Automated Code Repair to Ensure Spatial Memory Safety

William Klieber*, Ruben Martins§, Ryan Steele*, Matt Churilla*, Mike McCall*, David Svoboda*
* Software Engineering Institute, Carnegie Mellon University
§ School of Computer Science, Carnegie Mellon University

***Abstract*—We present a technique for repairing C code to protect against potential violations of spatial memory safety. Many existing techniques can harden software against memory bugs as part of a compiler pass. However, this creates dependencies on the compiler and makes it difficult to fine-tune or even inspect the repairs. We propose an automated technique for repairing the source code to eliminate spatial memory vulnerabilities.**

Performing the repair at the source-code level introduces a new challenge: analysis and transformation are most easily done on an intermediate representation (IR), but existing techniques using IRs have fundamental limitations in regards to translating changes back to the level of source code. We break this challenge into two parts: (1) translating changes at the level of the IR to the abstract syntax tree (AST) level, and (2) translating changes at the AST level back to the original source-code text.

Preemptively repairing potential memory bugs leads to a trade-off between performance overhead and memory safety. While for safety-critical applications this trade-off may be acceptable, for other applications we can reduce the performance overhead by only repairing suspicious locations.

We implemented our approach in a tool called ACR and show that it can repair spatial memory vulnerabilities on buggy programs from the Software Verification Competition. Additionally, we also ran ACR on medium-size programs and preliminary results show the scalability of ACR for thousands of lines of code. Finally, we integrated ACR with static analysis tools and show that the performance overhead is small when repairing only locations that are flagged by a static analyzer.

***Index Terms*—code repair, memory safety, code transformation**

I. Introduction

Memory violations are among the most common and most severe types of vulnerabilities. For the past 3 years, spatial memory violations were 15% of CVEs in the NIST National Vulnerability Database and 24% of the critical-severity CVEs. Researchers in academia and industry have studied memory bugs for decades [1]. One approach to guarantee memory safety of C programs is to extend the C language with memory-safe pointers [2], [3], [4], [5]. However, this approach has the downside of not supporting legacy code since existing code would need to be rewritten in a new safe language. Alternatively, the structure of pointers can be extended to track the bounds of the objects it points to so that all memory reads and writes are within the bounds of the memory region [6], [7], [8], [9], [10], [11], [12], [13]. These approaches guarantee memory safety by inserting bound checks during the compilation process, rather than changing the source code.

In this work, we propose to repair a C program against potential violations of spatial memory safety at the source-code level. Our goal differs from traditional program repair [14], [15] since we do not repair the functionality of the program but instead repair potential security vulnerabilities so that the program cannot be exploited by an attacker. Performing the repair at the source-code level instead of doing it during the compilation process introduces new challenges. Analysis and transformation of code are most easily done at an intermediate representation (IR) level. However, existing approaches are not able to perform backward translation from IR to source code while preserving the structure of the original code. We tackle this challenge by breaking it into two parts: (1) translating changes from the IR level to abstract syntax tree (AST) level, and (2) translating changes from the AST level back to the original source-code. For the first part, we show that it is possible to carefully design AST↔IR transformation rules and repair transformations together, so that repairs done at the IR level are easily lifted to the AST level. For the second part, we have modified Clang to produce an AST annotated with corresponding locations in the original source-code files.

Repairing all potential memory spatial vulnerabilities can lead to a large performance overhead, due to inserted bounds checks and propagation of bounds metadata. While this can be acceptable for safety critical applications, this may be an issue for other applications. To reduce this overhead, we can perform *partial repairs* where we take as input a set of suspicious source-code locations and only guarantee the memory safety of those locations. These suspicious locations can be given by either a systems analyst or a static analysis tool.

We implemented our approach in a tool called ACR and evaluated it on programs with spatial memory bugs from the Software Verification Competition and on programs from the SPEC CPU2006 benchmarks. To increase confidence in our implementation, we ran SYMBIOTIC [16], a verification tool for memory safety, on the programs repaired by ACR for the Software Verification Competition. To evaluate the performance of the files repaired by ACR, we repaired larger programs from the SPEC CPU2006 benchmarks. Preliminary results show that the repaired programs are on average around 46% slower than the original programs. However, when only repairing a subset of locations that were flagged by a commercial program analyzer, the average time overhead of running the partially repaired program decreases to 19%, being as low as 3% for some programs. Even when there is

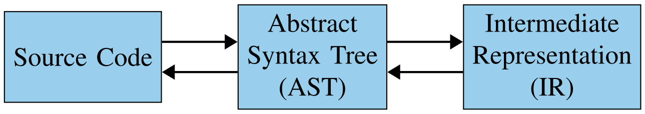

Fig. 1: Overview of the source-to-source repair

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

Fig. 2: Fat-pointer structure definition

```
// before repair
char* p = malloc(size);
while ((c = nondet_char()) != 0) {
  *p = c;
  p = p + 1; }
```

```
// after repair
struct FatPtr_char p = malloc_FatPtr_char(size);
while((c = nondet_char()) != 0) {
  *bound_check(p) = c;
  p = fatp_add(p, 1); }
```

Fig. 3: Example of repair using fat pointers

a large overhead, this can be an acceptable trade-off for the guarantee that these programs will not have spatial memory violations. If efficiency is critical, then we can repair only the more suspicious locations and substantially improve the performance of the system.

