

SCAIFE and ACR

Static Analysis Classification and Automated Code Repair

For SwA CoP/NNSA
September 2021

Dr. Lori Flynn

Dr. William Klieber

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Copyright 2021 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.

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.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

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.

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

DM21-0791

Rapid Adjudication of Static Analysis Results During CI

with SCAIFE modular static analysis classification system

Dr. Lori Flynn (PI)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

SEI Team:

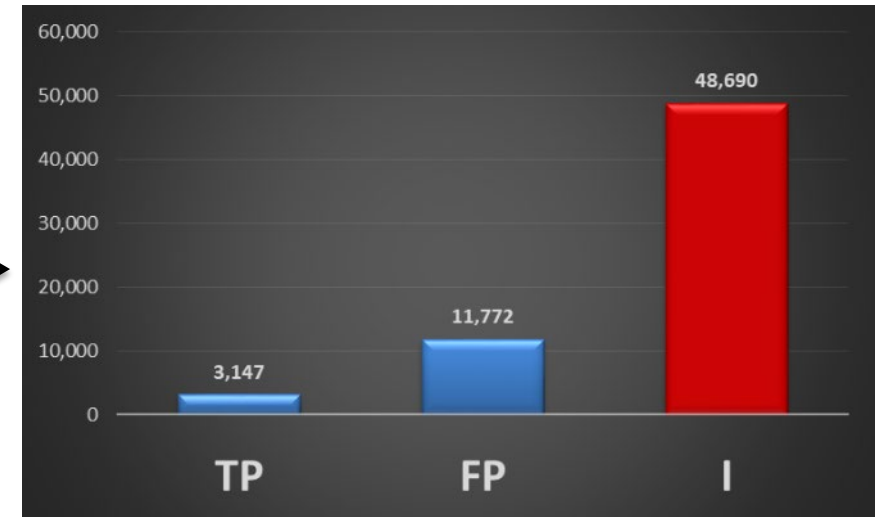
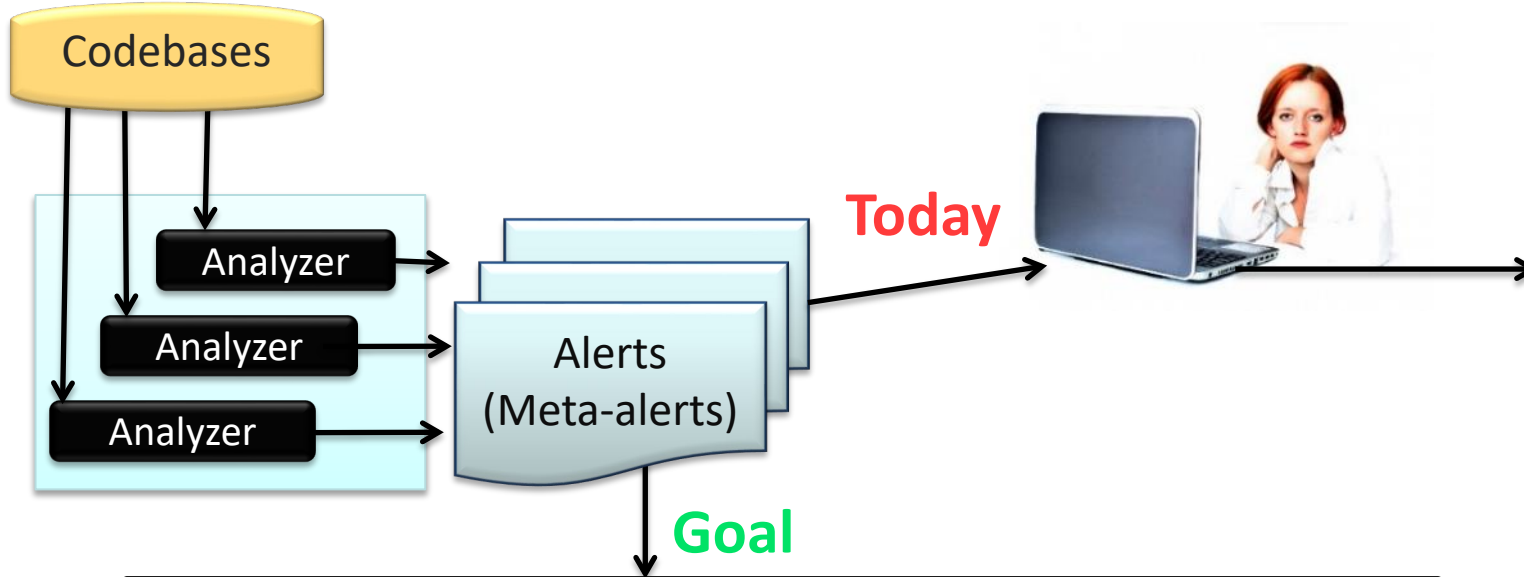
Matt Sisk
Ebonie McNeil
David Svoboda
Joseph Yankel
Tyler Brooks
Dustin Updyke
Jeffrey Mellon
Rhonda Brown
Lyndsi Hughes (tester)
Joseph Sible (tester)
Wei-ren Murray (tester)

SCAIFE

classification system

Problem: too many static analysis alerts
Solution: automate handling

A **meta-alert** is a static analysis result for a particular line number, filepath, and code flaw condition (e.g., CWE-190).



Continuous Integration (CI) – optional Systems that precisely and with high recall, classify at least as many manually-adjudicated meta-alerts as:

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

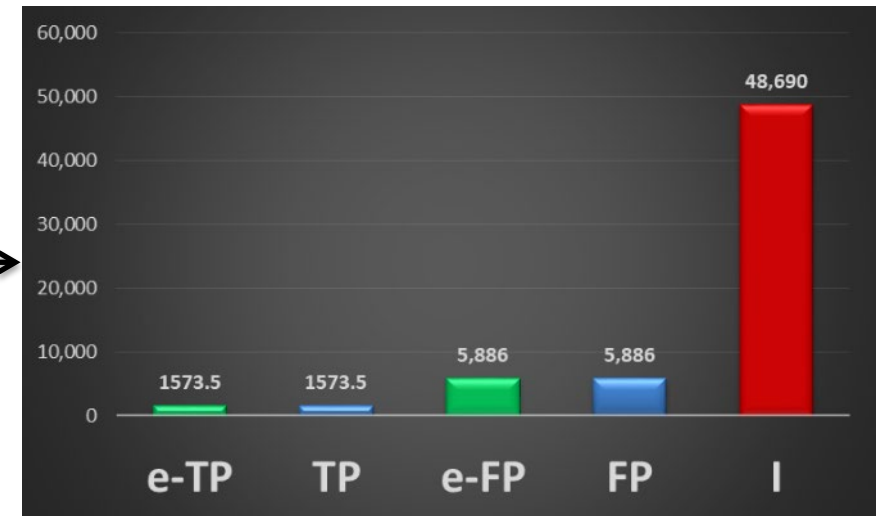


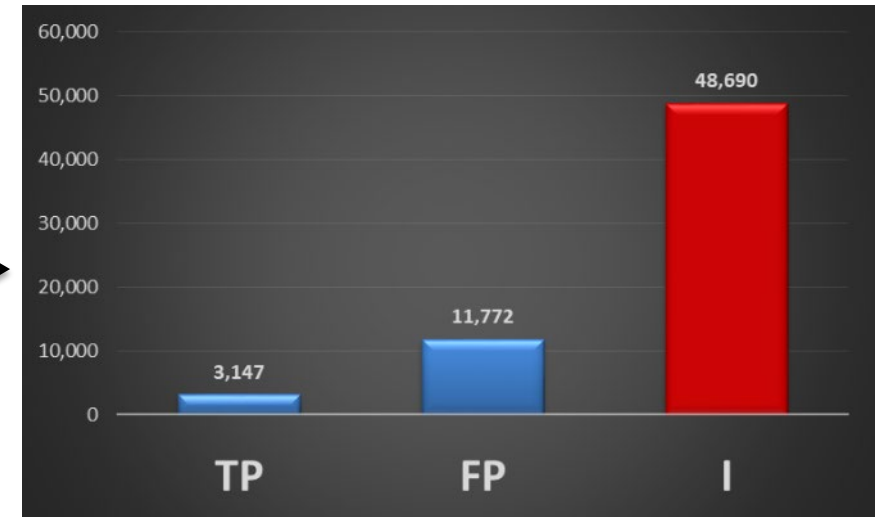
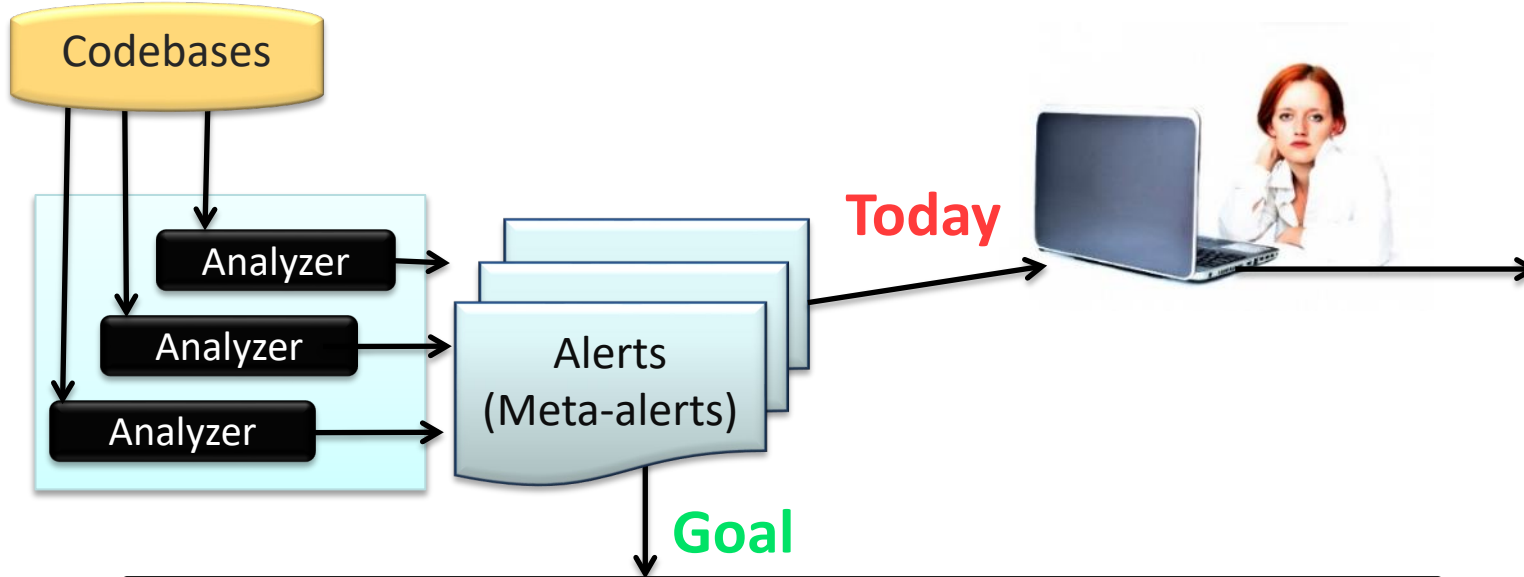
Image of woman and laptop from <http://www.publicdomainpictures.net/view-image.php?image=47526&picture=woman-and-laptop> "Woman And Laptop"

SCAIFE

classification system

Problem: too many static analysis alerts
Solution: automate handling

A **meta-alert** is a static analysis result for a particular line number, filepath, and code flaw condition (e.g., CWE-190).



