Carnegie Mellon University
Software Engineering Institute

Automated Code Repair for C/C++ Static Analysis

David Svoboda
Lori Flynn
William Klieber
Michael Duggan
Nicholas Reimer
Joseph Sible

September 2025

TECHNICAL REPORT
CMU/SEI-2025-TR-007
DOI: 10.1184/R1/29905805

CERT Division

https://www.sei.cmu.edu

This material is based upon work funded and supported by the United States Department of Defense under Air Force Contract No. FA870225DB003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM25-1009

Table of Contents

List of Figures

List of Tables

Abstract

Static analysis (SA) tools produce many diagnostic alerts indicating that source code in C or C++ may be defective and potentially vulnerable to security exploits. Many of these alerts are false positives. Identifying the true positive alerts and repairing the defects in the associated code are huge efforts; automated program repair (APR) tools can help. Our experience showed us that APR can reduce the number of SA alerts significantly and reduce the manual effort required of analysts to review code.

This engineering experience report details the application of design, development, and performance testing to an APR tool we built that repairs C/C++ code. Its repairs are simple and local. Furthermore, our findings convinced the maintainers of the CERT Coding Standards to re-evaluate and update the metrics used to assess when violations of guidelines are detectable or repairable. We discuss engineering design choices made to support goals of trustworthiness and acceptability to developers.

1 Introduction

Static analysis (SA) is a standard method of inspecting code for flaws that could lead to vulnerabilities and incorrect behavior. Manual adjudication of SA alerts as true or false positive is costly, requiring expertise and time, and often there are too many alerts to inspect them all. Consequently, some alerts are prioritized for adjudication, leaving others uninspected, which results in unknown risks in production code.

In this report, we present our engineering experience developing an automated program repair (APR) tool called Redemption to help developers handle large sets of alerts in C or C++ code. The tool accomplishes this by repairing as many alerts as it can in three flaw categories: null pointer dereferences, uninitialized value reads, and code that has no effect. We chose these categories because they produced many alerts, are critical to security, and are well known. Section 4 details our selection process.

False positives are a major problem of SA tools, as no tool can correctly adjudicate all alerts with 100% accuracy. We must therefore assume that some of the alerts to be repaired might be false positives and that we might be unable to judge whether they are false positives. We therefore seek to repair the code anyway; that is, we modify the code to no longer report the alert without changing its current behavior. If our APR tool can reliably determine that the alert is a false positive, our tool can report the alert as false positive rather than repairing it. In general, developers prefer to identify and discard false positives rather than repair them.

Designing an APR tool that preserves good code behavior introduces many engineering considerations, which we discuss in Section 5. For example, a good repair conforms to the program's strategy of error handling, but determining this strategy can be tricky. Still, we were able to accurately repair 94% of the alerts in one codebase, and we showed that APR can eliminate a large percentage of SA alerts, reducing the manual effort of analysts when reviewing code. The success of our APR tool prompted an update of the metrics in the CERT Coding Standards, as detailed in Section 4.1.

This report is organized as follows. Section 2 describes the magnitude of the problem. Section 3 analyzes the requirements for a flaw category to qualify for APR. Section 4 analyzes the CERT C guidelines [CERT 2023a] based on the requirements in the previous section and identifies the three guidelines selected. Section 4.5 discusses obstacles and difficulties we encountered when building the tool, and Section 6 describes the results of testing our solution to determine if it fulfilled our requirements. Section 7 discusses related work with other APR strategies and solutions. Section 8 summarizes positive and negative results and lessons learned.

This report's contributions include the following: Our case study analyzing and targeting code flaws to automatically repair, specific code/algorithm examples of repairs, details of our engineering implementation, and discussion of issues to consider when creating a system to do automated code repair. We also published the code for our APR tool [SEI 2025] and a dataset [Svoboda 2023] to accompany this report so that our results can be validated.

2 The Problem

Auditing and repairing the average C/C++ codebase takes a significant amount of manual effort. In a NIST study, 66% of application security tool findings were determined to be irrelevant [Delaitre 2018]. Another study found that triaging each finding takes an average of 10 minutes [Anderson 2018]. According to CodeDx, "A tool returns an average of 10,000 results on an application. With a 66% irrelevancy rate, this is equal to 132 days spent reviewing false positives" [Synopsis 2023].

We also analyzed audit and repair effort. Our analysis is based on alert density—the ratio of the total number of alerts in a codebase to its size—which we measured in alerts per thousand lines of significant code (kSigLoC). Alerts may be true or false positives and also some tools may not alert about some code defects, so our alert density is a rough estimate. The alert density of the C/C++ codebases in our analysis is 85,268 alerts / 233.9 kSigLoC =364.5 alerts/kSigLoC.

Suppose that the average C/C++ codebase is 1,957 kSigLoC; this is the average size of a set of proprietary codebases that we analyzed. Further suppose that the average kSigLoC produces 364.5 SA alerts, as we found in our analysis. How long does it take to audit an alert and repair the code if the alert is a true positive? To answer this, we looked at a large study by Google, which found that it took an average of 117 seconds to audit one alert [Ayewah 2010]. Let us guess another 117 seconds is required to fix each alert. (Some alerts, like buffer overflows, usually take much more than 117 seconds to fix, but we focus on alerts that are simple to fix, such as null pointer dereferences. If a developer makes many simple fixes at once, then 117 seconds per alert becomes reasonable.) The developers in the study manually fixed 32% of the alerts [Ayewah 2010]. The average time for a person to deal with an alert (i.e., to audit it and fix it if necessary) was 117 sec + 32% ∗ 117 sec = 154.44 sec/alert. Using the above-mentioned alert density of 364.5 alerts per kSigLoC, we get an average per-person effort (for a kSigLoC) of 154.44 sec/alert * 364.5 alerts/kSigLoC = 56,293 sec/kSigLoC. Thus, implied effort for an average codebase of 1,957 kSigLoC is 1,957 kSigLoC ∗ 56,293 person·sec/kSigLoC = 3.5 person-years.

Most organizations do not have this much effort available to address SA alerts, and they consequently ignore most alerts, leaving many true positives unrepaired.

3 Repair Category Requirements

To qualify as automatically repairable, a code defect must have a repair solution that can be automatically implemented by a repair tool. The tool may use the abstract syntax tree (AST) and intermediate representation (IR) of the code in which the repair will be placed. What follows are several aspects of CERT coding guidelines and our analysis of their suitability to target and judge code flaw repairs by our APR tool. (Those aspects could also be derived for other coding standards or code flaw taxonomies, which would not change our analysis of suitability related to our APR tool.)

Trustworthy: Repairs should not "break" the code, whether the alert is true or false. This means APR patches should not change expected functionality; add security flaws; cause unit, performance, or integration tests to fail; or disable compilation.

For example, the typical repair for an alert about a memory leak is to add a call to `free()`. However, if the code is correct and the alert is a false positive, adding a call to `free()` is likely to cause a double-free and/or a use-after-free. Consequently, repairs for memory leaks can be "breaking" repairs. See Section 5.4 for more information.

Locality: Some vulnerabilities can be assigned to a specific line of code but still require changes in multiple places in the codebase. They can't be fixed by changing just the flagged line of code or just the function that contains this line. For example, buffer-overflow alerts are typically not locally repairable if the buffer was allocated in a different function than that of the flagged line of code because there is no way to determine the capacity of the buffer using only information available at the alerted line of code.

Developer Acceptance: Once a repair is applied, it has two liabilities. First, it may break the code. Second, the repair might preserve intended behavior but be rejected by developers in favor of a manual repair or even no repair to the code. Determining what changes developers will accept is complicated, and we do not fully address it here. We believe developers are less likely to accept repairs that change many lines of code than repairs that change one line or a few lines of code [Sadowski 2015, Mechtaev 2015].

For example, it is better to reliably determine that an alert is a false positive rather than to repair it, as making any change to source code may mean that developers' previously-correct mental model of the code differs from the changed code. As another example, a repair tool should not repair an alert that has already been repaired, as a double repair may not change how the code behaves but would be rejected by developers.

Error Handling: Some violations of secure-coding guidelines can be repaired but require an error-handling strategy. For example, a signed-integer overflow might be repaired by inserting a check for whether overflow would occur and handling the case where it would. Which error-handling strategy to use depends on the codebase's error-handling policy and may also depend on the function. As an engineering choice for our tool, we assume the existence of a "standard" error-

handling mechanism. Our error-handling strategy is expected to prevent execution from continuing to a line whose execution would cause a fault.

