Using Safety Properties to Generate Vulnerability Patches

Zhen Huang*†, David Lie†, Gang Tan*, Trent Jaeger*
* Pennsylvania State University † University of Toronto
zhen.huang@psu.edu, lie@eecg.toronto.edu, gtan@cse.psu.edu, tjaeger@cse.psu.edu

Abstract

Security vulnerabilities are among the most critical software defects in existence. When identified, programmers aim to produce patches that prevent the vulnerability as quickly as possible, motivating the need for *automatic program repair* (APR) methods to generate patches automatically. Unfortunately, most current APR methods fall short because they approximate the properties necessary to prevent the vulnerability using examples. Approximations result in patches that either do not fix the vulnerability comprehensively, or may even introduce new bugs. Instead, we propose *property-based* APR, which uses human-specified, program-independent and vulnerability-specific safety properties to derive source code patches for security vulnerabilities. Unlike properties that are approximated by observing the execution of test cases, such safety properties are precise and complete. The primary challenge lies in mapping such safety properties into source code patches that can be instantiated into an existing program.

To address these challenges, we propose Senx, which, given a set of safety properties and a single input that triggers the vulnerability, detects the safety property violated by the vulnerability input and generates a corresponding patch that enforces the safety property and thus, removes the vulnerability. Senx solves several challenges with property-based APR: it identifies the program expressions and variables that must be evaluated to check safety properties and identifies the program scopes where they can be evaluated, it generates new code to selectively compute the values it needs if calling existing program code would cause unwanted side effects, and it uses a novel *access range* analysis technique to avoid placing patches inside loops where it could incur performance overhead. Our evaluation shows that the patches generated by Senx successfully fix 32 of 42 real-world vulnerabilities from 11 applications including various tools or libraries for manipulating graphics/media files, a programming language interpreter, a relational database engine, a collection of programming tools for creating and managing binary programs, and a collection of basic file, shell, and text manipulation tools.

I. Introduction

Fixing security vulnerabilities in a timely manner is critical to protect users from security compromises and to prevent vendors from losing user confidence. A recent study has shown that creating software patches is often the bottleneck of fixing security vulnerabilities [19]. As a result, an entire line of research has studied Automated Program Repair (APR) [16], [20], [26], [27], [30], [34], [39], [40], [54], [55], which takes a program and a vulnerability and automatically provides a patched program that fixes the vulnerability.

Like patches produced by human developers, patches generated by APR tools aim to fix a defect or vulnerability. Most existing tools rely on a set of positive/negative example inputs to find a patch that makes the program behaves correctly on those examples. They check if the patched program satisfies the positive example inputs but result in errors on the negative example inputs [30], [34], [36]. However, it is often difficult to obtain a complete set of example inputs, and the patched program may not behave correctly on other inputs or the vulnerability that the patch intends to fix may still be exploitable given other inputs. Therefore, they can result in generating patches that either do not fix the vulnerability, or even worse, introduce new bugs. We call this traditional method "example-based".

In this paper, we advocate a different approach, which we term "property-based" APR. Property-based APR relies on program-independent, vulnerability-specific, human-specified safety properties. An example of such a safety property might be that a program should not access beyond the end of a buffer to rule out a buffer overflow vulnerability. The advantage of this property-based approach is that a small set of safety properties can be specified once and used on a vast number of programs without the need to specify anything specific about each of the programs, or collect a comprehensive set of test cases. Moreover, such properties are inherently precise and complete. In comparison to example-based APR, property-based APR produces patches that work for all possible inputs; this is especially important for security, which requires us not leave holes for attackers.

Property-based APR differs from previous tools that enforce a safety property comprehensively. Instead, APR enforces each property with respect to a particular vulnerability and thus can take advantage of the vulnerability's context to generate an efficient and custom patch. For instance, SoftBound [32] instruments a program to enforce memory safety on all buffers and often results in high performance overhead. Property-based APR in contrast takes a specific vulnerability (e.g., a buffer-overflow vulnerability) as input and generates a patch that targets that vulnerability.

The challenge of property-based APR boils down to developing a method that ensures that the appropriate safety property is enforced. Unfortunately, property-based APR has received limited attention in the literature [24], and has several outstanding challenges that limit its applicability.

This work is mostly done while the first author was a Ph.D candidate in the Department of Electrical and Computer Engineering, University of Toronto.

First, because the safety properties enforced are vulnerability-specific, the APR tool must identify the correct property to enforce for a given vulnerability. In this work, we assume that the input to the APR tool is only an input that triggers the vulnerability, possibly crashing the program. As a result, the APR tool must also correctly identify the vulnerability and the corresponding safety property to enforce.

Second, our goal is to generate source code patches that can be adopted by developers. Because the safety properties are generic and program-independent, they must be mapped onto the variables in a program to generate a source code patch. However, sometimes not all program constructs corresponding to a safety property are available in the same program scope. For example, the size of a buffer may be stored in a variable available only in the function that allocates the buffer but not in the function accesses the buffer. This requires a program analysis that is able to generate equivalent expressions for those safety property and select an intersecting set that is in scope at a point in the program.

Third, some terms in a safety property may have to be mapped onto expressions involving not only program variables and constants, but also function calls. For example, a program may always calculate the size of an object by making a function call because the size is dynamic. Because function calls may have side-effects, a patch must be careful not to call any function with side-effects. As a result, an APR tool need to check if a function has side-effects and may even need to generate *new* functions in a program that compute required values without introducing unwanted side effects.

Finally, many vulnerabilities depend on the number of times a loop iterates. A naïve approach would simply check the safety property on each loop iteration, resulting in performance overhead. To reduce the performance impact of generated patches, an APR tool should be aware of loops and generate a patch that checks the safety property once outside of the loop to avoid performance overhead.

In this paper, we propose Senx, which addresses the above challenges to generate source code patches for security vulnerabilities automatically using such vulnerability-specific safety properties. Although Senx can in theory generate patches for vulnerability for which a safety property can be specified, we demonstrate Senx for three important classes of vulnerabilities: buffer overflow, bad cast, and integer overflow. We find that Senx is able to produce correct patches for over 76% of the vulnerabilities. The main reason Senx fails to generate a patch for the rest is that it is unable to find a place in the program source code where all variables needed to evaluate the safety property are in scope, which would require changes to function prototypes to allow those variables to cross those scopes.

This paper makes the following main contributions:

```
char* rev(const char *inp, char *out) {
    // reverse a string
    //  inp is the input string
    //  out is an output buffer
    if (inp != NULL) {
        int i, len = strlen(inp);
        // Failed to check if (len + 1 <=
            size_of_out)
        for (i = 0; i < len; i ++)
            out[i] = inp[len - i];
        out[i] = '\0';
        return out;
    }   else
        return "###";
}

void main(int argc, char *argv[]) {
    int size = atoi(argv[1]) + 1;
    char *out = (char *)malloc(size);
    // patch: if (strlen(argv[2]) + 1 > size) ...
    printf("%s\n", rev(argv[2], out));
}
```

Listing 1: A program that reverses an input string. It contains a buffer overflow in function `rev`.