CI-optional systems that **precisely and with high recall, classify at least as many manually-adjudicated meta-alerts as:**

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

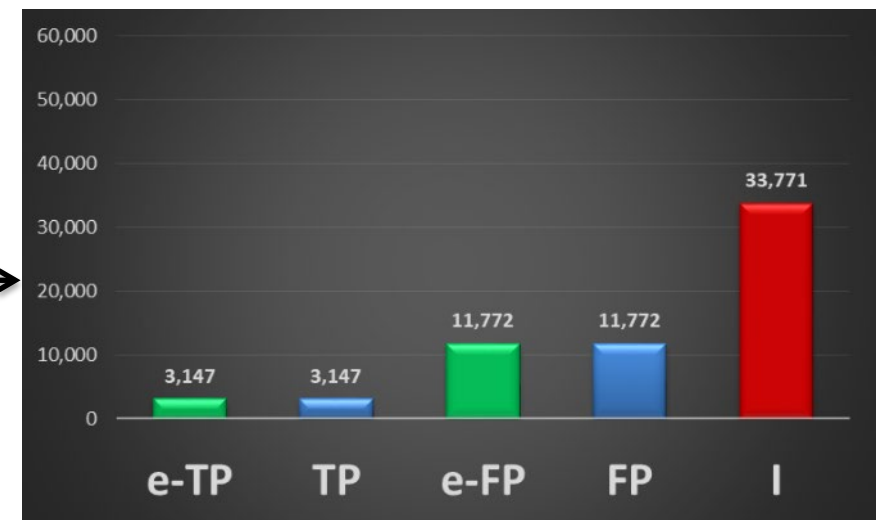


Image of woman and laptop from <http://www.publicdomainpictures.net/view-image.php?image=47526&picture=woman-and-laptop> "Woman And Laptop"

SCAIFE Static Analysis Classifiers Detail

Designed for use by machine learning novices, with settings that can be tweaked by experts

Labeled static analysis meta-alerts used to create classifiers:

- Manually adjudicated meta-alerts (true positive, false positive)
- Test suites (e.g., Juliet): SCAIFE automatically adjudicates meta-alerts
- User chooses labeled data sets, classifier, active learning, and other options

Modular ability to add different types of classifiers, active learning, and hyper-parameter optimization methods.

Built-in options:

- Classifiers: XGBoost, Random Forest, LightGBM
- Active Learning (adaptive heuristics): Similarities, K-Nearest Neighbors, and Label Propagation
- Hyper-parameter optimization: Bayesian Optimization

SCAIFE Classification System

Designed to be used in a wide variety of systems, with many other tools

Full SCAIFE system includes all 5 modules

Modular system designed to work with different user interfaces and static analysis tools

- SARIF static analysis format
- SCARF format (DHS SWAMP)
- Various tools and versions, with standard method for adding new tools

Use SCAIFE for a single code version or a codebase in a CI system

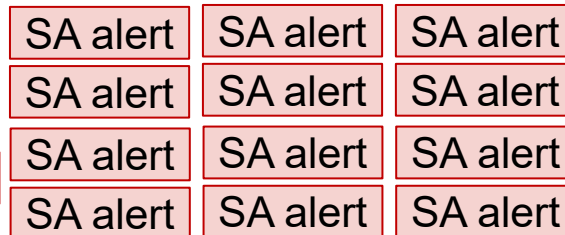
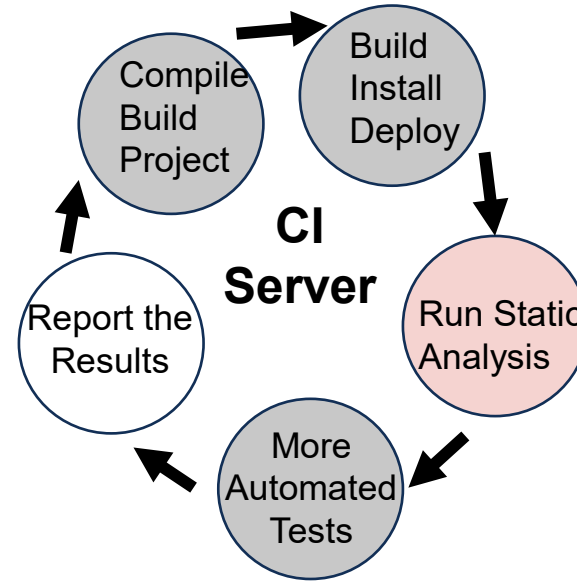
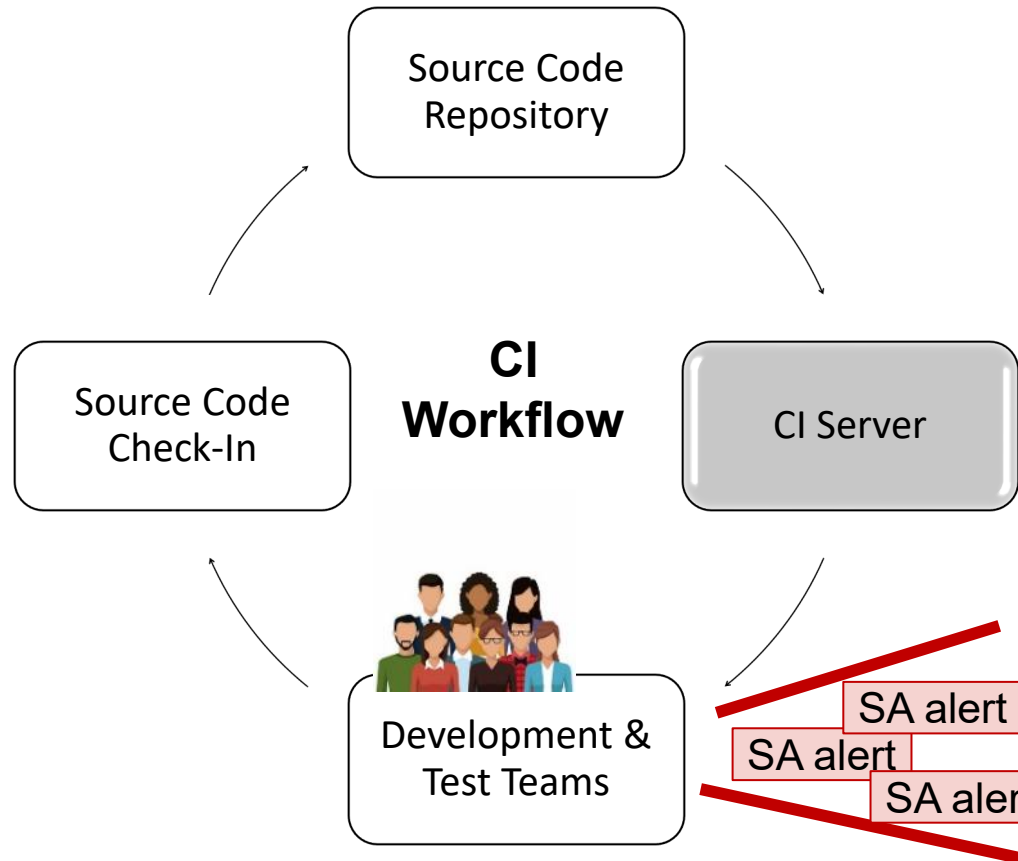
- CI system: updates to code and static analysis for the new code version

FY20-21: Rapid Adjudication of Static Analysis Results During CI



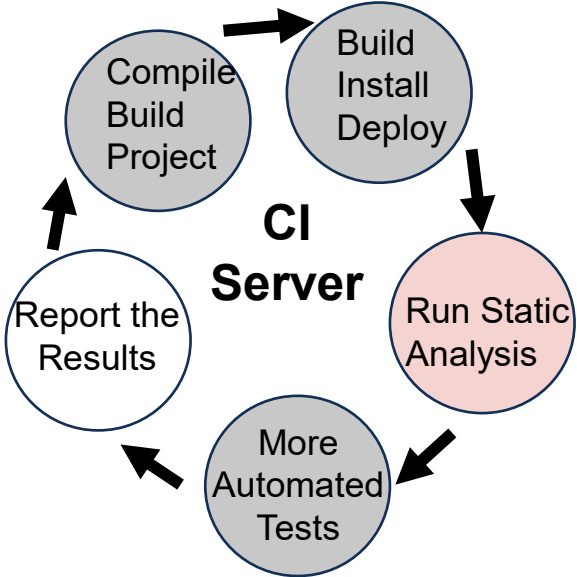
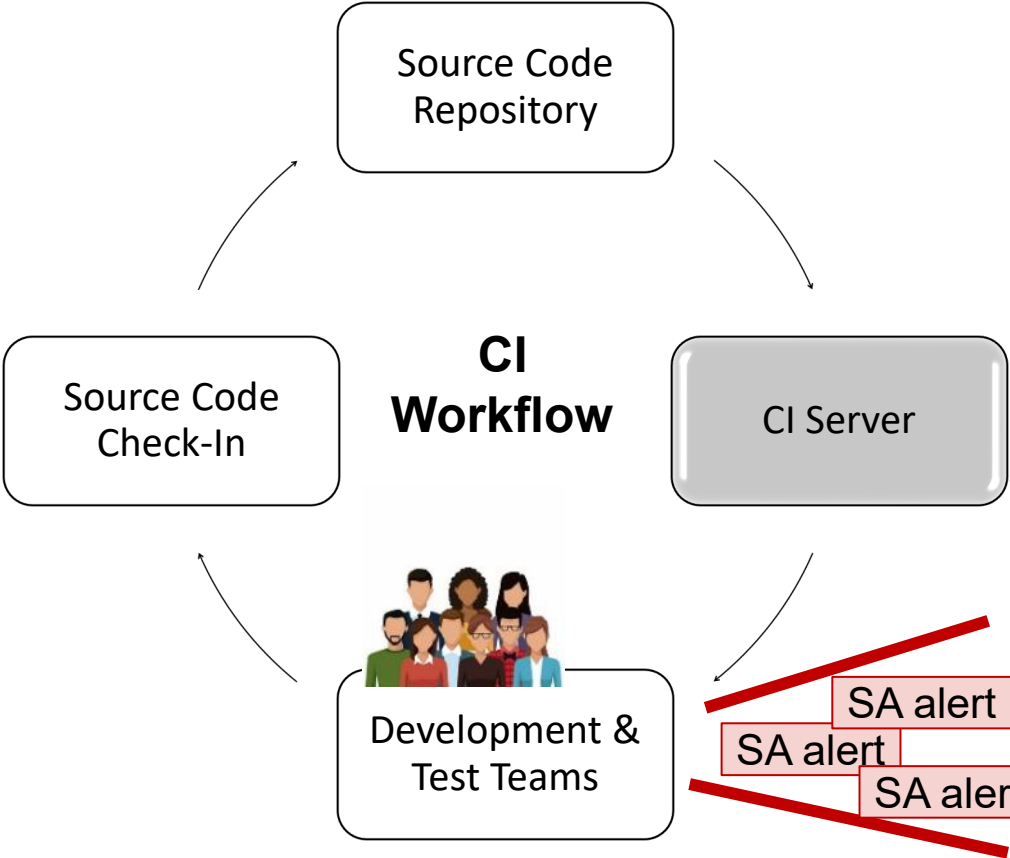
- **Issue addressed:** It takes too much time to adjudicate (i.e., audit) static analysis meta-alerts during continuous integration (CI).
- **Novel approach:** During CI builds, use **classifiers** with **precise cascading** and **CI/CD features**.