For example, a null pointer dereference can be repaired by inserting (before the dereference) comparison of the pointer with null and then handling the error. This repair is local. A null pointer dereference repair for a false-positive alert means that the code receives an extra null check on a pointer that can't be null. This introduces a minuscule slowdown on the code. Null-dereference repairs are non-breaking. For more information, see Section 5.2.

Static Analysis: Static analysis is usually best done on an intermediate representation such as LLVM IR. However, repair is usually easiest to do using the AST, since mapping AST-level changes back to the source-code text is easier than mapping IR-level changes. For the most part, our repair tool does not perform its own static analysis; instead, it relies on the output of external SA tools. While some SA tools put great effort into preventing false positives, not all SA tools do. A potential problem is that an SA tool might continue to flag a piece of code after our tool has applied a repair to it. This is undesirable because applying our tool again in the future will add redundant repairs to the code. Our repair tool does a small amount of its own static analysis to avoid repairing an alert that has previously been repaired. Also, if our tool detects that an alert is dependent [Svoboda 2016a] on another alert that is being repaired, it does not repair the dependent alert.

Automatic Detection: This turns out not to be required for repairable categories. A flaw category might be automatically repairable while not being automatically detectable. For example, null pointer dereferences can be automatically repaired, but accurate automatic detection of null pointer dereferences is a hard problem.

4 Repair Category Analysis

Our tool is designed to repair alerts that report flaws. Flaws can be partitioned by various coding standards, such as the CERT C Coding Standard [CERT 2023a] or MITRE CWEs [MITRE 2023a]. This section describes how we selected three CERT C guidelines to repair out of its more than 300 total guidelines.

4.1 Remediation Cost

Each CERT Coding Standard guideline includes several metrics about the cost of complying with the rule. Previously, it employed a "Remediation Cost" metric that indicated whether compliance with the rule was automatically detectable or whether an alert was automatically repairable. This metric presumed that a guideline that can be automatically repaired is not easy to fix unless it is also automatically detectable. As noted in Section 3, our approach to APR violates this assumption. The success of this project and tool prompted an adjustment to the metrics. Now, the "Detectable" metric addresses whether a static analysis tool can automatically determine if code violates the guideline with high precision and recall. The "Repairable" metric addresses whether an automated repair tool can reliably fix an alert by making local changes, plus whether the repair can be guaranteed not to break the code even if the alert is a false positive.

4.2 CERT Guideline Impact

The simplest metric to rate guidelines' impact is to count how many alerts map to each guideline in a set of SA alerts for a set of codebases. In a preliminary analysis, we used summary SA statistics from C/C++ codebases performed as part of SCALe audits [CERT 2014, Flynn 2018]. These codebases consisted of 234 kSigLoC. These codebases produced 85,268 alerts that allegedly violated 124 distinct CERT C or C++ guidelines [CERT 2023a]. Fortunately, 57,922 alerts (67.9%) violated just eight CERT guidelines, as shown in Table 1. If we automatically repaired 80% of these eight violations, that would fix 54.3% of the alerts. (When we speak of repairing an alert, we actually mean repairing the defect in a codebase that causes the alert to be reported.) The first rule alone (INT31-C) generates 21,264 alerts or 24.9% of the alerts! Fixing 80% of these resolves 20.0% of all alerts. A similar analysis on different codebases might produce different "top eight" CERT guidelines.

Table 1: CERT Guidelines

Guideline	Title	Priority	Alerts
INT31-C	Ensure that integer conversions do not result in lost or misinterpreted data	6	21,264
INT13-C	Use bitwise operators only on unsigned operands	6	11,007
EXP00-C	Use parentheses for precedence of operation	4	6,602
PRE03-C	Prefer typedefs to defines for encoding types	2	6,093
DLC00-C	Const-qualify immutable objects	1	4,863

Guideline	Title	Priority	Alerts
EXP05-CPP[1]	Do not use C-style casts	12	3,294
EXP12-C	Do not ignore values returned by functions	4	2,905

4.3 Flaw Frequency Background

Determining which code defects or alerts are most common requires a big dataset, and the choice of dataset heavily influences which alerts are most common. Many codebases are not public, nor are their SA results (whether or not they include manual adjudications of the alerts as true or false positive). Theoretically, the study of large datasets like GitHub code scans could be helpful. GitHub requires that users have write permission to view an alert summary for a repository [GitHub 2025].

One study involving the Coverity SA tool's alerts for four highly used, open-source codebases (each over 100,000 lines of code) found only 67 of the possible 193 alert types were produced, and 80% of the alerts came from 20% of the alert types. This study published the top 10 types and average time to manual code fixes for actionable alerts per type (24−660 days) [Imtiaz 2019]. The top-10 list could be used to target development of APR for code flaws that could cause those alerts, and APR time-savings could be estimated using their fix-time data. Our selected category CWE-476 "NULL pointer dereference" relates to items on their list: "4. Explicit null dereferenced," "5. Dereference after null check," "6. Dereference before null check," and "8. Dereference null return value." Our selected repair category CWE-561 "Dead Code" relates to their item "3. Logically dead code." Our selected category CWE-908 "Use of Uninitialized Resource" relates to (but includes more than) their item "9. Uninitialized scalar variable." A large-scale analysis of C/C++ vulnerabilities in the CVE database and associated open-source code from 2006 to 2023 listed the top 10 CWEs associated with the CVEs [Ni 2024]. Our selected category CWE-476 is fourth on their list, but CWE-908 and CWE-561 are not in their list.

Some SA tool vendors publish aggregated results of scans with their tools that include flaw frequency per language. For example, Veracode reports are currently based on assessments of over 14 trillion lines of code, though the flaw types listed are not as granular as CWEs [Veracode 2020].

A codebase or development team may have an unusual set of most frequent code defects, due to their coders' strengths and weaknesses, tools, and the codebase. Analysis of their alerts could help target code repairs to develop for them.

Alert counts or true-positive alert counts are not the only factors to consider when prioritizing effort to develop automated repairs. Other important factors are the estimated impact of the code defect if not fixed, amount of human effort required to make the automated fix (e.g., if a human needs to provide input to guide the particular automated repair used), and impact of a repair on code comprehension by a developer that affects code maintenance.

[1] EXP05-CPP has been withdrawn, but it appears in older SCALe audits.

4.4 Frequency Analysis

Based on the preliminary analysis referenced by Section 4.2, we conducted a frequency analysis of SA alerts produced using open-source SA tools on several open-source codebases. The SA tools we used are clang-tidy version 15.0.7 [Clang-tidy 2023], Cppcheck version 2.9 [Cppcheck 2023], and CERT Rosecheckers [CERT 2023b]. The codebases we analyzed are Zeek version 5.1.1 [Zeek 2023] and Git version 2.39.0 [Git 2023]. We also analyzed dos2unix version 7.4.3 [Dos2Unix 2023] but did not include its results here. Our dos2unix data is in our related dataset publication [Svoboda 2023]. Dos2unix is a small codebase requiring the compilation of just three C files. It is too small to produce useful results for publication, but its size works well for development testing.

To measure the frequency of each CERT guideline, we attempted to map each alert to a single guideline. Some alerts had no suitable CERT guideline to map to. A small sample of our results is shown in Table 2. Each row lists a CERT guideline, and the "Alerts" columns indicate how many alerts that map to the guideline were reported by the corresponding tool on the corresponding codebase. For each tool and codebase combination, there is a "Rank" column; this is used to indicate the most prolific guidelines with flaws reported by the tool on the codebase. The CERT rules also list their priority, plus an assessment of whether the rule is automatically repairable or not. The table concludes with a "Total" row, which contains the sum of all alerts that map to a CERT guideline, and a "Guidelines" row, which contains the number of distinct CERT guidelines for which there was at least one alert.

So that others can validate our analysis, we published data we produced and details about the open-source tools and code we used [Svoboda 2023]. (Note: This dataset may differ slightly from the numbers reported in this report due to slightly different versions of the tools used.) The dataset includes instructions for running the SA tools, a Dockerfile to conveniently obtain the SA tools, raw SA tool output, parsed SA data and aggregate analyses, and SA data augmented with CERT guideline and CWE data.

Table 2: Excerpt of Alert Counts and Ranking for Tools, Codebases, and Guidelines

			Git						Zeek					
			Cppcheck		clang-tidy		Rosecheckers		Cppcheck		clang-tidy		Rosecheckers	
Guideline	Priority	Repairable	Rank	Alerts	Rank	Alerts	Rank	Alerts	Rank	Alerts	Rank	Alerts	Rank	Alerts
MSC13-C	2	yes	1	228	9	458			1	1,179		16		
DCL19-C	2	no	2	63					2	434				
DCL01-C	2	yes	3	42			4	2,465	4	89			1	2,553
MSC12-C	2	yes	4	25			10	721	3	131			8	480
Total				420		48,233		27,162		2,466		18,806		10,885
Guidelines				13		29		49		24		41		45

Note: "Total" is the sum of alerts for the tool and codebase, and "Guidelines" is the number of CERT guidelines flagged by at least one alert
A blank cell in an "Alerts" column indicates that there were 0 alerts.