- We describe how safety properties are specified in Senx and demonstrate three example safety properties for buffer overflow, bad cast, and integer overflow vulnerabilities.
- We describe the design of Senx, a property-based automatic patch generation system that uses novel program analysis techniques: expression translation, loop cloning, and access range analysis.
- We prototype Senx on top of the KLEE symbolic execution engine and evaluate it on a corpus of 42 vulnerabilities across 11 popular applications, including PHP interpreter, sqlite database engine, binutils utilities for creating and managing binary programs, and various tools or libraries for manipulating graphics/media files. Senx generates correct patches in 32 of the cases and aborts the remainder because it is unable to determine semantic correctness in the other cases. The evaluation demonstrates that all three techniques are required to generate patches, and that failure to find a common function scope in which to place a patch is the most frequent reason for failure.

The structure of this paper is as follows. We motivate our work in Section II. In Section III, we define and characterize the problem Senx addresses. Section IV and Section V describe the design and implementation of Senx respectively. Particularly we describe how Senx addresses the second challenge in Section IV-D, the third and fourth challenges in Section IV-C. We present evaluation results in Section VII and discuss related work in Section VIII. Finally we conclude in Section IX.

II. Motivation

We discuss the limitations of state-of-art automatic patch generation tools that Senx aims to address in this section. We use the program in Listing 1 as the target program, which is adopted from a real-world buffer overflow vulnerability CVE-2012-0947 in a popular media stream processing library [42].

This program reverses an input string. It takes two inputs from the command line, a string and an integer that specifies

Type	argv[1]	argv[2]	output	expected output
P	1	A	A	A
P	2	AB	BA	BA
N	1	ABC	CBA	###
N	2	ABC	CBA	###

TABLE I: Test inputs and outputs for the program in Listing 1. Type 'P' test inputs are positive test inputs, while type 'N' test inputs are negative test inputs.

the length of the string, and outputs the reversed string. If an error occurs, the program outputs "###". To do this, it dynamically allocates a temporary buffer based in a value passed to it, and copies the input into the buffer. Like the real vulnerability, the allocation of the output buffer and the processing of the input string are implemented in two different functions. This example is typical of programs that process audio/video streams.

The buffer overflow occurs when the specified length, from the input integer, is smaller than the actual length of the input string. The buffer overflow can be fixed by adding a check that enforces that the actual length of the string is smaller than the allocated size of the buffer it is being copied into. Because the buffer size is only known in `main`, the check should be added at line 19 and compare `size` with `strlen(argv[2])`. While such checks are easily added by human developers to patch vulnerabilities — indeed such a check is found in the human-generated patch for the vulnerability this code example is based on [42] — it poses challenges for current APR tools. We describe these challenges in more detail below.

Example-based approaches. Many APR tools use example inputs as the basis for fixing vulnerabilities [29], [30], [34], [55]. For example, SemFix and Angelix collect path constraints to generate fixes [30], [34].

This approach leads to two problems. First, the constraints generated often only capture constraints based on the concrete values used in the test cases and not constraints on the relationships between program variables. To illustrate, Table I lists typical test inputs needed to use such tools, which we imagine might be used by such tools for our example in Listing 1.

Given these test cases, SemFix and Angelix would see that `argv[1]` having a value of `1` or `2` are not correlated with the negative test cases, since they take on those values in both positive and negative test cases. Thus, it would incorrectly infer that `strlen(argv[2]) < 3` needs to be added to the code for it to be correct. The incorrect patch is generated because the suite of test cases did not include a positive test case with `strlen(argv[2]) > 2`. This illustrates the shortcoming of example-based systems as they can easily fall prey to missing cases in the test suite, which are notoriously difficult to make complete.

Property-based approaches. AutoPaG [24] also uses a safety property-like predicate to create patches. However, AutoPaG only handles one type of vulnerability, buffer overflows, so it does not need to identify the type of vulnerabilities to enforce the appropriate safety property—it fails to generate a correct patch if the vulnerability is any type other than a buffer overflow.

In addition, AutoPaG cannot generate a patch if the location that the safety property needs to be enforced is not in the same function where the vulnerability occurs. In our example, the buffer overflow occurs in the `rev` function, but the patch must be places in `main`.

Finally, AutoPaG enforces its safety property by instrumenting the code at runtime. In Listing 1, AutoPaG would instrument and check the buffer size inside the `for` loop on line 8, causing performance overhead. In contrast, Senx symbolically extracts the memory range the loop accesses by analyzing the loop bounds, allowing it to check the safety property outside of the loop.

III. Problem Definition

We begin by defining what a Senx patch is, what guarantees a Senx patch provides and how Senx's vulnerability-specific safety properties are defined.

A. Patch

To generate a patch, Senx requires an input that can trigger the target vulnerability. Typically, this is the type of input that one could derive from a proof-of-concept exploit or an input generated by a fuzzer that can crash the program. From this, Senx generates a patch that ensures that all safety properties it supports hold, where each safety property corresponds to a particular type of vulnerability. A Senx patch can take one of two forms: a) detects if a safety property no longer holds and if so, raises an error to direct program execution away from the path where the vulnerability resides (we call this a check-and-error patch); b) prevents a safety property from being violated (we call this a repair patch).

B. Safety Properties

Each safety property corresponds to a vulnerability type, and is an abstract boolean expression that when mapped to concrete variables in a program can be evaluated. We describe the two types of safety properties Senx currently supports.

Buffer overflows. A buffer overflow occurs when a series of memory accesses traversing a buffer crosses from a memory location inside the buffer to a memory location outside of the buffer. The corresponding Senx safety property defines two abstract values: a memory access and a buffer. Senx uses the term buffer to refer to any bounded memory region, which may include structs, objects or arrays. A memory access may correspond to an array dereference or pointer dereference, but must occur inside a loop. This safety property covers both the case when the memory access exceeds the upper range of the buffer and the case when the memory access falls below the lower range (sometimes called a buffer underflow).

Bad casts. This safety property checks that a memory accesses that results from a offset from a base pointer is less than the upper bound of the buffer the base pointer is pointing to. While such a vulnerability may occur for a variety of reasons, it

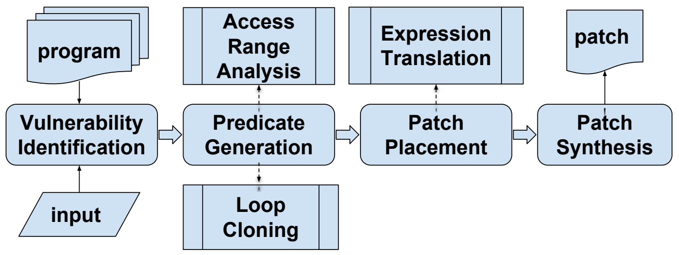

Fig. 1: Workflow of Senx: each rounded rectangle represents a step in Senx's patch generation, each rectangle with vertical bars represents a component of Senx.

is most often caused when a programmer mistakenly casts a pointer to a type that is incompatible with the object the pointer points to. This safety property can prevent both bad casts for simple structs and objects, as well as nested structs and objects.