This paper makes the following key contributions:

- We propose an approach that can translate from source code to an IR, perform transformations at the IR level, and map back to a modified version of the source code with the desired transformations.
- We implemented our approach in a tool called ACR that can repair a diversity of C programs and ensure that they are spatially memory safe.
- We allow partial repairs where ACR can just repair a subset of locations that can be flagged by a commercial static analyzer.

II. Overview

Memory safety is often divided into two parts: spatial and temporal. A program has a *spatial memory violation* if a write or read is performed beyond the bounds of a memory region. A program has a *temporal memory violation* if a write or read is done to a region after it has been deallocated. In this work, we focus on assuring *spatial memory safety* of C programs.

Figure 1 shows an overview of our repair approach. The input to our system is a C program and the output is a repaired C program that avoids any potential spatial memory violation. Although we want to make repairs at the level of the original source code, in many aspects it is easier to analyze and repair the code at the level of an intermediate representation (IR). Our solution is to *reversibly* transform the source code to an IR, make repairs on the IR, and then translate the repairs back to the original source code. We break this procedure into three stages: a source→IR stage (Section III), repair of the code at the IR level (Section V), and transforming the repairs from IR to source code (Section VI).

A common approach to ensure spatial memory safety is to use fat pointers [6], [7]. A fat pointer is a structure containing three fields: the raw pointer itself, the base of the memory region, and the size of the memory region, in bytes. For each pointer type $T*$, we introduce a fat-pointer type as defined in Figure 2.

Fat pointers have the disadvantage of increasing the time and memory overhead of the program and having compatibility issues with external libraries. Other approaches have been proposed, where the pointer structure is not modified but the bound information is encoded directly in the memory layout of the pointer [8], [12], [10], [11]. However, we advocate that the use of fat pointers is a better fit for repair of source code since they can be used to make local repairs that do not require global changes to the program and are more portable.

Figure 3 shows an example of our approach on a small example. The fat pointers and functions used during the fattening process are defined in a header file and do not need to be modified by the programmer. We can observe that the repaired code is faithful to the original code and can be easily interpreted and modified.

Note that the repair, including the bound checks, is done at the source code level. This contrasts with prior work that guarantees spatial memory safety via bound checks that are introduced during compilation [6], [7], [8], [12], [10], [11]. In our case, we want the repairs to be readable by programmers (as opposed to present only in the binary executable). Some advantages of repairing the source code with our approach when compared to a compiler pass are:

- Repairs to the source code are easily audited.
- Repairs to the source code can be tweaked to improve performance, if necessary. Usually, only a small percentage of a codebase is performance-critical.
- Doing repair as part of the compilation makes the build process dependent on using a compiler with such functionality. This can cause future problems if the repair technology is no longer maintained[1] or if the code needs to be compiled on a different platform.

III. From Source Code to IR

For clarity of presentation, and due to space constraints, in this paper we consider only a subset of C. (However, our implementation handles most of standard C, but lacks support for some uncommonly-used features such as variable-length arrays.) Additionally, we use a different syntax than C for

[1] This concern applies mainly to custom repair tools, not to tools such as AddressSanitizer [17] that are integrated into a major compiler.

- t_1 = $t_2 \star_b t_3$;
- t_1 = $\star_u t_2$;
- t_1 = $\star t_2$;
- $\star t_1$ = t_2;
- t_1 = &v;
- t_0 = t_f(t_1, ..., t_n);
- t_0 = (*type*){$t_1, ..., t_n$};
- t_1 = (*type*)t_2;
- t_1 = & t_2->*field*;
- return t;
- goto *label*;
- *label*:;
- ; /* empty statement */
- *var-decl*;
- t_1 = x; /* where x is a variable, function, or literal */
- if (t) goto $label_1$ else goto $label_2$;

Fig. 4: Statements in the IR language. A "t" (with or without a subscript) denotes a temporary variable introduced by the transformation from the original source code to the IR.

types and declarations, so that the type is not intermingled with the identifier being declared, in order to avoid confusion when discussing transformations for using fat pointers: We write "ptr<*type*>" to denote the type of a pointer to an object of type *type*, and we write "*type* *var*;" to denote a declaration of a variable *var* of type *type*.

Our intermediate representation (IR) language is shown in Figure 4. We allow each syntactic unit (i.e., expression, statement, etc.) to be annotated with a set of tuples that convey information about how to transform back to original source code. Given a syntactic unit u, we write "(u)@*tag*" to denote that u is annotated with *tag*, where *tag* is a tuple.