.

4.5 Analysis Summary and Decisions

In deciding which guidelines to repair, we considered alert frequency, CERT Prioritization metrics, and guidelines that have analogous CWEs in the CWE Top 25 [MITRE 2023b]. We used these to rank the CERT guidelines from most to least worthy of our efforts, then selected the following top three for our work:

1) EXP34-C (CWE-476) Null-Pointer Dereferences:

- 5th most alerts generated by Cppcheck on Git
- 6th most alerts generated by Cppcheck on Zeek
- 11th in the 2022 CWE Top 25 Vulnerabilities
- Priority 18 (the highest) in the CERT C Coding Standard

2) EXP33-C (CWE-457) Read of Uninitialized Value:

- 8th most alerts generated by Cppcheck on Git
- 2nd most alerts generated by clang-tidy on Git
- 1st most alerts generated by clang-tidy on Zeek
- Priority 12 (2nd highest) in the CERT C Coding Standard

3) MSC12-C (CWE-561, CWE-1164) Remove Code that Has No Effect:

- 4th most alerts generated by Cppcheck on Git
- 10th most alerts generated by Rosecheckers on Git
- 8th most alerts generated by Rosecheckers on Zeek

5 Engineering Challenges

This section covers some of the issues we encountered while building our APR tool and our solutions for addressing them.

5.1 Implemented Repairs

In this section, we describe the repair algorithms for the categories we selected for repair.

For a null pointer dereference, we use a `null_check()` macro to protect any pointer that might be null. We replace any expression *x* that might represent a null pointer with the value `null_check()`. This macro is defined in a file `acr.h`, and it effectively invokes error-handling code if *x* is null. Here *x* can be an expression; it need not be a variable. By default, the error-handling code aborts the program via `abort()`, but it can be overridden it to have more specific behaviors (e.g., return `NULL` if inside a function that returns `NULL` on an error).

For an uninitialized value read, we simply go to the variable's declaration and set it to 0 (or the equivalent of 0 for the variable's type; e.g., it would be 0.0 for floats).

For ineffective code (i.e., code that has no effect), we remove the ineffective portion of the statement or expression. One example of ineffective code is an evaluation of some expression, with the result assigned to a variable that is never read (e.g., `x = foo(...);`). Our repair for that preserves the expression and removes the assignment (e.g., `(void)foo(...);`). Casting a function's return value to `(void)` is a common indication that the function's return value is to be ignored. See Exception 1 of CERT rule EXP12-C for more information [CERT 2025]. This code is ineffective only if the assignment can be proved not to invoke a C++ constructor, destructor, overloaded assignment operator, or overloaded conversion or cast operators, which might have side effects. Also, Ineffective Code (MSC12-C) is a CERT recommendation (sometimes correct to do), not a rule (always correct to do), and this presented additional complications. Therefore, we added a `REPAIR_MSC12` environment variable, so the MSC12-C repair is only done if the user explicitly specifies to include it.

5.2 Error Handling

As previously noted, many automated repairs require the existence of an error-handling strategy that the program can employ if it detects that it can prevent an error, such as dereferencing a null pointer. The C language has many techniques for handling errors, but deciding which error-handling strategy is best to employ can be tricky [Svoboda 2016b]. CERT guideline ERR00-C recommends determining an error-handling strategy, and many coding projects do so. If a coding project has an error-handling convention, ideally our repair tool will recognize and use it. There are several common conventions:

- Terminate the program (i.e., call `abort()` or `exit()`).
- Return an "invalid" value, such as `NULL` or `EOF`. Or return prematurely, for void functions.
- Transfer control elsewhere (e.g., use `goto` or `longjmp()` or raise a signal).

The error-handling strategy need not terminate the program, but it must not allow execution to resume normally. While our repair tool can employ some intelligence to determine what error-handling strategy to use, it should also allow users to specify the error-handling strategy.

We conducted an informal analysis on dos2unix [Dos2Unix 2023], Git [Git 2023], and Zeek [Zeek 2023] to identify their error-handling strategies. Dos2unix doesn't seem to have one. Git has an `error()` function that takes arguments in the style of `printf()`, reports the error, and returns. Git also has a `die()` function that calls `exit()` (but can probably be overridden). Zeek has a `sec` enum type that indicates various errors, often used as return values (`sec::none == 0`), plus several enum types that indicate various non-error states (e.g., `connection_state`).

We examined the null-pointer (EXP34-C) alerts in dos2unix, Git, and Zeek. Based on this analysis, we settled on the following heuristic for error handling:

- If a function returns an `int` (or any integer type that is not an enum), then look for all return statements. Any return statement with a value distinct from the final return statement (at the end of a function) indicates a suitable value to return upon error.
- A function that returns an enumeration class is harder, unless the enum class provides an obvious error value that we can use. Our tool does not currently handle that.
- Likewise, if a function returns a pointer type, then look for all return statements. If it returns null anywhere in the function, but a potentially non-null value at the end, then assume that the function can return null upon error.
- If the above fails, default to `abort()`.

More intelligent error-handling code is currently beyond our ability to detect.

Our tool allows the user to provide custom error-handling code, such as a `die()` function.

5.3 Conditional Compilation Directives

C's preprocessor may be the biggest reason why automated repair of C/C++ remains difficult [Medeiros 2017]. The nature of the preprocessor allows the compiler to produce different code depending on which configuration of macro definitions is known to the compiler. Therefore, a C source file represents not one program but a family of one or more programs, each of which differs by a unique configuration of macro definitions provided to the compiler.

We decided that our repair tool (like many SA tools and all compilers) need only operate on one macro configuration at a time, and that macro configuration must produce a working C program. Our tool will also strive to ensure that repairs made to one macro configuration will not break other macro configurations. To repair multiple configurations, you must be able to build each such configuration.

One concern with conditional preprocessor directives (e.g., `#if` / `#ifdef` / `#ifndef`) is that not all macro configurations are necessarily compilable. While it is possible to write conditional preprocessor directives such that all configurations compile, this is often not done. If a program fails to compile when the macro `FOO` is not defined, then the question of how a repair might affect the `-UFOO` configuration is itself ill-defined. Input to a repair tool should include at least one

macro configuration that is known to compile properly. If a user wants a program to maintain correctness in macro configurations other than the default one specified, the user should indicate which configurations the repair tool must support and preserve.

Most repairs to code will be simple additions of repair macros. For the following analysis, we assume that the repair takes an expression, such as `a+b`, and replaces it with a macro, such as `acr_safe_add(a, b, goto handle_error)`.

There are several ways that conditional preprocessor directives can interact with this type of repair:

1. **Independent**: If the expression to be repaired contains no conditional preprocessor directives, we call it independent. (The expression may still be contained between matching conditional preprocessor directives.) In this case, the expression is okay to replace with a repaired version, as shown in Figure 1:

```
  x =
#ifdef WINDOWS
  a+b; /* expression to
be repaired */ #else /*
LINUX */
  c;
#endif
```

```
  x =
#ifdef WINDOWS
  acr_safe_add(a, b,
goto handle_error);
#else /* LINUX */
  c;
#endif
```

Figure 1: Independent Original (left) and Repaired (right)

2. **Embedded**: For a repair that replaces `a + b` with `acr_safe_add(a, b, goto handle_error)`, the embedded case is when the subexpressions *a* or *b* contain an `#if` conditional preprocessor directive (or a variation such as `#ifdef`) and all matching `#else`, `#elif`, and `#endif` conditional preprocessor directives. Often the embedded case allows the repair to be done safely, as shown in Figure 2:

```
  x =
#ifdef WINDOWS
  a
#else /* LINUX */
  c;
#endif
  +b;
```

```
  x = acr_safe_add(
#ifdef WINDOWS
  a
#else /* LINUX */
  c;
#endif
, b, goto handle_error);
```

Figure 2: Embedded Original (left) and Repaired (right)

However, there are pathological instances of the embedded case, such as in Figure 3 where `FOO` is false. Here, repairing the addition if `FOO` is false will break the code if `FOO` is true. Likewise, repairing the addition if `FOO` is true breaks the code if `FOO` is false. This is because if `FOO` is true, the conditional contains two statements, but if `FOO` is false, the conditional contains only one.