Integer overflows. An integer overflow occurs when a variable is assigned a value larger or smaller than what can be represented in the variable. A vulnerability from an integer overflow can occur if the result of an overflow is then used to allocate a buffer, resulting in a buffer that is far smaller than expected. As a result, the corresponding safety property checks that value used in a memory allocation is not the result of an integer overflow.

As a prototype, we have started with only these three vulnerability classes. Nevertheless, they are representative enough to capture a good percentage of CVE vulnerabilities. We conducted an informal analysis on the CVEs reported in 2018 [15]. At the time of our analysis, 8,507 vulnerabilities have been reported and the most popular vulnerability categories are denial of service, code execution, and overflow. By randomly examining 100 CVE reports for each of the three vulnerability categories, we find that 25% of CVE vulnerabilities fall into buffer overflow, bad cast, and integer overflow. We believe the principles behind Senx can be extended to other vulnerability classes, which we intend to do in future work.

IV. Design

A. Overview

The goal of Senx is to generate source code patches that can be easily verified and adopted by developers. Senx generates patches in four distinct steps: vulnerability identification, predicate generation, patch placement, and patch synthesis, as illustrated in Figure 1.

First, during *vulnerability identification*, Senx takes a program and an input that can trigger a vulnerability, and outputs the violated safety property, the *vulnerability point* [4] in the program, and the source code expressions for the execution trace. Senx leverages concolic execution using the vulnerability-triggering input to generate the execution trace for the input. Senx then checks the execution trace to identify which safety property is violated and the point at which safety property is violated, the vulnerability point, which also serves as the candidate location for the patch. Because not every program operation affects safety properties, Senx checks safety properties only for the set of operations that can affect them. For example, Senx checks for overflow only on integers that affect the size of for memory allocations. The execution trace is transformed into expressions for synthesizing a source-level patch. Since we aim to patch the source code, these expressions must conform to the syntax of the programming language of the target program.

Second, in *predicate generation*, Senx takes the violated safety property, which also implies the type of the vulnerability, and the source code expressions generated by vulnerability identification, and outputs a predicate required to prevent the safety violation. Predicate generation maps the safety property identified in the previous step to concrete expressions over variables and function calls in the source code of the program. Thus, the resulting predicate represents the safety property in terms of source code expressions suitable for synthesizing the source level patch.

Third, during *patch placement*, Senx uses the vulnerability point found in vulnerability identification and the predicate produced in predicate generation to find a program location to insert the patch. This step solves the problem of finding a program location where all necessary variables in the predicate are in scope. Senx uses *expression translation*, described in Section IV-D, to translate all necessary variables into the common scope. For check-and-error patches, Senx requires the current scope to have some error handling code to call. It uses Talos [19] to find and select the error handling code.

Finally, in *patch synthesis*, Senx takes the patch location and the instantiated predicate to generate the patch code. Currently the only vulnerability class for which a repair patch can be generated are integer overflows. In all other cases, Senx generates a check-and-error patch that checks the patch predicate and calls the error handling code if the predicate evaluates to true.

B. Vulnerability Identification

The concolic execution engine Senx uses during vulnerability identification is based on KLEE [7]. Senx executes the program in LLVM IR instructions with the vulnerability-triggering input until a safety property is violated. It labels this point the vulnerability point. The main output of this phase is the vulnerability point, the violated safety property, and a set of symbolic expressions that represent the symbolic values of the variables in the executed program.

We modified KLEE's LLVM IR execution engine to extract expressions that retain enough information to be easily translated back into source code. For the most part, this involves storing the source code symbol names along with the LLVM IR instructions.

Senx also extends KLEE with support for complex data types such as C/C++ structs, making it possible to translate symbolic expressions back into source-code syntax for patch generation. For example, a field of a struct must be attached to its parent object, and the generated syntax changes depending on whether the parent object is referenced using a pointer or with a variable holding the actual object. Arrays and structs can also be nested and the proper syntax must be used to denote the level of nesting relative to the top level object. Senx has its own IR that records these relationships so a value can be converted back to a source code expression that contains the parent struct or array's name and not just the field name or array index. In this way, Senx can recover the full expression for a variable such as `foo→f.bar[10]`, where as without these extensions, Senx would only know that the variable corresponds to the 10th element in an array.

C. Predicate Generation

The objective of predicate generation is to take the abstract boolean expression of the violated safety property identified in the previous phase and map it to concrete expressions in the program source code. Senx uses the variables that violated the safety property at the vulneraility point as a starting point for generating the predicate.

The case of the integer overflow property is the simplest. In this case, the property is violated when an integer that has overflown is used to determine the size of a buffer allocation. As a result, Senx generates a predicate that would prevent the integer variable from overflowing. There are two possible causes for such a vulnerability. In the first case, the program assigns the result of an overflowed operation into a variable whose size is larger than the size of the operation (e.g., assigning the result of an overflowed 32-bit multiplication into a 64-bit variable). In this case, Senx generates a repair patch by modifying the type of the operation so that the overflow does not happen (e.g., changing the 32-bit multiplication into a 64-bit multiplication). In the second case, there is no such assignment. So Senx generates a check-and-error predicate that checks whether the result is smaller than expected (for example, if the result of an addition is smaller than either of its inputs).

A predicate to detect a bad cast takes the form

$$(mem\_access > \mathit{buffer\_upper} || mem\_access < \mathit{buffer\_lower})$$

To generate this predicate, Senx must map mem_access to a source code expression that gives the address of the memory access and develop expressions that capture $buffer\_upper$ and $buffer\_lower$. To extract the expression for the memory access, Senx traverses nested struct or class relationships to generate an expression that includes any parent structs or objects that are needed to generate a proper source code expression. For example, to determine the offset of a field in a nested struct, Senx first determines the offset of the field from its immediate parent struct, and recursively accumulates the offset of each level of nesting until the full offset from the base struct can be determined.

For a bad cast that involves a memory access with a base pointer and an offset, the lower bound of the buffer is given by the base pointer. To generate the expression for the upper bound, Senx needs to find the point where the buffer is allocated so that it can extract the size of the buffer. Senx first tries to match the accessed buffer with an allocation point using intraprocedural path-sensitive dataflow analysis. If this does not resolve to a unique allocation point, Senx refines the analysis using interprocedural call history derived from the execution of the program on the vulnerability-triggering input. Once Senx finds the allocation point, it derives the size from the expression passed to the allocation function (i.e., `malloc` or some variant). This computation is sound if the expressions extracted are sound and the memory allocation is correctly identified, which is true if the application only uses standard memory allocation functions that Senx understands. Our Senx prototype currently only supports standard libc and C++ memory allocation functions.

