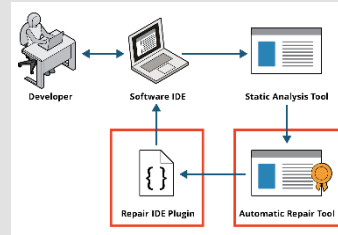


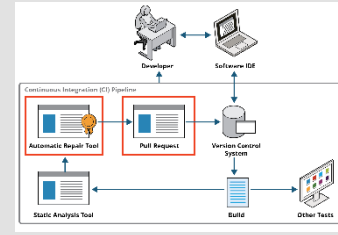


```
143 145 void *realloc(void *ptr, size_t size) {
144 146 #ifndef HAVE_MALLOC_SIZE
145     void *realptr;
147+    void *realptr = NULL;
146 148 #endif
147     size_t oldsize;
148     void *newptr;
149+    size_t oldsize = 0;
150+    void *newptr = NULL;
149 151
150 152 if (ptr == NULL) return zmalloc(size);
```

via



or

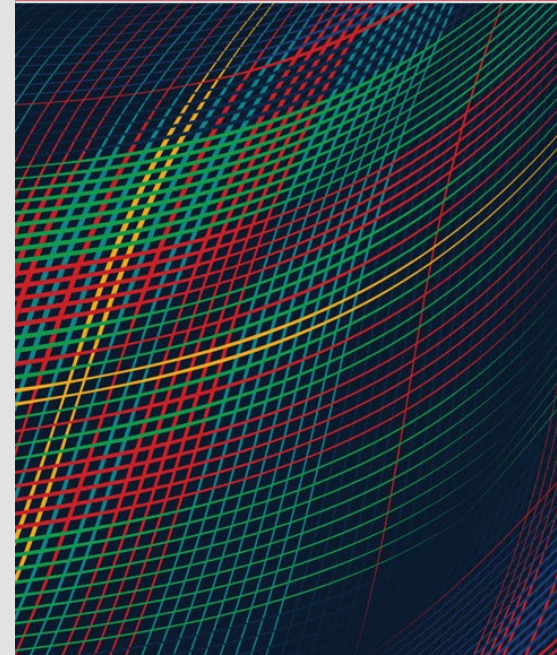


# Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

for NDIA's 27th Systems & Mission Engineering Conference

OCTOBER 2024

Lori Flynn, PhD  
David Svoboda (PI)



Carnegie Mellon University 2024

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.

References herein to any specific entity, 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 nor of Carnegie Mellon University - Software Engineering Institute by any such named or represented entity.

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.

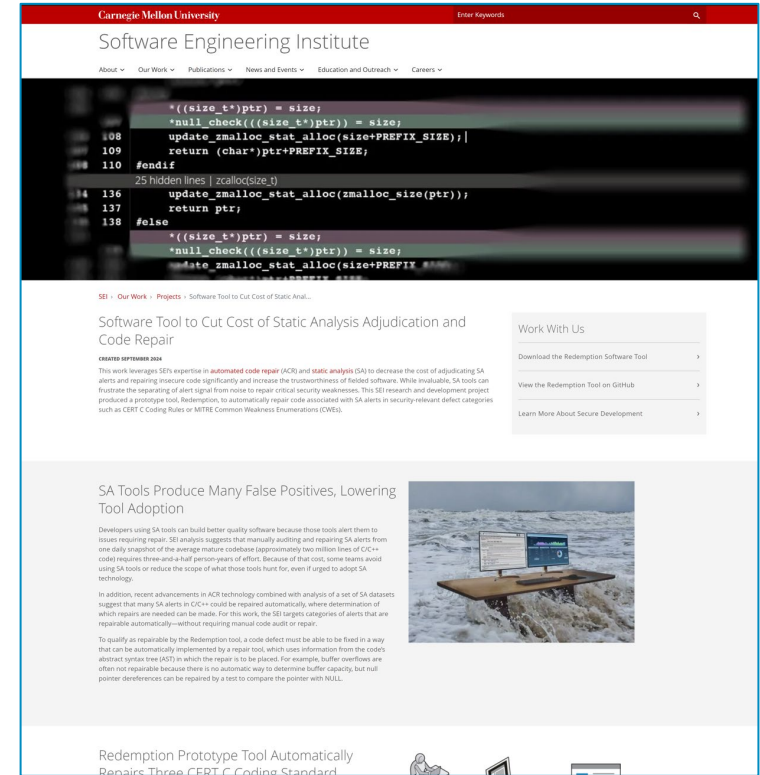
This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

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

DM24-1310

# Agenda

- Problem: static analysis (SA) alert deluge
- Our tool repairs source code associated with alerts
- Design choices
- Tool use during development, test, and evaluation
- Development methods
- Test results
- Demo
- How can this work be extended to help you?



Project page [https://www.sei.cmu.edu/our-work/projects/display.cfm?customel\\_datapageid\\_4050=497941](https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=497941)



# Problem: static analysis (SA) alert deluge



## Case study of 5 C/C++ audited codebases

- 239 kSLoC
- 364.5 alerts/kSLoC
- **85,268 SA alerts**
- **Repairs for 8 CERT rules would resolve 57,922 alerts (68%)**

## Average CERT-audited C/C++ program is 2 MSLoC

- 117 seconds to audit one alert\*
- **15.5 person-years to audit all alerts**
- If 32% of alerts are true and 117 seconds per repair → **5 person-years to fix all true alerts**

\* Ayewah, Nathaniel. & Pugh, William. The Google FindBugs fixit. Pages 241-252. In *Proceedings of the 19<sup>th</sup> International Symposium on Software Testing and Analysis*. July 2010.  
<https://doi.org/10.1145/1831708.1831738>

# 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.

# Collaborator Experience

**Of the languages that our collaborator uses, they told us that C code tends to exhibit the most vulnerabilities.**

**One collaborator's 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.

# C/C++ Automated Program Repair (APR) Tools

## Template-based APR tools have a pre-set method to repair a defect

- **Visual Studio Code** has some APR for C/C++
- **Eclipse IntRepair** open-source APR tool for [integer overflows, buffer overflows, and more](#) (per research [papers](#)) is an extension to the C/C++ Development Tools (CDT) plugin
- **Automated Code Repair** (SEI's Dr. Will Klieber) APR for buffer overflows in C. It converts pointers to fat pointers, potential for changes throughout the codebase
- **clang-tidy** has recent APR fixes for many C/C++ [checkers](#)
- **Clang's** new JSON API outputs the AST in an easy-to-parse JSON file, useful for developing APRs

