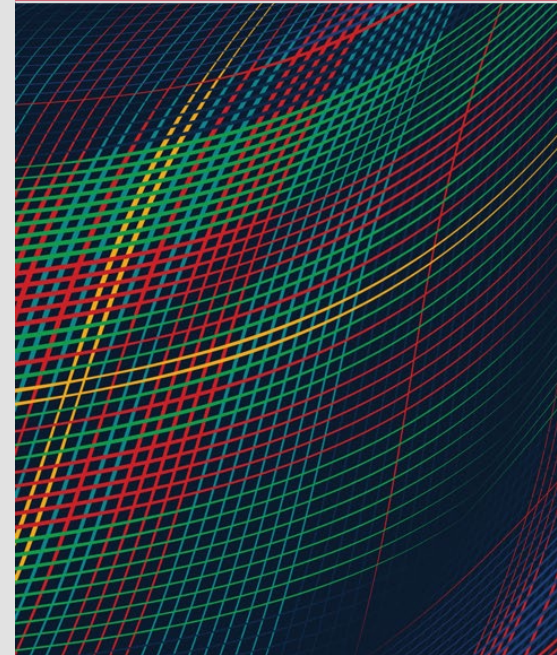


# Automated Repair of Static Analysis Alerts

David Svoboda  
Software Security Engineer



# Document Markings

Copyright 2023 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](mailto:permission@sei.cmu.edu).

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

DM23-0915

# Agenda

- **The Problem**
- **The Solution**
- **Testing**
- **Lessons Learned**
- **Conclusion**

Automated Repair of Static Analysis Alerts

# The Problem

# Does the DoD Require Use of Static-Analysis Tools?

- From the [Application Security & Development \(ASD\) Security Technical Implementation Guide \(STIG\)](#):
  - According to [V-222624, \*The ISSO must ensure active vulnerability testing is performed\*](#), Use of automated scanning tools accompanied with manual testing/validation which confirms or expands on the automated test results is an accepted best practice when performing application security testing.
- The [NIST Computer Security Resource Center \(CSRC\)](#) documents recommendations for
  - [RA-5: Vulnerability Monitoring and Scanning](#)
  - [SA-11: Developer Testing and Evaluation](#)

[Parasoft](#), [Coverity](#), and [Perforce](#) all suggest that their SA tools help you achieve compliance with the Defense Information Systems Agency's (DISA's) ASD STIG.

# Project Summary

**Problem:** There are too many static-analysis (SA) tool alerts to audit them all. Even if we knew which alerts were true positives, there are still too many true positives to repair all of them manually.

**Solution:** Automatically repair 80% or more of each type of SA alert in a way that both preserves soundness and makes the alert disappear when the code is reanalyzed.

**Approach:** Choose 3 (later 10) categories of alerts to repair, build a tool to repair alerts, and verify it can fix >80% of alerts in each category.

\* The number of alerts repaired depends on the particular codebase, SA tool(s), and developers. For example, some developers avoid particular code flaws, or they repeatedly program particular code flaws. Some codebases may be likely to have (or lack) certain types of code flaws due to the software architecture and whether the organization invests in handling its technical debt. Other Software Engineering Institute (SEI) projects address these issues.

# Static Analysis Tool Alerts: Le Déluge



## Experience from analyzing multiple large-sized project with 2-3 million LoC:

Static analysis produces 50,000 alerts

- i.e. 16-25 alerts / kLoC

Which would take 3 analysts 33 weeks to manually review.

- i.e. 100 person-weeks = 2 person-years
- This implies 216 seconds / alert

Source: Aesia Cohen from USArmy C5ISR  
as part of the Software Assurance Toolset (SwAT) Overview Brief  
December 2023

# Collaborator Experience

Of the languages that our collaborators use, C code tends to exhibit the most vulnerabilities.

Their process is

- Filter alerts based on a preset list of CWEs and (if time permits) analyze the *most critical* remaining alerts.
  - About 20% of (unfiltered) alerts are deemed to be true positives.
- Fix ~90% of the true positives.

In a Google study, 32% of alerts are addressed in their code.

Some collaborators believe that SA tools in a CI pipeline are more worthwhile than the SA tool's own GUI

- Others don't use CI pipelines (yet)



# Code Repair State of the Art

## Refactoring Code

- **Eclipse** has Java code refactoring built-in since 2001, but not C/C++ code.
  - Research extensions to the Eclipse C/C++ Development Tools (CDT) plugin support C/C++ code repair for [integer overflows and buffer overflows](#), according to a 2015 peer-reviewed [paper](#).
- **Visual Studio Code** for C/C++ code provides some automated repairs plus simple code refactoring.
  - As far as we've been able to tell, it does not include the automated repairs we've developed.
- **clang-tidy** now has recent refactoring and application programming interface (API) rewriting for C/C++.
- **clangd** provides easy integration of Clang to editors and integrated development environments (IDEs).

## Automated Code Repair (ACR) Project

(Principal Investigator: Dr. Will Klieber)

- Automatically rewrites C/C++ code to address specific problems
- Imposes major infrastructure change to entire C code (to solve buffer overflows and convert most pointers to “fat pointers”)

These results, combined with the prominence of C code in the DoD and the evolution of **clang-tidy's** rewriting API, suggest that now is the time to pursue automated repair of SA tool alerts.

Automated Repair of Static Analysis Alerts

# The Solution

# Technical Approach

Make cheap, local fixes.

Only fix code associated with an SA alert.

Ensure that fixes are sound and do not change the behavior of good code.