In Figure 5, we show the transformations used for transforming the original AST to IR and for transforming the repaired IR back to an AST. Each transformation is indicated by two boxes connected with a bidirectional arrow. The code in the top box is semantically equivalent to the code in the bottom box, provided that any temporary variables that are defined in the bottom box but not the top box are fresh (i.e., not occur elsewhere in the program). When transforming from the original AST to IR, code that matches the top box is transformed into code in the bottom box. Likewise, when transforming from the repaired IR to AST, code that matches the bottom box is transformed into code in the top box.

Each AST node of the original AST is annotated with a pair $\langle$L, *loc*$\rangle$ where *loc* is a tuple of the form $\langle$*filename*, *byte_offset_start*, *byte_offset_end*, *ast_id*$\rangle$, identifying the location of the source-code text corresponding to the AST node.[2] Here, *ast_id* is a unique number for each AST node; it is used to identify AST nodes in the event that multiple AST nodes have the same source-code range (e.g., in a macro expansion, all the resulting AST nodes have a source-code range corresponding to the macro use).

Before the repair, we store a mapping *LocToOrigAST* that maps the L tuple of each AST node to the original AST node.

Regarding the sizeof operator: In C, the argument to sizeof expression is not always evaluated if it is an expression. Generating IR for the argument of sizeof complicates the semantic correspondence of the IR to the AST, if the argument to sizeof has side-effects. However, the only negative effect in our case is that our points-to analysis becomes less precise. We consider it an acceptable trade-off.

IV. Analysis for Fattening Eligibility

Fattening a pointer that is stored in memory changes the memory layout of the program. Fattening a parameter to a function changes the program's binary interface. In either case, the fattening might break the program in the presence of external code (i.e., code not available to the repair tool) or platform-specific code. Accordingly, we consider a pointer to be *ineligible* to be fattened in the following cases:

- Pointers in memory passed to external code are ineligible.
- If a function is passed to external code, then the function's arguments and return value are ineligible to be fattened.
- If memory can be accessed via different types of pointers, then pointers *stored in* that memory are ineligible. (E.g., if a pointer to an array of pointers is cast to char* to read/write the individual bytes of the pointers, then the pointers *in* the array cannot be fattened.)

Given a type ty and a set of locations L, we write "$FattenTy(ty, L)$" to denote the fattened type of an object originally of type ty that is stored in L, except that if any location in L is ineligible for fattening, then ty must not be fattened, so $FattenTy(ty, L) = ty$. Given a variable v whose memory location is ℓ and whose declared type is ty, we write "$FattenVarType(v)$" to denote $FattenTy(ty, \{\ell\})$.

V. Repairs at the IR Level

Our tool inserts code to perform a bounds check immediately before dereferencing a fat pointer. That is, it replaces a pointer dereference "$\star p$" with "$\star$*bound_check*(p)", where *bound_check* is a function that aborts the program if the pointer is out-of-bounds and otherwise returns the raw pointer. In Standard C, we can define such a function for each pointer type T. Alternatively, we can define a single macro bound_check that works with all fat-pointer types, using widely-supported extensions to Standard C introduced by gcc. There are also other auxiliary functions/macros, which are defined similarly, but are not shown due to space constraints.

Figure 6 shows how we repair statements in the IR language (due to lack of space, not all could be shown). If a declaration of a local variable that holds a fat pointer lacks an initializer, we add an initializer (setting all fields to 0) to prevent an invalid fat pointer. We repair calls to standard-library functions to instead call wrapper functions that we developed, which do bounds checking. The function "fatten_raw" converts a raw pointer of unknown range to a faw pointer, using 0 as the lower bound and SIZE_MAX as the size.

We repair a declaration of a variable v of pointer type by replacing the declared type with $FattenVarType(v)$. We repair function declarations and definitions by fattening the return type and the types of arguments (if they are pointer types), using the *FattenTy* function.

We repair struct definitions by fattening the fields of the struct as necessary. More precisely, if the original type

[2] The "L" in "$\langle$L, *loc*$\rangle$" is just a literal letter "L", to indicate that the annotation specifies a location, as opposed to other types of annotations.

Expression statement

```
(e;)@⟨L, loc⟩
↕
(;)@⟨expr-stmt⟩@⟨L, loc⟩
t = e;
```

where e is not an assignment to a temporary. The purpose is to ensure that all arithmetic operations, function calls, etc. occur on the right-hand side of an assignment.

Address-of

```
tt = (&a[i])@⟨L, loc⟩;
↕
(;)@⟨array-address⟩
tt = ((a+i))@⟨L, loc⟩;
```

Return

```
(return e;)@⟨L, loc⟩
↕
(;)@⟨return⟩@⟨L, loc⟩
tt = e;
tret = tt;
return tret;
```

Note: *tret* is redundant but simplifies the repair operations.