Rapid Adjudication of Static Analysis (SA) Meta-Alerts During CI



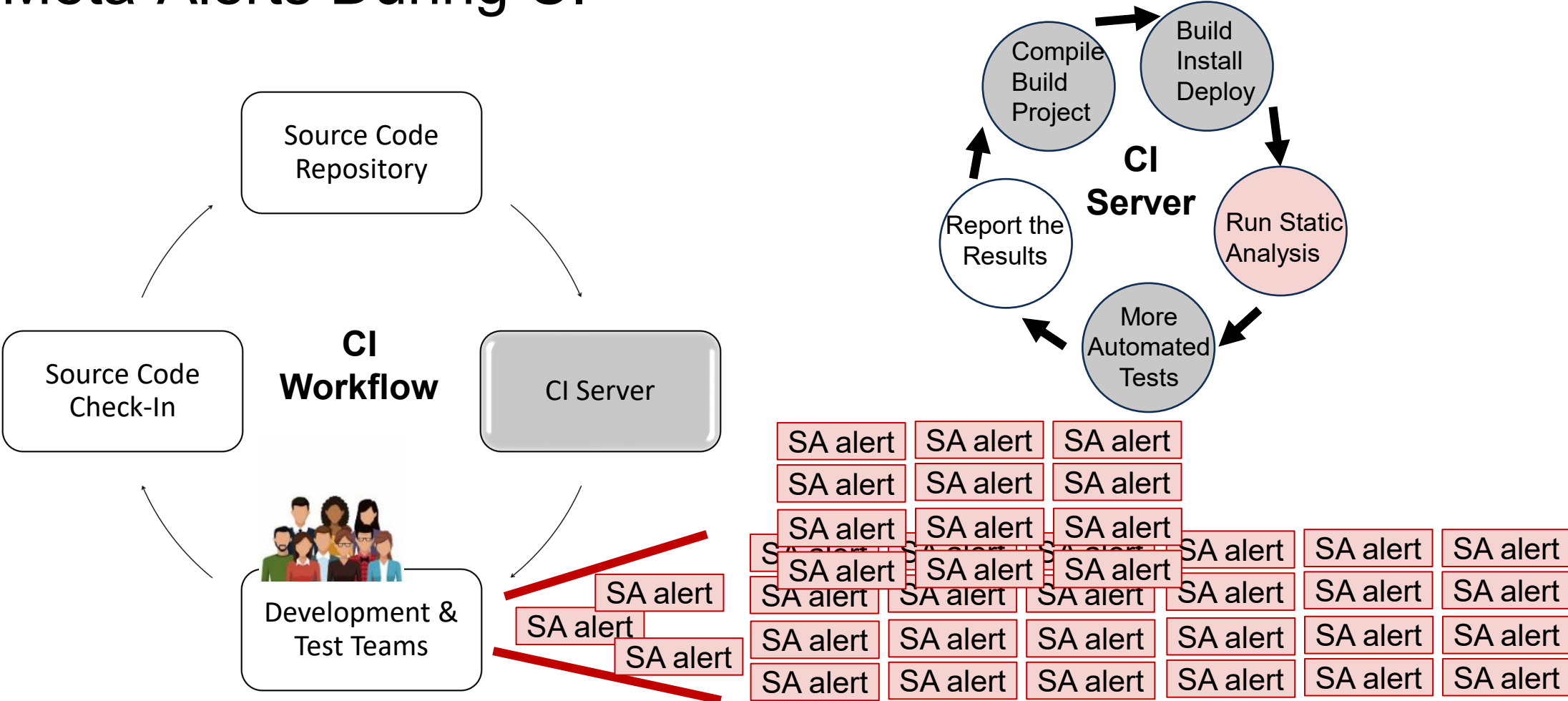
Alert: An SA warning (with a tool checker ID, line #, filepath, message)
AlertCondition: An alert mapped to a code flaw taxonomy item (e.g., CWE-190)
Meta-alert: mapped to by the set of alertConditions that differ only by checker ID.
Adjudication and classification at the meta-alert level.

Rapid Adjudication of Static Analysis Meta-Alerts During CI

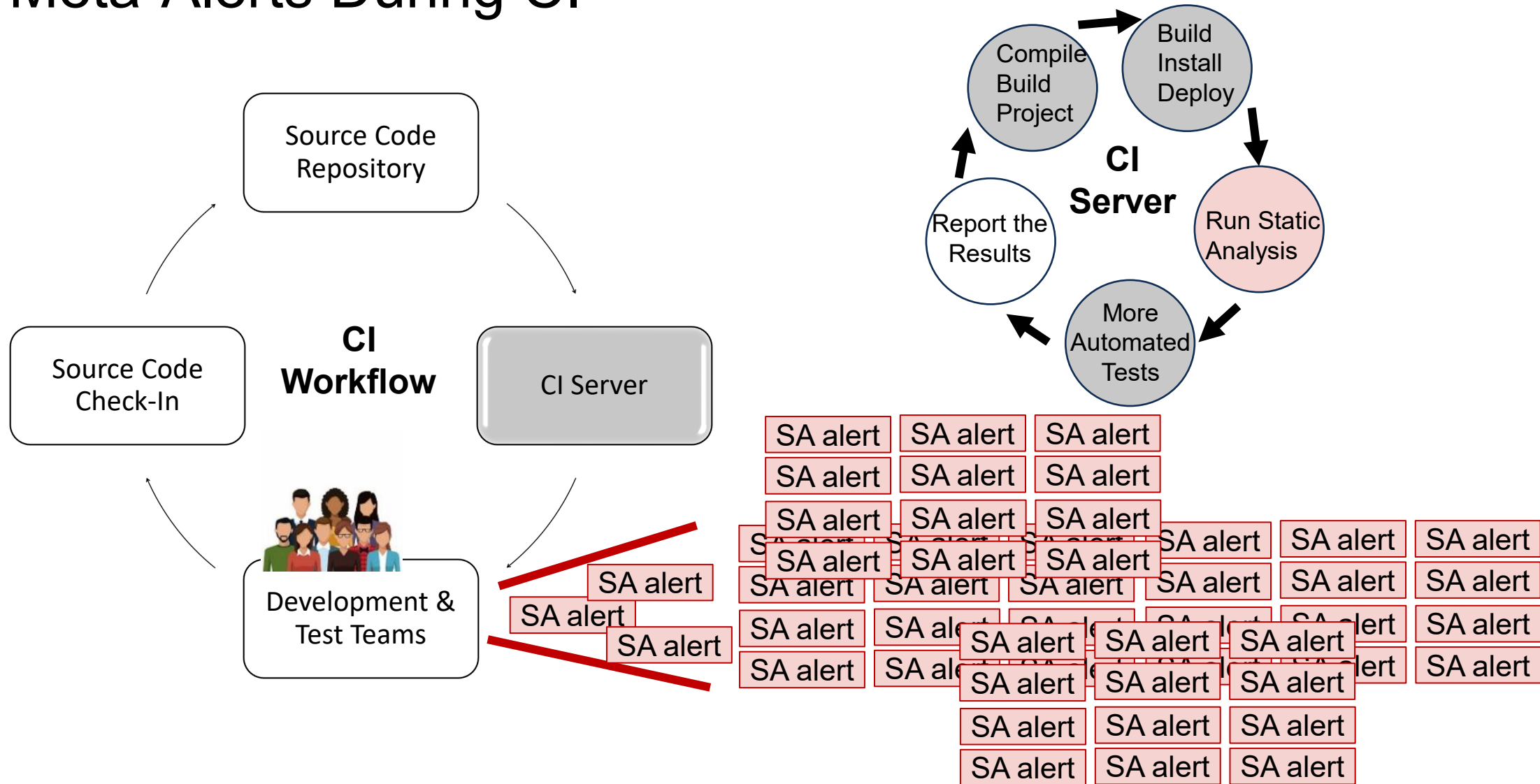


SA alert	SA alert	SA alert	SA alert	SA alert	SA alert
SA alert	SA alert	SA alert	SA alert	SA alert	SA alert
SA alert	SA alert	SA alert	SA alert	SA alert	SA alert
SA alert	SA alert	SA alert	SA alert	SA alert	SA alert

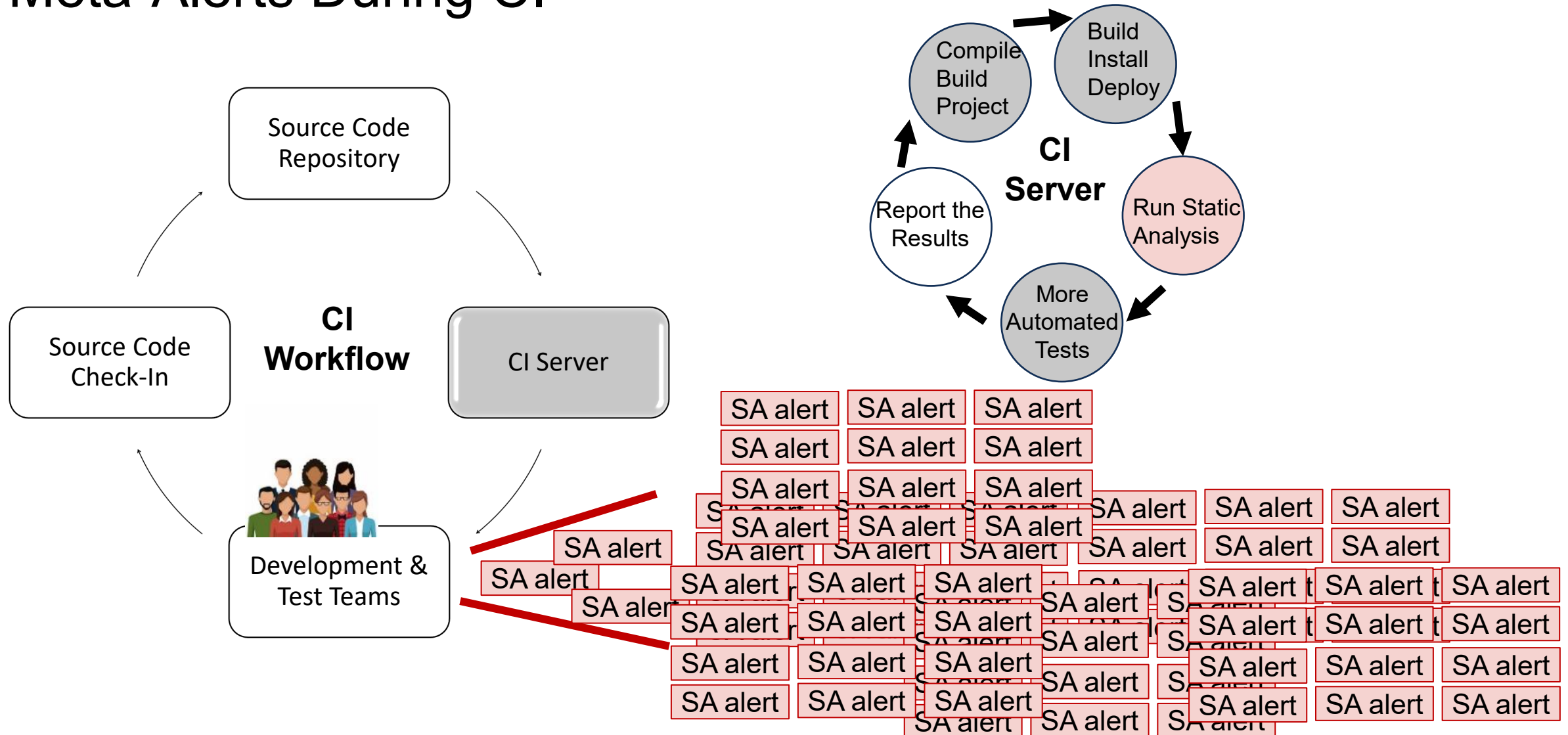
Rapid Adjudication of Static Analysis Meta-Alerts During CI



Rapid Adjudication of Static Analysis Meta-Alerts During CI

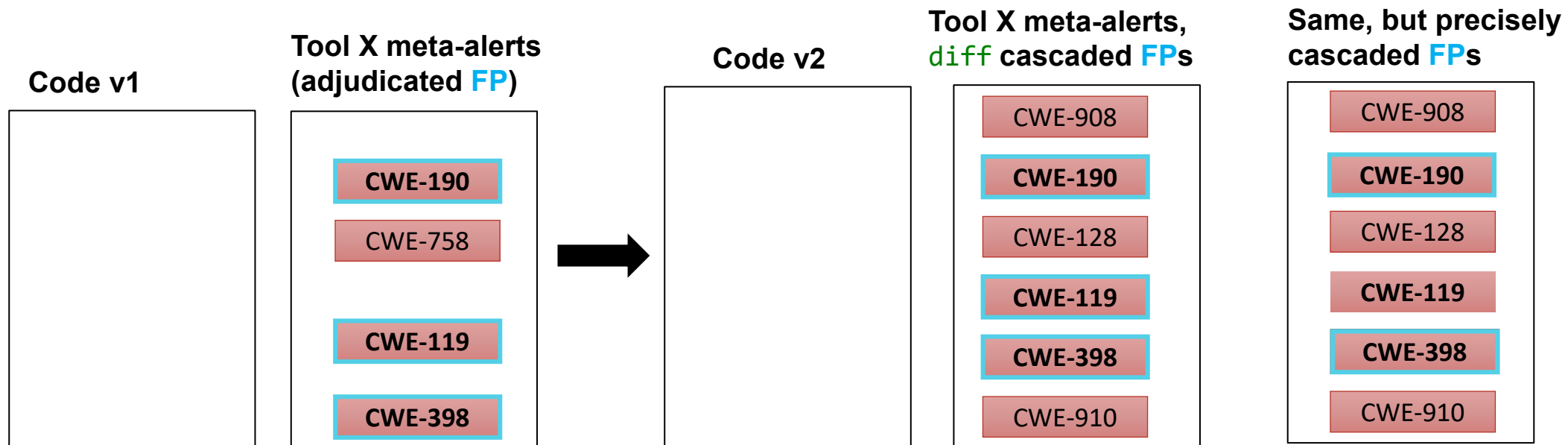


Rapid Adjudication of Static Analysis Meta-Alerts During CI



SCAIFE has 2 Types of Meta-Alert Adjudication Cascading

- For code versions 1 and 2, can a manual adjudication (e.g., true, false) for a meta-alert from v1 be applied to a meta-alert for code v2?
- Imprecise cascading happens on a per-file analysis and uses regular expression and/or line numbers.
- Precise cascading means analysis across a whole program using control flow, data flow, and type flow.