3. **Mixed**: In the Mixed case, the expression to be repaired contains conditional preprocessor directives in a way that doesn't match the Embedded case. This case is not as easy to repair as the Independent or Embedded cases. In some cases, it is feasible to refactor the code so that it matches the independent or embedded case, as shown in Figure 4. In general, however, the mixed case is not readily repairable.

   Conditional Compilation Directives Outcomes: We developed a function that takes an expression (specified by its byte range in the source code) and returns `True` if it is Independent (i.e., it contains no conditional preprocessor directives). If an expression is not Independent, our tool declines to repair it.

```
#include <stdio.h>
void main() {
  int x=0, y=0, z=0;
  int a=1, b=2, c=3;
  int* e = &c;
  y = (
#if FOO
  32);
  z = (a
#else
  * e
#endif
  ) + c;
printf("y = %d, z = %d\n", y, z);
}
```

Figure 3: Pathological Example of Conditional Directives

```
x =
#ifdef WINDOWS
  a +
#else /* LINUX */
  a *
#endif
  b;
```

```
x =
#ifdef WINDOWS
  a + b
#else /* LINUX */
  a * b
#endif
  ;
```

Figure 4: Mixed Original (left) and Refactored (right)

5.4 Trustworthy Repairs

Making repairs that do not break code can be challenging. For example, we designed repairs to avoid splitting any expression into multiple statements. This avoids complications that arise when modifying the text of the source code at the character level, rather than at the AST level.

For example, consider the snippet from coreutils [GnuLib 2023] shown in Figure 5 (from `regexec.c`, lines 3549–3554). Repairing the `bitset_set_all (accepts)` line in a way that splits it into two statements would require insertion of curly brackets. For the repair to work with both possible values of `RE_ENABLE_I18N`, the opening curly brace would appear after the `#endif`, not on the same line as the `else` statement.

```
#ifdef RE_ENABLE_I18N
  if (dfa->mb_cur_max > 1)
    bitset_merge (accepts, dfa->sb_char);
  else
#endif
    bitset_set_all (accepts);
```

Figure 5: Excerpt from Coreutils

As another example, consider the snippet in Figure 6. Splitting any of the expressions on the `for` line into multiple statements would require relocating them elsewhere (either inside the body of the `for` loop or outside the loop).

```
for (int x = foo(); x < bar(); x++) {
  baz();
}
int x = 42;
```

Figure 6: A Snippet Involving a `for` Statement

Splitting these examples is not impossible but does introduce extra complexity that can be avoided by introducing auxiliary functions so that the expressions can be replaced with other expressions instead of splitting into multiple statements.

Inserting null checks on expressions is more complicated than it seems. Consider the line of code shown in Figure 7. Cppcheck warns that `parent_names` might be null when incremented and dereferenced. However, in this code, `parent_names` is used as an lvalue [Meneide 2023]; its value is incremented. Adding a null check to `parent_names` must preserve the lvalue, lest the resulting code fail to compile. We solve this problem by providing two `null_check` macros, one for rvalues (i.e., expressions that are not used as lvalues) and one for addressable lvalues (i.e., expressions that represent an assignable memory address). We were unable to find a single macro that solves both cases. We have also not solved the problem for non-addressable lvalues; such a case did not appear in any of the code we tested.

```
trace2/tr2_tgt_normal.c:175:49:
  while ((parent_name = *parent_names++)) {
```

Figure 7: Code from Git's `trace2/tr2_tgt_normal.c`

5.5 Applicability

We integrated our repair tool into a private Gitlab CI pipeline. Our tool's repairs can also be viewed in Visual Studio Code [Microsoft 2023a]. Since each repair modifies only one line of

code, they can be easily compared against unrepaired code using a utility such as diff(1); consequently, many tools that can inspect and modify diffs can be used on repaired code. This applies to both Gitlab and VS Code, which provide interfaces for inspecting changes and accepting or rejecting them.

The combination of features in our APR tool may be useful for particular environments. Its relative simplicity combined with its use of Clang AST and LLVM IR might appeal to some APR developers. It works with the output of three SA tools and can easily be extended to others. It can be used on-premises without requiring expensive high-performance systems, as opposed to some APR tools that require cloud processing or highly performant LLMs or deep neural networks. Although the industry standard for APR use involves patches as suggestions to developers rather than batch processing [Eladawy 2024], for some purposes batch processing might be desirable. It can be used on C codebases that don't have test suites yet. Batch repair on a codebase lacking a test suite involves risk if done by our tool, which lacks formal verification of algorithm or software, though other tests of the patched code (compilability, fuzzing, etc. per Section 7) or manual review could provide some degree of confidence in the repairs.

6 Case Study

We implemented repairs for the three flaws and tested them on Git and Zeek. We conducted several tests to make sure our repairs comply with our constraints.

6.1 Sample Alert Test

In this test, we ran our SA tools on our codebases and collected alerts reporting violations of the three CERT guidelines we provided repairs for. Table 3 shows statistics for each combination of tool, codebase, and guideline. Some cells are empty because the SA tool produced no alerts.

The first row indicates the total count of SA alerts. We provide additional analysis for the Git alerts, rerunning the SA tools on the repaired version of Git, and counting how many alerts remained unrepaired. The first row for each cell contains the number of repaired alerts, total number of alerts, and percentage of repaired alerts to total alerts.

The second row contains sampling data. Since manually validating all of these repairs was impractical given project constraints, we randomly sampled five alerts from each cell (five alerts for each guideline, tool, and codebase), so the results may not achieve statistical significance. We evaluated whether each alert was a true or false positive and whether our tool should repair it. Our goal was to improve the tool's reliability at repairing alerts until each cell showed an 80% success rate.

The tool achieved a 100% success rate for all EXP33-C and EXP34-C alerts. It correctly repaired or recognized as false all of the null-dereference or uninitialized-variable alerts.

For MSC12-C (code that has no effect), our success rate was lower. MSC12-C alerts were flagged because

- a local variable was initialized or assigned but never subsequently read
- a label was never accessed via `goto` (This is often code generated by tools like yacc(1).)
- an unsigned expression was checked for being less than 0, which is impossible by definition

For these reasons, we did not select five MSC12-C alerts at random but strove to maximize diversity. We picked five categories of alerts randomly, and then one alert from each category, as a stratified random sample.

Although all these alerts could be repaired, the repairs would not necessarily improve the code. MSC12-C is a recommendation, not a rule in the CERT coding standards because it is not always a good idea to make the changes suggested by MSC12-C. Consequently, achieving an 80% satisfaction ratio for MSC12-C became a low priority, and these repairs are disabled by default. The failure of the MSC12-C tests to reach the 80% repair rate contrasts with the success of the other two repair categories; their successes were not a foregone conclusion. Finally, the MSC12-C repairs are deterministic and can make repairs faster than a human; hence, they are still useful when enabled.

Table 3: Alert Test Results

Guideline	Git clang-tidy	Git Cppcheck	Git Rosecheckers	Zeek clang-tidy	Zeek Cppcheck	Zeek Rosecheckers
EXP33-C	8,654 / 9,157 (94.5%) 5 / 5 (100%)	1 / 1 (100%) 1 / 1 (100%)		5,225 5 / 5 (100%)	24 5 / 5 (100%)	
EXP34-C	72 / 77 (93.5%) 5 / 5 (100%)	11 / 20 (55.5%) 5 / 5 (100%)		44 5 / 5 (100%)	52 5 / 5 (100%)	14 5 / 5 (100%)
MSC12-C		18 / 25 (72%) 1 / 5 (20.0%)			131 2 / 5 (40.0%)	480

Note: Each cell has two rows. The first row is the total number of alerts generated; for Git, this cell also contains the number of repaired alerts, the total number of alerts, and the percentage of alerts that were repaired. The second row addresses a sample of five alerts, indicating the number of sample alerts repaired, total number of alerts in the sample, and percentage of alerts repaired satisfactorily.

6.2 Integration Test

In this test, we built the unrepaired codebase and ran its own testing mechanisms. We then repaired the alerts generated for that codebase. We then rebuilt the codebase and ran it through its own tests. If test results for original and repaired code are identical, our experiment is successful. Both Git and Zeek have extensive tests for detecting bugs or regressions. Our repairs had no effect on the output of these tests; therefore, our integration test was successful.

6.3 Performance Test

This test confirmed that the repairs imposed on code did not significantly impede performance. In this test, we built and tested the unrepaired codebase and measured the time to completion. We compared this time against the time to build and test the repaired codebase. Ideally the repaired code would be as fast as the unrepaired code.

Table 4 shows the timing data from executions of the test suites for Git and Zeek. Each test suite is invoked by the "make test" command. For each test suite, and for each repair state (repaired vs. unrepaired), we ran two timing tests.