Declaration with initializer

```
(type var = e;)@⟨L, loc⟩
↕
(;)@⟨decl-init⟩@⟨L, loc⟩
(type var;)@⟨has-init⟩@⟨IR⟩
var = e;
```

Compound Assignment

```
tt = ((*L_ptr ★= RHS))
     @⟨L, loc⟩;
↕
(;)@⟨subexpr, tt⟩@⟨L, loc⟩;
tL = L_ptr;
tR = RHS;
tt = (*tL ★= RHS);
```

Binary Operators

```
tt = (e1 ★b e2)@⟨L, loc⟩
↕
(;)@⟨subexpr, tt⟩@⟨L, loc⟩
t1 = e1;
t2 = e2;
tt = t1 ★b t2;
```

Array index, l-value

```
tt = (a[i] ≛ e)@⟨L, loc⟩;
↕
(;)@⟨subexpr, tt⟩@⟨arr-idx⟩
tt = (*(a+i) ≛ e)@⟨L, loc⟩;
```

where “$\overset{*}{=}$” denotes any assignment operator (“=”, “+=”, etc.).

sizeof

```
tt = (sizeof(e))@⟨L, loc⟩;
↕
(;)@⟨sizeof⟩@⟨L, loc⟩
t1 = e;
tt = sizeof(t1);
```

where e is an expression.

Notation: A variable whose name begins with “t” denotes a temporary variable introduced by the AST↔IR transformation.
In the forward (AST→IR) direction, statements already in IR form are not further transformed, even if they match a transformation rule.

Fig. 5: A selection of transformations between the AST and the IR (some are omitted due to space constraints)

Compound Assignment

```
tt = (*tp += tR);
↓
tt = fatp_assign_add(
  bound_check(tp), tR);
```

where *tp* is a fattenable pointer.

Memory Write

```
*tp = tR;
↓
*bound_check(tp) = e;
```

where e is defined as $e = tR$ unless the memory location being written to is marked a non-fattenable and *tR* is a fat pointer, in which case $e = tR$.rp.

Pointer arithmetic

```
tt = tp + tn;
↓
tt = fatp_add(tp, tn);
```

where *tn* is an integer and *tp* is a fattenable pointer. A similar transformation exists for “*tn* + *tp*” and for “*tp* − *tn*”.

Memory Read, case 1

```
tt = *tp;
↓
tt = *bound_check(tp);
```

If the location being read holds a non-fattenable pointer, then we replace “`tt = *tp;`” with “`tt = fatten_raw(ty′, *bound_check(tp));`”.

Address of a field of a struct

```
tt = &ts->field;
↓
tt = fat_fld_addr(
    ty, field, ts);
```

where $ty = \mathit{FattenVarType}(tt)$

Returning a raw pointer

```
tret = tt;
return tret;
↓
tret = tt.rp;
return tret;
```

Side condition: The return value of the function containing the return statement is a pointer type and is unfattenable.

External function call, returning a pointer

```
tt = tf(t1, ..., tn);
↓
tt = tf(d1, ..., dn);
```

where t_f is an external function, and, for $i \in \{1, \ldots, n\}$,

$$d_i = \begin{cases} t_i.\texttt{rp} & \text{if } t_i \text{ is a pointer} \\ t_i & \text{if } t_i \text{ otherwise} \end{cases}$$

If *tt* is a pointer type, then a further transformation is applied:

```
tt = tf(e1, ..., en);
↓
tt = fatten_raw(ty′,
    tf(e1, ..., en));
```

Fig. 6: Transformations for repairing IR (some are omitted due to space constraints)

of a field is *ty*, then the corresponding type in the repaired code will be *FattenTy*(*ty*, *FieldLocs*), where *FieldLocs* is the set of memory locations that may store the field. The macro `define_field_addr(Foo, field_i, ty′_i)` expands to a definition of a function that takes a fat pointer to a struct of type *Foo* and returns a fat pointer to field *field* of the struct. In Figure 6, we use the macro `fat_fld_addr` to call this function. The bounds of the fat pointer returned by `fat_fld_addr` are narrowed (with respect to the fat pointer for the struct) to the bounds of the field. (If the field is an array, the narrowed bounds encompass the whole array.) This narrowing of bounds protects against the *sub-object corruption* problem [13]. Some programs intentionally take a pointer to a field and perform pointer arithmetic to get a pointer to the containing `struct`; narrowing must be avoided in such cases.

The wrapper for `malloc` clears its memory, like `calloc`. This is needed to ensure the integrity of fat pointers stored in heap-allocated memory. `fat_realloc` never increases in-place the size of an already-allocated region of memory; it always allocates a new region and copies the old data. This is because resizing in-place would require adjusting the bounds of all existing fat pointers to the resized memory region.

A. Reducing performance impact of bounds checks

Inserting a bounds check before every memory access can significantly slow down a repaired program. While such performance impacts might often be acceptable if they are planned for at the design stage, significantly reducing the performance of existing in-operation systems is often rejected. Therefore, we added an option to repair only those memory accesses that are flagged by an external static analysis tool. With this option, any expression of the form "`*bound_check(`e`)`" is converted to "`*(`e`.rp)`" if it originated from an unflagged line.