SCAIFE Architecture

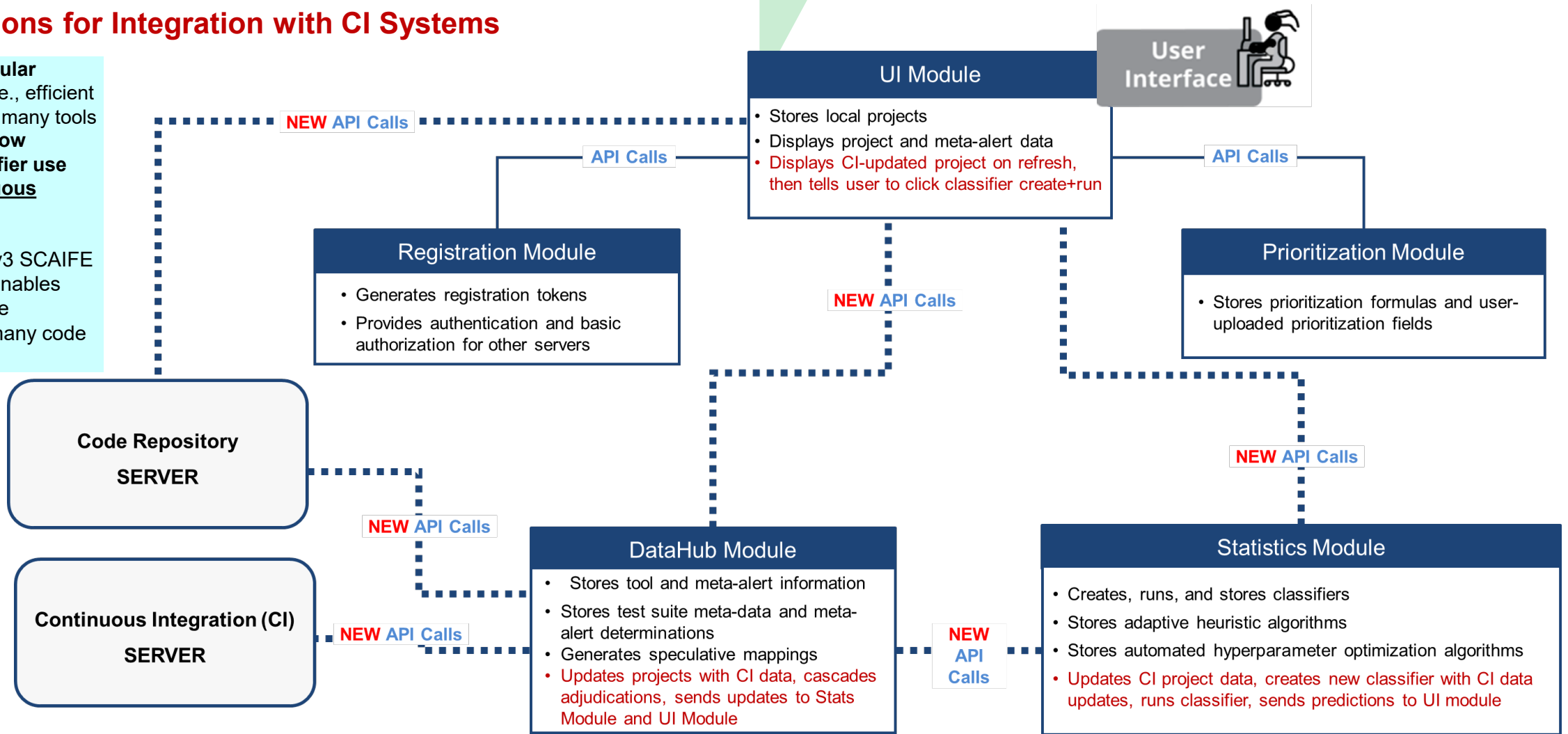
Modifications for Integration with CI Systems

SCAIFE's modular architecture (i.e., efficient integration with many tools and systems) **now enables classifier use during continuous integration.**

The OpenAPI v3 SCAIFE API definition enables automated code generation in many code languages

Any static analysis tool can instantiate APIs to become a UI Module. For example

- SEI SCALe
- DHS SWAMP
- CCDC C5ISR SwAT
- Other aggregator tools
- Single static analysis tools

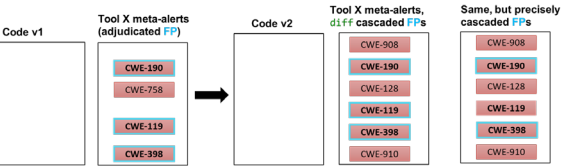


Goal: Enable practical automated classification, so all meta-alerts can be addressed

FY21 Select Artifacts

- **Adjudication cascading**

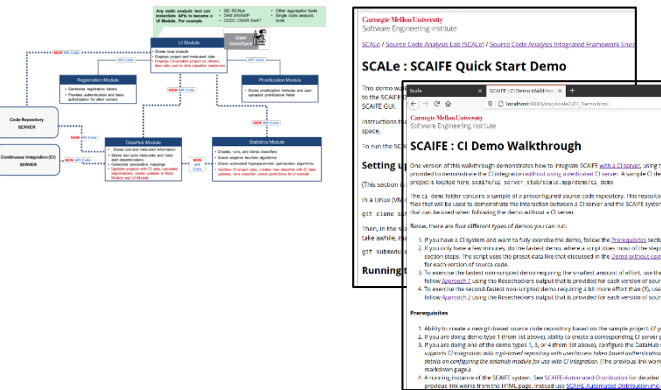
- Diff-based cascading integrated in SCAIFE
- Generated diff-based cascading test data for comparison with precise cascading
- Collaboration with Dr. Le's team from Iowa State University:
 - Precise cascader development
 - Generated test data for precise and diff-based cascading



scaife.online.2.0.0.tar.gz (138 MB)

- **SCAIFE v2.0.0 release**

- automatically uses CI update data (code, static analysis) to update SCAIFE projects
- Includes hands-on demos to walk users through it on their own system
 - 4 demos: One for use with full CI systems, others for use without CI systems
- Additional training demos, code, and documentation enhancements



Release ok for DoD contractors, now!
 (Previously Dist F: only for 5 DoD orgs)
 scaife.online.2.0.1.tar.gz (115 MB)

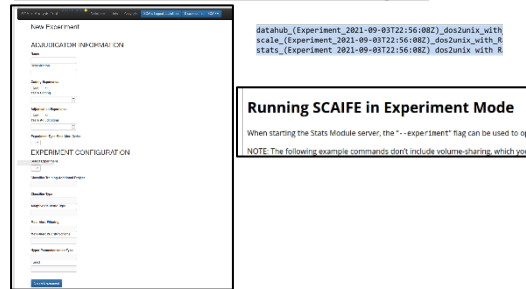
- **First Dist D SCAIFE release: SCAIFE v2.0.1**

FY21 Select Artifacts

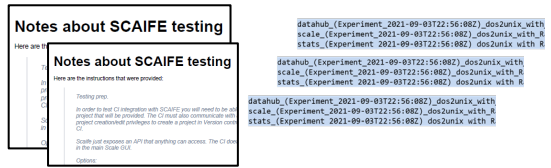


- **Paper** "Test suites as a source of training data for static analysis alert classifiers" by Lori Flynn, William Snaveley, and Zachary Kurtz to ICSE-associated Conference on Automation of Software Test (AST) 2021 <https://conf.researchr.org/home/icse-2021/ast-2021> and video <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=737855>

scaife.online.3.0.0.tar.gz

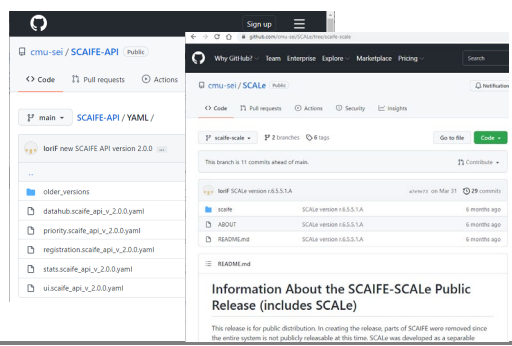


- **SCAIFE v3 release** (this week)
 - Contains much-enhanced performance metrics collection:
 - Experiment mode, auto-setup experiments with configuration files + datasets, collect metrics, auto-end, and export data
 - Metrics include (among others): classifier precision and recall, counts of adjudicated vs. high-confidence predicted, and key step latencies, CPU use (max, avg), bandwidth use (max, avg), memory use (max, avg)
 - Java test suites now fully usable by SCAIFE
 - CI updates in SCAIFE include Stats module and new classifier predictions



- **SCAIFE release test results and analysis:**
 - SEI CI experts did the CI demo, provided feedback (Lyndsi Hughes and Joe Sible)
 - External collaborators started testing SCAIFE v2 and providing feedback
 - Next (hopefully FY21!): SCAIFE v3 collaborator testing, analysis of test results

- **GitHub publications of SCAIFE API** <https://github.com/cmu-sei/SCAIFE-API> (Sept'21 release soon)
- **GitHub publications of SCAIFE UI module (SCALE) code** at <https://github.com/cmu-sei/SCALE/tree/scaife-scale> (Sept'21 release soon)



Current Work

- Writing research papers
- Gathering and analyzing test data.
- Interested potential testers welcome!
- Special report for collaborators

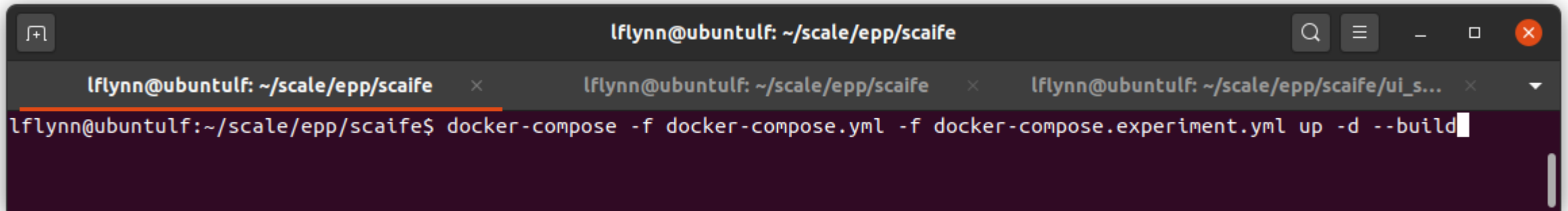
Detail on testing

- Interested potential testers welcome!

Involves

1. SCAIFe startup (docker-compose command on Linux or Mac machine)

Start SCAIFE



```
lflynn@ubuntulf: ~/scale/epp/scaife
lflynn@ubuntulf: ~/scale/epp/scaife
lflynn@ubuntulf: ~/scale/epp/scaife/ui_s...
lflynn@ubuntulf:~/scale/epp/scaife$ docker-compose -f docker-compose.yml -f docker-compose.experiment.yml up -d --build
```

Detail on testing

- Interested potential testers welcome!

Involves

1. SCAIFe startup (docker-compose command on Linux or Mac machine)
2. Select experiment from list (rest auto-fills)

Select Experiment in Drop-Down

SCAIFe Analysis Tool SCALE at CERT Definitions Help Copyright SCALE Logout: scaleUser Disconnect from SCAIFe

New Experiment

ADJUDICATOR INFORMATION

Name

Organization

Coding Experience

Years Coding

Adjudication Experience

Years Adjudicating

Experiment Type Meta Alert Order

EXPERIMENT CONFIGURATION

Select Experiment:

Classifier Training Additional Project

Classifier Type

Adaptive Heuristic Type

Meta-Alert Filtering

Meta-Alert Priority Scheme

Hyper-Parameterization Type

Fused

Detail on testing

- Interested potential testers welcome!