Finally, the predicate to detect a buffer overflow has the form

$$\begin{aligned}(&mem\_access\_upper > \mathit{buffer\_upper} \quad || \\ &mem\_access\_lower < \mathit{buffer\_lower})\end{aligned}$$

The procedure for extracting the buffer range expressions is based on the bad cast case but with some small differences. Because there is no base pointer for the memory access of a buffer overflow, Senx instead infers which buffer the illegal memory access was likely trying to access by recording the buffer of the last legal access made by the same instruction. The size of the buffer is then computed similarly by finding the allocation point of the buffer. In this case, rather than a `sizeof()` argument being passed to the memory allocation function, it may involve an arithmetic expression passed to the memory allocation function to allocate a variable-size array. This difference does not affect Senx as its goal is to extract the expression and inject it into the source code patch, where it will be compiled and evaluated at runtime by the program.

For the buffer overflow predicate, the other requirement is to extract expressions for the range of memory access in the loop that issues a set of sequential memory access. This is more complex because the number of iterations the loop can execute may vary. To compute the memory access range, Senx uses two complementary loop analysis techniques: *loop cloning* and *access range analysis*. Both loop cloning and access range analysis take as input a function `F` in the target program and an instruction `inst` that performs the faulty access in the buffer overflow, and returns the symbolic memory access range $[A_1, A_n]$ of `inst`. This symbolic access range can then be compared with the allocation range in a generated patch for safety.

Access Range Analysis. Senx relies on LLVM's built-in loop canonicalization functionality [8] to perform access range analysis, which computes the access range of canonicalized loops. Loop canonicalization seeks to convert the loop into a standard form with a pre-header that initializes the loop iterator variable, a header that checks whether to end the loop, and a single backedge. Extracting the access range for a single loop in this way is straightforward. The main difficulty is extending this to handle nested loops.

Access range analysis is implemented for nested loops using the algorithm described in Algorithm 1. It analyzes the loops

```
1 char *foo_malloc(x,y) {
2         return (char *)malloc(x * y + 1);
3 }
4
5 int foo(char *input) {
6+        if ((double)(cols+1)*(size/cols) + 1 >
7+                          rows * (cols + 1) + 1)
8+                return -1;
9         char *output = foo_malloc(rows, cols + 1);
10        if (!output)
11                return -1;
12        bar(p, size, cols, output);
13        return 0;
14 }
15
16 void bar(char *src,int size,int cols,char *dest) {
17        char *p = dest;char *q = src;
18        while (q < src + size)  {
19                for (unsigned j = 0; j < cols; j++)

20                        *(p++) = *(q++);
21                *(p++) = '\n';
22        }
23        *p = '\0';
24 }
```

Listing 2: A buffer overflow CVE-2012-0947 with a patch (prefixed with '+').

enclosing a memory access instruction `inst` in function `F`, starting with the innermost loop and iterating to the outermost, accumulating increments and decrements on the loop induction variables including the pointer used by `inst`.

We use the loop in `bar` of Listing 2 as an example of how Algorithm 1 can be applied to a nested loop. In this case, `bar` is function `F` and `inst` is the memory write using pointer `p` at line 42. For each loop, Sen retrieves the loop iterator variable and the bounds of it by calling helper function `find_loop_bounds`. Senx also gets the list of induction variables of the loop and their *update*, which is the fixed amount that an induction variable is increased or decreased by on each loop iteration by calling another helper function `find_loop_updates`. In our example, we have $iter =$ `j`, $initial = 0, end =$ `cols` and `j` $\mapsto 1,$ `p` $\mapsto 1,$ `q` $\mapsto 1$ in $updates$ for the innermost `for` loop from lines 19-20.

Algorithm 1 then symbolically accumulates the update to each induction variable to a data structure referred to by acc, which maps each induction variable to an expression denoting the accumulated update to the induction variable. As for the example, it will store `j` $\mapsto 1,$ `p` $\mapsto 1,$ `q` $\mapsto 1$ into acc for the innermost for loop. After that, it synthesizes the expression to denote the total number of iterations for the loop. At line 16 of the algorithm, we will have $count =$ `cols` which is simplified from `(cols-0)/1`.

Having the total number of iterations, it multiplies the accumulated update for each induction variable by the total number of iterations. So acc will have `j` $\mapsto$ `cols`, `p` $\mapsto$ `cols`, `q` $\mapsto$ `cols` after the loop from line 18 to 19 in Algorithm 1.

Once this is done, it moves on to analyze the next loop enclosing *inst*, which in Listing 2 is the while loop enclosing the inner for loop. As a consequence, we will have

Algorithm 1 Finding the access range of a memory access.

Input: F: a function
$inst$: a memory access instruction in F
Output: $acc\_initial$: initial address acccessed by $inst$
acc_end: end address accessed by $inst$

```
 1: procedure ANALYZE_ACCESS_RANGE
 2:   ▷ $acc$: accumulated updates to induction variables
 3:   $acc \leftarrow \emptyset$
 4:   $innermost\_loop \leftarrow innermost\_loop(inst)$
 5:   $outermost\_loop \leftarrow outermost\_loop(inst)$
 6:   $visited \leftarrow \emptyset$
 7:   for $l \in [innermost\_loop, outermost\_loop]$ do
 8:     $iter, initial, end \leftarrow$find_loop_bounds$(F, l)$
 9:     $updates, visited \leftarrow$find_loop_updates$(l, visited)$
10:     ▷ Symbolically add up induction updates
11:     for $var, upd \in updates$ do
12:       $acc\{var\} \leftarrow$ sym_add$(acc\{var\}, upd)$
13:     end for
14:     ▷ Symbolically denote the number of iterations of
        $l$ as $count$
15:     $upd\_iter \leftarrow updates\{iter\}$
16:     $count \leftarrow$sym_div(sym_sub$(end, initial), upd\_iter)$)
17:     ▷ Symbolically multiply induction updates by the
        number of iterations of $l$
18:     for $var, upd \in acc$ do
19:       if $\neg$is_initialized_in_last_loop$(var)$ then
20:         $acc\{var\} \leftarrow$sym_mul$(acc\{var\}, count)$
21:       end if
22:     end for
23:   end for
24:   $ptr \leftarrow$get_pointer$(inst)$
25:   $first\_inst \leftarrow$loop_head_instruction$(outermost\_loop)$
26:   ▷ Find the definition of $ptr$ that reaches $first\_inst$
27:   $acc\_initial \leftarrow$reaching_definition$(F, first\_inst, ptr)$
28:   $acc\_end \leftarrow$sym_add$(acc\_initial, acc\{p\})$
29:   return $acc\_initial, acc\_end$
30: end procedure
```

$iter =$ `q`, $initial =$ `src`, $end =$ `src+size` and `p` $\mapsto 1$ in $updates$ at line 10 of the algorithm, `j` $\mapsto cols,$ `p` $\mapsto cols + 1,$ `q` $\mapsto$ `cols` in acc and $count =$ `size/cols` at line 17 of the algorithm, and finally `j` $\mapsto$ `cols`, `p` $\mapsto$ `(cols+1)*(size/cols)`, `q` $\mapsto$ `size` in acc. Note that the algorithm will not multiply the number of iterations of the loop to `j` because `j` is always initialized in the last analyzed loop, the innermost for loop.

