Automated Repair of Static Analysis Alerts

SEPTEMBER 6, 2023

David Svoboda
Software Security Engineer

Carnegie Mellon University
Software Engineering Institute

Copyright 2023 Carnegie Mellon University

Project Summary

Problem: Static analysis (SA) tools produce many alerts on C/C++ code, many of which are false positives.

Solution: Automatically repair 80% or more of each type of SA alerts 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.

Outcome and Impact

- **Year 1:** Enhance the coverage of the repair prototype tool so that it can automatically repair 80% of alerts in 3 different security-relevant defect categories (e.g., 3 CERT C Coding Rules or MITRE Common Weakness Enumerations [CWEs]).
- **Year 2:** Department of Defense (DoD) collaborators will run our repair prototype tool integrated into (1) their development environment and (2) a continuous integration (CI) pipeline if they use one. Our goal is that, by using our tool, our collaborators will reduce their auditing and manual repair effort by up to 80% for each alert in 10 CERT rules or CWEs. We plan to adjust our strategy using the data from Year 1 to achieve 80% effort reduction.

Agenda

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

Automated Repair of Static Analysis Alerts

The Problem

Carnegie Mellon University
Software Engineering Institute

Does the DoD Require Use of Static-Analysis Tools?

Carnegie Mellon University Software Engineering Institute

From the Application Security & Development (ASD) Security Technical Implementation Guide (STIG):

- Section 4.1 of the file U_ASD_V5R1_Overview.pdf (updated October 26, 2020) notes the following:
  - Application code scanners should be utilized whenever possible.
    - Application code scanner is the ASD STIG's term for an SA tool.
- Finding ID V-222624 notes the following:
  - 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...

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.

Summary

Carnegie Mellon University Software Engineering Institute

Problem

- There are too many SA tool alerts to audit them all.
- Even if we knew which alerts were true positives, there are still too many to repair all of them manually.

Solution

- Create a repair tool that takes a set of alerts of one category (e.g., integer overflow).
- Automatically repair the source code for each alert, without auditing it.
- Do this for as many categories as possible (3 categories in FY23 and 10 in FY24).
- Allow the collaborator to run the tool in their environment.

* The number of alerts repaired depends on the particular codebase, SA tool(s), and developers. For example, some developers avoid particular code flaws or 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.

SA Tool Alerts: Le Déluge –1

5 C/C++ Audited Codebases

- 239 kSigLoC
- 85,268 SA alerts
- 364.5 alerts / kSigLoC
- 57,922 alerts (67.9%) violate just 8 CERT rules.
  - Fixing 80% of these 8 rules fixes 54% of all alerts.
- The average audited codebase has 1,957,040 LoC.

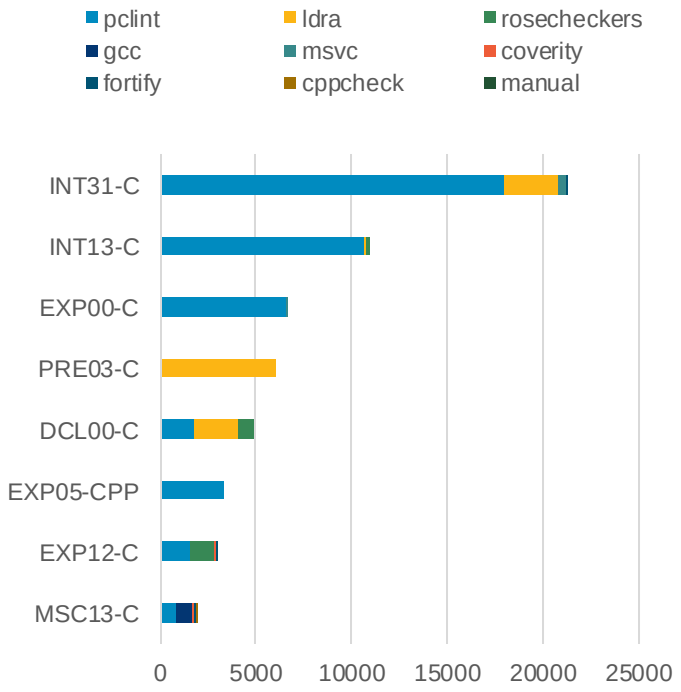
117 Seconds/Alert for Auditing*

- If we also assume 117s to fix each true positive, then
- it will take **3.43 person-years** to audit and fix all alerts in the average C codebase
  - as part of software sustainment

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

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.