Involves

- SCAIFe startup (docker-compose command on Linux or Mac machine)
- Select experiment from list (rest auto-fills)
- Adjudicate meta-alerts, working from top of list. (list re-orders after adjudications)

Adjudicate Meta-alerts (Do Top of List)

Project: dos2unix/rosecheckers project

New Alert + Condition Order by Path ASC, Line ASC Fused View: On

All IDs: Verdict: Unknown Previous: -- Path:

Line: Checker: All Checkers Tool: All Tools Condition: All Taxonomy: View All

Category: All Shuffle Seed:

Filter Clear Filters

Showing 1 to 10 of 218 | Meta-alerts per page: 10 Go Previous 1 2 3 4 5 6 7 8 9 ... 21 22 Next

Select all 218 Meta-alerts: SCAIFe Mode: Connected Run Classifier: automation: Random Forest 0 Classify

Manual Verdicts: 20
Predicted Verdicts: 0

ID	Flag	Verdict	Supplemental	Notes	Previous	Path	Line	Message	Checker	Tool	Condition	Title	Label	Confidence	Category	AlertCondition Pri	Sev	Lik	Rem	Pri	Lev	CWE_Lik
381 (d)	[]	[Unknown]	Edit	0	0	/dos2unix-7.2.2/common.c	265	Exclude user input from format strings	FIO30-C	rosecheckers_oss	FIO30-C	Exclude user input from format strings	false	0.0	I		3	3	2	18	1	
382 (d)	[]	[Unknown]	Edit	0	0	/dos2unix-7.2.2/common.c	267	Exclude user input from format strings	FIO30-C	rosecheckers_oss	FIO30-C	Exclude user input from format strings	false	0.0	I		3	3	2	18	1	
383 (d)	[]	[Unknown]	Edit	0	0	/dos2unix-7.2.2/common.c	268	Exclude user input from format strings	FIO30-C	rosecheckers_oss	FIO30-C	Exclude user input from format strings	false	0.0	I		3	3	2	18	1	
384 (d)	[]	[Unknown]	Edit	0	0	/dos2unix-7.2.2/common.c	270	Exclude user input from format strings	FIO30-C	rosecheckers_oss	FIO30-C	Exclude user input from format strings	false	0.0	I		3	3	2	18	1	
385 (d)	[]	[Unknown]	Edit	0	0	/dos2unix-7.2.2/common.c	276	Exclude user input from format strings	FIO30-C	rosecheckers_oss	FIO30-C	Exclude user input from format strings	false	0.0	I		3	3	2	18	1	
386 (d)	[]	[Unknown]	Edit	0	0	/dos2unix-7.2.2/common.c	277	Exclude user input from format strings	FIO30-C	rosecheckers_oss	FIO30-C	Exclude user input from format strings	false	0.0	I		3	3	2	18	1	

```
261 printf(_(" -863 use DOS code page 863 (French Canadian)\n"));
262 printf(_(" -865 use DOS code page 865 (Nordic)\n"));
263 printf(_(" -7 convert 8 bit characters to 7 bit space\n"));
264 if (is_dos2unix(program))
265 printf(_(" -b, --keep-bom keep Byte Order Mark\n"));
266 else
267 printf(_(" -b, --keep-bom keep Byte Order Mark (default)\n"));
268 printf(_(" -c, --convmode conversion mode\n"));
269 convmode ascii, 7bit, iso, mac, default to ascii\n"));
270 printf(_(" -f, --force force conversion of binary files\n"));
271 #ifdef DO_UNICODE
272 #if (defined(MINGW2) && !defined(CYGWIN))
273 printf(_(" -gb, --gb18030 convert UTF-16 to GB18030\n"));
274 #endif
275 #endif
276 printf(_(" -h, --help display this help text\n"));
277 printf(_(" -i, --info=FLAGS display file information\n"));
278 file ... files to analyze\n");
```

Detail on testing

- Interested potential testers welcome!

Involves

1. SCAIFe startup (docker-compose command on Linux or Mac machine)
2. Select experiment from list (rest auto-fills)
3. Adjudicate meta-alerts, working from top of list. (list re-orders after adjudications)
4. Send exported 3 files back if possible, else just provide qualitative feedback on testing.

Experiment End: Exports 3 Files

```
datahub_(Experiment_2021-09-03T22:56:08Z)_dos2unix_with_Random_Forest_and_K-Nearest_Neighbors.json  
scale_(Experiment_2021-09-03T22:56:08Z)_dos2unix_with_Random_Forest_and_K-Nearest_Neighbors.json  
stats_(Experiment_2021-09-03T22:56:08Z)_dos2unix_with_Random_Forest_and_K-Nearest_Neighbors_2021-09-03_22:58:46_324347.json
```

Carnegie Mellon University
Software Engineering Institute

Automated Code Repair to Ensure Spatial Memory Safety

Dr. Will Klieber(PI)

SEI Team:
Ryan Steele
Matt Churilla
David Svoboda
Mike McCall
Ruben Martins (CMU SCS)

Automated Code Repair (ACR) for Memory Safety

Problem: Software vulnerabilities constitute a major, growing threat.

- Spatial memory violations are among the most common and most severe types of vulnerabilities.
 - 15% of CVEs in the NIST NVD and 24% of critical-severity CVEs.
 - iPhone iOS CVE-2019-7287 (exploited by Chinese government, according to <https://techcrunch.com/2019/08/31/china-google-iphone-uyghur/>)
 - Android Stagefright (2015)

Automated Code Repair (ACR) tool as a black box

Solution: Automatically repair source code to assure spatial memory safety. Abort program (or call error-handling routine) before memory violation.

Approach: Repair program to use *fat pointers* to track bounds and insert a bounds check before memory accesses.

Input: Buildable codebase written in C

Output: Repaired source code that is still human-readable and maintainable

ACR Tool



Why repair at the source-code level?

Repair of source code

Repair as a compiler pass

Easily audited (if desired).

Must trust the tool.

Repairs can easily be tweaked to improve performance, if necessary.

Difficult to remediate performance issues caused by repair.

Changes to source code are frequent and easily handled.

Changes to the build process may be more difficult, more error-prone, and create unwanted dependencies.

Okay to do slow, heavy-weight static analysis; produces a persistent artifact.

Slowing down every build is not okay.

Fat pointers

We replace raw pointers with **fat pointers**:

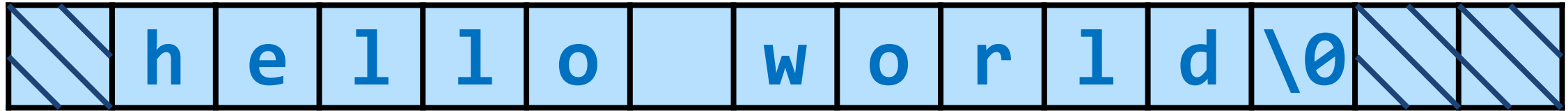
- A *fat pointer* is a struct that includes the pointer itself as well as bounds information.
- Before dereferencing a fat pointer, a bounds check is performed.
- For each pointer type T^* , we introduce a fat-pointer type defined as follows:

```
struct FatPtr_T {  
    T*      rp;    /* raw pointer */  
    char*   base; /* of allocated memory region */  
    size_t  size; /* of allocated memory region, in bytes */  
};
```

Fattening of pointers has been performed as a compiler pass:

- Todd Austin et al. “Efficient detection of all pointer and array access errors.” *PLDI*, 1994.
- Wei Xu et al. “An efficient and backwards-compatible transformation to ensure memory safety of C programs.” *ACM SIGSOFT*, 2004.

Fat pointer example



Original:
 p

Repaired:
 $p.rp$
 $p.base$
 $(p.base + p.size - 1)$

Example of tool output

Original Source Code

```
1
2
3 #define BUF_SIZE 256
4 char nondet_char();
5
6 int main() {
7     char* p = malloc(BUF_SIZE);
8     char c;
9     while ((c = nondet_char()) != 0) {
10         *p = c;
11         p = p + 1;
12     }
13     return 0;
14 }
```

Repaired Source Code

```
1 #include "fat_header.h"
2 #include "fat_stdlib.h"
3 #define BUF_SIZE 256
4 char nondet_char();
5
6 int main() {
7     FatPtr_char p = fatmalloc_char(BUF_SIZE);
8     char c;
9     while ((c = nondet_char()) != 0) {
10         *bound_check(p) = c;
11         p = fatp_add(p, 1);
12     }
13     return 0;
14 }
```

Performance overhead

Full Repair: Guard every memory access.

Partial Repair: Run static-analysis tool and insert bounds checks only for memory accesses flagged by the static analyzer.

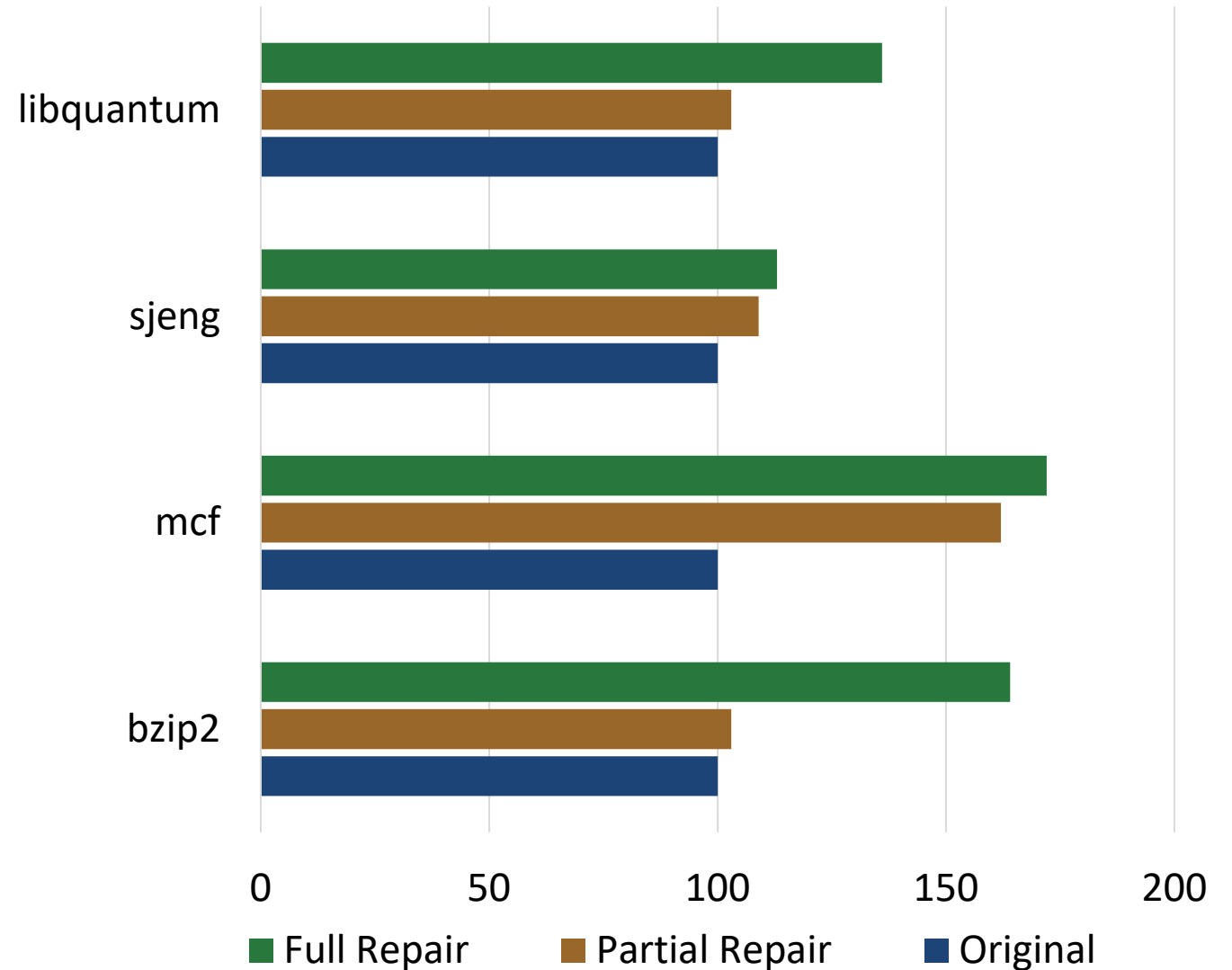
Small performance overhead:

- 3% for libquantum
- 9% for sjeng
- 3% for bzip2

Still large overhead for programs with many pointers:

- 62% for mcf

(Currently we still fatten all pointers, even if we never use the bounds.)