- A repair does not break the code, even if the alert was a false positive.
- The tool should be *idempotent* (i.e., the tool will not modify code it already repaired).

```
char *f(int a, int b) {  
    int x;  
    int sum = a + b;  
    /* ... */  
}
```

If the mathematical value of **a+b** cannot be stored in an **int**, the behavior is undefined.



```
char *f(int a, int b) {  
    int x;  
    int sum;  
    if ((b > 0) && (a > (INT_MAX - b))) ||  
        ((b < 0) && (a < (INT_MIN - b)))) {  
        /* Handle error */  
    }  
    sum = a + b;  
    /* ... */  
}
```

# Technical Approach

Make cheap, local fixes.

Only fix code associated with an SA alert.

Ensure that fixes are sound and do not change the behavior of good code.

- A repair does not break the code, even if the alert was a false positive.
- The tool should be *idempotent* (i.e., the tool will not modify code it already repaired).

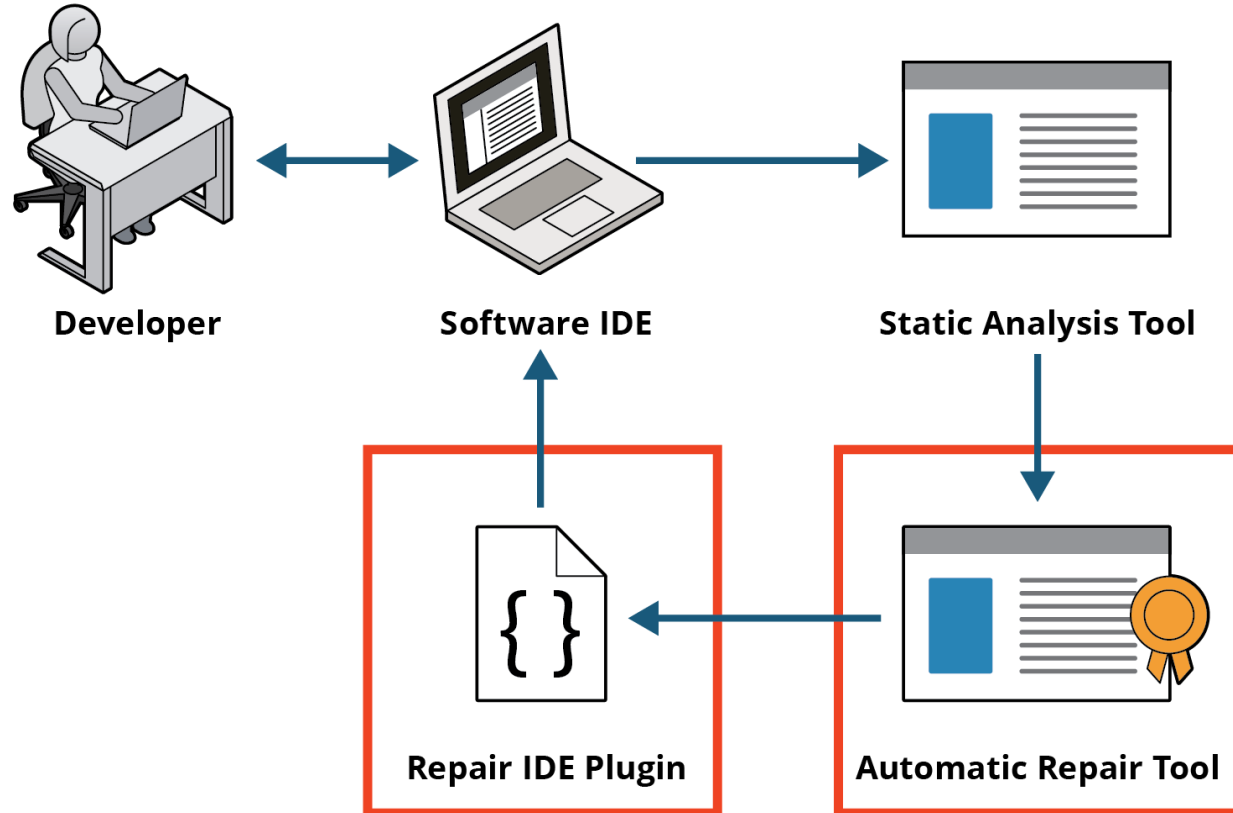
```
char *f(int a, int b) {  
    int x;  
    int sum = a + b;  
    /* ... */  
}
```

Possible integer  
overflow?