Rationale for project: 1. Significant DoD use of C code, 2. `clang`'s new JSON API, and 3. we did not find any OSS APR tool documentation that explicitly states a fix for "CERT C secure coding rule violations"

## Learning-based APR tools use AI/ML/LLMs, past bugfixes, & more to make new patches

- Contact Lori [lflynn@sei.cmu.edu](mailto:lflynn@sei.cmu.edu) about collaboration on APR research involving learning-based methods

# Our tool repairs source code associated with alerts

<u>Category</u>	<u>CERT Rule ID</u>	<u>CWE ID</u>	<u>Repair</u>
Null Pointer Dereference	<a href="#">EXP34-C</a>	<a href="#">CWE-476</a>	Insert null check
Uninitialized Value Read	<a href="#">EXP33-C</a>	<a href="#">CWE-908</a>	Initialize variable at declaration
Ineffective Code	<a href="#">MSC12-C</a>	<a href="#">CWE-561</a>	Delete ineffective code

```
*((size_t*)ptr) = size;  
*null_check(((size_t*)ptr)) = size;  
update_zmalloc_stat_alloc(size+PREFIX_SIZE); |
```



Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

# Design choices

# Design choices

1. Make cheap, local fixes.
2. Only fix code associated with an SA alert.
3. Goal: Fixes are sound and do not change the behavior of good code.
  - A repair should not break the code, even if the alert was a false positive.
4. 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;  
    /* ... */  
}
```

# Design choices

1. Make cheap, local fixes.
2. Only fix code associated with an SA alert.
3. Goal: Fixes are sound and do not change the behavior of good code.
  - A repair should not break the code, even if the alert was a false positive.
4. 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;  
    /* ... */  
}
```



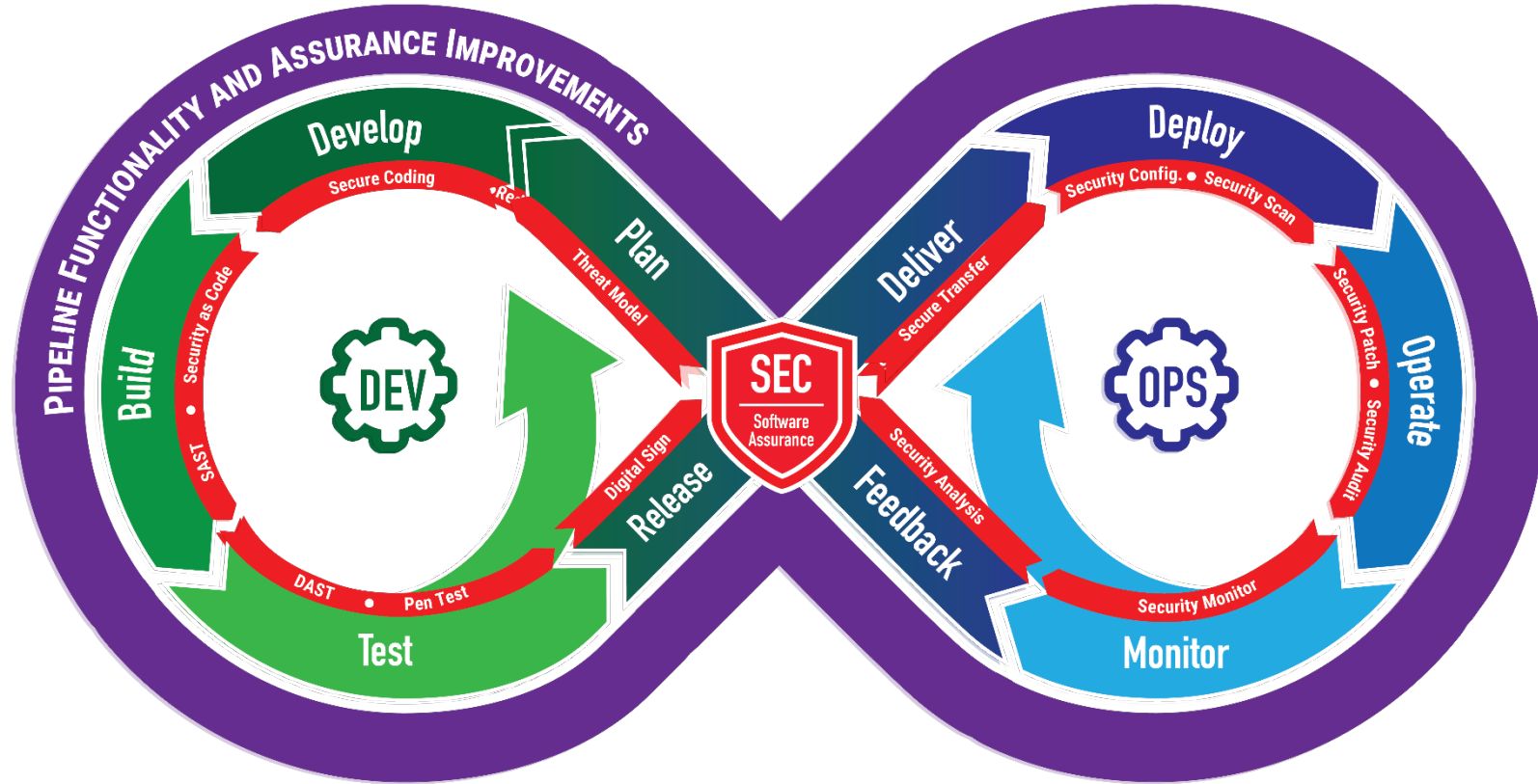
```
char *f(int a, int b) {  
    int x;  
    int sum = SAFE_ADD(a, b,  
    /* Handle error */  
    );  
    /* ... */  
}
```

Possible integer overflow?