Limitations

We cannot guarantee memory safety in the presence of:

- Non-standard pointer tricks (e.g., XOR-linked lists)
- Reuse of memory for different types (except via unions)
- Concurrency
 - Race conditions can cause memory corruption
- External code that accesses program memory
 - If the program's data structures are accessed by external binary code, the pointers inside them cannot be fattened.
 - We can identify such pointers using a whole-program points-to analysis with an *allocation-site* abstraction.

Carnegie Mellon University
Software Engineering Institute

Decompilation for Binary Software Assurance

Dr. Will Klieber(PI)

SEI Team:
David Svoboda
Ruben Martins (CMU SCS)

Decompilation for Binary Software Assurance (FY21—22)

- Goal: Increase software assurance of components available only in binary form.
 - Decompile and perform static analysis on decompiled code.
 - Make localized repairs to functions of the binary.
- We are adapting an existing open-source decompiler (Ghidra) to produce decompiled code suitable for static analysis and repair.
 - Existing decompilers were developed for aiding manual reverse engineering.
 - They were not designed to produce recompilable code.
 - Gap: Decompiled code often has semantic inaccuracies and syntactic errors.
- Key technical steps:
 - Determine which individual functions have been correctly decompiled.
 - Run static analysis and localized repair on correctly decompiled functions.
 - Recombine repaired functions with the original binary files (e.g., using DDisasm).

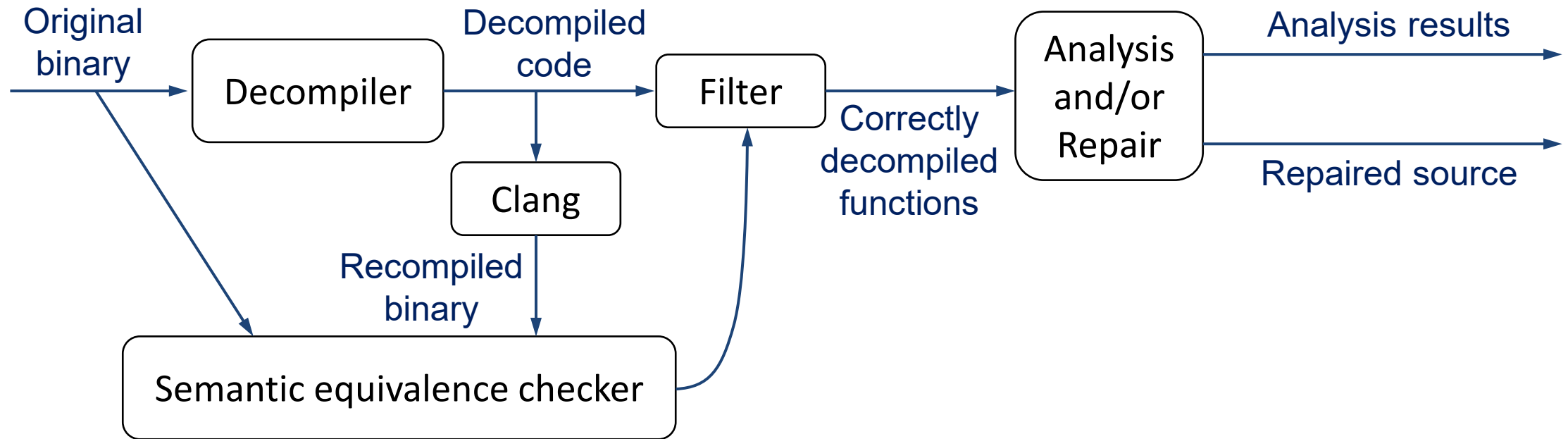
Decompilation for Binary Software Assurance (continued)

- A perfect decompilation of the entire binary isn't necessarily required to get significant benefit, as long as enough relevant functions can be correctly decompiled.
- Main contributions of our work:
 - We develop a semantic-equivalence checker to determine which individual functions are correctly decompiled.
 - We improve the syntactic and semantic correctness of decompiler output. We will offer our improvements to the mainline branch of Ghidra used by DoD.
 - Measure how well static analysis and automated repair work on decompiled code.
- This work, if successful, will enable DoD to find and fix potential vulnerabilities in binary code that might otherwise be cost-prohibitive to investigate or repair manually.

State of the Art – Recompile of decompiled code

- Zhibo Liu and Shuai Wang. “How far we have come: testing decompilation correctness of C decompilers.” *ACM Int’l Symposium on Software Testing & Analysis (ISSTA)*, July 2020.
 - Tested **synthetic** code **without input or nondeterminism**.
 - Ghidra: out of 2504 test cases (averaging around 250 LoC), 93% were correctly decompiled.
 - Only **unoptimized** code. No structs, unions, arrays, or pointers.

Pipeline for use on in-the-wild binaries



Code Recompilation

The table shows the percentage of source-code functions that are extracted as recompileable (i.e., syntactically valid) C code.

SPEC 2006
Benchmarks

Project	Source Functions	Recomp Functions	Percent
dos2unix	40	17	43%
jasper	725	377	52%
lbm	21	13	62%
mcf	24	18	75%
libquantum	94	34	36%
bzip2	119	80	67%
sjeng	144	93	65%
milc	235	135	57%
sphinx3	369	183	50%
hmmmer	552	274	50%
gobmk	2,684	853	32%
hexchat	2,281	1,106	48%
git	7,835	3,032	39%
ffmpeg	21,403	10,223	48%
Average			52%

Types of syntactic errors

Count	Error type
609	Request for member in something not a structure or union
706	Invalid operands to binary operator
910	Other
2,972	Use of undeclared identifier
1,224	void value not ignored as it ought to be
1,153	too many arguments to function
3,434	too few arguments to function
11,008	Total

SCAIFE-ACR Integration

Integrated Static Analysis Classification and Automated Code Repair for CI

Problem: DoD organizations that develop code or analyze code security need to make code more secure, with as little costly manual effort as possible. Automated code repair (ACR) tools can fix some code flaws, and automated SA classifiers can save manual work adjudicating static analysis (SA) results, but they may not work well together as-is.

Solution: A system that can modularly incorporate a wide variety of SA classifiers and ACRs that increases the percent of high-severity SA results¹ addressed automatically after applying ACR, designed for CI. “Automatically” means “automatically repaired” or “automatically classified with confidence 70% or greater” (provided that the classifier has a precision and recall 70% or greater).

Approach: Building on what we’ve learned and the tools from the previous 6 years of Line-funded projects (SEI ACR⁴ and SCAIFE²), we will develop a modularly integrated SCAIFE-ACR system for CI, then use it to measure impact of SEI ACR on the percent of high-severity SA results addressed automatically. We test it with open-source code and collaborators use it in their systems on their code. Also, novel use of ACR fix data to improve classifier predictions, and measure effectiveness with classifier precision and recall comparisons.

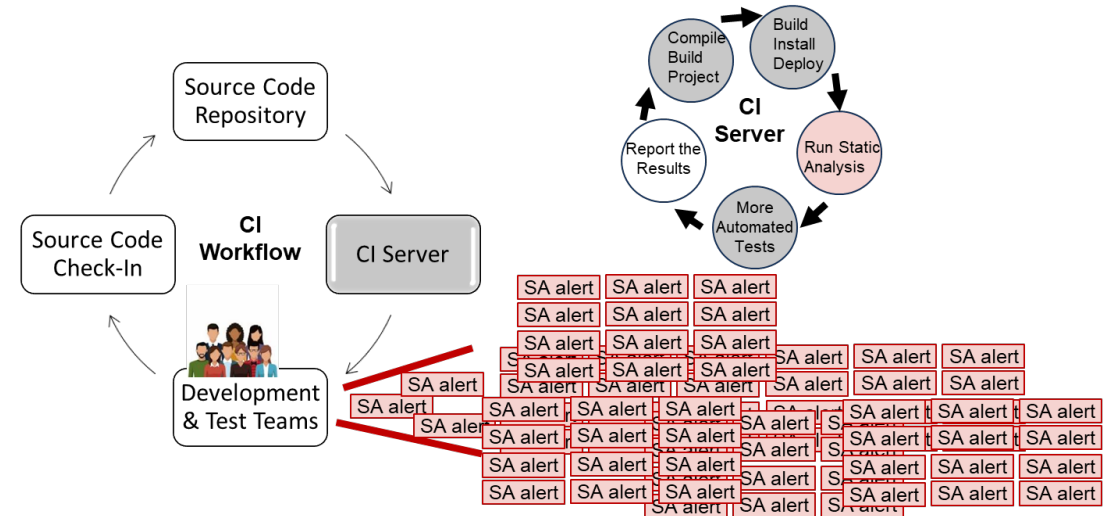
1. **High-severity SA results:** warnings for top 25 dangerous CWE and CERT coding rules with severity level 3
2. **SCAIFE:** modular static analysis classification system with GUI developed by L. Flynn’s SEI projects
3. **ACR:** uses a semantic representation of code to fix code flaws
4. **SEI ACR:** ACR tooling for memory safety in C developed by W. Klieber’s SEI projects

Auto-Labelled ACR Fixes

1. An ACR fix is made to the code (was version 1, now v2 in new branch)

ACR possible?	Priority	Auto-Repair?	Manual Adjudication	Classifier Confidence True (%)	Condition
yes	8890	YES			CWE-190
yes	8889	NO			INT31-C
yes	8888	YES			CWE-191
yes	8887	YES			CWE-79
yes	8886	YES			CWE-787
yes	8885	YES			CWE-125

2. The ACR code fixes (v2) are committed to the remote repository. Then, the CI server builds and tests the code, and sends results back to the development+test team and to SCAIFE.



3. SCAIFE fuses SA results into meta-alerts. If last code push is an auto-repair, SCAIFE checks if a meta-alert for the repaired condition re-appears on matched lines* of repaired code. If yes, it auto-labels the meta-alert False if fix marked 'reliable' by the ACR. New feature "auto-repair" for classifier. No ACR auto-labels True.)

SEI ACR does not prove the code v1 meta-alert was True, but it fixes many memory safety violations

ACR possible?	Priority	Auto-Repair?	Manual Adjudication	Classifier Confidence True (%)	Condition	Auto-Adjudication?	Line	Filepath
yes	8885	YES		88	CWE-125	N/A	10	dir4/FileX

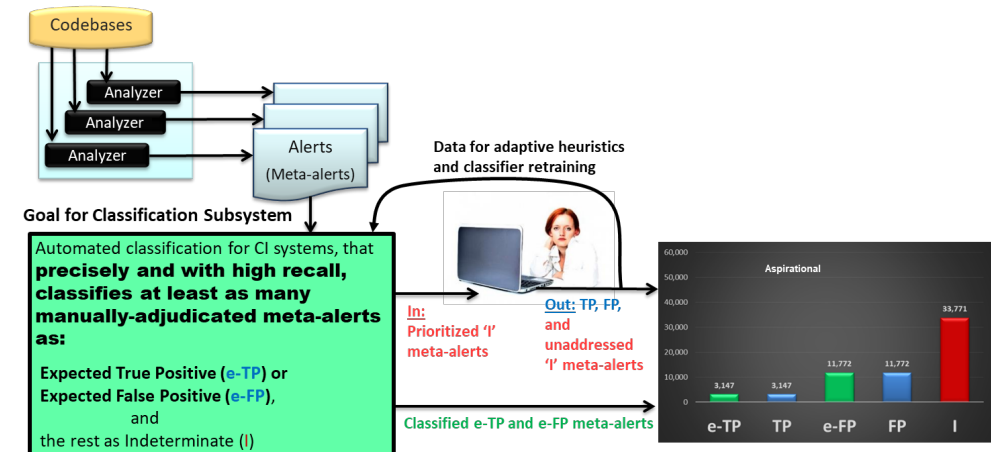
Code v1, ACR-fixed meta-alert

N/A	7000	PREVIOUS		80	CWE-125	FALSE	10	dir4/FileX
-----	------	----------	--	----	---------	-------	----	------------