VI. IR to Source

We reconstruct the AST from the repaired IR by applying (in reverse direction) the transformations in Figure 5 (along with other transformations that we do not have space to show).

We first determine which AST nodes were modified by the repair. We do this by comparing each AST node of the repaired AST to the corresponding node of the original AST (as determined by the $LocToOrigAST$ table).

Let us say that an AST node is a *regen* node if either it is modified or its location overlaps a *regen* node. (E.g., all nodes of a macro expansion share the same location, so if a node of a macro expansion is modified, then all nodes of the macro expansion must be marked as *regen* nodes.)

Let us say that a *regen* node is a *top-level regen node* if it is not the descendent of any other *regen* node. For each top-level regen node $node$, we replace the original source (as indicated by the L tag of $node$) with the result of generating source-code text from $node$ in the usual manner of generating source code from an AST node, except we re-use the original source-code text of an AST subnode if neither it nor any of its descendents are marked as *regen*. (Thus, macro uses that don't need repairs are preserved verbatim.) We use heuristics to add line breaks, semicolons, and indentation where appropriate.

If a header file is included multiple times via the "`#include`" directive, we create a repaired version for each inclusion point. If each repaired version is identical, we write this common repaired version as the repaired header file. If there are differences, we write a separate header file for each different version and flag it for manual review.

In order to be able to apply our tool again to code that has already been repaired, we implement a defattening operation, which converts all fat pointers back to raw pointers.

VII. Evaluation

We have implemented our approach in a tool called ACR. To evaluate ACR we used C programs from the Software Verification Competition (SVCOMP) [18] and programs from the SPEC CPU2006 (SPEC2006) benchmarks [19] for testing the performance of our approach on larger projects. Our evaluation aims to answer the following questions:

Q1. Can ACR automatically repair spatial memory bugs?
Q2. How much is the code bloat of the repaired code?
Q3. What is the memory and time overhead of the repair?
Q4. What is the overhead when repairing only locations flagged by a program analysis tool?

To answer these questions, we performed a series of experiments on these benchmarks.[3] All experiments are conducted on a laptop with a 2.3 GHz Intel Core i5 and 8 GB of memory, running Ubuntu 18.04 operating system in a Docker container. The original and repaired programs were compiled with `gcc` 7.5.0 using optimization setting "`-O3`".

A. Benchmarks, Verification, and Static Analysis Tools

We collected 52 programs from SVCOMP that only have spatial memory bugs such as invalid pointer dereferences. These programs range from 15 to 153 lines of code, having 39.1 lines of code on average. Even though these C programs are small in size, they have common bugs made by programmers and are used to test the capabilities of verification tools.

To evaluate that ACR generates memory-safe repairs, we used the verification tool SYMBIOTIC [16], [20]. This tool is one of the best verification tools for memory safety in the Software Verification Competition of 2019 and 2020. In this work, we consider the 2020 version of SYMBIOTIC available at the SVCOMP website. SYMBIOTIC uses KLEE [21] as its symbolic engine to check the verification conditions of memory safety. Note that SYMBIOTIC does not report incorrect results since it does not use any over- or under-approximations. However, since SYMBIOTIC uses symbolic execution it does not scale for programs with infinite paths since these cannot be fully symbolically executed in finite time. For these programs, SYMBIOTIC will not be able to prove memory safety and is only useful to detect bugs that have shallow executions.

To evaluate the overhead of ACR on larger programs, we used the bzip2, mcf, sjeng, and libquantum programs from the SPEC2006 benchmarks. These programs contain multiple files and have between 1,571 and 10,515 lines of code. They cover a wide range of applications, from compression (bzip2), combinatorial optimization (mcf), playing chess (sjeng), to quantum computing (libquantum) and are easy to benchmark since they include inputs for which they can be tested.

To reduce the overhead of the repairs generated by ACR, we also support partial repairs where we only insert bounds checks for program locations flagged by a static analysis tool as a potential memory violation. To evaluate the impact of partial repairs, we run Coverity version 2018.01 [22], a state-of-the-art commercial static analyzer that can flag locations with potential NULL pointer dereferences and buffer overflows.

B. Repair of spatial memory bugs in SVCOMP programs

We ran SYMBIOTIC with a time limit of 900 seconds per program. Table I shows the result of SYMBIOTIC which can be one of the following: "Safe" if SYMBIOTIC was able to prove memory safety, "Buggy" if a bug was found, "Unknown" if SYMBIOTIC could not prove memory safety or find a bug due to limitations of the verification tool, and "Timeout" if SYMBIOTIC did not terminate within 900 seconds. SYMBIOTIC found bugs in the majority of the original files. For the repaired

[3] Logs and benchmarks of the experiments are available for download at https://figshare.com/s/394170207df298856651. ACR's source code will be made publicly available at https://github.com/cmu-sei.