After analyzing all the loops enclosing *inst*, the algorithm gets the pointer *ptr* used by *inst* and performs reaching definition dataflow analysis to find the definition that reaches the beginning of the outermost loop. As for the example, we will have $ptr =$ `p` and the assignment `p=dest` at line 16 of `bar` as the reaching definition for `p`. From this reaching definition, it extracts the initial value of `p`, $acc\_initial =$ `dest`. Finally it gets the end value of p, acc_end = dest+(cols+1)*(size/cols)+1 by adding the initial value dest to the accumulated update of p, (cols+1)*(size/cols) from acc, plus the last write via pointer p at line 22. Hence it returns [dest,dest+(cols+1)*(size/cols)+1] as the expressions denoting the access range $[A_1, A_n]$.

Our access range analysis can be considered as a form of pattern-based loop analysis [14] with several differences. On one hand, access range analysis aims to derive the number of loop iterations as an expression involving program variables and/or constants, while pattern-based loop analysis aims to derive the number of loop iterations as a constant. On the other hand, access range analysis requires loops be normalized to fit the pattern required by pattern-based loop analysis, and relies on loop canonicalization to normalize loops.

Loop Cloning. Access range analysis cannot be applied to loops that LLVM cannot canonicalize. For those loops, Senx uses loop cloning. As an example, consider the loop in Listing 3, where Senx applies loop cloning to produce new code code that preserves the number of loop iterations, but removes code that causes side-effects. The new code can then be used to safely return the access range in the generated patch.

Because the patch must be inserted into a function where both the access range and allocation range are available, loop cloning first searches on the call between the function where the buffer is allocated and the function where the loop resides (i.e. F). If no such function can be found, Senx will not be able to generate a patch. Otherwise, it designates the function it finds $\texttt{F}_p$ and then clones each function $\texttt{F}_i$ along the call chain from F until $\texttt{F}_p$ into the new code that returns the access range. As a result, each $\texttt{F}_i$ is either a direct or indirect caller of F or is F itself.

Loop cloning needs to satisfy two requirements: 1) F must compute the access range and pass the access range to its caller; 2) any direct or indirect caller of F must pass the access range that it receives from its callee upwards to the next function along the call chain. Each $\texttt{F}_i$ is cloned using the following steps.

1) Loop cloning clones the entire code of $\texttt{F}_i$ into $\texttt{F}_i$_clone.
2) Using program slicing, it removes all statements that are not needed in order to compute the access range or pass the access range to $\texttt{F}_p$. If $\texttt{F}_i$ is F, it retains statements on which the execution of inst is dependent. If $\texttt{F}_i$ is a direct or indirect caller of F, it retains statements on which the call to F is dependent.
3) It changes the return type of $\texttt{F}_i$_clone to void and removes any return statement in $\texttt{F}_i$_clone.
4) It adds two output parameters start and end to $\texttt{F}_i$_clone. If $\texttt{F}_i$ is F, it inserts statements immediately before the (nested) loops to copy the initial value of the pointer or array index used in the faulty access into start, and statements immediately after the loops to copy the end value of such pointer or array index into end. If $\texttt{F}_i$ is a caller of F, it changes the call statement to include the two output parameters in the list of call arguments.

```
 1 int decode(const char *in, char *out) {
 2   int i;
 3   char c;
 4   i = 0;
 5   while ((c = *(in++)) != '\0') {
 6     if (c == '\1')
 7       c = *(in++) - 1;
 8     out[i ++] = c;
 9   }
10   return i;
11 }
12 
13 char* udf_decode(const char *data, int datalen) {
14   char *ret = malloc(datalen);
15   if (ret && !decode(data+1, ret)) {
16     free(ret);
17     ret = NULL;
18   }
19   return ret;
20 }
```

Listing 3: A complex loop involving a complex loop exit condition and multiple updates to loop induction variable on multiple execution paths.

```
 1+  void decode_clone(const char *in, char *out, char
         **start, char **end) {
 2     char c;
 3+    *start = in;
 4     while ((c = *(in++)) != '\0') {
 5       if (c == '\1')
 6         c = *(in++) - 1;
 7     }
 8+    *end = in;
 9   }
10 
11   char* udf_decode(const char *data, int datalen) {
12     char *ret = malloc(datalen);
13+    char *start, *end;
14+    decode_clone(data+1, ret, &start, &end);
15     if (ret && !decode(data+1, ret)) {
16       free(ret);
17       ret = NULL;
18     }
19     return ret;
20   }
```

Listing 4: A cloned and sliced loop that no longer contains any statements that have side-effects and returns the number of iterations. Statements prefixed with '+' are added or modified by Senx to count and return the number of loop iterations.

After cloning each $\texttt{F}_i$, loop cloning inserts a call to the last cloned function into $\texttt{F}_p$, which returns the access range in start and end. Subsequently a patch will be synthesized to leverage the returned access range.

To see how loop cloning works, consider the example in Listing 3, which presents a loop adapted from a real buffer overflow vulnerability CVE-2007-1887 [38] in PHP, a scripting language interpreter. The buffer overflow occurs in function decode. The loop features a complex loop exit condition and multiple updates to loop induction variable in that depend on the content of the buffer that in points to. The result of loop cloning is shown in Listing 4. Loop cloning is invoked with decode as F, and the faulty access at line 5 as `inst`. It first finds that function `udf_decode` is on the call chain to `decode` and in which the allocation range is available. Because `udf_decode` directly calls `decode`, it only needs to clone `decode`. It then clones function `decode` into `decode_clone`, after which it applies program slicing to `decode_clone` with line 5 and variable `c` and `in` that are accessed at line 5 as the slicing criteria. `decode` also has a potential write buffer overflow at line 8, but in this example, we focus on generating a predicate that will check whether `in` can exceed the end of the buffer it points to. The program slicing uses a backward analysis and removes all statements that are irrelevant to the value of `c` and `in` at line 5, including line 2, 4 and 8. After program slicing, it changes the return type of `decode_clone` into `void` and removes all return statements. And it adds two output parameters `start` and `end` to the list of parameters of `decode_clone`. Then it inserts a statement at line 3 to copy the initial value of `in` to `start` before the loop and a statement at line 8 to copy the end value of `in` to `end` after the loop. Finally it inserts into function `udf_decode` a call to `decode_clone` at line 14 and a statement to declare `start` and `end` at line 13.