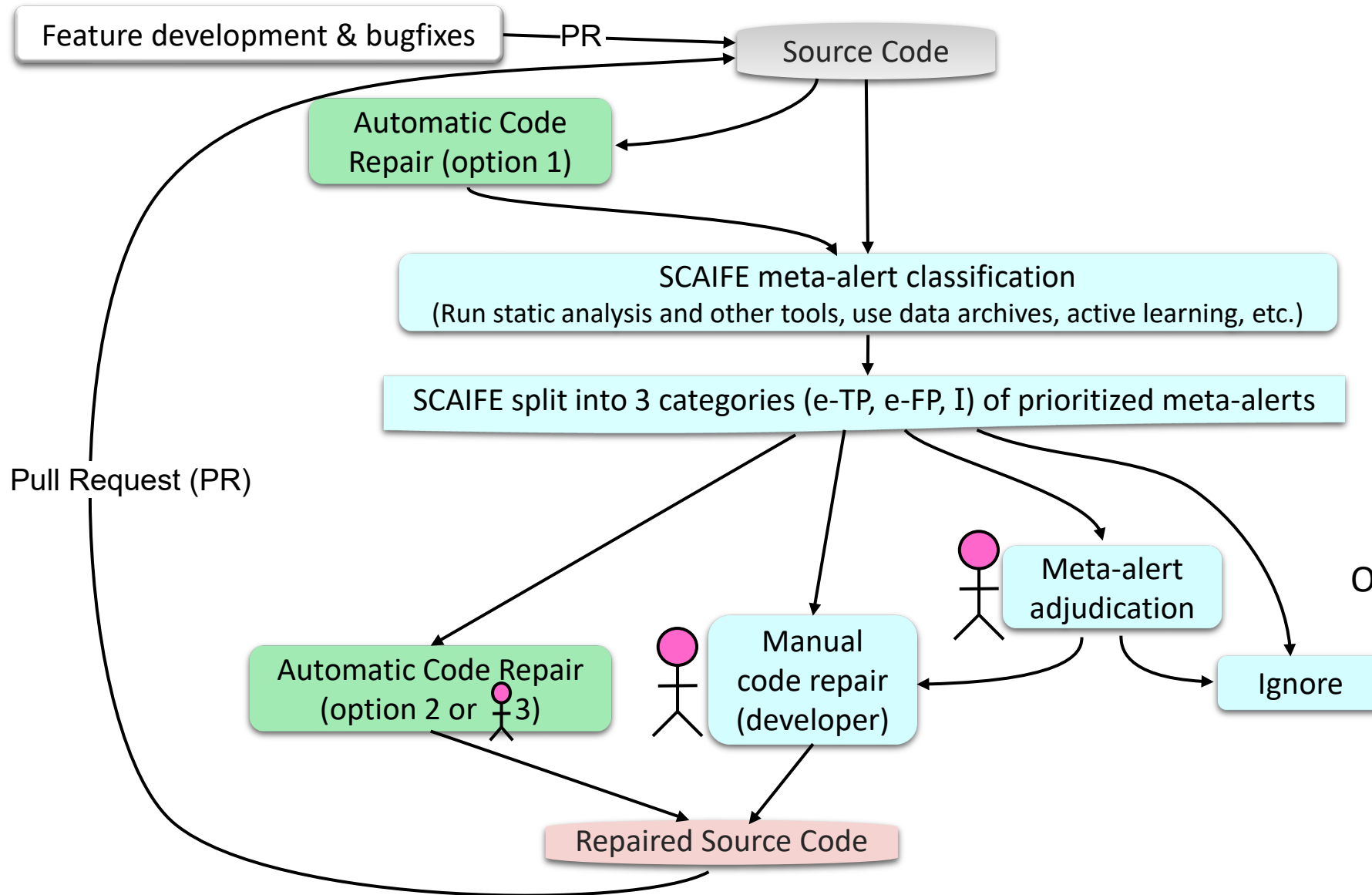
Code v2, same location and condition: meta-alert auto-labeled FALSE

* Lines may be matched using POSIX diff program, possibly enhanced with extra matching information related to the ACR system. E.g., a memory access that previously took 1 line might be expanded by the ACR to take 3 lines in one file plus a new function in another file, and any of those locations would count as a match.

4. The new labeled data (meta-alert auto-labeled false and associated data) is used to improve the SA classifier predictions for all remaining not-yet-adjudicated meta-alerts, using adaptive heuristics and/or occasional classifier retraining.



ACR & SCAIFE Meta-Alert Classification



- ACR option 1: Make all possible automatic repairs (worse runtime overhead, better safety)
- ACR option 2: Only repairs higher-priority meta-alerts (less runtime overhead, but might leave unfixed vuls)
- ACR option 3: Analyst views potential repairs for high-priority meta-alerts and approves or rejects each individually.

One path down could happen:

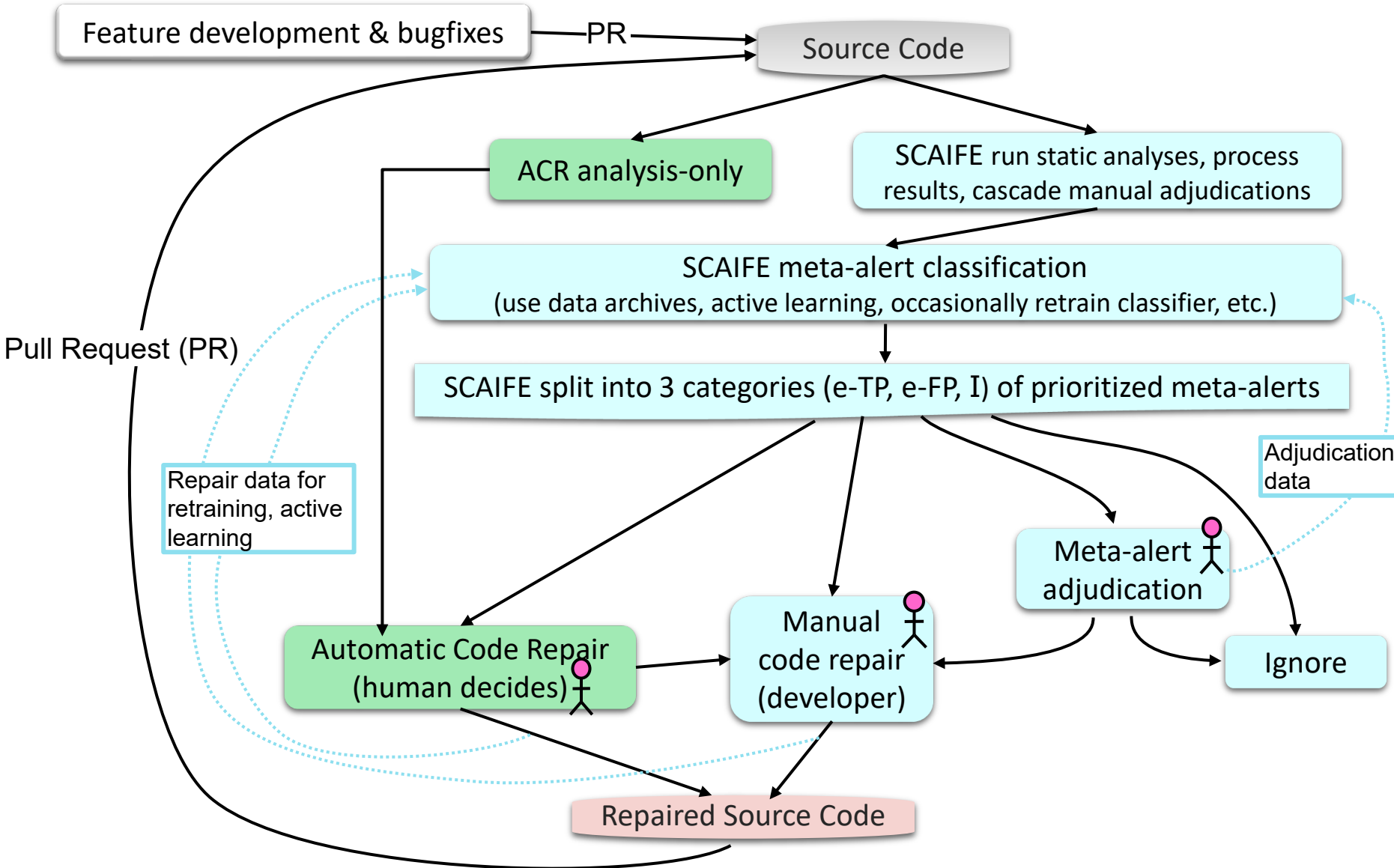
- during one continuous integration build cycle (cycle until build passes, then repeat for new builds)
- during a code security analysis+fix

ACR & SCAIFE Meta-Alert Classification

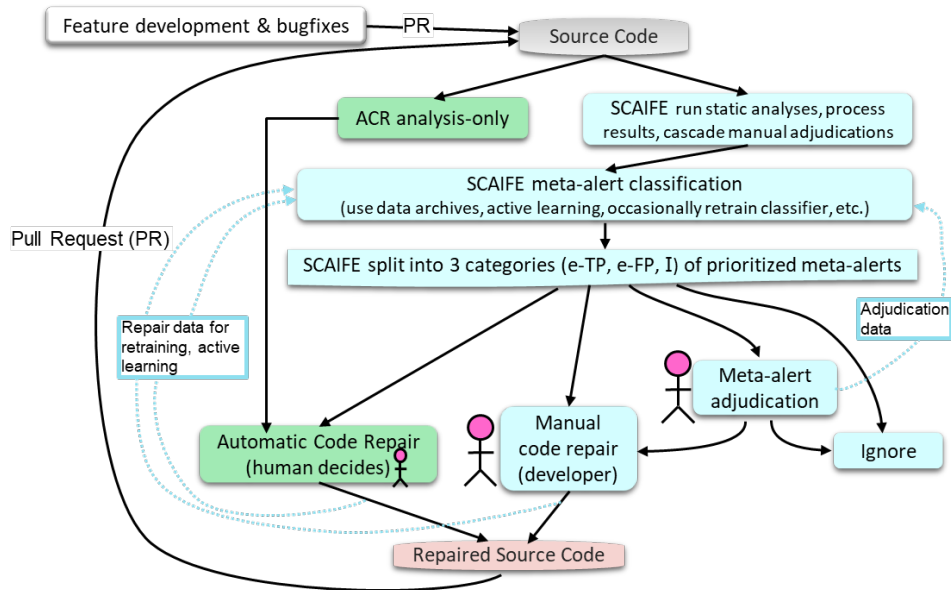
Example: SCAIFE and ACR option 3 (manual decisions)

Parallel manual effort possible:

- ACR decisions
- manual code repair
- meta-alert adjudication



ACR & SCAIFE: Latency Impacts on Design



- ACR option 1: Make all possible automatic repairs (worse runtime overhead, better safety)
- ACR option 2: Only repairs high-priority meta-alerts (less runtime overhead, but might leave unfixed vuls)
- ACR option 3: Analyst views potential ACR repairs for high-priority meta-alerts and approves or rejects each individually.

One vertical flow could happen:

- for a CI build, with SCAIFE / ACR results that the CI system can use
- during a code security analysis + fix
- for one code commit, with manual repairs/analyses working down priority lists

- **For fast ACR and SCAIFE analyses**, the analyses (and potential repairs) could be potentially be done for each CI build.
- **For slower ACR and SCAIFE analyses**, (e.g., for large codebases) the envisioned design would have the analysis be done on a code version (e.g., on a particular commit to a repository). After, ACR (1,2, or 3) and SCAIFE “I” adjudications and “e-TP” manual repair continue on that code version until new analyses are available for a later commit.
- This project will **measure latencies** to determine if ACR and SCAIFE analyses should be done in parallel (as shown) or if some ACR analyses should be done after SCAIFE analyses, to reduce ACR analysis latency.

GUI Mock-up

List of meta-alerts*, ACR-fixable at top. Click to see associated code and reliability. SCAIFE GUI meta-alert list has many more fields: line, filepath, notes, etc.

* Meta-alert: fused SA alerts mapped to same condition from taxonomy of code flaws (e.g., CWE-190)

ACR possible?	Priority	Auto-Repair?	Manual Adjudication	Classifier Confidence True (%)	Condition
yes	8890	ACCEPT			CWE-190
yes	8889	REJECT			INT31-C
yes	8888	ACCEPT			CWE-191
yes	8887	ACCEPT			CWE-79
yes	8886	ACCEPT			CWE-119
yes	8885	PENDING DECISION		88	CWE-787
yes	8884	PENDING DECISION		86	CWE-125
yes	8883	PENDING DECISION		85	CWE-89
yes	8882	PENDING DECISION		84	CWE-200
yes	8881	PENDING DECISION		82	CWE-416

.....

no	1000	N/A		90	CWE-352
no	999	N/A		85	CWE-787
no	998	N/A		84	CWE-22
no	997	N/A		82	CWE-476
no	996	N/A		81	CWE-287
no	995	N/A		80	CWE-434
no	994	N/A		78	CWE-732
no	993	N/A		77	CWE-94
no	992	N/A		73	CWE-522
no	991	N/A		65	EXP34-C

Bold text specifies current code view

Diff View for ACR Fixes

Source Code for Manual Adjudication

For this ACR fix, select files on right to see edits below (red text) that will be made if fix is accepted.