TABLE I: Verification status returned by SYMBIOTIC

	#Safe	#Buggy	#Unknown	#Timeout
Original	0	48	1	3
Repaired	27	0	13	12

TABLE II: Code bloat in SPEC2006 programs

	Original		Repaired	
	LOC	Bytes	LOC	Bytes
bzip2	5,720	244,869	7,251	416,370
mcf	1,571	63,348	1,600	81,194
sjeng	10,515	305,311	10,215	398,563
libquantum	2,589	102,143	2,532	122,585

versions, SYMBIOTIC proved memory safety of 27 benchmarks and did not find any bugs in the remaining ones. This increases our confidence in the correctness of ACR implementation. Since proving safety is harder than finding bugs, SYMBIOTIC had more timeouts for the repaired files.

The repair time of ACR is negligible for these small benchmarks. ACR takes between 0.4 and 1.2 seconds, with an average time of 0.7 seconds to repair each of these benchmarks.

C. Repair of spatial memory bugs in SPEC2006 programs

To have a better understanding of the overhead of the code repaired by ACR in terms of code bloat, partial repairs, time and memory performance, we analyzed the impact of running ACR on larger programs from the SPEC2006 benchmarks. We did not analyze these programs with SYMBIOTIC since SYMBIOTIC doesn't scale to the programs this large.

Code bloat. Table II shows the code bloat of the repair, measured in both lines of code (LOC) and bytes of source code. In terms of lines of code, all except bzip2 have less than a 5% change in the number of lines of code. Due to expansion of macros, repairing bzip2 increased the number of lines of code by 27%. On sjeng and libquantum, the repair decreased the number of lines of code; this is due to formatting changes, e.g., in function definitions, the return type of a function was rewritten to be on the same line as the name of the function, rather than its own line. The number of bytes increases between 20% (libquantum) and 70% (bzip2). In the case of bzip2, the size of the repaired file increased substantially since ACR needed to expand macros to repair them. Also, our current implementation fattens every local variable of pointer type. In future work, we will avoid fattening variables where unnecessary for bound checks (when repairing only those lines flagged by the external static analyzer).

TABLE III: Running time of ACR

	Src-to-AST	Repair	AST-to-Src	Total
bzip2	11.85	43.76	8.27	63.88
mcf	11.91	12.90	1.29	26.10
sjeng	15.65	55.78	8.16	79.81
libquantum	11.26	16.25	2.05	29.56

TABLE IV: Number of locations flagged by a static analyzer that correspond to bounds check

Program	bzip2	mcf	sjeng	libquantum
Flagged LOC	10	5	9	5

Running time of ACR. Table III shows the running time of ACR. When running ACR on a larger project, we distinguish the running time of three different phases done by ACR: (i) parsing the source code into AST (Src-to-AST), (ii) repairing the AST (Repair), (iii) transforming the repaired AST into repaired source code (AST-to-Src). The repair time of ACR dominates the running time of the tool. (This includes the time for transforming the AST to IR and then transforming the repaired IR back to AST.) For instance, to repair bzip2, ACR took 63.88 seconds, where 11.85 seconds were spent to parse the source code into AST, 43.76 seconds to perform the repair and 8.27 seconds to transform the repaired AST into repaired source code. Note that ACR is written in Python and not optimized for performance but even in its current form can repair several thousands of lines of code in a few minutes.

Partial repairs. ACR can optionally take as input a set of lines of code flagged by an external static analyzer; ACR will limit insertion of bounds checks to only those lines that are flagged. To evaluate the performance of ACR when considering partial repairs, we ran a commercial static analyzer to flag locations with potential NULL pointer dereferences[4] and buffer overflows. Table IV shows the number of locations that were flagged and correspond to bound checks that can be enforced by ACR. We can observe that for these programs, the number of flagged locations is relatively small, which avoids inserting potentially superfluous bound checks.

It may be noted that the partial repair of mcf added a significant runtime overhead even though bounds checks are added to only a small number of memory accesses. This is because we fattened all pointers, regardless of whether the bounds metadata was actually needed for a bounds check. The mcf program spends a lot of time traversing a large linked list, so the increased size of pointers significantly increases the transfer of data from RAM to the CPU, and the bandwidth of the memory bus can be a bottleneck. The overhead can be significantly decreased by fattening only pointers that actually need to carry the bound metadata. However, the full-fattening approach that we use here might be more representative of very large programs, since only a single flagged memory access is necessary to require fattening of all pointers that can feed into the flagged memory access. Using a compressed pointer format (e.g., [10], [23]) may reduce runtime overhead.

Time overhead. We analyzed the time overhead of the repair by running each program in the testing input included in their distribution. The running times vary between 39 seconds for compressing a text file with bzip2 and 304 seconds for

[4] Dereferencing NULL directly will be caught by hardware on x86, but won't necessarily be caught if NULL is used as the base address of an array in conjunction with a large subscript.