The new code produced by loop cloning must not have any side-effects. If such a side-effect free slice cannot be produced, Senx aborts patch generation.

Function Calls. In some cases, the extracted expressions contain the results of function calls. In such cases, Senx must be careful that it does not call functions in a generated predicate that have side-effects.

Senx defines a side-effect as: 1) a possible change to the memory visible outside of a function, or 2) an invocation of a system call that has external impact, or 3) an invocation of a function that has any side-effect. We refer to a function that has no side-effects as a *safe* function. First, it considers any write to a global variable as a side-effect. To be conservative, it also considers any write via a pointer argument passed to the function as a side-effect. Second, it uses a white-list of common API functions and system calls that have no external impact such as changing file system state or outputting to a device or the network. An invocation of a function or system call on the white-list is considered as having no side-effects.

Senx uses a flow-sensitive, context-insensitive intraprocedural static analysis to identify the list of safe functions. In the beginning, the list contains only the functions on the white-list. Senx performs the analysis for every function of a target program and adds each function that has no side-effects to the list.

For each function, Senx maintains a set of its callee functions that might be "unsafe". Whenever a new function *F* is added to the list, Senx removes *F* from the set of "unsafe" callee functions for each function that calls *F*. When such a set becomes empty for a function, Senx considers the function as "safe" and also adds the function to the list.

D. Expression Translation

Because the patches Senx generates are source code patches, the predicate of the patch must be evaluated in a single function scope. However, sometimes the allocation range is computed in one function scope, while the memory access range is computed in a different function scope. So the expression denoting the allocation range and the expression denoting the memory access range are not both valid in a single function scope. To make the expressions both valid in a single function scope, one possible solution is to send the expression valid in a source function scope as a call argument to a destination function scope where the expression is not valid. This approach requires adding a new function parameter to the destination function, and adding a corresponding call argument at every call site of the destination function. We decided not to use this approach because it requires code changes to any function on the call chain from the source function to the destination function. In addition, unrelated functions that call any of the changed functions will also have to be changed, resulting in a very intrusive patch.

Expression translation solves this problem by translating an expression exp_s from the scope of a source function f_s to an equivalent expression exp_d in the scope of a destination function f_d. It does not require adding new function parameters or call arguments like the aforementioned solution. Senx uses expression translation to translate both the buffer size expression and memory access range expression into a single function scope where the predicate will be evaluated. We call this process *converging* the predicate.

At a high level, expression translation can be considered as a form of lightweight function summarization [17]. While function summarization establishes the relations between the inputs to a function and the outputs of a function, expression translation establishes the relations between the inputs to a function and a subset of the local variables of the function. It works by exploiting the equivalence between the arguments that are passed into the function by the caller and the parameters that take on the argument values in the scope of the callee. Using this equivalence, expression translation can iteratively translate expressions that are passed to function invocations across edges in the call graph. Formally, expression translation can converge the comparison between an expression exp_a, the memory access range expression in f_a, and exp_s, the buffer size expression in f_s iff along the set of edges $\mathbb{E}$ connecting f_a and f_s in the program call graph, an expression equivalent to either exp_a or exp_s form continuous sets of edges along the path such that exp_a and exp_s can be translated along those sets into a common scope.

Note that variables declared by a program as accessible across different functions such as global variables in C/C++ do not require the translation, although the use of such kind of variables is not very common. We refer to both function parameters and such kind of variables collectively as nonlocal variables. And we refer to an expression consisting of only nonlocal variables as a nonlocal expression.

The low-level implementation of expression translation in Senx consists of two functions. Function `translate_se_to_scopes`, listed in Algorithm 2, is the core of expression translation. It translates a particular expression $expr$ to the scope of all candidate functions along the call stack $stack$.

Algorithm 2 Translating an expression to the scope of each function on the call stack.

```
Input: stack: a call stack consists of an ordered list of call instruction
    expr: the expression to be translated
    inst: the instruction to which expr is associated
Output: translated_exprs: the translated expr in the scope of each caller function on the call stack
1: procedure TRANSLATE_SE_TO_SCOPES
2:   ▷ Translate expr to an expression in which all the variables are the parameters of func
3:   $func$ ←get_func($inst$)
4:   $expr$ ←make_nonlocal_expr($func, inst, expr$)
5:   if $expr \neq \varnothing$ then
6:     for $call \in stack$ do
7:       ▷ Substitute each parameter variable in expr with its correspondent argument used in call
8:       $expr$ ←substitute_parms_with_args($call, expr$)
9:       $func$ ←get_func($call$)
10:      $translated\_exprs[func] \leftarrow expr$
11:      $expr$ ←make_nonlocal_expr($func, call, expr$)
12:      if $expr = \varnothing$ then
13:        break
14:      end if
15:    end for
16:  end if
17:  return $translated\_exprs$
18: end procedure
```

We illustrate how it works with the code in Listing 2. For simplicity, we use source code line numbers to represent the corresponding instructions. To translate the buffer size involved in the buffer overflow, Senx finds that the buffer is allocated from a call to `malloc` at line 2 from the call stack that it associates with each memory allocation, and invokes `translate_se_to_scopes` with $stack$ =[line 9], $expr$ =“`x*y+1`”, $inst$ =line 2 , $func$ = `foo_malloc`. The function first converts “`x*y+1`” into a definition in which variables are all parameters of `foo_malloc`, which we call a nonlocal definition, if such conversion is possible. This conversion is done by function `make_nonlocal_expr` listed in Algorithm 3, which tries to find a nonlocal definition for each variable in $expr$ and then substitutes each variable with its matching nonlocal definition. `make_nonlocal_expr` relies on `find_nonlocal_def_for_var`, which recursively finds reaching definitions for local variables in a function, eventually building a definition for them in terms of the function parameters, global variables or the return values from function calls. Note that a nonlocal definition can only be in the form of an arithmetic expression without involving any functions. In this case, the resulting $expr$ is also “`x*y+1`” because both `x` and `y` are parameters of `foo_malloc`.