Reliable fix: True

[File X](#)

[File Z](#)

[File Y](#)

[File Q](#)

Original Source Code File X

```
#define BUF_SIZE 256
char nondet_char();

int main() {
    char* p = malloc(BUF_SIZE);
    char c;
    while ((c = nondet_char()) != 0) {
        *p = c;
        p = p + 1;
    }
    return 0;
}
```

Repaired Source Code File X

```
#include "fat_header.h"
#include "fat_stdlib.h"
#define BUF_SIZE 256
char nondet_char();

int main() {
    FatPtr_char p =
        fatmalloc_char(BUF_SIZE);
    char c;
    while ((c = nondet_char()) != 0) {
        *bound_check(p) = c;
        p = fatp_add(p, 1);
    }
    return 0;
}
```

GUI Mock-up

Changed classifier confidences and order: ACR labeled meta-alerts are used by adaptive heuristic and when retraining the classifier.

ACR possible?	Priority	Auto-Repair?	Manual Adjudication	Classifier Confidence True (%)	Condition
yes	8890	ACCEPT			CWE-190
yes	8889	REJECT			INT31-C
yes	8888	ACCEPT			CWE-191
yes	8887	ACCEPT			CWE-79
yes	8886	ACCEPT			CWE-119
yes	8885	ACCEPT			CWE-787
yes	8884	REJECT			CWE-125
yes	8883	ACCEPT			CWE-89
yes	8882	ACCEPT			CWE-200
yes	8881	ACCEPT			CWE-416

.....

no	1000	N/A	True positive		CWE-352
no	999	N/A	True positive		CWE-787
no	998	N/A	False positive		CWE-611
no	997	N/A	True positive		CWE-476
no	996	N/A		82	EXP34-C
no	995	N/A		80	CWE-434
no	994	N/A		78	CWE-732
no	993	N/A		77	CWE-94
no	992	N/A		73	CWE-522
no	991	N/A		65	CWE-22

Bold text specifies current code view

Diff View for ACR Fixes

Source Code for Manual Adjudication

Source Code `dos2unix-7.2.2/common.c`

```

427  fname_len = strlen(dir) + strlen("/d2utmpXXXXXX") + sizeof (char);
428  if (!(fname_str = malloc(fname_len)))
429      goto make_failed;
430  sprintf(fname_str, "%s%s", dir, "/d2utmpXXXXXX");
431  *fname_ret = fname_str;
432
433  free(cpy);
434      while ((c = nondet_char()) != 0) {
435  #ifdef NO_MKSTEMP
436      name = mktemp(fname_str);
437      *fname_ret = name;
438      if ((fd = fopen(fname_str, W_CNTRL)) == NULL)
439          goto make_failed;

```

SCAIFE and SEI ACR: Current and Future Work (1/2)

SEI-ACR:

- Currently works on some small/medium (10 kLOC) real-world codebases.
- Still needs additional development to consistently handle real-world codebases.
 - We estimate about 8 person-weeks of effort.

SEI SCAIFE:

- Modular system, containerized with Docker.
- Combines the results from multiple SA tools.
- Manual adjudication of meta-alerts via a GUI.
- Can create labeled data for classifiers from test suites.
- GUI-based specification, training, and running of static analysis classifiers.
- Formally defined APIs for each module, using OpenAPI v3.
 - Parts can be swapped out (e.g., classifier, active learning, user interface).
- It has some continuous integration (CI) functionality.
 - It needs further development to support CI builds: Stats Module re-classification should start automatically, plus add a 'CI Orchestrator' to ensure system consistency.

SCAIFE and SEI ACR: Current and Future Work (2/2)

Integrating ACR with SCAIFE:

- GUI for previewing and accepting/rejecting auto-repairs.
- Define the API for communications between ACR and SCAIFE.
- Develop API-related code.
- Containerize SEI ACR.
- Add code hooks to record metrics to analyze for the project.


Transitioning into production use:

- Enable use in a DevSecOps pipeline.
- Expected to require:
 - security hardening,
 - performance improvements,
 - possibly a cloud-ready design.

Select Performance Metrics

Among many other metrics we plan to gather:

- Validation of our tool on codebases representative of what DoD encounters in software assurance. Metrics:
- Compare pre- and post-repair
 1. classifier precision & recall;
 2. adjudication cascading (compare counts and manually verify random selections)
- Counts of auto-repaired code (lines, SEI ACR suggestions accepted)
- Performance per goal: percent automatically-handled meta-alerts, precision, recall



SCAIFE and ACR:
Tools & Research

Invitation to Collaborate

Interest in SCAIFE-ACR Integration?

To do the integration work described, we would need more funding.

Would you or your org be interested? If so, please contact us!

Invitation to Test

We invite you to test SCAIFE and ACR tools:

- Full SCAIFE system release limited to DoD and DoD contractors (Distribution D)
- Testing does *not* have to include data sharing
- SCAIFE classification performance release needs testers ASAP.
- If interested please contact us: lflynn@cert.org and weklieber@cert.org

Deployment and testing support by SCAIFE:

- release system Docker-containerized, with configuration files (ports, URLs, names) to ease integration in wide variety of systems
- comes with documentation, much-extended in last year per collaborator feedback
- hands-on demos and tutorials, for quick start

Deployment and testing support by ACR:

- Coming soon: release system to be Docker-containerized
- Coming soon: hands-on demos and tutorials, for quick start

Thanks + Contact Info

Thank you for listening!

Questions?

Feedback and potential funding/collaborations are welcome.

Dr. Lori Flynn

lflynn@sei.cmu.edu

Dr. William Klieber

weklieber@sei.cmu.edu

Carnegie Mellon University
Software Engineering Institute
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Backup slides

State of Practice/Art: ACR

Many ACR tools exist for many languages [1]; it is an active area of research and development.

- **SEI ACR** [2] advanced state of the art of ACR for C memory vulnerabilities, with a technique for automatically repairing all potential violations of spatial memory safety in C source code, modulo specified limitations. The recently-discovered sudo bug CVE-2021-3156 is an example of a memory vulnerability that SEI ACR fixes. **SEI ACR developers are part of this project, enabling cost-efficient development of the integrated system. Also, memory safety is important.**
- **Repairnator** [3] is an open-source CI-integrated modular ACR system for incorporating a wide variety of ACR tools. The pipeline takes a failing CI build ID and tries to replicate the bug and then repair it with different ACR tools. It can be configured to automatically create pull requests (PRs) for ACR fixes that pass build tests. **Repairnator has CI integration and GUI view of repair to accept/reject. It isn't integrated with an SA classification system for manual adjudication, doesn't run repairs unless a build fails, nor make particular ACR and classifier functionality work well together, nor improve SA classifiers with ACR fix info.**
- **DeepFix** [4] is an ACR that fixes C language errors by deep learning (AI) trained to predict incorrect program locations along with the fixed code. **Like many ACR tools, it uses AI prediction of code flaws like SCAIFE, but only for a certain set of code flaws the tool itself identifies, not using general-purpose SA tool results. It isn't CI-integrated.**
- **AVATAR** [6] uses fix patterns of SA true positives to generate patch candidates to fix semantic bugs. **Similar to many ACRs, it uses SA but the manual process of fixing the remaining SA meta-alerts is not improved nor integrated with a classifier.**

[1] program-repair.org, <http://program-repair.org/>

[2] W. Klieber. "Automated Code Repair to Ensure Memory Safety", SEI Research Review, Nov. 2020.

https://resources.sei.cmu.edu/asset_files/Presentation/2020_017_001_648949.pdf

[3] Repairnator: an open-source platform for automatic program repair, <https://github.com/eclipse/repairnator>

[4] R. Gupta, S. Pal, A. Kanade, and S. Shevade. "DeepFix: Fixing common C language errors by deep learning", AAI 2017. <https://bitbucket.org/iiscseal/deepfix/>

[6] K. Liu, A. Koyuncu, D. Kim, and T. Bissyande. "AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations", IEEE Conference on Software Analysis, Evolution, and Reengineering, 2019. <https://github.com/SerVal-DTF/AVATAR>

State of Practice/Art: SA Classification and Multi-Tool Aggregation

There are many multi-SA-tool result aggregators (for code flaw coverage) and many SA classifiers (most single-tool). **We haven't found one integrated with ACR that enables manual adjudication of non-ACR meta-alerts. Of the candidates for this project, extensibility and modifiability are required features.**

- **SEI SCAIFE** previously advanced state of the art for SA classification for multiple tools. A modular extensible system where parts can be swapped out (e.g., classifier, user interface) using publicly-published formal APIs that can be used for auto-code generation of client calls and server stubs in many languages, and much of the code publicly available. **Not as fast, security-hardened, or with a slick GUI interface like commercial tools but has modularity and access needed for project. Experienced SCAIFE developers will work on proposed project, making the integrated system development cost-effective.**
- **CodeDX** [1] is fast, security-hardened, imports results from multiple SA tools, and has a slick GUI interface. But it's not useable for this project because we cannot modify it to incorporate ACR. **Proprietary: might classify, but not extensible/modifiable for this research.**
- **DHS SWAMP** [2] uses results from multiple SA tools, is security-hardened, CI-integrated, scalable for cloud systems. SWAMP-in-a-Box is open-source back end and has a proprietary front-end. **SWAMP doesn't record adjudications and doesn't incorporate classification unless CodeDx proprietary front-end does it.**
- **Software Assurance Tool (SwAT)** developed by Army CCDC C5ISR imports output of multiple tools, is scalable with integrations to cloud and is currently in the process of integrating use of classifiers by integrating with the SEI SCAIFE system. **SwAT is DoD-proprietary. CCDC C5ISR is in-process integrating SwAT with our SCAIFE modular SA classification system.**

[1] CodeDx <https://codedx.com/>

[2] SWAMP: Software Assurance Marketplace <https://continuousassurance.org/>

[3] Software Assurance Tool (SwAT) per correspondence with CCDC C5ISR collaborator.

[4] [Release of SCAIFE System Version 1.0.0](#) Provides Full GUI-Based Static-Analysis Adjudication System with Meta-Alert Classification

State of Practice: SA Classification

Some highlights about SA classification current state below.

Note: This proposal mostly involves modularly using third-party or pre-existing SA classifiers. The only proposed classifier research is to auto-label meta-alerts matching ACR-fixes, then use that data to improve classifier predictions. We will measure precision and recall, our team testing open-source codebases and DoD collaborators testing on their codebases. **We will use third-party commonly-used classifiers and adaptive heuristics.**

Some static analysis classifier results:

- 88-91% classifier precision using multiple classifier types on CERT-adjudicated meta-alerts and multiple static analysis tools as features [Flynn] ← Our work that led to SCAIFE, code from that is in SCAIFE.
- 85% accuracy in study at Google with FindBugs, Logistic Regression, adaptive prioritization of meta-alerts using code-fix decisions [Ruthruff]
- 62% precision for top 50 alerts, using locality, flaw type, code version number [Williams]
- 81% of true positive meta-alerts identified after investigating only 20% of the alerts by adaptively ranking meta-alerts using developer feedback, suppressing false positives and fixing true positives, and using more features such as alert type accuracy and code locality. [Heckman]
- A factor of 2-8 improvement over randomized meta-alert ranking, by using formula (Z-ranking) that adapts with SA adjudications. New labeled data can be dynamically used to re-order meta-alerts, as the analyst completes each meta-alert adjudication from the top of the ordered set of meta-alerts. [Kremenek]

[Flynn] "Prioritizing alerts from multiple static analysis tools, using classification models" *IEEE Workshop on Software Qualities and their Dependencies*, 2018.

[Ruthruff] 2008. Predicting accurate and actionable static analysis warnings: an experimental approach. *International Conference on Software Engineering*. ACM, 341–350.

[Williams] A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 2011.

[Heckman] Adaptively ranking alerts generated from automated static analysis. *Crossroads* 14, 1 (2007), 7.

[Kremenek] 2004. Correlation exploitation in error ranking. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 83–93.