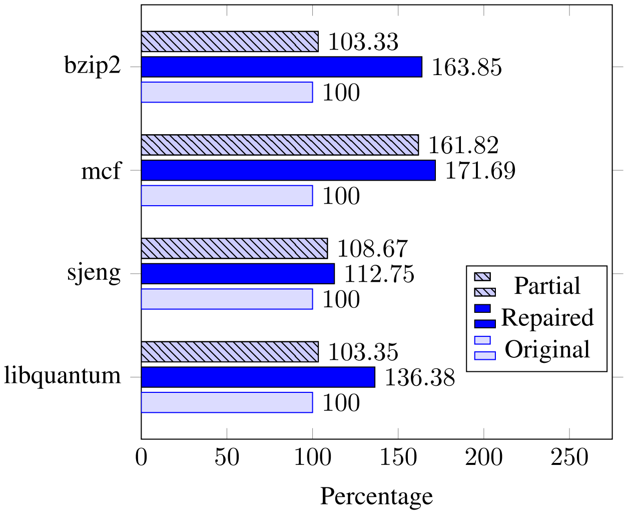

Fig. 7: Time overhead of the repaired and partially repaired source code for the SPEC2006 programs

performing combinatorial optimization using mcf. The time overhead for each program is presented in Figure 7 where we show the percentage of time that each program takes with respect to the original code. The execution time of the repaired version varies between 1.13× and 1.72× the time of the original version. This is roughly similar to times reported by Kroes et al. [11] for SoftBound (1.67×) and ASan (1.80×). Programs that make extensive use of pointers, such as mcf, have their performance more deteriorated when compared to programs that make fewer memory accesses.

Memory overhead. We analyzed the memory overhead on these programs and observed that for some programs (e.g., bzip2 and sjeng), the repaired program has less than a 1% increase in memory allocation over the original program. However, for programs that use lists extensively, there is a significant increase in memory usage. For instance, the libquantum program uses 1.4× more memory and the sjeng program uses 2× more memory than the original program.

VIII. Limitations and Discussion

ACR can already repair a diversity of C programs and protect against spatial memory bugs. However, the current prototype could be improved by addressing some of its limitations.

C language. We cannot fatten pointers that are stored in memory accessible by external code (e.g., libraries and dynamically loaded code). Pointers subjected to non-standard pointer manipulation (e.g., XOR linked lists) or type punning (e.g., to access individual bytes of a pointer stored in memory) also preclude fattening. We also do not prevent memory corruption due to race conditions in concurrent code.

Benchmark selection. The benchmarks we use may not be representative of real-world programs. Even though we used small programs to validate the memory safety repairs performed by ACR, we also analyzed the performance of ACR on the SPEC2006 programs which are closer to real-world programs than the small examples used for program verification. Although these programs are still far from the size of industrial programs, we expect the performance of ACR to scale to larger programs while maintaining the same overheads that were observed while repairing the SPEC2006 programs.

Code readability. One of our goals is to maintain the repaired code as similar as possible to the original code. Even though the changes to the code are small, it may be that the repaired code is harder to read and modify. As future work, we propose to perform a user study to validate our hypothesis that the repaired code is easy for programmers to modify. We will also consider code readability metrics [24] and extend them to our domain to show the maintainability of the repaired code.

IX. Related Work

A. Memory safe languages

A way to guarantee memory safety is to write code in a language that is memory safe. Cyclone [2] and Deputy [3] are examples of typesafe dialects of C. Similarly to our work, Cyclone also uses fat pointers to support pointer arithmetic. In contrast, Deputy does not change the pointer layout but requires the programmer to describe the bounds using program expressions. However, these approaches require the code to be written in these languages and cannot be used for legacy code.

CCured [4] uses whole-program analysis to divide the pointers into 3 kinds: (1) safe pointers that are never involved in pointer arithmetic or unsafe casts, (2) sequential pointers that are involved in pointer arithmetic but not unsafe casts, and (3) dynamic pointers that may be involved in both pointer arithmetic and unsafe casts. The C language is extended to support these pointers, represented internally as fat pointers.

Checked C [5] has been recently proposed as an extension to C that is designed to support spatial memory safety. Checked C does not change the pointer layout and introduces checked pointers that can be used by a programmer to specify single objects, arrays, and NULL-terminated arrays, where the bounds are specified explicitly. These bounds are used by the compiler to prove that a given memory access is safe; otherwise, a run-time bound check is inserted.

B. Pointers to assure memory safety

Ensuring memory safety by extending the structure of pointers to track the bounds of objects and checking if pointer arithmetic stays in bounds is a common approach to guarantee memory safety [6], [7], [8], [9], [10], [11], [12], [13]. For instance, Austin et al. [6] extended the notion of pointers to fat pointers and developed a source-to-binary compiler pass that inserts bounds checks using fat pointers. Alternatively, instead of changing the structure of the pointers, one can encode the bound information directly in the memory layout of the pointer [8], [12], [10], [11]. For instance, low-fat pointers [10] restrict the possible allocation sizes to a fixed finite set and allocate same-sized objects in one region of virtual address space. Another approach is done by SoftBound [13] where bounds are stored in a separate memory region. Delta Pointers [11] are also based on pointer tagging with modifications to the pointer structure to perform the overflow check at the hardware level, which allows it to be more efficient than other pointer tagging approaches.

