technology

Affordable
Be Affordable such that the cost of acquisition and operations, despite increased capability, is reduced and predictable

Trustworthy
Be Trustworthy in construction, correct in implementation, and resilient in the face of operational uncertainties

Capable
Bring Capabilities that make new missions possible or improve the likelihood of success of existing ones

Timely
Be Timely so that the cadence of fielding is responsive to and anticipatory of the operational tempo of the warfighter

- **Secure coding standards**: provide coverage target for static analysis tools, training for developers
- **SCALe** (static analysis alert auditing tool): provide implemented research features for others to use or adapt into their own tools

SOTA | **SOTP**

- **Automated Code Repair** to Ensure Memory Safety: inexpensive process results in more-secure code

Available Technology: Secure Coding Standards

Curated wisdom from thousands of contributors on community wiki since 2006

- **Use the standards to develop analysis tools and to train developers.**

SEI CERT C Coding Standard

- Free PDF download: cert.org/secure-coding/products-services/secure-coding-download.cfm
- Basis for ISO TS 17961 C Secure Coding Rules

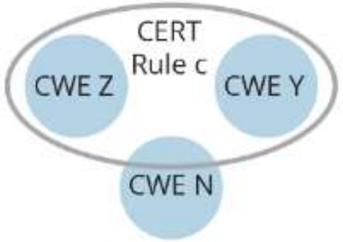
SEI CERT C++ Coding Standard

- Free PDF download (Released March 2017): cert.org/secure-coding/products-services/secure-coding-cpp-download-2016.cfm

CERT Oracle Secure Coding Standard for Java

- Latest guidelines available on CERT Secure Coding wiki: securecoding.cert.org

Available Technology: Secure Coding Standards (cont’d)



Precise mappings: Defines *what kind* of relationship, and if overlapping, *how.* Also *when* mapped and *which versions*.

Imprecise mappings → Precise mappings
(“*some* relationship”)