Algorithm 3 Making a nonlocal expression.

```
Input: f: a function
    inst: an instruction in f
    expr: the RHS expression associated with inst
Output: nonlocal_expr: the nonlocalized expr
1: procedure MAKE_NONLOCAL_EXPR
2:   ▷ mapping stores the nonlocal definition for each variable within expr
3:   $mapping \leftarrow \emptyset$
4:   for $var \in expr$ do
5:     if $\neg$ is_var_nonlocal($f, var$) then
6:       $def$ ←find_nonlocal_def_for_var($f, inst, var$)
7:       if $def = \varnothing$ then
8:         ▷ We cannot find a nonlocal definition for var
9:         return $\varnothing$
10:      else
11:        $mapping[var] \leftarrow def$
12:      end if
13:    end if
14:  end for
15:  ▷ Substitute the occurrence of each variable with its nonlocal definition
16:  $nonlocal\_expr$ ←substitute_vars($expr, mapping$)
17:  return $nonlocal\_expr$
18: end procedure
```

It then iterates each call instruction in $stack$, starting from line 9. For each call instruction, it substitutes the parameters in $expr$ with the arguments used in the call instruction. For line 9, it substitutes `x` with `rows` and `y` with `cols+1`, respectively, by calling helper function `substitute_parms_with_args`. As a consequence, “`x*y+1`” becomes “`rows*(cols+1)+1`”. Hence it associates “`rows*(cols+1)+1`” with function `foo` and stores the association in $expr\_translated$, because line 9 exists in function `foo`. After that, it tries to convert “`rows*(cols+1)+1`” into a nonlocal definition with respect to `foo`. At this point, it halts because both `rows` and `cols` are assigned with return values of calls to function `extract_int`. Otherwise, it will move on to the next function on the call stack and continue the translation upwards the call stack. However, in this case, expression translation is also able to translate the memory access range expression from the scope of `bar` into the scope of `foo`. Thus, Senx places the patch predicate in `foo`. If expression translation fails to converge the expression, Senx will abort patch generation.

V. Implementation

We have implemented Senx as an extension of the KLEE LLVM execution engine [7]. Like KLEE, Senx works on C/C++ programs that are compiled into LLVM bitcode [48].

We re-use the LLVM bitcode execution portion of KLEE, and as described in Section IV-B, to implement our expression builder, but do not use any of the constraint collection or solving parts of KLEE. For simplicity and ease of debugging, we represent our expressions as text strings. To support arithmetic operations and simple math functions on expressions, we leverage GiNaC, a C++ library designed to provide support for symbolic manipulations of algebra expressions [1].

We implement a separate LLVM transformation pass to annotate LLVM bitcode with information on loops such as the label for loop pre-header and header, which is subsequently used by access range analysis. This pass relies on LLVM's canonicalization of natural loops to normalize loops [8]. We extend LLVMSlicer [46] for loop cloning. To locate error handling code, we use Talos [19].

Our memory allocation logger uses KLEE to interpose on memory allocations and stores the call stack for each memory allocation. Senx extends KLEE to detect integer overflows and incorporates the existing memory fault detection in KLEE to trigger our patch generation. For alias analysis, Senx leverages DSA pointer analysis [22].

Senx is implemented with 2,543 lines of C/C++ source code, not including the Talos component [19] used to identify error handling code.

VI. Discussion

Our prototype of Senx supports three vulnerability types: buffer overflow, bad cast, and integer overflow. And we believe that safety properties can be used to produce patches for many other vulnerability types. Because our goal is to produce source code patches that can be adopted by developers, this approach can be applied to any vulnerability type as long as the safety property for the vulnerability type can be concretized to program expressions in the source code of target programs.

For example, a safety property can be written for temporal vulnerabilities such as time-of-check-to-time-of-use vulnerabilities with the information on what operations are considered as a check and what operations are considered as a use. A safety property can also be written for missing security check vulnerabilities with information on what operations require security checks and what API functions are used to perform those security checks. Although providing such information can require some effort from the developers, it only needs to be done for one time.

We note that additional instrumentation in the source code is needed for certain vulnerabilities. In our evaluation of Senx on real-world vulnerabilities we found that the most common reason that prevents Senx from generating a patch is the inability to find a common program scope where all program expressions required to synthesize a predicate are available. For such cases, the ability to create function clones with additional arguments that pass the required expressions between function scopes would enable Senx to also cover these cases. We plan to add such capability to Senx in our future work.

VII. Evaluation

First, we evaluate the effectiveness of Senx in fixing real-world vulnerabilities. Second, we evaluate the quality of the patches generated by Senx. We manually examine all the produced patches for correctness and compare them to the developer created patch. For the sake of space, we only describe two of the patches in detail. Third, we compare Senx against state-of-art APR tools including Angelix and SemFix. Last, we measure the applicability of loop cloning, access range analysis, and expression translation using a larger dataset.

App.	Description	SLOC
autotrace	a tool to convert bitmap to vector graphics	19,383
binutils	a collection of programming tools for managing and creating binary programs	2,394,750
libming	a library for creating Adobe Flash files	88,279
libtiff	a library for manipulating TIFF graphic files	71,434
PHP	the official interpretor for PHP programming language	746,390
sqlite	a relational database engine	189,747
ytnef	TNEF stream reader	15,512
zziplib	a library for reading ZIP archives	24,886
jasper	a codec for JPEG standards	30,915
libarchive	a multi-format archive and compression library	158,017
potrace	a tool for tracing bitmap graphics	20,512
Total	N/A	3,817,268

TABLE II: Applications for testing real-world vulnerabilities.

A. Experiment Setup

We choose vulnerabilities in popular applications for Senx to attempt to patch by searching online vulnerability databases [10], [15], [33], software bug report databases, developers' mailing groups [5], [37], [41], and exploit databases [35]. We focus on vulnerabilities that fall into one of the three types of vulnerabilities Senx can currently handle. We then select vulnerabilities that meet the following three criteria: 1) an input to trigger the vulnerability is either available or can be created from the information available, 2) the vulnerable application can be compiled into LLVM bitcode and executed correctly by KLEE, and 3) the vulnerable application uses standard memory allocation functions such as `malloc` to allocate memory as Senx currently relies on this to infer the allocation size of objects. Applications that use custom memory allocation routines are currently not supported by Senx. We obtain the vulnerability-triggering inputs or information about such inputs from the blogs of security researchers, bug reports, exploit databases, mailing groups for software users, or test cases attached to patch commits [2], [3], [6], [35], [44], [56].

From this, we select 42 real-world buffer overflow, bad cast, and integer overflow vulnerabilities along with proof of concept exploits to evaluate the effectiveness of Senx in patching vulnerabilities. The vulnerabilities are from 11 applications show in Table II, which include 8 media and archive tools and libraries, PHP, sqlite, and a collection of programming tools for managing and creating binary programs. The associated vulnerabilities consist of 19 buffer overflows, 13 bad casts, and 10 integer overflows.

All our experiments were conducted directly on these vulnerable applications on a desktop with quad-core 3.40GHz

Intel i7-3770 CPU, 16GB RAM, 3TB SATA hard drive and 64-bit Ubuntu 14.04.

B. How Effective is Senx in Patching Vulnerabilities?

For each vulnerability of an application, we run the corresponding program under Senx with a vulnerability-triggering input. If Senx generates a patch, we manually examine the patch for correctness. If Senx aborts patch generation, we examine what caused Senx to abort.