Table 4: Timing Tests of Test Suites for Git and Zeek

	User (minutes.seconds)	**System (minutes.seconds)**	**Elapsed (hours:minutes.seconds)**
Unrepaired Git	337.99	308.69	11:24.65
	341.06	318.04	11:38.17
Repaired Git	339.08	313.82	11:31.41
	339.54	311.17	11:28.83
Unrepaired Zeek	3.90	4.63	0:28.43
	3.50	4.71	0:28.45
Repaired Zeek	3.39	4.71	0:28.54
	3.77	4.77	0:28.49

The average difference in time to run the tests on these repaired and unrepaired codebases was less than the difference between identical runs of these codebases. Performance time was not significantly affected by our code repair. Two of Zeek's tests timed out after 60 seconds for both the original and repaired versions: `python-zeek` and `python-zeek-unsafe-types`.

6.4 Recurrence Test

This test confirms that the code was successfully repaired, according to the SA tools. In this test, we repaired the codebases and then ran our SA tools to generate alerts. We compared the alerts from the repaired code to the alerts from the unrepaired code. Ideally, both sets of alerts would be identical except that repaired alerts would be present in the unrepaired set but absent from the repaired set. We performed two recurrence tests with Git and Zeek, one with alerts produced by Clang-tidy version 16 and one with alerts produced by Cppcheck 2.9. Table 5 summarizes the

results. With Clang-tidy, for Git our tool repaired 8718 alerts, and Clang-tidy reported 8718 fewer alerts on the repaired code. For our tool repaired 32 alerts, but 39 fewer alerts were reported on the repaired code. So seven alerts disappeared without being explicitly repaired. In Zeek, the number of alerts for our two successful repair categories dropped from 76 to 34. There were problems with repairing and analyzing Zeek with Cppcheck, as detailed in the appendix.

Table 5: Alert Counts in Recurrence Tests for Git and Zeek

Guidelines Alerts Map to	Clang-tidy Original	Clang-tidy Repaired	Cppcheck Original	Cppcheck Repaired
Git				
EXP33-C	9157	500	1	0
EXP34-C	77	16	20	7
MSC12-C	0	0	25	7
All Our 3	9234	516	46	14
All Guidelines	48,233	40,840	420	381
Zeek				
EXP33-C	5225	173	24 (22)	13 (3)
EXP34-C	44	14	52 (29)	21 (7)
MSC12-C	0	0		
All Our 3	5269	187	76 (51)	34 (10)
All Guidelines	18,806	15,563	2466 (992)	3071 (899)

(Note: The parenthesized numbers indicate results without the two sqlite3.c files in Zeek. See the appendix for details.)

7 Related Work

The current state of the art and practice in repairing C/C++ code includes much ongoing research, and tools have made significant advances for more than two decades [Le Goues 2012a, Weimer 2010, Aberg 2003, Sidiroglou 2005, Necula 2002]. Automated program repair methods commonly use program test failures, SA alerts, and/or bug reports to identify potential defects, localize the code responsible for the defect, produce patches, and run tests to validate correct patching [Eladawy 2024]. This section discusses some dimensions of APR methods with tool examples (not comprehensive) and comparison to our work.

Continuous integration (CI): Repairnator is an opensource modular software repair system for incorporating a wide variety of repair tools into CI systems. The pipeline tries to replicate the bug in a failing CI build and repair it with different APR tools. It can be configured to create pull requests for repairs that pass build tests [Urli 2018]. It could use our APR tool.

AI: Many APR tools use artificial intelligence (AI) to predict code errors [Zhang 2023]. GitHub's Copilot [Git 2025] AI tool suggests code, including repairs, and is integrated with widely used integrated development environments (IDEs) and coding tools for many languages, including C/C++. DeepFix is an APR that fixes C language errors by deep learning trained to predict incorrect program locations along with the fixed code [Gupta 2017]. Our tool doesn't use AI.

Template-based repairs: AVATAR uses fixed patterns of true positives from static analysis to generate patch candidates to fix semantic bugs [Liu 2019]. Template-based repairs produce template-based fixes for matching code constructs and defect categories [Eladawy 2024]. Our tool makes template-based repairs.

IDE integration: Widely used IDEs such as Visual Studio Code [Microsoft 2023a] and Eclipse [Eclipse 2023] do some automated repairs for C/C++. The Microsoft Visual Studio IDE provides "Quick Actions" that enable applying a code repair for some code flaws, including for SA alerts from external tools [Microsoft 2023b, 2023c]. In future work, the alert fixes we discuss in this report could be integrated with that capability. For example, the Snyk plugin [Snyk Security 2025a] provides quick-fix repairs for 15 CWEs [Snyk Security 2025b]. IntRepair detects and analyzes C/C++ flaws and repairs various CWEs as an open-source plugin to the Eclipse IDE [Muntean 2019]. Its suggested repairs are viewable and can be accepted or rejected in the IDE with a mouse click or done as batch repairs [IntRepair 2023].

Simplicity as repair goal: Work by Klieber and colleagues provided automatic rewriting of C code to address memory safety problems by changing ordinary pointers to "fat pointers" that contain the bounds of allocated memory in addition to the raw pointer. Some of these repairs require wide-ranging changes to the codebase, such as when the repair changes the type of an element of a `struct` or changes the type of an argument to a function with many call sites. In contrast, our work targets making small local changes, with the goal of fixing many alerts of a few common types of defects. Other tools have similar goals of small repairs; for example, the

DirectFix APR tool attempts to find the simplest repairs, using partial MaxSAT constraint solving and component-based program synthesis [Mechtaev 2015].

Speed and simplicity: Unlike some APR tools such as IntRepair, our tool does not use a potentially time-consuming SMT solver to check flaws, instead relying on basic static analysis and compiler tools. Our focus also differs from those auto-repairs to target smaller, simpler-to-fix problems.

Preprocessor directives: As noted in the work of Medeiros 2017 and colleagues, many challenges for the current state of automated repair of C/C++ code can be attributed to the C preprocessor, which greatly complicates automated analysis and clean rewrites. That paper advocates a more disciplined approach to using the preprocessor's conditional directives, with preprocessor-oriented rewrites (as opposed to general code repair). Research on rewriting C code recognizes that high-level C or Fortran idioms can be inlined into function calls [Couto 2022]. The development version of the ROSE compiler framework supports modifying source code while preserving macros [Schordan 2003]. Software product lines (SPLs) are program families with code mutations (e.g., an `#ifdef` directive). SPLAllRepair is an SPL repair framework that uses variability encoding, bounded model checking, and SAT and SMT solvers on top of the AllRepair APR tool, experimentally resulting in correct C program repairs that were repaired more quickly (linear vs. exponential growth) than separately running AllRepair on each configuration [Dimovski 2024]. Automated refactoring of preprocessor directives have been applied manually or automatically to improve code understanding and maintainability and enable better use of APR tools [Medeiros 2017, van Tonder 2020], such as by expanding macros [Reiter 2022]. Our tool does not provide SPL-level repairs and efficiencies, but it does address some C preprocessor analysis complexities.

Binary repairs: Mayhem is an autonomous bot that finds and fixes C and C++ vulnerabilities in binaries [Avgerinos 2018]. In contrast, our tool repairs source code.

Mutational repairs: GenProg uses an extended form of genetic programming to repair code by using existing test suites to encode the defect and its required functionality. Early test results included fixing eight classes of defects in 1.25 million lines of C code in 16 programs [Le Goues 2012a].

Clang and clang-tidy: Clang's [Lattner 2008, Clang 2023a] clang-tidy [Clang-tidy 2023] program includes modern refactoring and rewriting APIs [Clang 2023b], and clangd [Clangd 2023] allows for easy integration into text editors and IDEs. Refactoring and rewriting APIs of clangd are designed to repair only code identified by clang-tidy. Our fixes address some issues that clang-tidy doesn't find (e.g., MSC12-C) and some that clang-tidy detects but does not auto-fix. For example, our tool repairs null pointer dereferences, but clang-tidy doesn't have auto-fixes for its associated checkers (`core.NonNullParamChecker`, `core.NullDereference`, `unix.cstring.NullArg`, `clang-analyzer-core.NonNullParamChecker`, and `clang-analyzer-core.NullDereference`) [LLVM Project 2025].

Other APR tools fix the three code flaws in C code that our tool repairs. Table 6 summarizes some of these tools.