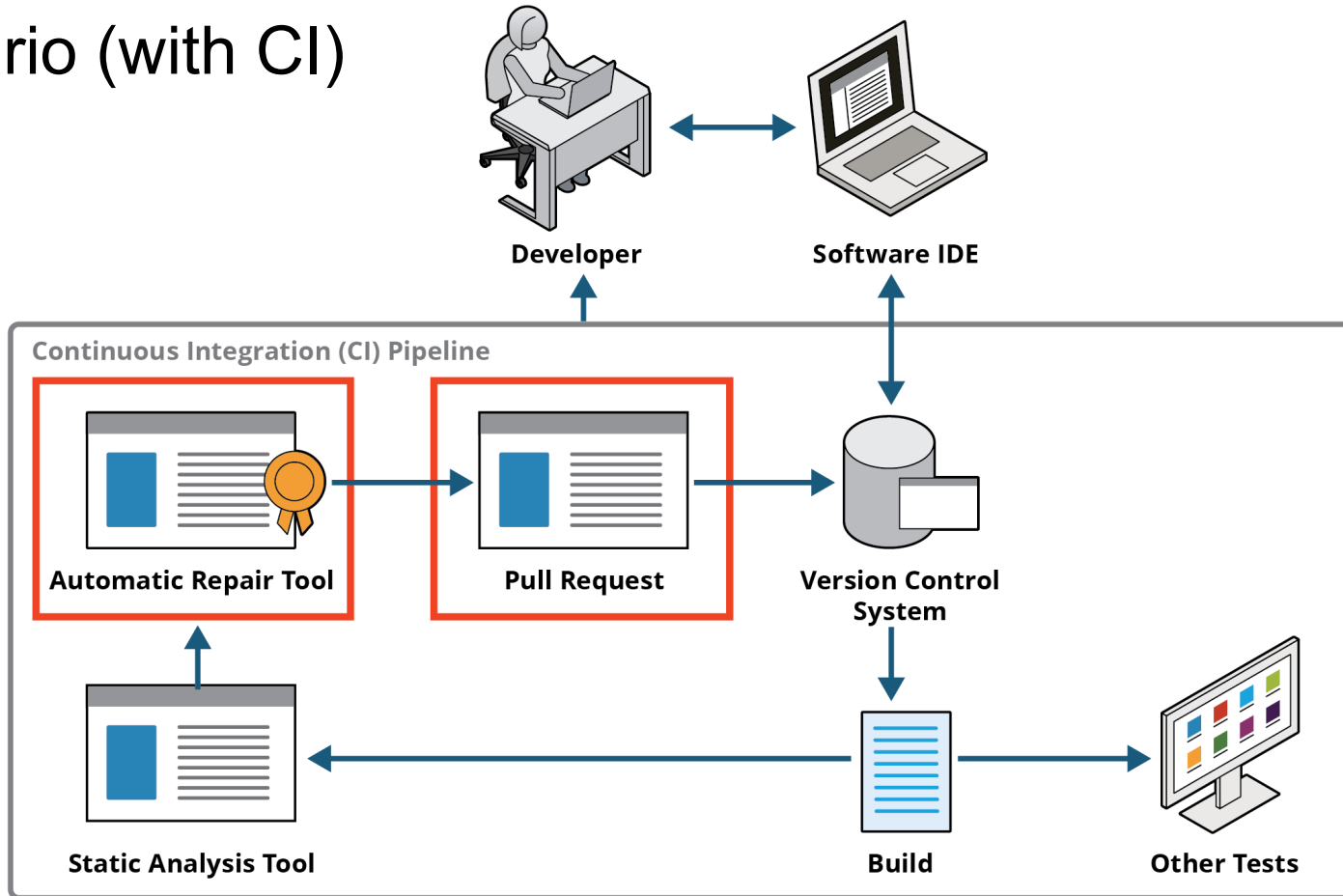


```
char *f(int a, int b) {  
    int x;  
    int sum = SAFE_ADD(a, b,  
        /* Handle error */  
    );  
    /* ... */  
}
```

# Usage Dataflow Scenario (Without CI)



# Usage Dataflow Scenario (with CI)



Automated Repair of Static Analysis Alerts

# Testing

# Verification Theory: Undefined Behavior



Typically, code that violates a CERT rule causes undefined behavior (UB).

- EXP33-C / CWE-457: Reading an uninitialized variable
- EXP34-C / CWE-476: Dereferencing a null pointer

Read garbage value

Crash

Platforms may define platform-specific behaviors.

ISO C only constrains programs without UB.

- UB means the platform may do anything.



Compilers may assume UB cannot happen.

- This makes subsequent behavior unpredictable.



# Verification



Our repair algorithms do the following:

- Replace code with UB with error-handling code (e.g., termination).
- Possibly run additional operations or checks on code with no UB.
  - These operations or checks must **NOT** change the behavior.

**Limitation:** Cannot reliably repair code that depends on

- Undefined behavior (UB)
- Performance or timing issues

# Components for Testing

SA alerts were produced by running SA tools over the following OSS codebases:

- [git](#) (v2.39.0, C)  
Has internal test systems with good test coverage.
  - All tests pass.
- [zeek](#) (v5.1.1, C++)  
Has internal test systems with good test coverage.
  - Many tests currently fail.

So far, we address only these three CERT guidelines:

- [EXP34-C](#) Dereferencing a null pointer
- [EXP33-C](#) Reading an uninitialized variable
- [MSC12-C](#) Code that is never executed

To test the repair tool, we produced >15,000 SA alerts using the following SA tools:

- [cppcheck](#) (v2.9)
- [clang-tidy](#) (v15.0.7)
- [CERT Rosecheckers](#)

We use an internal CI system to catch regressions.

# Tests & Experiments

## Regression Testing

*Verifies that each improvement to the tool does not cause bugs or failures to previously-working code.*

## “Stumble-Through” Tests

*Verifies that the repair tool does not crash or hang*

- Test the repair tool on all alerts in all codebases.
- The test fails if the tool crashes, hangs, or throws exceptions.

For this test, it does not matter whether the tool correctly repairs any alerts.

## Sample Alert Experiments

*Ensures repairs are correct*

**BUT** with >15,000 alerts to repair, we cannot test all of them!

For each tool/guideline/codebase,

- Pick N random alerts; N=5 for now. For each alert,
  - Manually check if ACR did the right thing:
    - Repaired correctly or correctly refused to repair.
  - Until ACR does the Right Thing on  $\geq 80\%$  of alerts, Fix ACR bugs and re-run experiment.

## Integration Experiments

*Verifies that repairs did not change the behavior of code*

- Run the repair tool on all codebases.
- Compile the codebases, run their internal testing mechanisms.

The experiment is successful if all codebase-specific internal tests pass.

## Performance Experiments

*Confirms that repairs do not significantly impede performance*

- Compile original codebases; run their internal testing mechanism.
  - Measure the time and usage of the testing mechanisms.
- Run the repair tool on all codebases.
- Compile the codebases; run their internal testing mechanisms.
  - Measure the time and usage of the testing mechanisms.