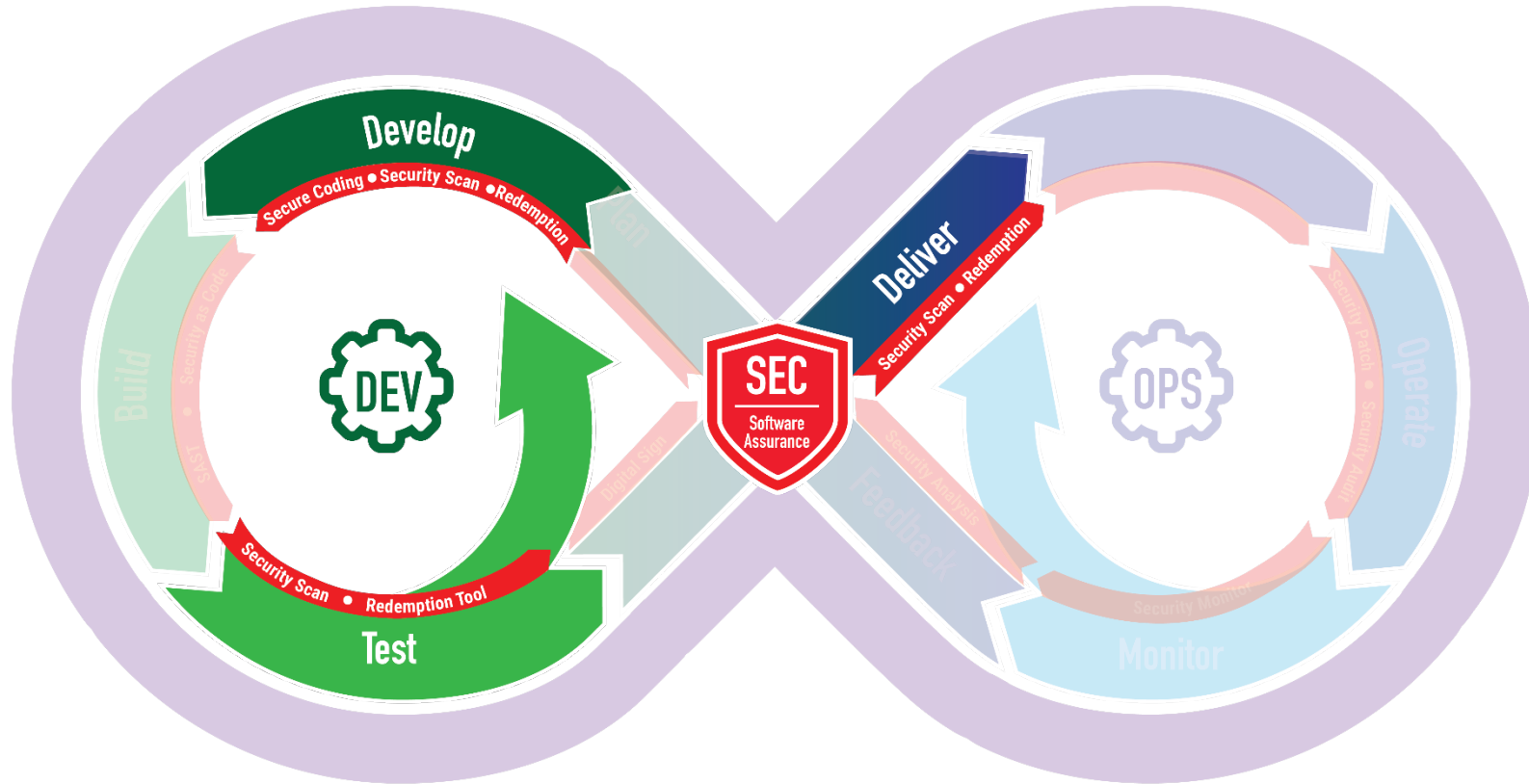
Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