Table 6: Some APR Tools that Fix the Same CWEs in C Code as Our Tool (Not Comprehensive)

CWE-476 NULL Pointer Dereference	
Paper Describing APR Tool That Fixes This CWE	**Detail About the APR Tool**
GenProg [Le Goues 2012a]	Genetic programming
SemFix [Nguyen 2013]	Symbolic execution, constraint solving
Prophet [Long 2016]	Learning-based
SapFix [Marginean 2019]	Template-based repairs, used at Meta
SPR [Long 2015]	Staged program repair, condition synthesis
FootPatch [van Tonder 2018]	Repairs SA alerts from Infer w. symbolic heaps
ExtractFix [Gao 2021]	Template-based, constraints from sanitizers
EffFix [Zhang 2024]	Incorrectness separation logic
VulRep [Wei 2023]	Based on vuln-inducing and fixing commits
VulMatch [Cao 2024]	Based on pretrained code model
Flexirepair [Koyuncu 2020]	Semantic patches
Coccinelle [Lawall 2018, INRIA 2025]	Semantic patches, Linux kernel
Conch [Xing 2024]	Built on Infer, repairs Infer SA alerts, search- & constraint-based
CWE-561 Dead Code	
Paper Describing APR Tool That Fixes This CWE	**Detail About the APR Tool**
Systematic Code and Asset Removal Framework (SCARF) [Shackleton 2023]	Used at Meta
Genetic mutation APR with operators delete, insert, & swap [Le Goues 2012b]	delete operator removes dead code
CWE-908 Use of Uninitialized Resource	
Paper Describing APR Tool That Fixes This CWE	**Detail About the APR Tool**
Prophet [Long 2016]	Learning-based, starts with set of good patches
SPR augmented [Mehne 2018]	Search-based, location-selection, testcase pruning
PAR [Kim 2013]	Manually created repair templates summarize good patches

Tests: For APR tool-use in industry, it is standard practice to run tests after creating a patch candidate (test suites, compilability, general SA tools, dynamic analysis tools such as fuzzers, and sometimes naturalness analyzers), and then for test-passing patches to submit a potential patch for human review before acceptance [Eladawy 2024, Yang 2023, Chen 2021]. Template-based non-learning APR tools like PAR and Redemption don't require the codebase to have a pre-existing test suite. Tools do exist to automatically create test suites, and active research is being done in

that area [Lemieux 2023, Schäfer 2023, Zhang 2011, Babić 2011], but if the new tests fail, work is required to fix the code to make them pass (or to determine the new tests aren't good).

Multiple SA tools: TRICORDER is a code analysis ecosystem integrated into the production Google development toolchain, including 30 SA tools (as of 2014), and provides APR patches for SA alerts as fix suggestions [Sadowski 2015]. Some of the many APR tools that fix the same CWEs in C code as our tool are listed (not comprehensively) in Table 6. Our tool is not a comprehensive framework but does provide APR fixes for alerts from multiple SA tools.

Reduction of SA alerts: Google's TRICORDER analysis showed that violations of code flaw checks (which produce SA alerts) sharply change from increasing to decreasing when SA alerts for that flaw appear and APR patches are provided during code (self-)review before check-in [Sadowski 2015]. One striking example of this used patches for clang-tidy checker `readability-redundant-smartptr-get` [LLVM Project 2025].

8 Lessons Learned

Manually adjudicating and repairing SA alerts is costly, requiring years of work per codebase in our case study. Our development and test results support the idea that an automatic repair tool could significantly reduce that expense by applying repairs to some of the most frequent alert types.

We learned much while planning and building our repair tool that generalizes for others building APR tools. We found that some alerts for some secure coding guideline violations are automatically repairable, even if SA tools have poor accuracy in identifying defective code. This prompted an update of the metrics used to assess when guidelines are detectable or repairable in the CERT Coding Standards.

Many codebases provide their own error-handling routines, while others use standard C constructs like `abort()`. APR tools should try to use the codebase's error-handling method.

Some C and C++ codebases use conditional preprocessor directives like `#ifdef`, which cause a single source file to represent a family of similar source files, not all of which are valid. APR tools should strive not to break the code for any macro configuration or at least to explicitly specify a particular configuration for the repair to target.

We outlined some obstacles with automated repair, such as conditional preprocessor directives appearing inside expressions. We demonstrated in Figure 3 that it is a simple task to create examples of code that are difficult to repair automatically.

Of the three categories our tool repairs, two satisfied our goal of not breaking the code, supporting their possible safe batch repair of multiple alerts. The repair *techniques* do not change the behavior of the code on good execution traces, in contrast to techniques that use genetic algorithms or AI, which cannot make such a guarantee. We do not claim that our tool is free of bugs. Our repair for null-pointer dereferences is to insert code that checks whether the pointer is null immediately before the dereference. Thus, our repair does not change the behavior on good traces, where null pointers aren't dereferenced. Our repair for use-before-initialization of variables is to add an initial value at the declaration. Reading an uninitialized variable is undefined behavior[2] in C, so on every good trace, the initial value we assign will be overwritten before the variable is read. Thus, our change of adding an initial value doesn't alter the behavior of any good trace. Our tool's repair for MSC12-C does not change code behavior but is sometimes considered inferior by developers, so it is disabled by default. The repair is deterministic (in contrast, humans make mistakes) and accepting APR patches is faster than human coding, so it can still be useful.

[2] To be precise, C23 and later C standards would describe an uninitialized read as "indeterminate behavior," which is sometimes but not always "undefined behavior."

We confirmed that these repairs do not significantly impede performance. Re-running the SA tools on our repaired Git codebase did not produce the repaired alerts or new alerts (except possibly on `sqlite3.c`; see the appendix).

We learned that many APR tools repair many flaw types with many techniques, some shown to be highly effective against large test sets, some with highly-modular frameworks and scalable test systems, and some widely used in IDE plugins and industry toolchains.

We published our APR tool, manual adjudications, and automatic repairs plus the dataset of SA alerts discussed in the report, so others may build on them or use them to test against other APR tools. One way to build on our work would be to do similar code flaw prioritization analysis as in this report, to extend other APR tools to address more code flaw types. Also, the combination of features in our APR tool may be useful for particular environments. In the future, our fixes could be incorporated into clang-tidy, using repairs described in this report for their matching checkers. Our tool could be integrated more deeply into IDEs such as VS Code or Eclipse [Eclipse 2023]. We plan to repair alerts for more coding rules and incorporate techniques for repairs of multiple macro configurations.

Appendix

We encountered the following problems during the recurrence test for Zeek, with results in Table 1. No other tables were affected by these problems.

First, a different version of Cppcheck was run on the original Zeek than was run on repaired Zeek (2.4.1 vs. 2.9). This might be the cause for some variance in the counts.

Second, we had trouble with the two `sqlite3.c` files embedded in Zeek, for several reasons: Each file has about 400 SPLs, and Cppcheck takes over 40 hours to analyze one such file. Thus, trying to reproduce previous results on this file was prohibitive. Also, our tool only repaired one SPL, so it could have no effect on alerts on code that are `#ifdef`'d out. Finally, these files each consist of one C file that contains the entire Sqlite source code. Since Sqlite is external to Zeek, repairs should be made to the sqlite3 source itself rather than to the copy used by Zeek.

Due to time constraints, we were unable to measure the effect of repairing MSC12-C alerts with Zeek.

References

URLs are valid as of the publication date of this report.

[Aberg 2003]
Aberg, R.A. et al. On the Automatic Evolution of an OS Kernel Using Temporal Logic and AOP. Pages 196–204. *In Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. October 2003. https://ieeexplore.ieee.org/document/1240307

[Anderson 2018]
Anderson, P. Measuring the Value of Static-Analysis Tool Deployments. *IEEE Security & Privacy*. Volume 10. Issue. 3. January 2012. Pages 40–47. https://ieeexplore.ieee.org/document/6143915

[Avgerinos 2018]
Avgerinos, T. et al. The Mayhem Cyber Reasoning System. *IEEE Security & Privacy*. Volume 16. Number 2. March/April 2018 Pages 52–60. https://ieeexplore.ieee.org/document/8328972

[Ayewah 2010]
Ayewah, N. & Pugh, W. The Google FindBugs fixit. Pages 241-252. In *Proceedings of the 19th International Symposium on Software Testing and Analysis.* July 2010. https://dl.acm.org/doi/10.1145/1831708.1831738

[Babić 2011]
Babić, D. et al. Statically-Directed Dynamic Automated Test Generation. Pages 12–22. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. July 2011. https://dl.acm.org/doi/10.1145/2001420.2001423

[Cao 2024]
Cao, X. et al. Enhancing Vulnerability Repair Through Summarization of Repair Patterns and Optimal Matching. *Journal of Systems and Software*. Volume 230. December 2025. Article 112528. https://www.sciencedirect.com/science/article/pii/S0164121225001967