*Time should be <5% slower. Memory usage should be equivalent.*

## Recurrence Experiments

*Verifies that repaired alerts are not reported or re-repaired*

- Run the repair tool on all codebases.
- Re-run SA tools on all codebases, and compare alerts generated with original alerts.
- The experiment is successful if repaired alerts are no longer reported by an SA tool.
- Re-run the ACR tool on the repaired codebase's new alerts.
- Ideally, the ACR tool should do nothing since what remains are only the alerts it could not repair.
- If a repaired alert recurs, the ACR tool should report it as a false positive.

Automated Repair of Static Analysis Alerts

# Lessons Learned

# Challenges with Collaborator Data

- Collaborator used Fortify, Checkmarx, and Cppcheck, which are all managed by CodeDX.
- Collaborators' data associates alerts with CWEs, not with CERT rules.
- #1 cppcheck CWE: CWE-398: Code Quality
  - CWE-398 is a category that encompasses 11 more specific CWEs, including
    - CWE 457: Use of Uninitialized Variable (aka CERT EXP33-C)
    - CWE 476: NULL Pointer Dereference (aka CERT EXP34-C)
- We can repair a line of code associated with EXP34-C or CWE-476, but **not** with CWE-398, which is too vague.
- **Solution:** Obtain the tool's checker associated with each alert, and map it to a CERT rule (i.e., ignore CWE).
- **Solution:** Map to a CERT rule from the CWE number and message, which contains technical details.

# Collaborator feedback: Mind Your **PMSGBOXPARAMSA**

How should you initialize a variable of type **PMSGBOXPARAMSA** ?

- a) **PMSGBOXPARAMSA** `pmbp = 0 ;`
- b) **PMSGBOXPARAMSA** `pmbp = 0.0 ;`
- c) **PMSGBOXPARAMSA** `pmbp = NULL ;`
- d) **PMSGBOXPARAMSA** `pmbp = { } ;`
- e) **PMSGBOXPARAMSA** `pmbp = [ ] ;`



Correct



What Redemption  
currently does

Providing the wrong repair **breaks** code.

Getting the answer right requires a type database.

# Clang Integration



Clang can serialize its abstract syntax tree (AST) as JavaScript Object Notation (JSON).

- **We can deserialize this in Python; no C++ is required.**

However, Clang's AST serialization is not complete.

The ASTs of complex types in typedefs are completely serialized:

```
typedef void (*sighandler_t) (int) ;
```

But ASTs of complex types in variable declarations are not:

```
void *handle_signal(int) ;
```

Clang merely provides a hard-to-parse string in the JSON.

We intend to modify Clang to provide digestible type info in the JSON.

**The Clang community has some [interest](#) in accepting our fixes.  
The SEI Legal team has given us approval to send Clang a patch with our fix.**

# Resulting Repair Rate: Two out of Three ain't bad

So far, we address only these three CERT guidelines:

- EXP34-C  
Dereferencing a null pointer
- EXP33-C Reading an uninitialized variable
- MSC12-C Code that is never executed

## *Problems with MSC12-C:*

In **zeek/cppcheck**:

103 / 131 alerts are about unused **goto** labels.

- Should not repair

All Rosecheckers  
alerts were false  
positives

In **git/cppcheck**:

16 / 25 alerts are about unused same (recurring) code:

```
argc = parse_options(argc, argv, ...);
```

**argc** never used



# Minimum Success Criteria

## 1. Usable (from the command line)

- *Users will try the command-line tool before adopting it into CI pipeline*

## 2. Trustworthy

- The repairs it makes are correct
- That is, the tool should be transparent about what it does, by:
  - Documenting how it manages each type of repair
  - Provide a preview of which repairs can be made to a codebase

## 3. Nontrivial

- Should repair a nonempty subset of alerts
- 80% not necessary...perhaps 10%.

## Stretch Goals:

- The tool must run without net access (e.g. in a SCIF)
- Should be able to repair legacy code (that won't build in the tool's own container)

# Demos!

## Repairs of:

- Single C file
- Multi-file OSS program: **dos2unix**
- OSS program w/dependencies: **wrk**
  - Cannot build on Redemption platform

## In environment:

- Command Line
- Gitlab CI (private on Azure)
- Visual Studio Code

## Demos documented in:

- README files in Redemption distro
- Videos

```
1+ #include "acr.h"
2+
1 3 /* zmalloc - total amount of allocated memory aware version of malloc()
2 4 *
3 5 * Copyright (c) 2009-2010, Salvatore Sanfilippo <antirez at gmail dot com>
98 hidden lines | zmalloc(size_t)
102 104 update_zmalloc_stat_alloc(zmalloc_size(ptr));
103 105 return ptr;
104 106 #else
105 - *((size_t*)ptr) = size;
107+ *null_check(((size_t*)ptr)) = size;
106 108 update_zmalloc_stat_alloc(size+PREFIX_SIZE);
107 109 return (char*)ptr+PREFIX_SIZE;
108 110 #endif
25 hidden lines | zcalloc(size_t)
134 136 update_zmalloc_stat_alloc(zmalloc_size(ptr));
135 137 return ptr;
136 138 #else
137 - *((size_t*)ptr) = size;
139+ *null_check(((size_t*)ptr)) = size;
138 140 update_zmalloc_stat_alloc(size+PREFIX_SIZE);
139 141 return (char*)ptr+PREFIX_SIZE;
140 142 #endif
141 143 }
142 144
143 145 void *zrealloc(void *ptr, size_t size) {
144 146 #ifndef HAVE_MALLOC_SIZE
145 - void *realptr;
147+ void *realptr = NULL;
146 148 #endif
147 - size_t oldsize;
148 - void *newptr;
149+ size_t oldsize = 0;
150+ void *newptr = NULL;
149 151
150 152 if (ptr == NULL) return zmalloc(size);
151 153 #ifdef HAVE_MALLOC_SIZE
```