Collaborator Experience

Of the languages that our collaborators use, C code tends to exhibit the largest number of vulnerabilities.

The process is

- Filter alerts based on a pre-set list of CWEs; 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.

SA tools in a CI pipeline benefit our collaborators more than the SA tool's GUI.

Code Repair State of the Art

Carnegie Mellon University Software Engineering Institute

Refactoring Code

- Eclipse has refactored Java code since 2001.
- Visual Studio (VS) Code refactors newer languages but not C/C++.
- **`clang-tidy`** now has recent refactoring & 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)

- Provides automatic rewriting of C/C++ code to address specific problems.
- Imposes major infrastructure change to entire C code. (To solve buffer overflows, 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

Carnegie Mellon University
Software Engineering Institute

Technical Approach

Make cheap, local fixes.

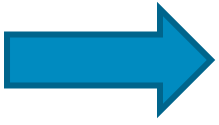
Only fix code associated with an SA alert.

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

- If an alert is a false positive, it does not break the code.
- 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.

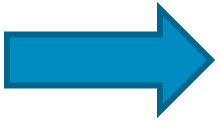
Only fix code associated with an SA alert.

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

- If an alert is a false positive, it does not break the code.
- 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 */
  );
  /* ... */
```

Command Line Tool (M2) –1

Inputs

Codebase (path to C/C++ source code file)

Build Command

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

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.

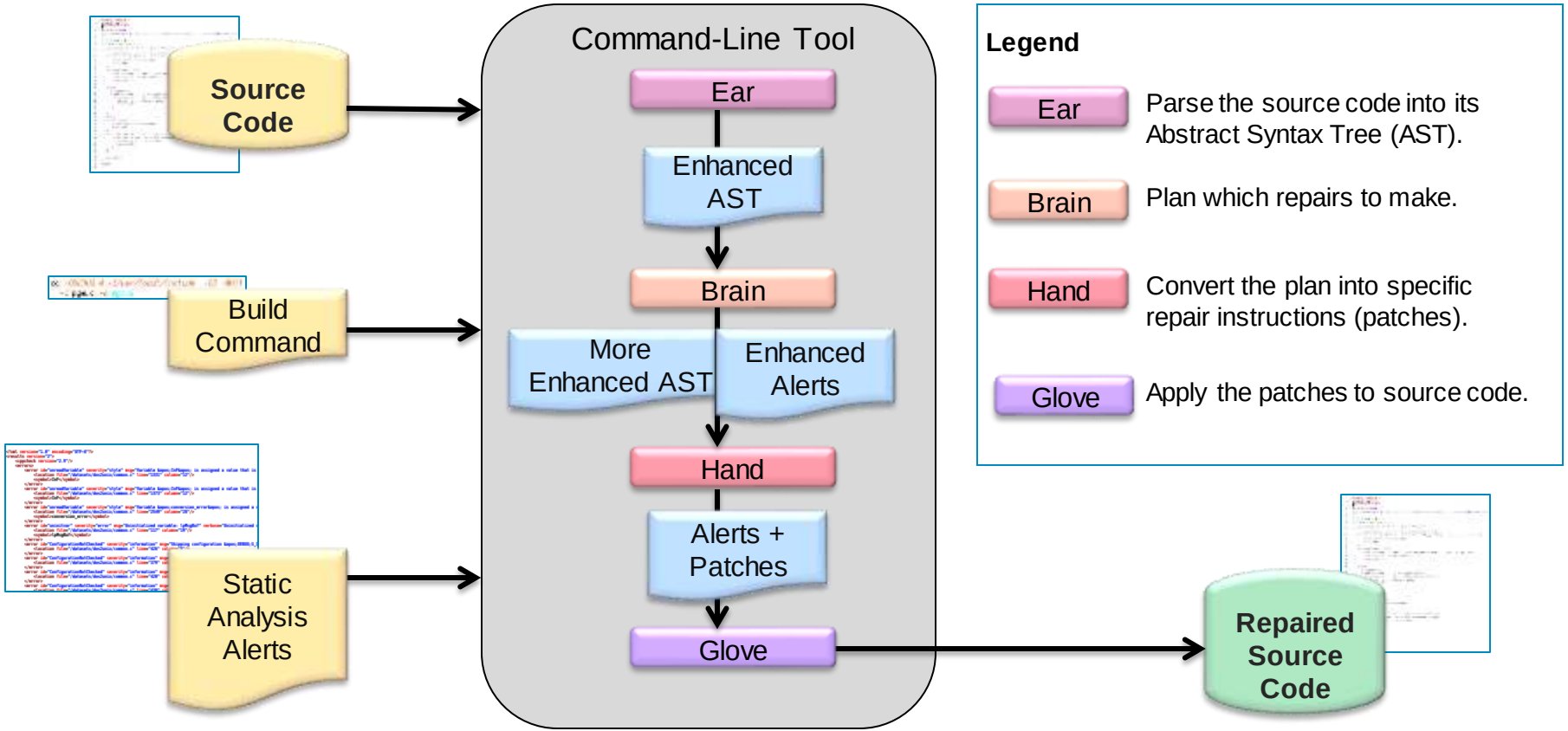
OR

- Explain in a text message why it cannot be repaired.

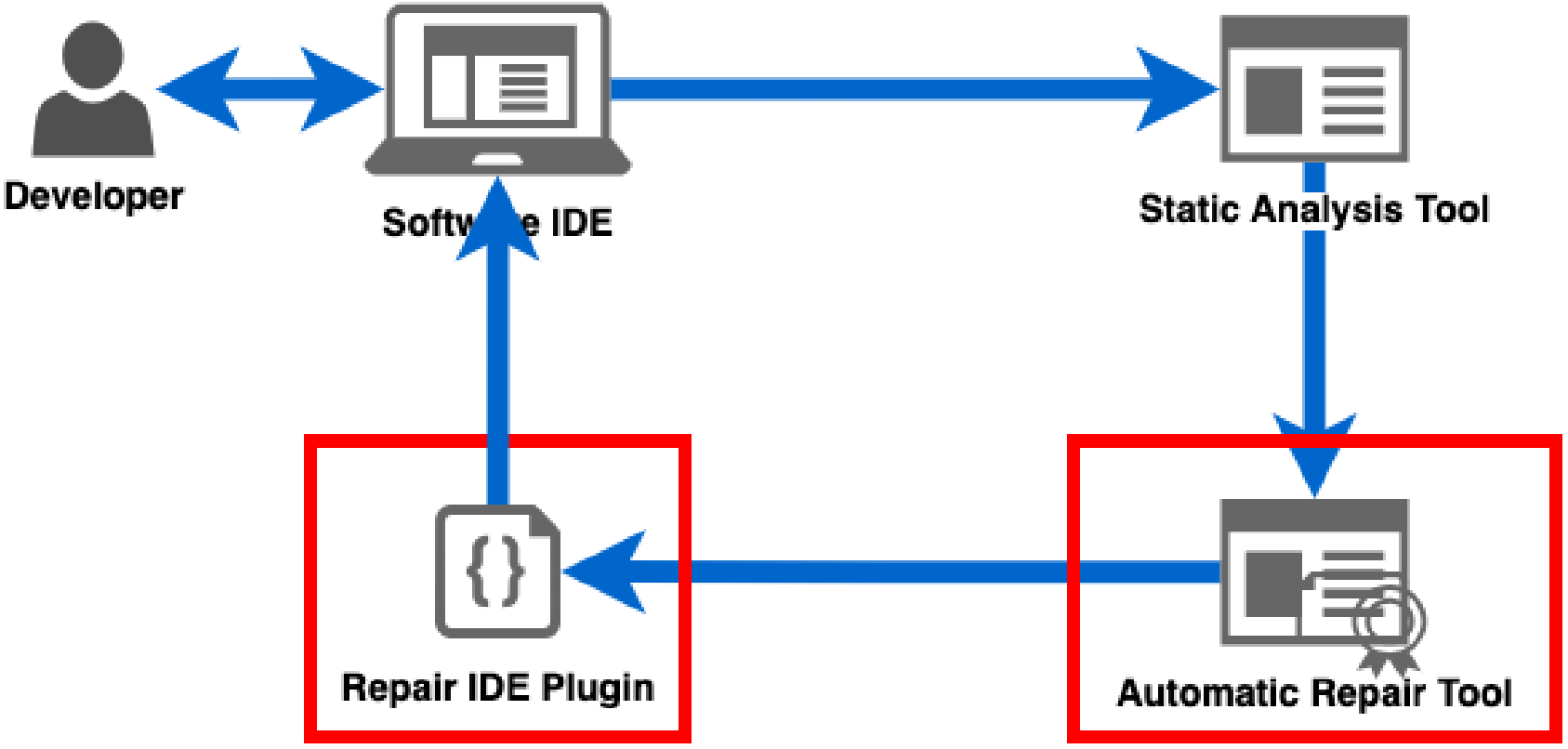
All patches should be independent (i.e., they repair distinct regions of code).

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

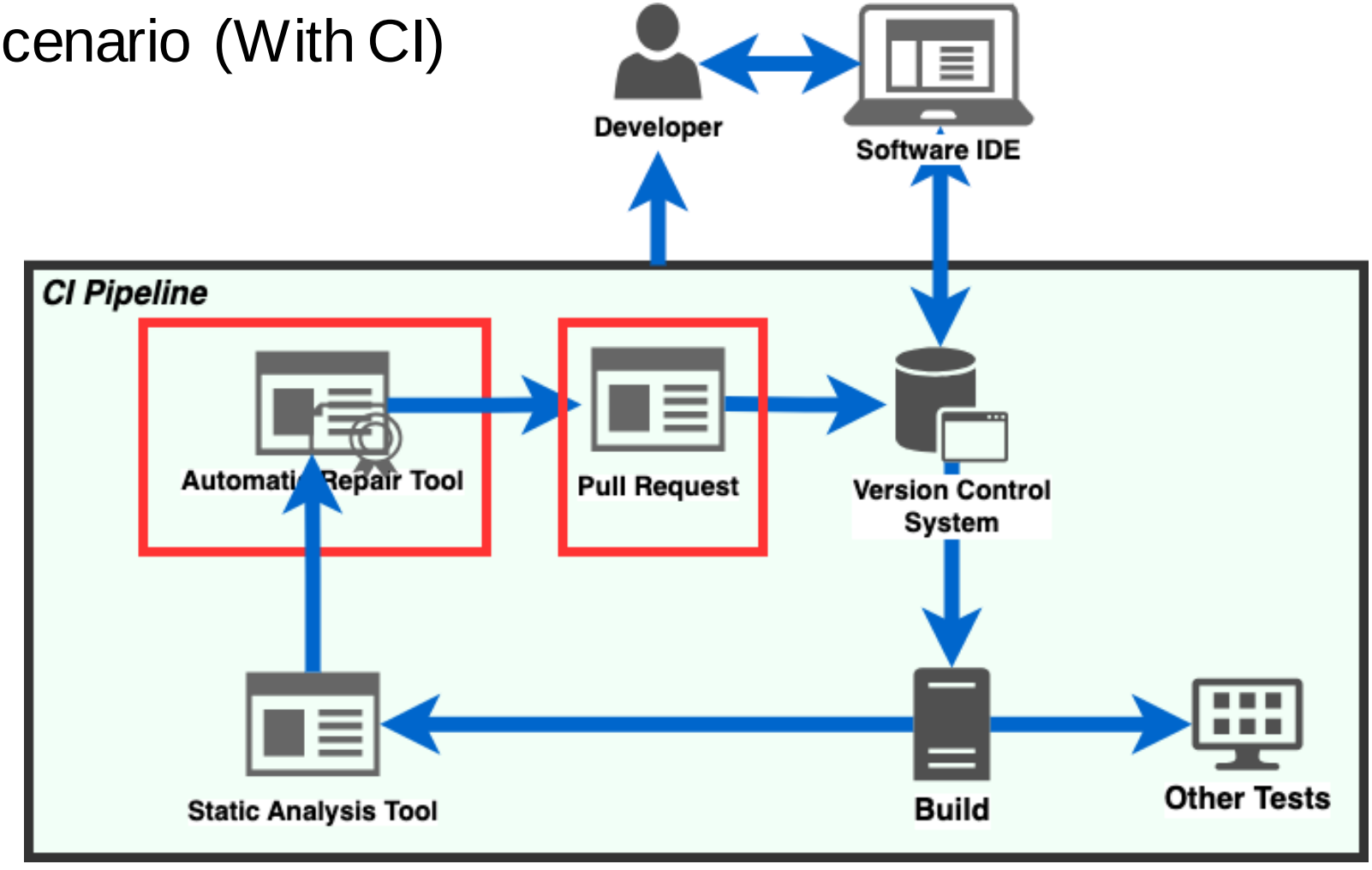
Command Line Tool (M2) –2



Usage Dataflow Scenario (Without CI)



Scenario (With CI)



Automated Repair of Static Analysis Alerts

Testing

Carnegie Mellon University
Software Engineering Institute

Undefined Behavior

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

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

Undefined Behavior

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

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

Platforms may define platform-specific behaviors.

Undefined Behavior

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

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

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.

Informal 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 should **NOT** change the behavior.

Limitation: Code that relies on UB cannot be reliably repaired.

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.

For FY23, 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.

Types of Tests

Carnegie Mellon University Software Engineering Institute

"Stumble-Through" Testing

Verifies that 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.

Alert Testing

Ensures repairs are correct

- For each tool/guideline/codebase,
  - Pick N random alerts; N=5 for now. For each alert,
    - Manually check if ACR did the right thing.
    - Repair correctly or refuse to repair.
- If ACR does not do the right thing on >=80% of alerts,
  - Fix ACR bugs and re-test.

With >15,000 alerts to repair, we cannot test all of them.

Integration Testing

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 test passes if all codebase-specific internal tests pass.

Recurrence Testing

Verifies that repaired alerts are not reported or re-repaired

- Run the repair tool on all codebase.
- Re-run SA tools on all codebases, and compare alerts generated with original alerts.
- The test passes if all repaired alerts are no longer reported by an SA tool.
- Re-run the ACR tool on the repaired codebase 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.

What Makes a Repair Satisfactory?

Carnegie Mellon University Software Engineering Institute

Git has the following code in dir.h:

```
535  static inline int ce_path_match(struct index_state *istate,
536                                  const struct cache_entry *ce,
537                                  const struct pathspec *pathspec,
538                                  char *seen)
539   {
540       return match_pathspec(istate, pathspec, ce->name, ce_namelen(ce), 0, seen,
541                             S_ISDIR(ce->ce_mode) || S_ISGITLINK(ce->ce_mode));
541   }
```

The `ce_namelen` macro is defined in `cache.h`:

```
240 #define ce_namelen(ce) ((ce)->ce_namelen)
```

What Makes a Repair Satisfactory?

Carnegie Mellon University Software Engineering Institute

Git has the following code in `dir.h`:

```
535  static inline int ce_path_match(struct index_state *istate,
536                                  const struct cache_entry *ce,
537                                  const struct pathspec *pathspec,
538                                  char *seen)
539   {
540       return match_pathspec(istate, pathspec, ce->name, ce_namelen(ce), 0, seen,
541                             S_ISDIR(ce->ce_mode) || S_ISGITLINK(ce->ce_mode));
541   }
```

A developer would insert a null check here.

clang-tidy complains of a potential null dereference on this variable (and nowhere else).

The `ce_namelen` macro is defined in `cache.h`:

```
240 #define ce_namelen(ce) ((ce)->ce_namelen)
```

We do not second-guess our SA tools.

What Makes a Repair Satisfactory?

Carnegie Mellon University Software Engineering Institute

Git has the following code in `dir.h`:

```
535  static inline int ce_path_match(struct index_state *istate,
536                                  const struct cache_entry *ce,
537                                  const struct pathspec *pathspec,
538                                  char *seen)
539    {
540        return match_pathspec(istate, pathspec, ce->name, ce_namelen(NULL_CHECK(ce,
abort())), 0, seen,
541                              S_ISDIR(ce->ce_mode) || S_ISGITLINK(ce->ce_mo    ;
541    }
```

A developer would insert a null check here.

clang-tidy complains of a potential null dereference on this variable (and nowhere else).

Repairing just this variable (and no others) is satisfactory.

The `ce_namelen` macro is defined in `cache.h`:

```
240 #define ce_namelen(ce) ((ce)->ce_namelen)
```

Automated Repair of Static Analysis Alerts

Lessons Learned

Carnegie Mellon University
Software Engineering Institute

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		
~~INT30-C~~	~~Ensure that unsigned integer operations do not wrap~~	~~190~~	~~13~~

Challenges with Collaborator Data

- Collaborators used Fortify, Checkmarx, and Cppcheck, all managed by CodeDX.
- Collaborators' data associates alerts with CWEs, not 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.

Clang Integration

Clang can serialize its AST as 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 fix this in Clang itself.

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.

Automated Repair of Static Analysis Alerts

Conclusion

Future Work

Carnegie Mellon University Software Engineering Institute

In FY24

- Transition the tool to new DoD collaborators.

Later

- Add more code repairs based on list of theoretical repairs from FY23.
- Add support for more SA tools.
- Add support for more IDEs.
- Integration of repair tool into CI pipeline enables repairs of more types of defects (e.g., indentation defects, coding-style deviations).
- Use repair tool to detect and resolve inconsistencies (e.g., in error handling)

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?

The Automated Repair Team

David Svoboda
Principle Investigator
Software Security Engineer

Will Klieber
Software Security Engineer

Lori Flynn
Senior Software Security Researcher

Ebonie McNeil
Technical Engagement Lead

Carmelita D Chichester
Project Manager

Joseph Sible
Associate Software Engineer

Ryan Karl
embedded Software Engineer

Robert Schiela
CSFTechnical Manager

Email: info@sei.cmu.edu

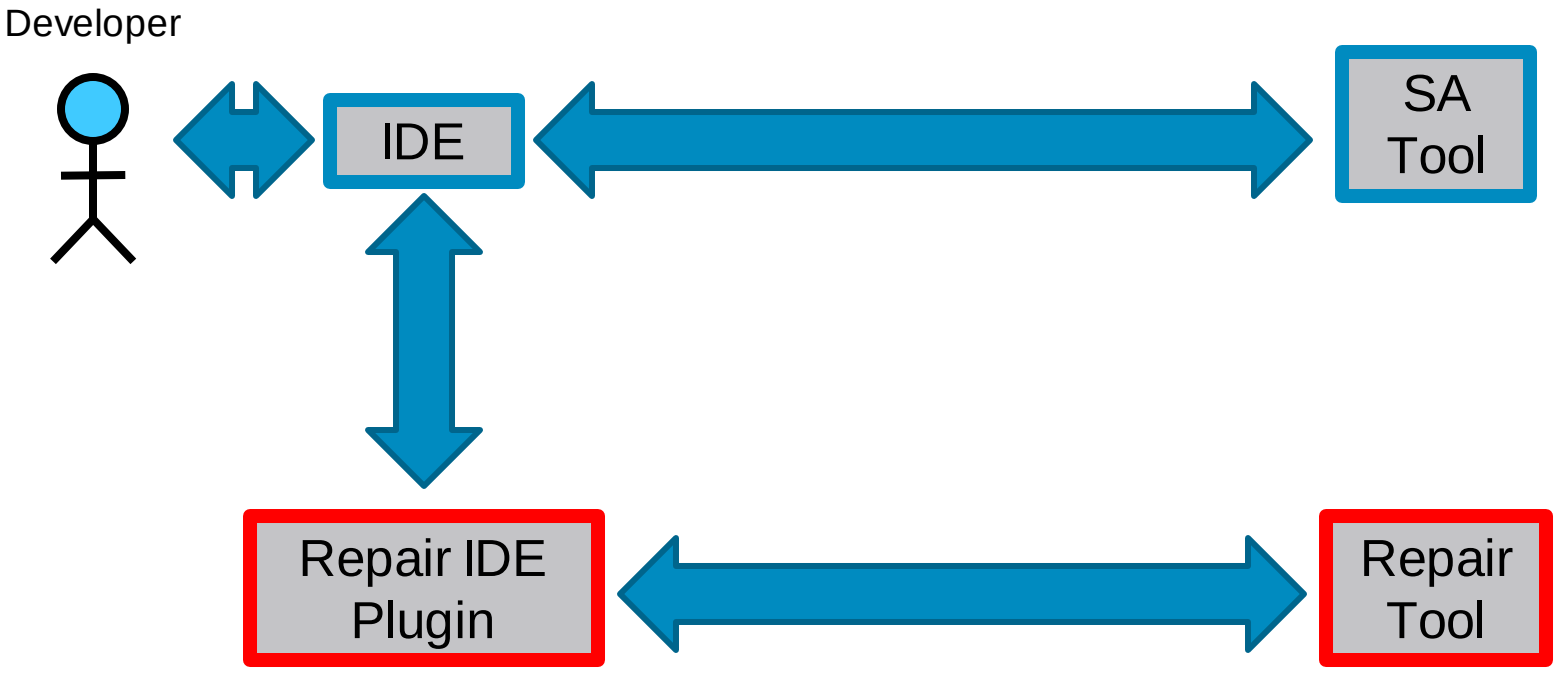
Carnegie Mellon University
Software Engineering Institute

Backup Slides

FY23 Line: Redemption of False Positives (HotCRP #13)

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

Carnegie Mellon University Software Engineering Institute

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)

Carnegie Mellon University Software Engineering Institute

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
- 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.

“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.

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, ER	6.2	✓	x				
M3.		6.2		x	x	x		
M8.		6.2		x	x	x		
M4. DoD transfer & test of command-line tool	M2 (A1)	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

A collaborator wants a repair of OSS Zeek v4, which is mostly C++ code. We did not plan to handle C++ code, but it should be possible. The collaborator sent us data running the following: Fortify, Checkmarx, and Cppcheck via CodeDX on Zeek v4.