[CERT 2014]
Source Code Analysis Laboratory (SCALe) (Collection*). Software Engineering Institute Website*. November 11, 2014. https://www.sei.cmu.edu/library/source-code-analysis-laboratory-scale-collection/?page=1

[CERT 2025]
CERT Secure Coding Group. EXP12-C Do Not Ignore Values Returned by Functions. *Carnegie Mellon University Software Engineering Institute.* May 2, 2025 [accessed]. https://wiki.sei.cmu.edu/confluence/x/mtYxBQ

[CERT 2023a]
CERT Secure Coding Group. SEI CERT Coding Standards. *Carnegie Mellon University Software Engineering Institut*e. May 11, 2023 [accessed]. https://wiki.sei.cmu.edu/confluence/display/seccode

[CERT 2023b]
CERT. CERT Rosecheckers. *GitHub.* May 15, 2023 [accessed]. https://github.com/cmu-sei/certrosecheckers

[Chen 2021]
Chen, L. Fast and Precise On-the-Fly Patch Validation for All. Pages 1123–1134. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. May 2021. https://ieeexplore.ieee.org/document/9402121

[Clang 2023a]
Clang. *Clang.llvm.org*. May 15, 2023 [accessed]. https://clang.llvm.org/

[Clang 2023b]
Clang's Refactoring Engine. *Clang.llvm.org.* May 15, 2023 [accessed]. https://clang.llvm.org/docs/RefactoringEngine.html/

[Clangd 2023]
Clangd. What Is clangd? *Clangd.llvm.org*. May 15, 2023 [accessed]. https://clangd.llvm.org/

[Clang-tidy 2023]
Clang-tidy. Extra Clang Tools 22.0.0 Git Documentation. *Clang.llvm.org*. May 15, 2023 [accessed]. https://clang.llvm.org/extra/clang-tidy/

[Couto 2022]
Couto, V. et al. Source Matching and Rewriting. *arXiv* [preprint]. February 5, 2022. https://arxiv.org/abs/2202.04153

[Cppcheck 2023]
Danmar. Cppcheck. *GitHub.* May 15, 2023 [accessed]. https://github.com/danmar/cppcheck

[Delaitre 2018]
Delaitre, M.; Stivalet, B. C.; Black, P. E.; Okun, V.; Cohen, T. S.; & Ribeiro, A. *SATE V Report: Ten Years of Static Analysis Tool Expositions*. NIST SP 500-326. National Institute of Standards and Technology. 2018. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-326.pdf

[Dimovski 2024]
Dimovski, A. S. Mutation-Based Lifted Repair of Software Product Lines. Pages 12:1–12:24. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. September 2024. https://2024.ecoop.org/details/ecoop-2024-papers/21/Mutation-based-Lifted-Repair-of-Software-Product-Lines

[Dos2Unix 2023]
Dos2Unix. / unix2dos: Text File Format Converters. *Sourceforge*. May 15, 2023 [accessed]. https://dos2unix.sourceforge.io/

[Eclipse 2023]
Eclipse. *Eclipse Foundation*. June 7, 2023 [accessed]. https://www.eclipse.org/

[Eladawy 2024]
Eladawy, H. et al. Automated Program Repair: What Is It Good For? Not Absolutely Nothing! Pages 1021–1033. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. April 2024. https://ieeexplore.ieee.org/document/10548723

[Flynn 2018]
Flynn, L. Improve Your Static Analysis Audits Using CERT SCALe's New Features. *Software Engineering Institute Website*. December 18, 2018. https://www.sei.cmu.edu/library/improve-your-static-analysis-audits-using-cert-scales-new-features-2/

[Gao 2021]
Gao, X. et al. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM).* Volume 30. Number 2. Article 14. May 2018. https://dl.acm.org/doi/10.1145/3418461

[Git 2023]
Git. *GitHub*. May 15, 2023 [accessed]. https://github.com/git/git

[Git 2025]
GitHub Copilot. *GitHub*. April 24, 2025 [accessed]. https://github.com/features/copilot

[GitHub 2025]
About Code Scanning. *GitHub*. April 17, 2025 [accessed]. https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning

[GnuLib 2023]
CoreUtils File Regexec.c. GnuLib. *GitHub*. May 15, 2023 [accessed]. https://github.com/coreutils/gnulib/blob/master/lib/regexec.c

[Gupta 2017]
Gupta, R. et al. DeepFix: Fixing Common C Language Errors by Deep Learning. Pages 1345–1351. *In Proceedings of the AAAI Conference on Artificial Intelligence*. February 2017. https://ojs.aaai.org/index.php/AAAI/article/view/10742

[Imtiaz 2019]
Imtiaz, N. et al. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. Pages 323–333. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. October 2019. https://ieeexplore.ieee.org/document/8987507

[INRIA 2025]
Coccinelle: Impact on the Linux Kernel. *INRIA*. March 6, 2025 [accessed]. https://coccinelle.gitlabpages.inria.fr/website/impact_linux.html

[IntRepair 2023]
IntRepair. *GitHub*. May 30, 2023 [accessed]. https://github.com/TeamVault/IntRepair

[Kim 2013]
Kim, D. et al. Automatic Patch Generation Learned from Human-written Patches. Pages 802–811. In *2013 35th International Conference on Software Engineering (ICSE)*. May 2013. https://ieeexplore.ieee.org/document/6606626

[Klieber 2021]
Klieber, W. et al. Automated Code Repair to Ensure Spatial Memory Safety. Pages 23–30. In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*. June 1, 2021. https://ieeexplore.ieee.org/document/9474531

[Koyuncu 2020]
Koyuncu, A. et al. FlexiRepair: Transparent Program Repair with Generic Patches. *arXiv* [preprint]. November 2020. https://arxiv.org/abs/2011.13280

[Lattner 2008]
Lattner, C. LLVM and Clang: Next Generation Compiler Technology. *LLVM.org*. May 17, 2008. https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.pdf

[Lawall 2018]
Lawall, J. et al. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. Pages 601–613. In *USENIX ATC '18: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. July 11, 2018. https://dl.acm.org/doi/10.5555/3277355.3277413

[Le Goues 2012a]
Le Goues, C. et al. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*. Volume 38. Number 1. January 2012. Pages 54–72. https://ieeexplore.ieee.org/document/6035728

[Le Goues 2012b]
Le Goues, C. Representations and Operators for Improving Evolutionary Software Repair. Pages 959–966. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. July 2012. https://dl.acm.org/doi/10.1145/2330163.2330296

[Lemieux 2023]
Lemieux, C. et al. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre–trained Large Language Models. Pages 919–931. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. May 2023. https://ieeexplore.ieee.org/document/10172800

[Liu 2019]
Liu, K. et al. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. Pages 456–-467. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019. https://www.computer.org/csdl/proceedings-article/saner/2019/08667970/18uSxbVsX6g

[LLVM Project 2025]
Clang-tidy Checks. Extra Clang 18 Documentation. *LLVM Project*. May 4, 2025 [accessed]. https://clang.llvm.org/extra/clang-tidy/checks/list.html

[Long 2015]
Long, F. et al. Staged Program Repair with Condition Synthesis. Pages 166–178. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. August 2015. https://dl.acm.org/doi/10.1145/2786805.2786811

[Long 2016]
Long, F. et al. Automatic Patch Generation by Learning Correct Code. Pages 298–312. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. January 2016. https://dl.acm.org/doi/10.1145/2837614.2837617

[Marginean 2019]
Marginean, A. et al. SapFix: Automated End-to-End Repair at Scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. May 2019. https://ieeexplore.ieee.org/document/8804442

[Mechtaev 2015]
Mechtaev, S. et al. DirectFix: Looking for Simple Program Repairs. Pages 448–458. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. May 2015. https://ieeexplore.ieee.org/document/7194596

[Medeiros 2017]
Medeiros et al. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering*. Volume 44. Number 5. March 28, 2017. Pages 453–469. https://ieeexplore.ieee.org/document/7888579

[Mehne 2018]
Mehne, B. et al. Accelerating Search-Based Program Repair. Pages 227–238. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. April 2018. https://ieeexplore.ieee.org/document/8367051

[Meneide 2023]
Meneide et al. *Programming Languages—C, 5th ed. draft (n3149*). ISO/IEC 9899:2023 (E). ISO/IEC. 2023. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3088.pdf

[Microsoft 2023a]
Visual Studio Code. *Microsoft*. May 15, 2023 [accessed]. https://code.visualstudio.com

[Microsoft 2023b]
Quick Actions. *Microsoft*. May 31, 2023 [accessed]. https://learn.microsoft.com/en-us/%20visualstudio/ide/quick-actions?view=vs-2022