# Tool use during development, test, and evaluation

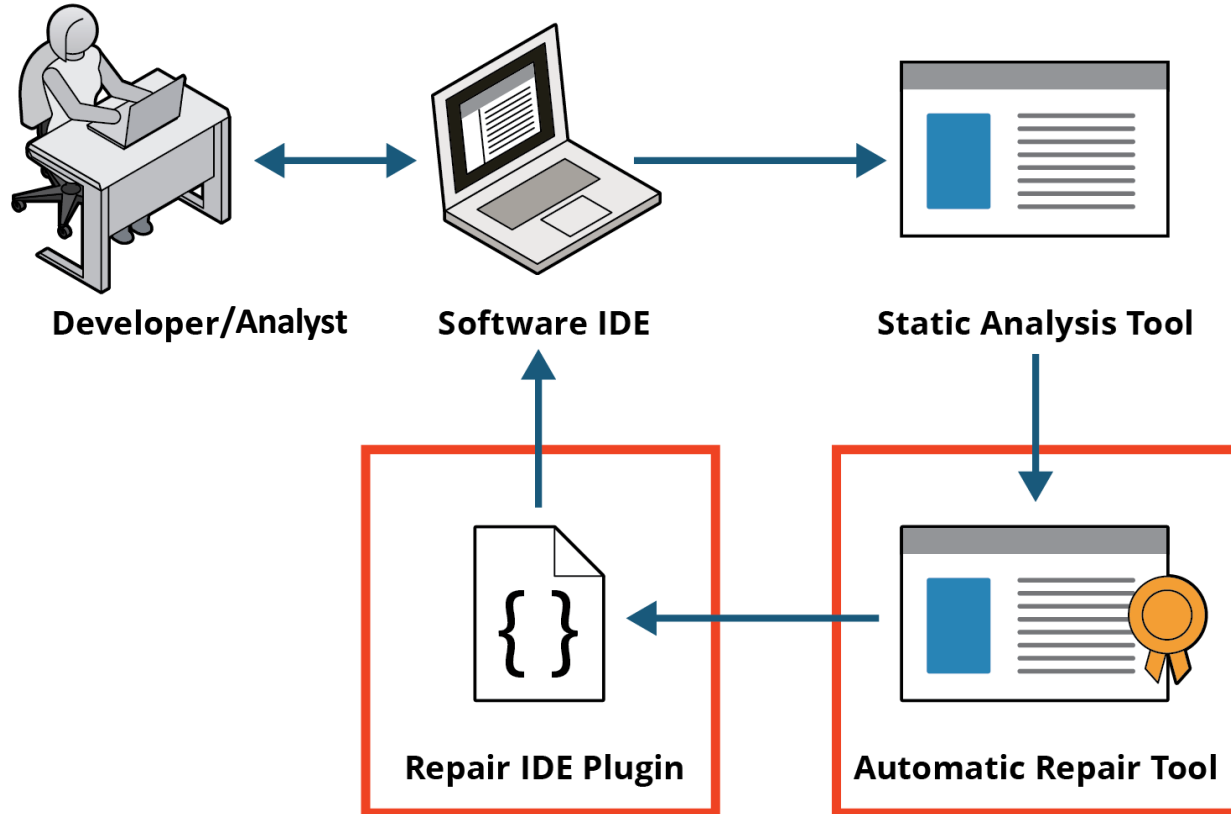
# Where to use Redemption in DevSecOps



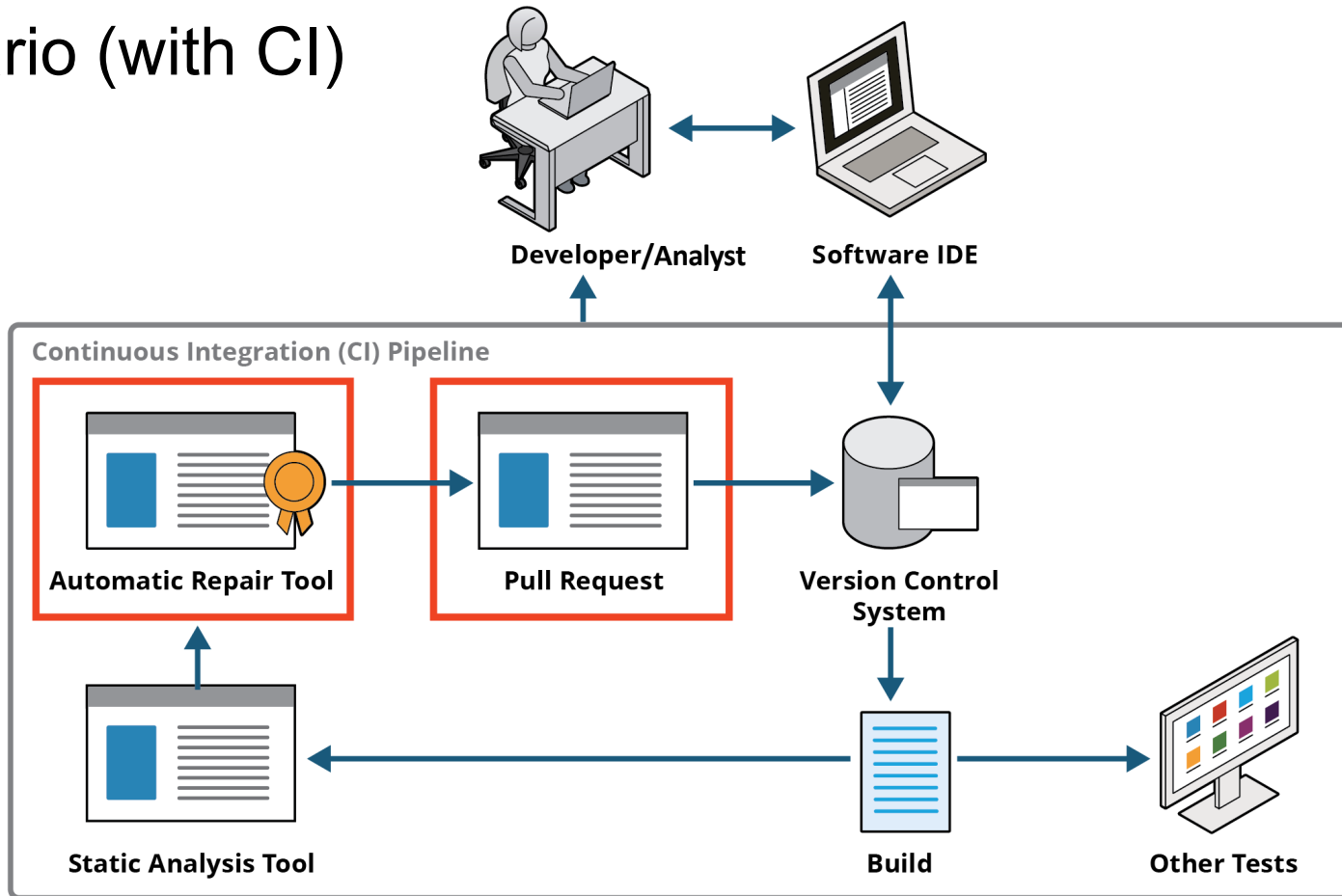
# Where to use Redemption in DevSecOps



# Usage Dataflow Scenario (Without CI)



# Usage Dataflow Scenario (with CI)

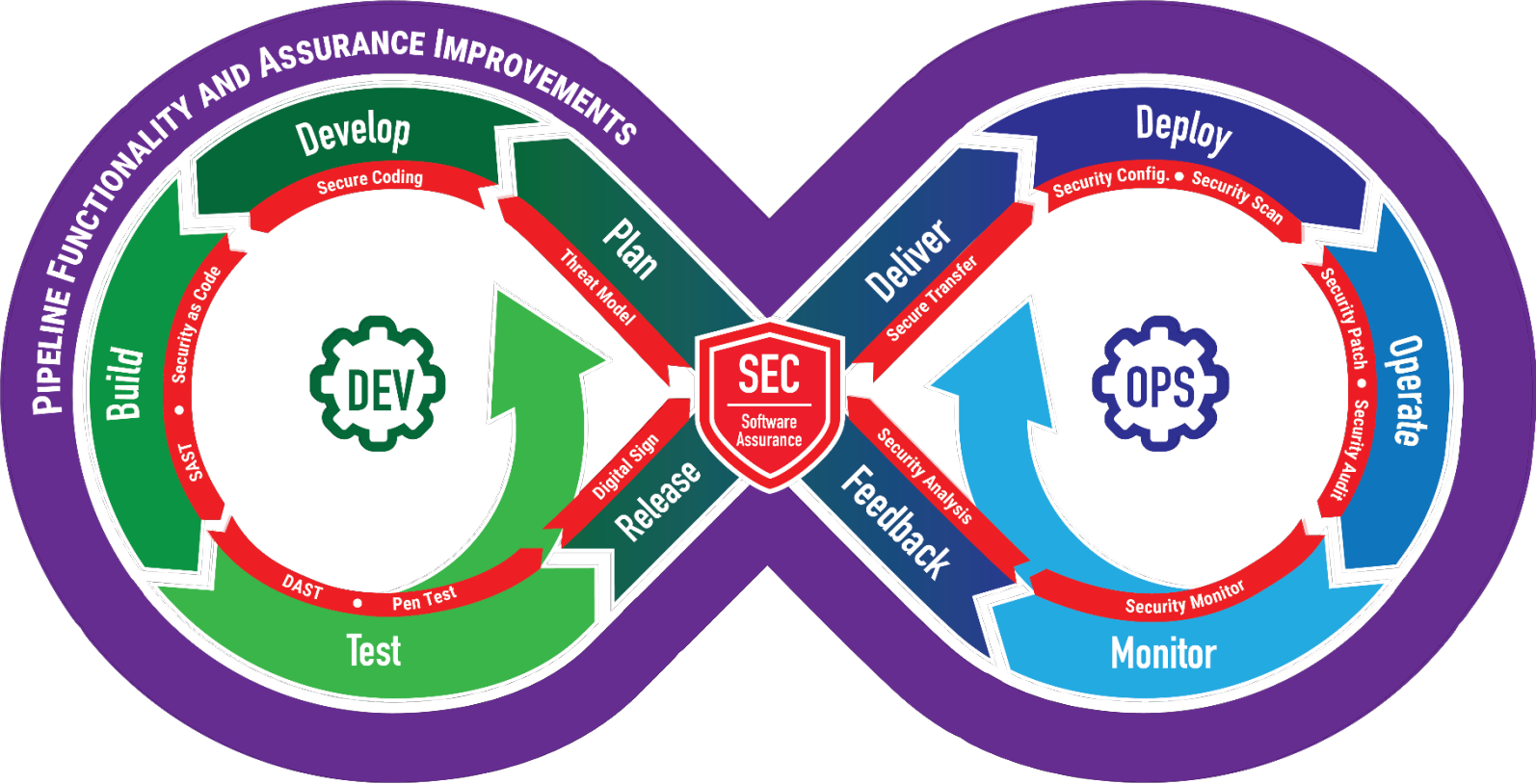




Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

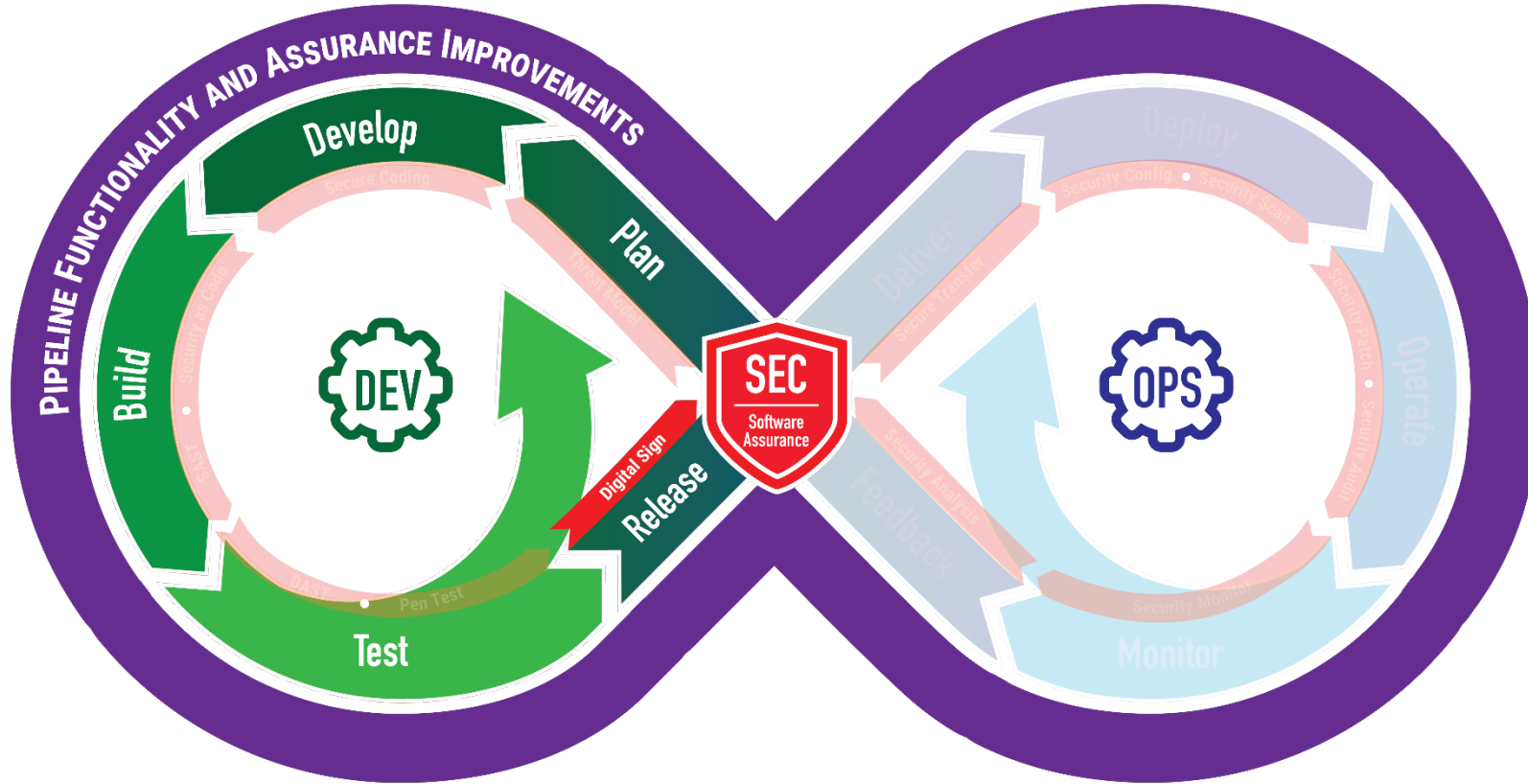
# Development methods

# Existing Redemption capabilities that help extending it



- Docker containerized
- Tests (unit, integration, performance, etc.)
- Modular code
- Documentation
- Demos
- Test code + static analysis alerts

# Existing Redemption capabilities that help extending it



- Docker containerized
- Tests (unit, integration, performance, etc.)
- Modular code
- Documentation
- Demos
- Test code + static analysis alerts

# How to Develop New Repairs

1. Choose code flaw to repair
2. Find or create test cases that need repair
3. Develop repair:
  - a. Determine repair site of flawed code using AST (.json) and LLVM IR (.ll) code
  - b. Implement “template” repair algorithm to repair the code
4. Run tests (unit, integration, performance etc.)
5. Iteratively address any bugs
6. Document repair method in README.md

Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

# Testing & test results

# 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.

ISO C only constrains programs without UB.

- UB means the platform may do anything.



International  
Organization for  
Standardization,  
Public domain, via  
Wikimedia  
Commons

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 (without repair).

We address these 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 ← All these tests currently pass

*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 ← Next slide

*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 APR did the right thing:
    - Repaired correctly or correctly refused to repair.
  - Until APR does the Right Thing on >=80% of alerts, Fix APR bugs and re-run experiment.