# Future Work

- Prototype 'Redemption' Tool currently available at: <https://github.com/cmu-sei/redemption>
- Have the tool repair 10 CERT rules up from 3
  - 80% of 10 CERT rules should repair 50% of all SA alerts.
- Have the tool repair 17 CERT rules
  - 80% of 17 CERT rules should repair 60% of all SA alerts
- Add support for more SA tools & IDEs
- Repair more "legacy" code (that won't build in the tool's own container)
  - Repair Windows code
- Repair more types of defects (e.g., indentation defects, coding-style deviations).
- Use the repair tool to detect and resolve inconsistencies (e.g., in error handling).
- Other techniques for addressing SA alerts. For example, see:
  - "Using LLMs to Adjudicate Static-Analysis Results" by Klieber & Flynn (SEI)  
(scheduled for tomorrow)

# The Automated Repair Team



**David Svoboda**  
Software Security Engineer  
Principal Investigator



**Will Klieber**  
Software Security  
Engineer



**Lori Flynn**  
Senior Software  
Security Researcher



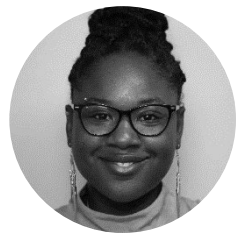
**Joseph Sible**  
Associate Software  
Engineer



**Michael Duggan**  
Reverse Engineer



**Nicholas H. Reimer**  
Engineer



**Ebonie McNeil**  
Technical Engagement  
Lead



**Robert Schiela**  
CSF Deputy Director



**Timothy Chick**  
Technical Manager

Email: [info@sei.cmu.edu](mailto:info@sei.cmu.edu)

# Backup Slides

# Future Work: Test Framework

Extend our test framework to be able to test other repair tools, such as:

- [IntRepair](#) does automated C/C++ integer overflow detection and repair (along with several repairs such as buffer overflows and sensitive dataflows) as an open-source plugin to Eclipse.
- P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert, “Intrepair: Informed repairing of integer overflows,” IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. 2225–2241, 2019. Tool at <https://github.com/TeamVault/IntRepair>, accessed May 30, 2023.
- [ErrDoc](#) provides automated repair to address defects related to error handling.
- Y. Tian and B. Ray, “Automatically Diagnosing and Repairing Error Handling Bugs in C,” ESEC/FSE 2017: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, August 2017, Pages 752–762, <https://doi.org/10.1145/3106237.3106300>

# Future Work: Meta-Repair Tool

Use test framework to evaluate repairs produced by various repair tools, including ours.

- Test framework should dictate for each CERT rule/CWE (+ SA tool):
  - How well does each repair tool perform on un-repaired code?

Develop a **meta-repair tool**, which encapsulates various repair tools, and recommends the best tool(s) to repair a codebase, given static analysis alerts associated with the codebase.

# Verification Theory: Undefined Behavior



Typically, code that violates a CERT rule causes undefined behavior (UB).

- EXP33-C: Reading an uninitialized variable
- EXP34-C: Dereferencing a null pointer



# Verification Theory: Undefined Behavior



Typically, code that violates a CERT rule causes undefined behavior (UB).

- EXP33-C: Reading an uninitialized variable      **Read garbage value**
- EXP34-C: Dereferencing a null pointer      **Crash**

Platforms may define platform-specific behaviors.

# Command Line Tool (M2) –1



## Inputs

Single C/C++ source file in codebase

## Build Command

It includes -D/-U macro definitions and other switches to let Clang parse a source code file.

```
cc -DDEBUG -I/usr/local/include -D2 -Wall  
-c pgn.c -o pgn.o
```

## Distinct SA Tool Alerts

Each alert contains the following:

- CERT rule
- Location where rule is being violated (e.g., line number, column number, end-line number, end column number)
- Message