[Microsoft 2023c]
Edit and Refactor C++ Code in Visual Studio. *Microsoft*. May 31, 2023 [accessed]. https://learn.microsoft.com/en-us/cpp/ide/writing-and-refactoring-code-cpp?view=msvc-170

[MITRE 2023a]
Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types. *MITRE*. May 15, 2023 [accessed]. https://cwe.mitre.org

[MITRE 2023b]
2021 CWE Top 25 Most Dangerous Software Weaknesses. *MITRE*. May 15, 2023 [accessed]. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[Muntean 2019]
Muntean, P. et al. IntRepair: Informed Repairing of Integer Overflows. IEEE Transactions on Software Engineering. Volume 47. Number 10. October 8, 2019. Pages 2225–2241. https://ieeexplore.ieee.org/document/8862860

[Necula 2002]
Necula, G.C. et al. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. Pages 213-228. In *International Conference on Compiler Construction*. January 2002. https://link.springer.com/chapter/10.1007/3-540-45937-5_16

[Nguyen 2013]
Nguyen, H. D. T. et al. SemFix: Program Repair via Semantic Analysis. Pages 772–781. In *2013 35th International Conference on Software Engineering (ICSE)*. May 2013. https://ieeexplore.ieee.org/document/6606623

[Ni 2024]
Ni, C. et al. Megavul: AC/C++ Vulnerability Dataset With Comprehensive Code Representations. Pages 738-742. In *Proceedings of the 21st International Conference on Mining Software Repositories*. July 2, 2024. https://dl.acm.org/doi/10.1145/3643991.3644886

[Reiter 2022]
Reiter, P. et al. Improving Source-Code Representations to Enhance Search-Based Software Repair. Pages 1336–1344. In *Proceedings of the Genetic and Evolutionary Computation Conference*. July 2022. https://dl.acm.org/doi/10.1145/3512290.3528864

[Sadowski 2015]
Sadowski, C. et al. Tricorder: Building a Program Analysis Ecosystem. Pages 598 – 608. In *ICSE '15: Proceedings of the 37th International Conference on Software Engineering*. May 16, 2015. http://dl.acm.org/doi/10.5555/2818754.2818828

[Schäfer 2023]
Schäfer, M. et al. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*. Volume 50. Number 1. January 2024. Pages 85–105. https://ieeexplore.ieee.org/document/10329992

[Schordan 2003]
Schordan, M. et al. A Source-to-Source Architecture for User-Defined Optimizations. In *Joint Modular Languages Conference*. 2003. https://www.osti.gov/servlets/purl/15004527

[SEI 2025]
Carnegie Mellon University Software Engineering Institute (CMU SEI). Redemption. *GitHub*. August 11, 2025 [accessed]. https://github.com/cmu-sei/ redemption/tree/main

[Shackleton 2023]
Shackleton, W. et al. Dead Code Removal at Meta: Automatically Deleting Millions of Lines of Code and Petabytes of Deprecated Data. Pages 1705–1715. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. November 30, 2023. https://dl.acm.org/doi/10.1145/3611643.3613871

[Sidiroglou 2005]
Sidiroglou, S. et al. Countering Network Worms Through Automatic Patch Generation. *IEEE Security & Privacy.* Volume 3. Number 6. November–December 2005. Pages 41–49. https://ieeexplore.ieee.org/document/1556535

[Snyk Security 2025a]
Snyk Security Visual Studio Code Extension. *Visual Studio*. February 28, 2025 [accessed]. https://marketplace.visualstudio.com/items?itemName=snyk-security.snyk-vulnerability-scanner-vs

[Snyk Security 2025b]
Snyk Security. C++ Rules. *Snyk Security*. February 13, 2025 [accessed]. https://docs.snyk.io/scan-with-snyk/snyk-code/snyk-code-security-rules/c++-rules

[Svoboda 2016a]
Svoboda, D. et al. Static Analysis Alert Audits: Lexicon & Rules. Pages 37–44. In *2016 IEEE Cybersecurity Development (SecDev)*. November 2016. https://ieeexplore.ieee.org/document/7839787

[Svoboda 2016b]
Svoboda, D. Beyond errno: Error Handling in "C." Page 161. In *2016 IEEE Cybersecurity Development (SecDev)*. November 2016. https://ieeexplore.ieee.org/document/7839814

[Svoboda 2023]
Svoboda, David; Klieber, Will; & Flynn, Lori. Alert Type Frequency Assessment of Open-Source Static Analysis Tools and Codebases. *Zenodo*. May 2023 [accessed]. https://doi.org/10.5281/zenodo.7958183

[Synopsis 2023]
Code Dx. Software Risk Manager: Triage Assistant. *Black Duck Software*. May 15, 2023 [accessed]. https://sig-synopsys.my.site.com/community/s/article/Software-Risk-Manager-Triage-Assistant.

[Urli 2018]
Urli, S. et al. How to Design a Program Repair Bot? Insights from the Repairnator Project. Pages 95–104. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. May 2018. https://dl.acm.org/doi/10.1145/3183519.3183540

[van Tonder 2018]
van Tonder, R. et al. Static Automated Program Repair for Heap Properties. Pages 151–162. In *Proceedings of the 40th International Conference on Software Engineering*. May 2018. https://ieeexplore.ieee.org/document/8453073

[van Tonder 2020]
van Tonder, R. et al. Tailoring Programs for Static Analysis via Program Transformation. Pages 824–834. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. October 2020. https://dl.acm.org/doi/10.1145/3377811.3380343

[Veracode 2020]
State of Software Security v11. Veracode. 2020. https://info.veracode.com/report-state-of-software-security-volume-11.html

[Wei 2023]
Wei, Y. et al. VulRep: Vulnerability Repair Based on Inducing Commits and Fixing Commits. *EURASIP Journal on Wireless Communications and Networking*. Volume 2023. Number 1. April 21, 2023. Page 34. https://dl.acm.org/doi/abs/10.1186/s13638-023-02242-7

[Weimer 2010]
Weimer, W. et al. Automatic Program Repair with Evolutionary Computation. Communications of the ACM. Volume 53. May 2010. Number 5. Pages 109–116. May 2010. https://dl.acm.org/doi/abs/10.1145/1735223.1735249

[Xing 2024]
Xing, Y. et al. What {IF} Is Not Enough? Fixing Null Pointer Dereference with Contextual Check. Pages 1367–1382. In *33rd USENIX Security Symposium (USENIX Security 24)*. August 12, 2024. https://dl.acm.org/doi/10.5555/3698900.3698977

[Yang 2023]
Yang, J. et al. A Large-Scale Empirical Review of Patch Correctness Checking Approaches. Pages 1203–1215. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. November 2023. https://dl.acm.org/doi/10.1145/3611643.3616331

[Zeek 2023]
Zeek. *GitHub.* May 15, 2023 [accessed]. https://github.com/zeek

[Zhang 2011]
Zhang, S. et al. Combined Static and Dynamic Automated Test Generation. Pages 353–363. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. July 2011. https://dl.acm.org/doi/10.1145/2001420.2001463

[Zhang 2023]
Zhang, Q. et al. A Survey of Learning–Based Automated Program Repair. *ACM Transactions on Software Engineering and Methodology*. Volume 33. Issue 2. December 23, 2023. Pages 1–69. https://dl.acm.org/doi/10.1145/3631974

[Zhang 2024]
Zhang, Y. et al. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. *ACM Transactions on Software Engineering and Methodology*. Volume 34. Issue 3. February 20, 2025. Article 69. https://dl.acm.org/doi/10.1145/3705310

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2025	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Automated Code Repair for C/C++ Static Analysis	FA8702-15-D-0002

6. AUTHOR(S)

David Svoboda, Lori Flynn, William Klieber, Michael Duggan, Nicholas Reimer, and Joseph Sible

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2025-TR-007

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Static analysis (SA) tools produce many diagnostic alerts indicating that source code in C or C++ may be defective and potentially vulnerable to security exploits. Many of these alerts are false positives. Identifying the true-positive alerts and repairing the defects in the associated code are huge efforts; automated program repair (APR) tools can help. Our experience showed us that APR can reduce the number of SA alerts significantly and reduce the manual effort of analysts to review code. This engineering experience paper details the application of design, development, and performance testing to an APR tool we built that repairs C/C++ code. Its repairs are simple and local. Furthermore, our findings convinced the maintainers of the CERT Coding Standards to re-assess and update the metrics used to assess when violations of guidelines are detectable or repairable. We discuss engineering design choic-es made to support goals of trustworthiness and acceptability to developers.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Code repair, C/C++, static analysis	35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102