## Integration Experiments ← All these tests currently pass

*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 ← All timing tests pass for git and zeek\*.

*Confirms that repairs do not significantly impede performance*

- Compile original codebases; run their internal testing mechanism.
  - Measure the time and memory 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 ← All these tests currently pass

*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 APR tool on the repaired codebase's new alerts.
- Ideally, the APR tool should do nothing since what remains are only the alerts it could not repair.
- If a repaired alert recurs, the APR tool should report it as a false positive.

# Test Results for Sample Alert Experiments

	git	git	git	zeek	zeek	zeek
	clang-tidy	cppcheck	rosecheckers	clang-tidy	cppcheck	rosecheckers
<b>EXP33-C</b>	9157	1		5225	29	
<b>EXP34-C</b>	77	20		44	53	14
<b>MSC12-C</b>		25	721		131	480

	git	git	git	zeek	zeek	zeek
	clang-tidy	cppcheck	rosecheckers	clang-tidy	cppcheck	rosecheckers
<b>EXP33-C</b>	100.0% (5/5) [0,0,5,0,0,0,0]	100.0% (1/1) [0,0,1,0,0,0,0]		100.0% (5/5) [1,0,4,0,0,0,0]	100.0% (5/5) [2,0,3,0,0,0,0]	
<b>EXP34-C</b>	100.0% (5/5) [4,0,1,0,0,0,0]	100.0% (5/5) [1,2,2,0,0,0,0]		100.0% (5/5) [4,2,0,0,0,0,0]	100.0% (5/5) [2,2,1,0,0,0,0]	100.0% (5/5) [5,0,0,0,0,0,0]
<b>MSC12-C</b>		20.0% (1/5) [1,0,0,4,0,0,0]			40.0% (2/5) [2,0,0,2,1,0,0]	

# Testing Result States for Sample Alert Experiments

Is_satisfactory	Is_repaired	Adjudication	Label
Satisfactory	Repaired	True/suspicious	A
Satisfactory	Repaired	False positive	C
Satisfactory	Not repaired	True/suspicious	None
Satisfactory	Not repaired	False positive	B
Unsatisfactory	Repaired	True/suspicious	F
Unsatisfactory	Repaired	False positive	G
Unsatisfactory	Not repaired	True/suspicious	D
Unsatisfactory	Not repaired	False positive	E

A+B+C = 100%  
of all alerts, for 2 rules

G = 0%  
Don't break code!

# Some repairs require human supervision (accept/reject)

## Some repair types are expected correct; others require human supervision

Not always a good idea to make the MSC12-C changes.

- MSC12-C (“Ineffective Code”) is a recommendation, not a rule in the CERT coding standard
- Repairs would not necessarily improve the code.

MSC12-C alerts are flagged for many reasons. For example:

- A label is never accessed via goto. Often generated by tools like yacc(1).
  - Removing the label may not change code behavior.
  - The label makes the code simpler. It might represent a node in a state diagram or DFA.

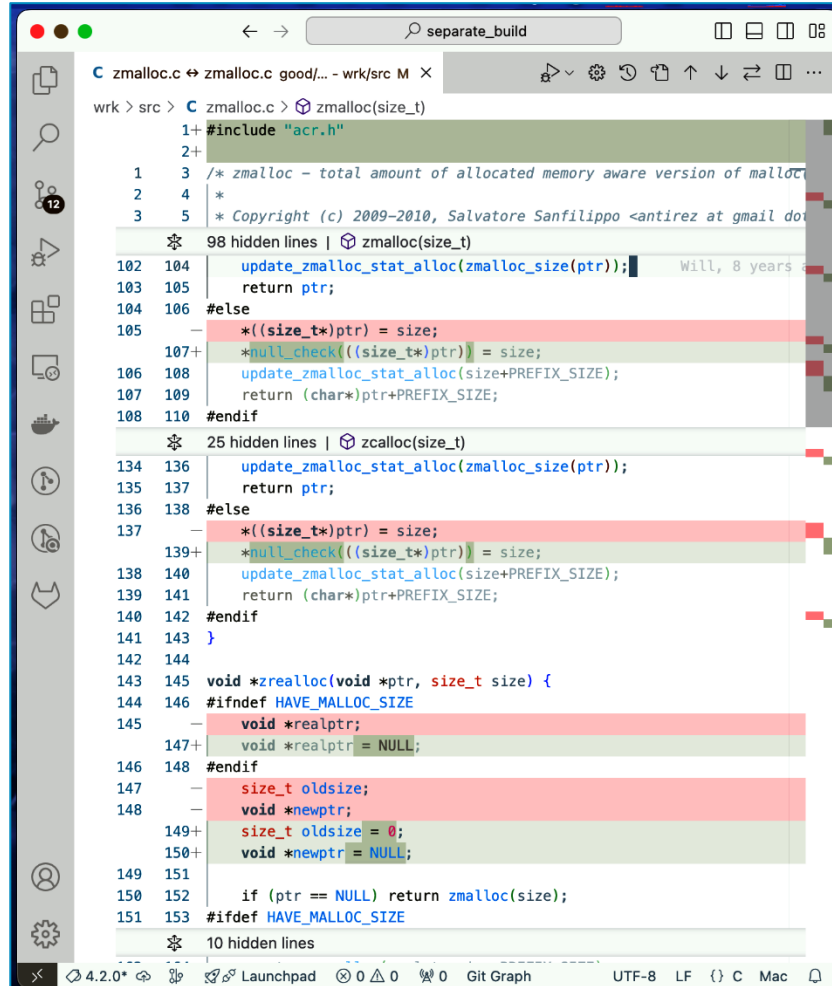
## MSC12-C repairs are disabled by default (enabled via environment variable)

Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

# Demo



# Merging repaired code with original code (1/3)



```
zmalloc.c ↔ zmalloc.c good/... - wrk/src M X
wrk > src > C zmalloc.c > zmalloc(size_t)
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
* 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
* 10 hidden lines
```