## Outputs

For each SA alert from input

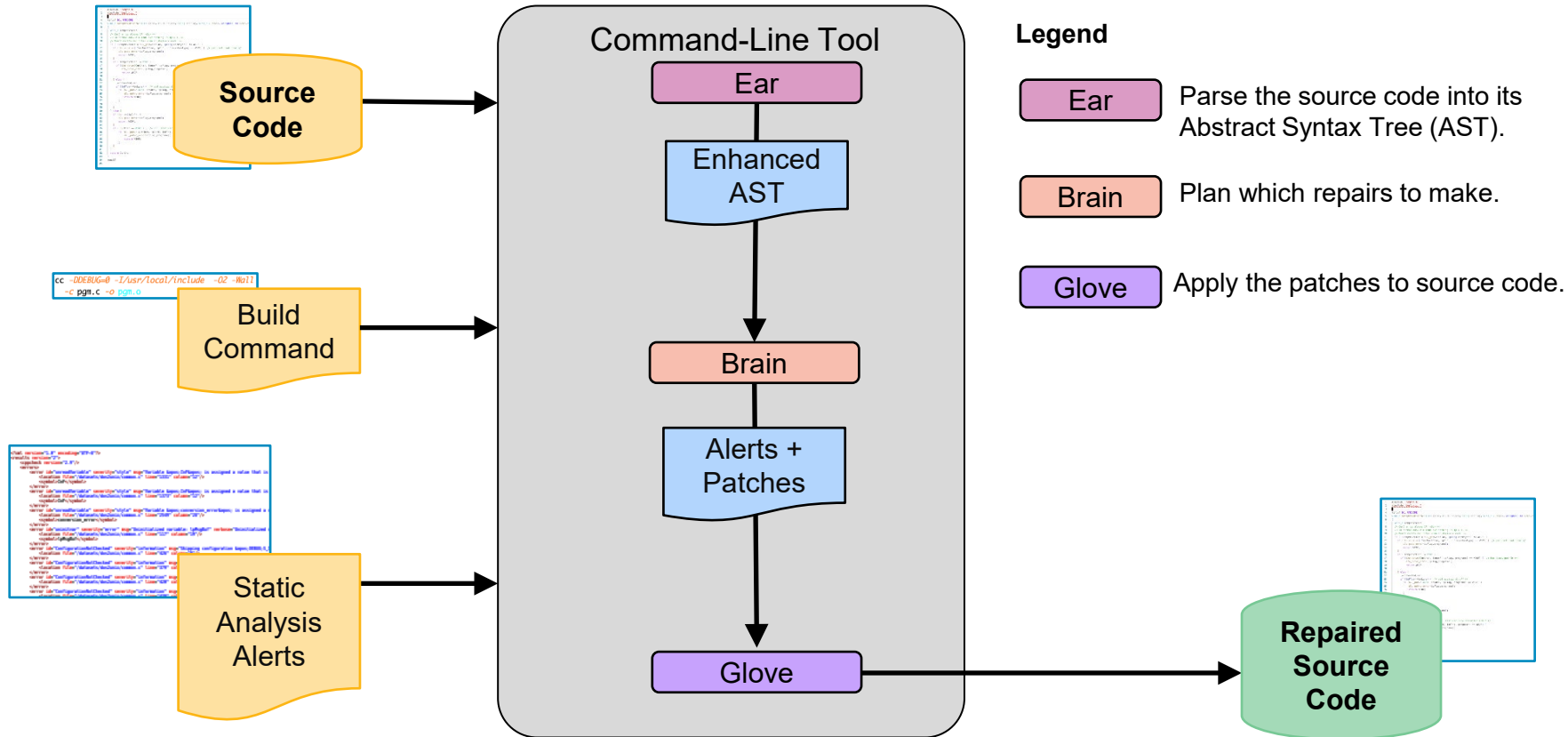
- Patch to repair the alert.
- OR
- Explain in a text message why it cannot be repaired.

All patches should be independent (i.e., they repair distinct regions of code).

There should also be a module that takes the output and applies all patches to the code.



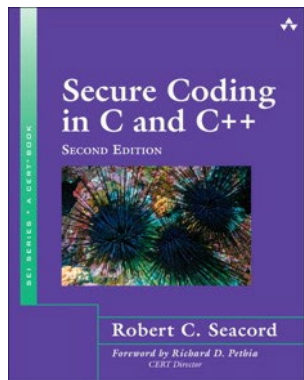
# Command Line Tool (M2) –2



# Questions for New Collaborators

- Do you develop software to be used by DoD? If not, can you propose patches to your development team?
- Do you have any suggestions for which CERT rules or CWEs we should repair?
- Do you use SA tools? How much effort do you expend in auditing or fixing SA alerts?
- What SA tools do you want us to support?
- Do you have a preferred CI/CD pipeline for us to integrate with? How should the repair tool indicate its repairs? (Pull requests?)
- What do you think constitutes a satisfactory repair? What if the SA tool fails to report important defects?
- Is there a general preferred IDE for us to integrate with?
- Do you use any automated-repair mechanisms?
- Do you have any questions or requests for us?

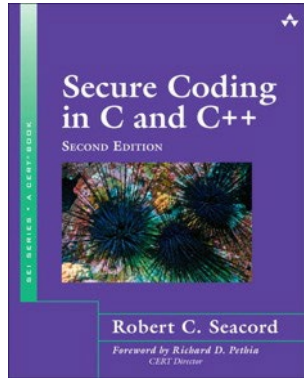
# Which CERT Rules to Repair?



1. Repair rules that collaborators requested.
2. Use some of the following rankings to prioritize rules.

Rank	Definition	Ranking	Vulnerability	Rationale
Popularity	Which rules are the most well-known?	CWE Top 25	Integer Overflow	#12 in CWE Top 25
Criticality	Which rules are ranked most critical by SA tools or rule taxonomies?	CERT Rule Priorities SA Tool Criticality Metrics	Null Pointer Dereference	CERT Priority 18
Volume	Which rules are indicated by the most alerts?	Run a tool on a codebase and count the alerts.	Integer Conversion Error	#1 in SCALe C/C++ Codebases