The run-time and memory overhead of approaches based on modified pointers were investigated by Kroes et al. [11]. Run-time overheads are 67% for SoftBound and 35% for Delta Pointers [11]. This overhead is comparable to the overhead we observed when using our approach. Memory overhead ranges from 64% for SoftBound to 0% for Delta Pointers. Even though our memory overhead is large for programs that use a lot of pointers such as lists, it has a small memory overhead for programs that are based on arrays.

The main difference between our approach and the ones described above is that we insert bound checks at the source code level instead of using a compiler pass. Repairing the source code instead of relying on the compiler has several advantages as discussed in Section II.

C. Source-code transformations

As mentioned earlier, Checked C [5] allows the programmer to use an extension of C with pointers that are memory safe. Existing C code can be intermixed with Checked C, but the plain-C parts do not benefit from the Checked C memory-safety protections. In an attempt to ease the transition to pure Checked C, an automatic porting from C to Checked C code has been recently proposed [25]. However, only between 26% and 45% of the pointers [25] were reported to be automatically converted to their safe variants and the transformed source code will have a mixture of safe and unsafe pointers.

Shaw et al. [26] identify some unsafe library functions related to strings and memory and propose to replace them with alternative safe functions. Additionally, they also use fat pointers for strings to prevent memory violations for strings. The code transformation is performed with the OpenRefactory/C [27] refactoring tool. This approach is limited since it can only prevent memory violations of a small fraction of functions and types and has no memory safety guarantees with respect to the entire program.

X. Conclusion

There has been much research on securing C programs against memory violations, but most of it has focused on doing repairs at compile-time, which has limitations that can make it less desirable than a repair of the source code. We have presented a technique for repairing the original source using fat pointers, and we have validated it on SVCOMP and SPEC2006 programs. As might be expected, programs with extensive traversals of large recursive data structures suffer the worst performance overhead. On programs where arrays dominate, the performance impact is more acceptable, particularly when bounds checks are limited to flagged source-code locations.

Acknowledgements

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM20-0206

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy (SP)*, 2013.

[2] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX ATC*, 2002.

[3] F. Zhou, J. Condit, Z. R. Anderson, I. Bagrak, R. Ennals, M. Harren, G. C. Necula, and E. A. Brewer, "SafeDrive: Safe and recoverable extensions using language-based techniques," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[4] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," in *POPL*, 2002.

[5] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: making C safe by extension," in *IEEE Cybersecurity Development (SecDev)*, 2018.

[6] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *PLDI*, 1994.

[7] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *ACM Symposium on Foundations of Software Engineering*, 2004.

[8] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009.

[9] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *ACM Symposium on Code Generation and Optimization*, 2012.

[10] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *ACM Conference on Compiler Construction*, 2016.

[11] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: buffer overflow checks without the checks," in *Proc. European Conference on Computer Systems*. ACM, 2018, pp. 22:1–22:14.

[12] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: memory safety for shielded execution," in *European Conference on Computer Systems (EuroSys)*, 2017.

[13] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," in *PLDI*, 2009.

[14] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, p. 5665, 2019.

[15] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: a survey," in *ICSE*, 2018.

[16] M. Chalupa, J. Strejcek, and M. Vitovská, "Joint Forces for Memory Safety Checking," in *Symposium on Model Checking Software*. Springer, 2018, pp. 115–132.

[17] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX ATC*, 2012.

[18] "Software Verification Benchmarks," https://github.com/sosy-lab/sv-benchmarks/, 2020, [Online; accessed 14-January-2021].

[19] "SPEC CPU2006," https://www.spec.org/cpu2006/, 2006.

[20] M. Chalupa, M. Vitovská, and J. Strejcek, "SYMBIOTIC 5: Boosted Instrumentation - (Competition Contribution)," in *TACAS*, 2018.

[21] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." USENIX Association, 2008, pp. 209–224.

[22] "Coverity," https://scan.coverity.com/, 2020, [Online; accessed 14-January-2021].

[23] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos *et al.*, "CHERI concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, 2019.

[24] R. P. L. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Trans. Software Eng.*, vol. 36, no. 4, pp. 546–558, 2010.

[25] A. Ruef, L. Lampropoulos, I. Sweet, D. Tarditi, and M. Hicks, "Achieving safety incrementally with Checked C," in *Conference on Principles of Security and Trust*. Springer, 2019, pp. 76–98.

[26] A. Shaw, D. Doggett, and M. Hafiz, "Automatically fixing C buffer overflows using program transformations," in *Conference on Dependable Systems and Networks*. IEEE Computer Society, 2014.

[27] M. Hafiz, J. Overbey, F. Behrang, and J. Hall, "OpenRefactory/C: an infrastructure for building correct and complex C transformations," in *Proc. Workshop on Refactoring Tools*. ACM, 2013, pp. 1–4.