# Merging repaired code with original code (2/3)

```

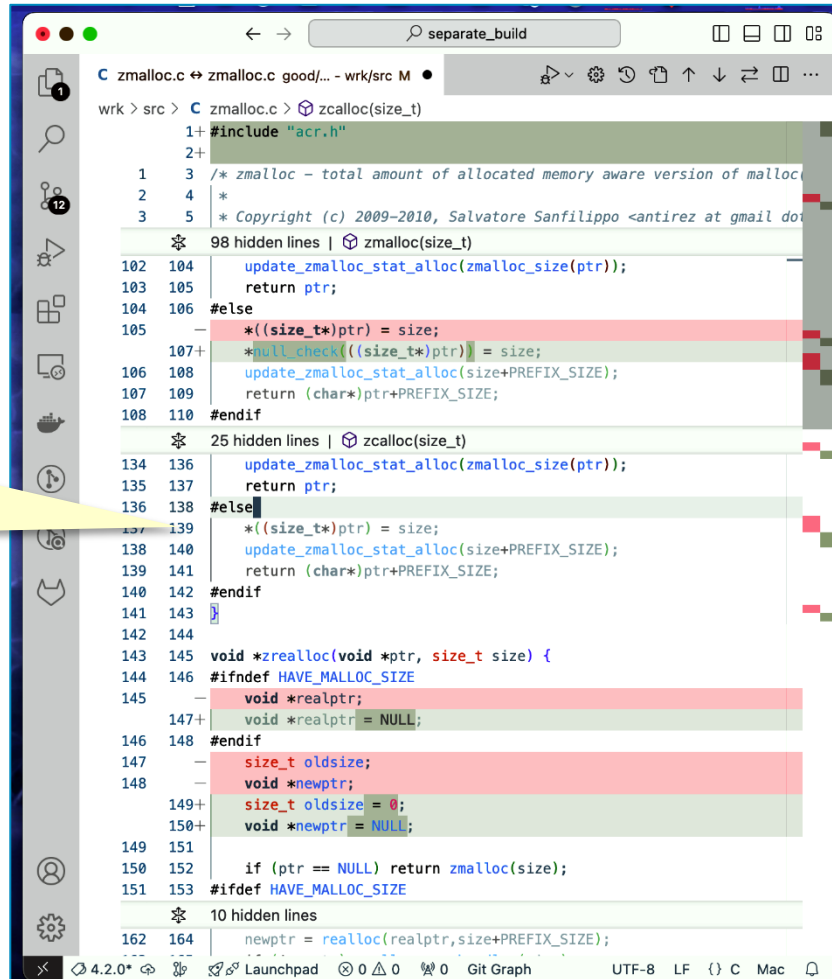
C zmalloc.c ↔ zmalloc.c good/... - wrk/src M X
wrk > src > C zmalloc.c > zcalloc(size_t)
1+ #include "acr.h"
2+
1 3 /* zmalloc - total amount of allocated memory aware version of malloc
2 4 *
3 5 * Copyright (c) 2009-2010, Salvatore Sanfilippo <antirez at gmail dot
98 hidden lines | zmalloc(size_t)
102 104     update_zmalloc_stat_alloc(zmalloc_size(ptr));
103 105     return ptr;
104 106 #else
105 -     *((size_t*)ptr) = size;
107+     *full_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
139 -     *((size_t*)ptr) = size;
139+     *full_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
148 -     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
10 hidden lines
  
```

If you dislike a repair, you can click on a line of code...

Copy changed line  
Revert this change



# Merging repaired code with original code (3/3)



```
zmalloc.c ↔ zmalloc.c good/... - wrk/src M
wrk > src > C zmalloc.c > zcalloc(size_t)
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
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
137 139 *((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
10 hidden lines
162 164 newptr = realloc(realptr, size+PREFIX_SIZE);
```

...and revert it!

Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

# How can this work be extended to help you?

# The Automated Repair Team



**David Svoboda**  
Senior Software  
Security Engineer  
Principal Investigator



**Will Klieber**  
Software Security  
Engineer



**Lori Flynn**  
Senior Software  
Security Researcher



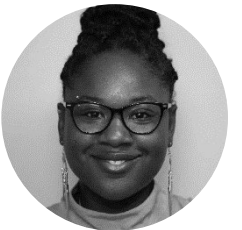
**Joseph Sible**  
Associate Software  
Engineer



**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)

# How can this work be extended to help you?



## Potential extensions

1. Add support for more static analysis tools
2. Repairs for more categories of SA alerts
3. Enhance Redemption's capability to work on MS Windows programs
4. Integrate more workforce tools, including IDEs and CI pipelines

## Related APR proposal

1. Lori is **looking for DoD/govt. collaborators** on her research project proposal involving **learning-based APR** (proposal due 11/11)
2. What APR feature(s) would make your organization likely to use it?
3. What are barriers to APR use at your org?

## Contact

**David Svoboda** svoboda@sei.cmu.edu

**Lori Flynn** lflynn@sei.cmu.edu

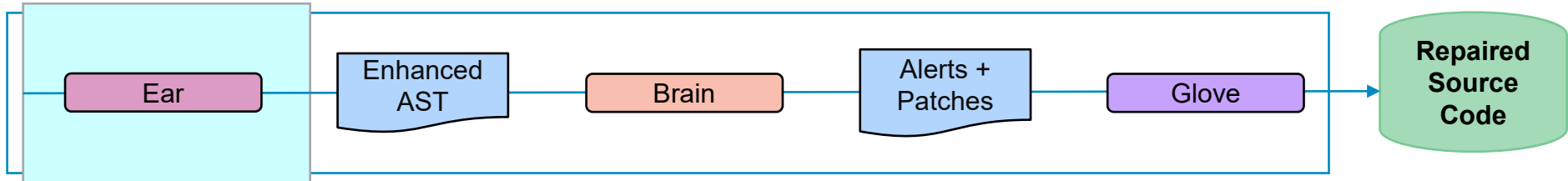
## Achievements highlights

- Developed APR tool that repairs 3 CERT coding rules and 3 mapped CWEs
- Tested tool on OSS codebases and collaborator code, with successful repairs
- Published code, OSS test results, use documentation, demo videos, presentations
- Published [dataset](#) for APR research & testing
- Research paper (pending acceptance)
- [Redemption project page](#) (links to tool, dataset, presentations, videos, paper, etc.)
- [Redemption tool](#) on [GitHub](#)