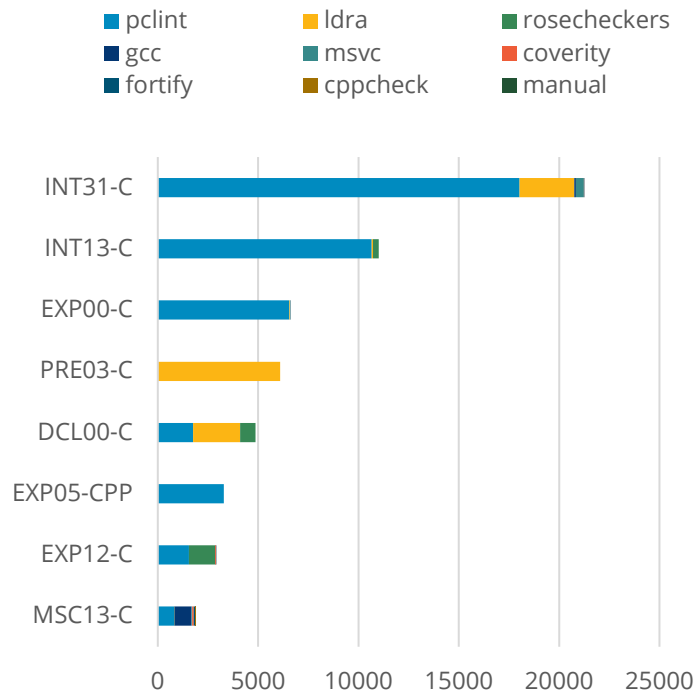
# Chosen Rules



Guideline	Title	CWE	2022 Top 25
EXP34-C	Do not dereference null pointers	476	11
EXP33-C	Do not read uninitialized memory	457	
MSC12-C	Detect and remove code that has no effect	561, 1164	
MSC13-C	Detect and remove unused values	563	
EXP12-C	Do not ignore values returned by functions	252	
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	192, 197	
EXP19-C	Use braces for the body of an if, for, or while statement		
ERR33-C	Detect and handle standard library errors	252	
INT32-C	Ensure that operations on signed integers do not result in overflow	190	13
DCL00-C	Const-qualify immutable objects		

# SA Tool Alerts: Le Déluge –2

## 5 C/C++ Audited Codebases

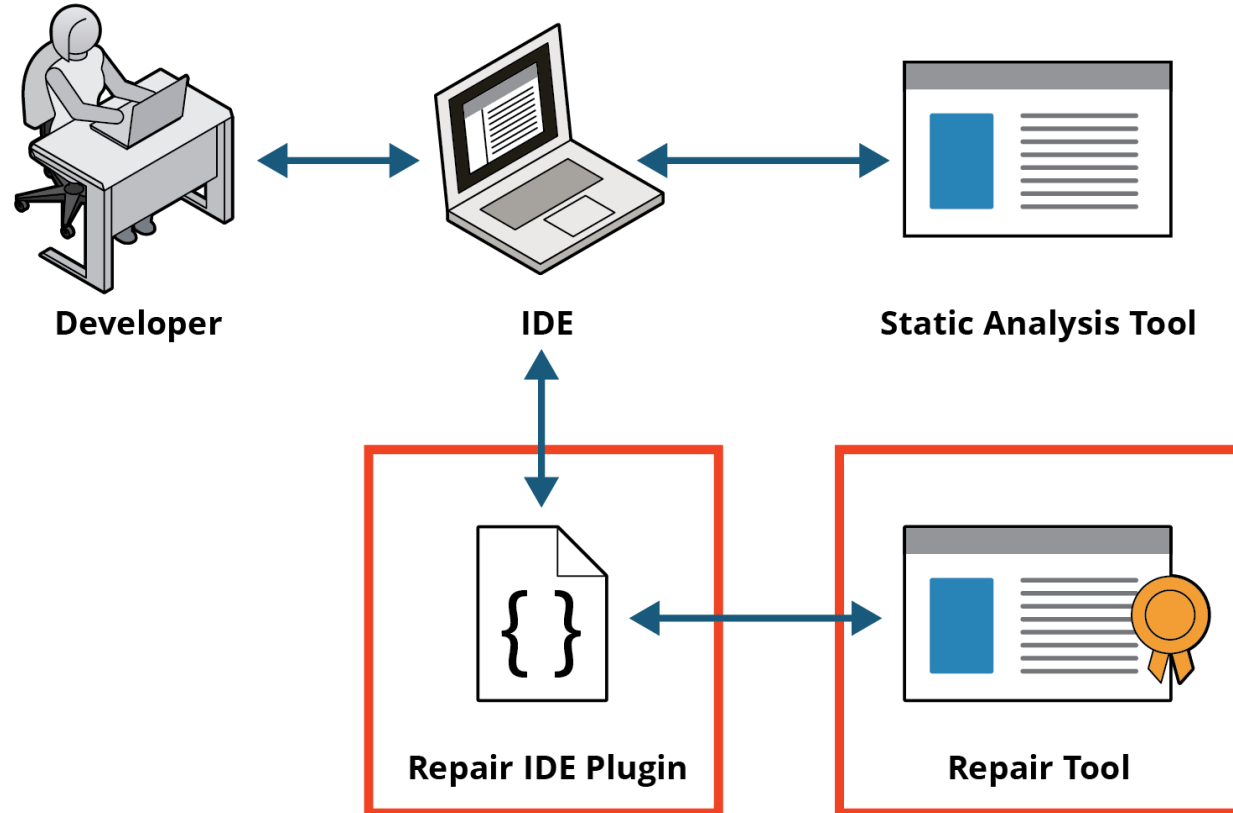


ID	Title
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data.
INT13-C	Use bitwise operators only on unsigned operands.
EXP00-C	Use parentheses for precedence of operation.
PRE03-C	Prefer typedefs to defines for encoding types.
DCL00-C	Const-qualify immutable objects.
EXP05-CPP	(VOID) Do not use C-style casts.
EXP12-C	Do not ignore values returned by functions.
MSC13-C	Detect and remove unused values.