Precise mappings on CERT C Standard wiki

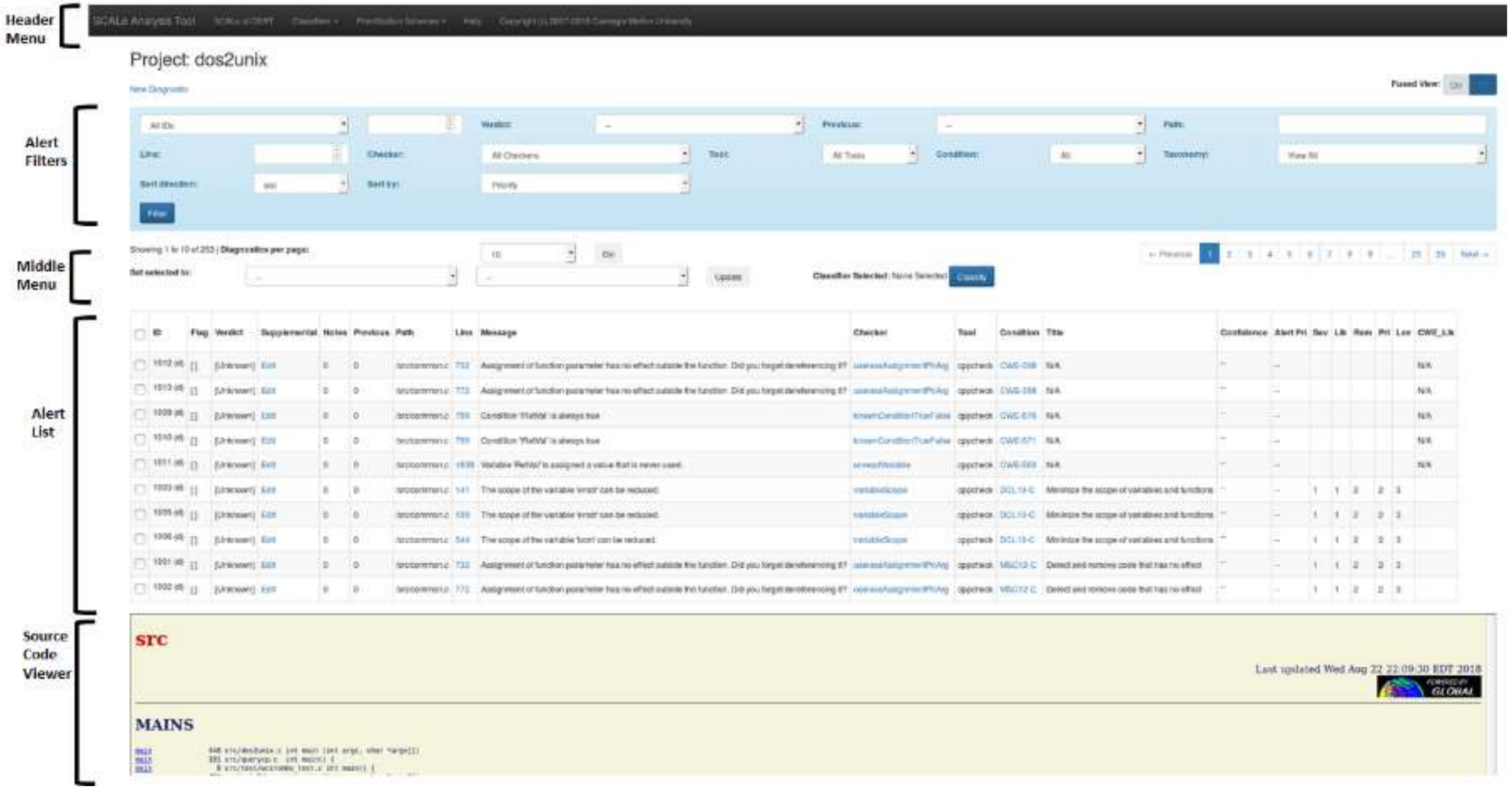
1. Per-rule CWE precise mapping
   - “CERT-CWE Mapping Notes” (set notation)
   - Table with taxonomy and relationship detail
2. Metadata for using Juliet Test Suite to test CERT rule coverage
   - Plan: create similar metadata for STONESOUP and other test suites