Static Analysis-Targeted Automated Repair to Secure Code and Reduce Effort

# BACKUP SLIDES

# Command Line Tool – Source Codebase

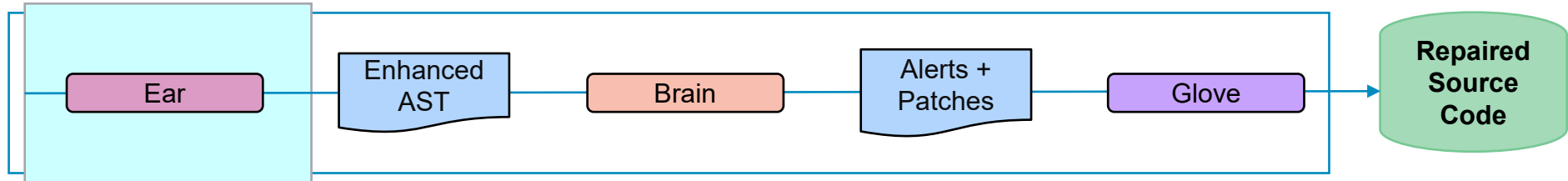


## Inputs

C/C++ source file(s) in codebase

```
1
2 int flag = 0;
3
4 #define NULL 0
5
6 /* Should return 0 upon error */
7 unsigned int foo1(int* p) {
8     if (flag) {
9         return 0;
10    }
11    return *p;
12 }
13
14 /* Should return -1 upon error */
15 int foo2(int* p) {
16     if (flag) {
17         return -1;
18     }
19    return *p;
20 }
21
22 /* Should return NULL upon error */
```

# Command Line Tool – Build Commands

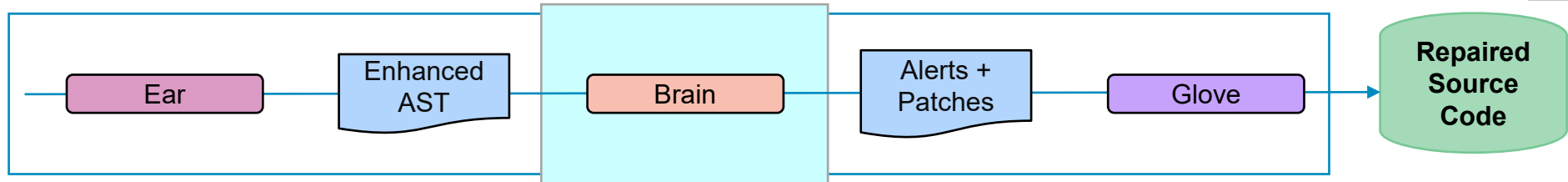


## Build Commands

Each command includes -D/-U macro definitions and other switches to let Clang parse each source code file.

```
cc -DDEBUG=0 -I/usr/local/include -O2 -Wall -c pgm.c -o pgm.o
```

# Command Line Tool – Static Analysis Alerts



## Next input: distinct SA Tool Alerts

Each alert contains the following:

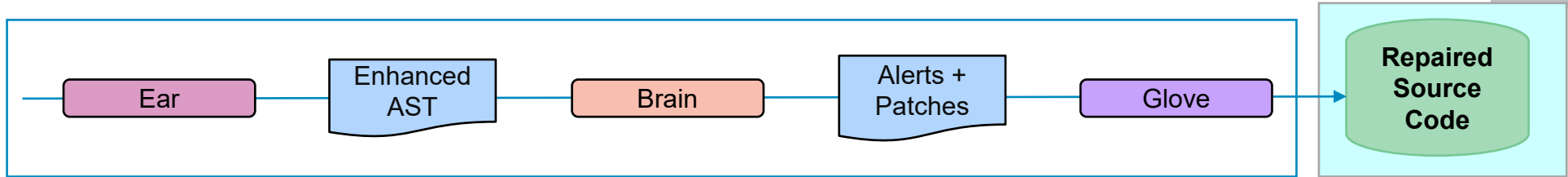
- CERT rule
- Location where rule is being violated (e.g., source code path, line number, column number, end-line number, end column number)
- Message

```

<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
  <cppcheck version="2.9"/>
  <errors>
    <error id="unreadVariable" severity="style" msg="Variable &apos;InF&apos; is assigned a
      <location file="/datasets/dos2unix/common.c" line="1331" column="12"/>
      <symbol>InF</symbol>
    </error>
    <error id="unreadVariable" severity="style" msg="Variable &apos;InF&apos; is assigned a
      <location file="/datasets/dos2unix/common.c" line="1373" column="12"/>
      <symbol>InF</symbol>
    </error>
    <error id="unreadVariable" severity="style" msg="Variable &apos;conversion_error&apos;
      <location file="/datasets/dos2unix/common.c" line="2549" column="28"/>
      <symbol>conversion_error</symbol>
    </error>
    <error id="uninitvar" severity="error" msg="Uninitialized variable: lpMsgBuf" verbose=""
      <location file="/datasets/dos2unix/common.c" line="117" column="19"/>
      <symbol>lpMsgBuf</symbol>
    </error>
    <error id="ConfigurationNotChecked" severity="information" msg="Skipping configuration
  
```



# Command Line Tool – Repaired Source Code



## 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)

```

1- #include "acr.h"
2-
3  /* zmalloc - total amount of allocated memory aware version of malloc
4  *
5  * Copyright (c) 2009-2010, Salvatore Sanfilippo <antirez at gmail dot
6  *
7  * 98 hidden lines | zmalloc(size_t)
8  *
9  * 102 104 update_zmalloc_stat_alloc(zmalloc_size(ptr)); Will, 8 years
10 105 return ptr;
11 106 else
12 107 * ((size_t*)ptr) = size;
13 108 * NULL check ((size_t*)ptr) = size;
14 109 update_zmalloc_stat_alloc(size+PREFIX_SIZE);
15 110 return (char*)ptr+PREFIX_SIZE;
16 111 #endif
17 112 * 25 hidden lines | zcalloc(size_t)
18 113 update_zmalloc_stat_alloc(zmalloc_size(ptr));
19 114 return ptr;
20 115 else
21 116 * ((size_t*)ptr) = size;
22 117 * NULL check ((size_t*)ptr) = size;
23 118 update_zmalloc_stat_alloc(size+PREFIX_SIZE);
24 119 return (char*)ptr+PREFIX_SIZE;
25 120 #endif
26 121 }
27 122
28 123 void *zrealloc(void *ptr, size_t size) {
29 124 #ifdef HAVE_MALLOC_SIZE
30 125 void *realptr;
31 126 void *realptr = NULL;
32 127 #endif
33 128 #ifdef HAVE_MALLOC_SIZE
34 129 size_t oldsize;
35 130 void *newptr;
36 131 size_t oldsize = 0;
37 132 void *newptr = NULL;
38 133
39 134 if (ptr == NULL) return zmalloc(size);
40 135 #ifdef HAVE_MALLOC_SIZE
41 136
42 137 * 10 hidden lines
  
```

# 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?