Milestones	Depends on	Fund	23 Q1	Q2	Q3	Q4	24 H1	H2
M1. Identify most crucial defect types, rates, and current repair effort		6.2	✓					
M5. Write paper (A3) based on 6.2 results submit to conference	M2	6.2			✓		x	
M2. Build command-line program (A1) repairs 80% of 3 conditions' alerts	M1, FR	6.2				x		
M10. Extend command-line tool (M2) to handle C++ code	M2	6.2				x		
M3. Build GUI tool (A2) to integrate into IDE	M2 (A1)	6.2					x	
M8. Build CI/CD pipeline (A4) around command-line program (A1)	M2 (A1)	6.2					x	
M4. DoD transfer & test of command-line tool	M2 (A1)	6.3					x	
M6. Command-line program now repairs 10 conditions' worth of alerts	M2 (A1)	6.2					x	
M7. Integration of GUI tool into collaborator environment	M3 (A2), M4	6.3						x
M9. Integration of command-line tool into collaborator CI/CD pipeline	M3, M8	6.3						x



# Usage Architecture Scenario (Without CI)



# Wins, Challenges, and Risks

## Wins

- Can leverage new Clang tech:
  - AST export
- Additional funding
  - Added C++ milestone
- Potential CMU Collaborator:
  - Prof. Hanan Hibshi, Information Networking Institute

## Challenges

- Rules caught by collaborator data differ from proposed rules and rules caught by our OSS data.
- Collaborator data associates alerts with general (i.e., vague) CWE categories.
- FY23 6.3 funding was dropped.
  - Re-ordered milestones

## Risks

- There are no more collaborators.
- 10 repairable rules mitigate  $x < 54\%$  of alerts.

# Collaborator: Brandon Bailey (Aerospace)

Brandon Bailey works at Aerospace as a security tester for U.S. Space Force SW, which develops the IDS/protocol decoder tool described in this RFI:

- [https://imlive.s3.amazonaws.com/Federal%20Government/ID109032996500052510795704167754182169199/Attachment\\_2\\_-\\_DCO-S\\_RFI.pdf](https://imlive.s3.amazonaws.com/Federal%20Government/ID109032996500052510795704167754182169199/Attachment_2_-_DCO-S_RFI.pdf)
- The software will be deployed operationally by Space Force.

Brandon can give us the results of running our tool on this software.

- Brandon is on the test team, which sends alerts and recommended fixes to the development team. If successful, our repairs will be deployed to Space Force.
- The plan is to adopt Gitlab Ultimate for their integration pipeline/release process by FY23, which should further integrate Brandon's team into the overall development cycle.

Brandon also maintains the SA tool integration in Gitlab (their DevSecOps pipeline) and will integrate our tool as well.

# Other Collaborators

- Mitchell Perry of C5ISR U.S. Army and Alan Sorensen of USAF AFMC have expressed interest in this project.

To obtain more collaborators, members of the Automated Repair team will

- Present at significant conferences (as part of a Secure Coding presentation):
  - DoD SwA COP
  - DoD Cyber COP
  - Cross Service Cyber T&E workshop
- Send email to more DoD contacts.

# Handling Errors

What should our tool instruct the program to do when it discovers an error (e.g., integer overflow) and `/* Handle error */` is not sufficient?

Some choices include

- `return;`
- `return NULL; /* or EOF */`
- `abort();`
- `signal(SIGINT, handler);`

The right choice depends on the code. How does the function currently handle other errors?

# Paper

David Svoboda, Dr. Will Klieber, and Dr. Lori Flynn drafted the paper, *Using Automated Code Repair to Fight Back the Deluge of False Positives*, which they will submit to a conference once the full test results are available.

# Tool Architecture

## Repair Library

### Command Line Tool

Library has a single `repair()` function, which takes the following as input:

- Path to C/C++ source file
- Set of alerts (perhaps in SARIF format), ignoring all alerts except the ones the tool knows how to fix

#### Function Outputs

- Repaired source file, perhaps in place
- Log of changes made

### IDE Plugin

IDE Plugin would provide a “search/replace” GUI for the repair tool.

The user can preview the repair of a single alert and decide whether to apply the repair.

Or the user can select a **Repair All** button and repair everything.

# Technical Transition/Impact

Milestones	Depends on	Fund Type	23 Q1	Q2	Q3	Q4	24 E1	E2
M1. Identify most crucial defect types, rates, and current repair effort		6.2	✓					
M2. Build command-line program (A1) repairing 80% of 3 conditions' alerts	M1, FR	6.2	✓	x				
M3. A collaborator wants a repair of OSS <a href="#">Zeek v4</a> , which is mostly C++ code. We did not plan to handle C++ code, but it should be possible.		6.2		x	x	x		
M8. The collaborator sent us data running the following: Fortify, Checkmarx, and Cppcheck via CodeDX on Zeek v4.		6.2		x	x	x		
M4. <del>Build transfer &amp; test of command line tool</del>	<del>M2 (A1)</del>	6.3	x	x	x	x		
M5. Write paper (A3) based on 6.2 results submit to conference	M2	6.2			x	x	x	
M6. Command-line program now repairs 10 conditions' worth of alerts	M2 (A1)	6.2			x	x	x	
M7. Integration of GUI tool into collaborator environment	M3 (A2), M4	6.3					x	x
M9. Integration of command-line tool into collaborator CI/CD pipeline	M3, M8	6.3					x	x



# “Ear” Module

## Inputs

- Codebase
- Build command
- ~~SA tool alerts~~

## Output

- A serialized (JSON) AST with enhancements.

It will use the build command to run Clang on the code, and have Clang produce an AST.

There will be submodules that enhance the AST with extra annotations (e.g., things best done here that need to examine the source code).

# “Brain” Module

## Inputs

- SA tool alerts
- Enhanced AST
- ~~Source code~~
- ~~Build command~~

## Output

- Enhanced alerts
- More enhanced AST

Each alert gets either a message explaining why it should not be repaired or a “green-light” indication. The message can include specific details about what kind of repair to do. One such detail would be what error-handling mechanism should be used. (e.g., “return NULL”).

# “Glove” Module

## Inputs

- Enhanced alerts with patches
- Source code

## Output

- Repaired source code

This approach is useful when doing an end-to-end repair.

It is not used by the GUI; the GUI uses enhanced alerts with patches to query the user about which repairs to make.