For code flaws you care about, understand your tool coverage

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

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

Available Technology: SCALe Static Analysis Alert Auditing Tool



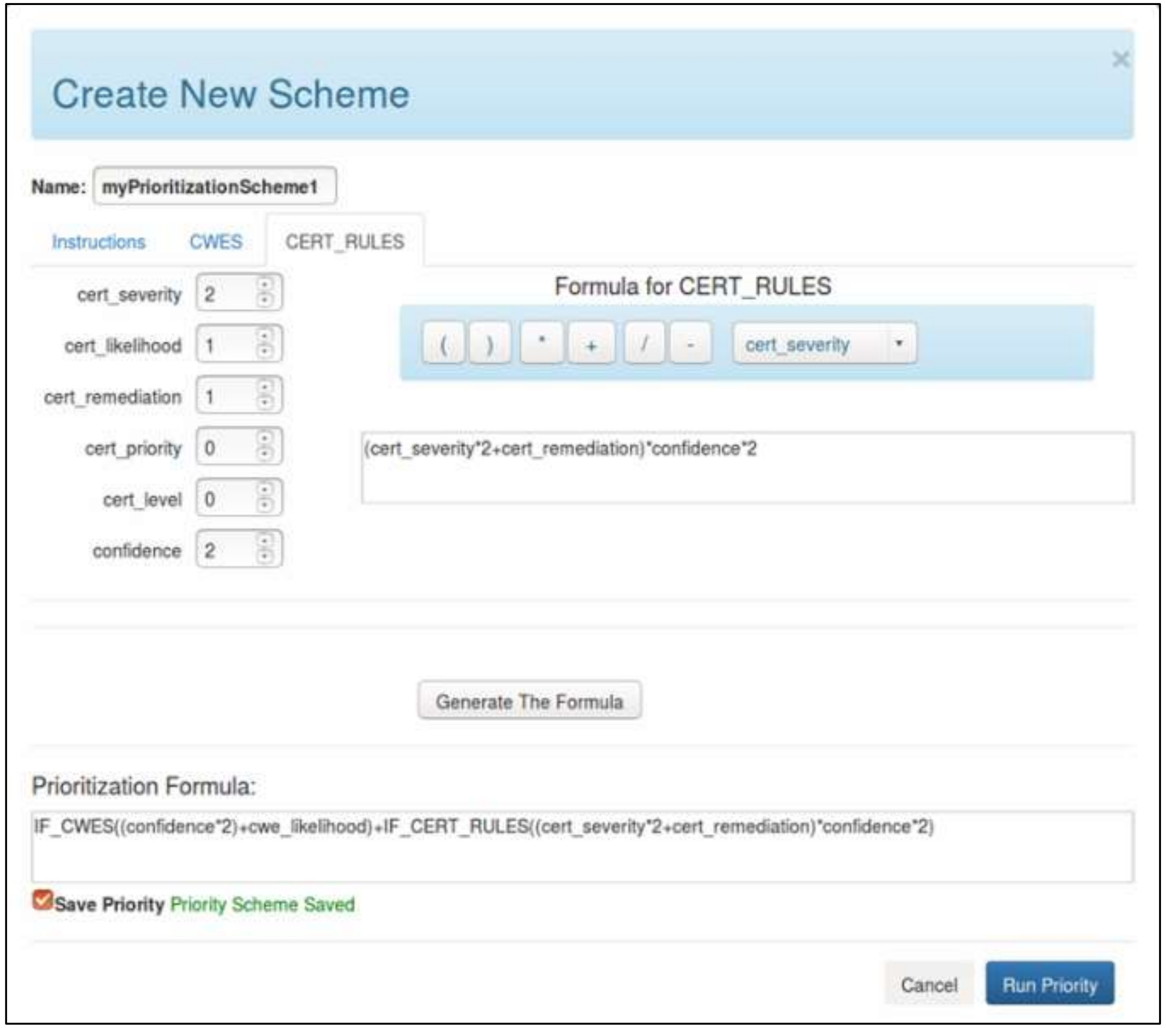
Used as a research platform

- Extend with new features
- Collaborators give us feedback
- Collaborators generate data required for our classifier research

Over last 3 years, new SCALe features are for classification and prioritization research.

- GitHub public release (SCALe v2), Aug. 2018
- SCALe v3 for research project collaborators

Available Technology: SCALe Static Analysis Alert Auditing Tool



Recent features include

- Alert fusion for {filepath, line, condition} reduces auditor effort
- Determinations history
- Automatically cascaded determinations from previous audits
- Classification schemes
- Prioritization schemes with mathematical formulas user can create and/or use
- User field uploads

Available Technology: Automated Code Repair to Ensure Memory Safety

Goal: Take a C codebase and repair potential bugs to enable a proof of memory safety.

What about distinguishing false alarms from true vulnerabilities?

- We simply apply a repair to all potential memory-safety vulnerabilities, at a cost of an often small runtime overhead. (Manual tuning might be needed for performance-critical parts.)

Available technology: Repair of integer overflows that lead to buffer overflows.

- Inferred specification: **inequality comparisons** involving array indices or bounds should behave as if normal (non-overflowing) arithmetic were used.
  - Includes `malloc`.
  - Excludes hash functions and crypto, where modular arithmetic is desired.
- We repair the code to satisfy this spec where possible.
- Tested on older versions OpenSSL and Jasper. Found and repaired known vuls with CVEs.