Our results are summarized in Table III. Column "Type" indicates whether the vulnerability is a 'B' buffer overflow, 'C' bad cast, or 'I' integer overflow. Column "Expressions" shows whether Senx can successfully construct all expressions that are required to synthesize a patch, as some code constructs may contain expressions outside of the theories Senx supports in its symbolic ISA. "Loop Analysis" describes whether loop cloning or access range analysis (ARA) is used if the vulnerability contained a loop. "Patch Placement" lists the type of patch placement: "Trivial" means that the patch is placed in the same function as the vulnerability and "Translated" means that the patch must be translated to a different function. "Data Access" describes whether or not the patch predicate involves complex data access such as fields in a struct or array indices. Finally, "Patched?" summarizes whether the patch generated by Senx fixes a vulnerability. The 10 vulnerabilities where Senx aborts generating a patch are highlighted in red.

Over the 42 vulnerabilities, Senx generates 32 (76.2%) patches, all of which are correct according to our three criteria. Of the 13 patched buffer overflows, loop analysis is roughly split between loop cloning and access range analysis (6 and 8 respectively). Senx elects not to use loop cloning mainly due to two causes. First, due to an imprecise alias analysis that does not distinguish different fields of structs correctly, the program slicing tool utilized by Senx may include instructions that are irrelevant to computing loop iterations into slices. Unfortunately these instructions call functions that can have side-effects so the slices cannot be used by Senx. Second, for a few cases the entire body of the loops is control dependent on the result of a call to a function that has side-effects. For example, the loops involved in CVE-2017-5225 are only executed when a call to `malloc` succeeds. Because `malloc` can make system calls, Senx also cannot clone the loops.

Senx must place 23.8% of the patches in a function different from where the vulnerability exists. This is particularly acute for buffer overflows (31.6% of cases), which have to compare a buffer allocation with a memory access range. This illustrates that expression translation contributes significantly to the patch generation ability of Senx, particularly for buffer overflows, which make up the majority of memory corruption vulnerabilities. Senx's handling of complex data accesses is also used in 48.5% of the patches, indicating this capability is required to handle a good number of vulnerabilities

Senx aborts patch generation for 10 vulnerabilities. The dominant cause for these aborts is that Senx is not able to converge to a function scope where all symbolic variables in the patch predicate are available. There is also one case (jasper-CVE-2017-5501) where Senx cannot find appropriate error-handling code to synthesize the patch. In these cases, the patch requires more significant changes to the application code that are beyond the capabilities of Senx. In other cases, Senx detects that there are multiple reaching definitions for patch predicates that it does not have an execution input for. Currently, Senx only accepts one execution path executed by the single vulnerability-triggering input. In the future we plan to handle these cases by allowing Senx to accept multiple inputs to cover the paths along which the other reaching definitions exist. Finally, Senx aborts for a couple of vulnerabilities because both loop cloning and access range analysis fail.

C. What is the Quality of the Produced Patches?

For each patch generated by Senx, we manually examine the patch for correctness. To determine if a patch is correct, we apply the three following tests: a) we apply the patch to the target program and verify that the vulnerability is no longer triggered by the vulnerability-triggering input, b) we run the built-in test suite provided by the vendor of the target program to verify that the entire test suite is passed, c) we check for semantic equivalence with the official patch released by the vendor, if available, by manually examining if the patch generated by Senx affects the behavior of the target program in the same way as the official patch, and semantic correctness by analyzing the code manually. We consider a patch is correct only when all three tests are passed. Our examination finds that all produced patches are correct.

Out of the 32 generated patches, we select 2 patches to describe in detail.

libtiff-CVE-2017-5225. This is a heap buffer overflow in libtiff, which can be exploited via a specially crafted TIFF image file. The overflow occurs in a function `cpContig2SeparateByRow` that parses a TIFF image into rows and dynamically allocates a buffer to hold the parsed image based on the number of pixels per row and bits per pixel. By using an inconsistent bits per pixel parameter, the attacker can cause libtiff to allocate a buffer smaller than the size of the pixel data and cause a buffer overflow.

When Senx captures the buffer overflow via running libtiff with a crafted TIFF image file, it first identifies that the buffer is allocated using the value of variable `scanlinesizein` and the starting address of the buffer is stored in variable `inbuf`. Hence it uses [`inbuf`, `inbuf + scanlinesizein`] to denote the buffer range. Senx then finds that the buffer overflow occurs in a 3-level nested loop and that the pointer used to access the buffer is dependent on the loop induction variable. Senx classifies the vulnerability as a buffer overflow.

Loop cloning fails because the loop slice is dependent on a call to `_TIFFmalloc`, which subsequently calls `malloc`. Thus, Senx applies access range analysis. Access range analysis detects that only the outer and inner-most loops affect the memory access pointer and from the extracted induction variables, computes the expression [`inbuf`, `inbuf+spp*imagewidth`] to represent the access range.

App.	CVE#	Type	Expressions	Loop Analysis	Patch Placement	Data Access	Patched?
sqlite	CVE-2013-7443	I	Determinate	—	Failed	—	✗
	CVE-2017-13685	I	Determinate	—	Trivial	Simple	✓
zziplib	CVE-2017-5976	B	Determinate	Cloned	Translated	Complex	✓
	CVE-2017-5974	I	Determinate	—	Translated	Complex	✓
	CVE-2017-5975	I	Determinate	—	Translated	Complex	✓
Potrace	CVE-2013-7437	C	Determinate	—	Trivial	Complex	✓
libming	CVE-2016-9264	I	Determinate	—	Trivial	Simple	✓
libtiff	CVE-2016-9273	B	Indeterminate	—	—	—	✗
	CVE-2016-9532	B	Determinate	Cloned	Trivial	Complex	✓
	CVE-2017-5225	B	Determinate	ARA	Trivial	Simple	✓
	CVE-2016-10272	B	Determinate	ARA	Translated	Simple	✓
	CVE-2016-10092	I	Determinate	—	Translated	Simple	✓
	CVE-2016-5102	I	Determinate	—	Trivial	Simple	✓
	CVE-2006-2025	C	Determinate	—	Trivial	Complex	✓
libarchive	CVE-2016-5844	C	Determinate	—	Trivial	Complex	✓
jasper	CVE-2016-9387	C	Determinate	—	Trivial	Complex	✓
	CVE-2016-9557	C	Determinate	—	Trivial	Complex	✓
	CVE-2017-5501	C	Determinate	—	Failed/Error handling	—	✗
ytnef	CVE-2017-9471	B	Determinate	Cloned	Trivial	Simple	✓
	CVE-2017-9472	B	Determinate	Cloned	Trivial	Simple	✓
	CVE-2017-9474	B	Determinate	Failed	—	—	✗
PHP	CVE-2011-1938	B	Determinate	ARA	Translated	Simple	✓
	CVE-2014-3670	B	Determinate	ARA	Translated	Complex	✓
	CVE-2014-8626	B	Determinate	Cloned	Trivial	Simple	✓
binutils	CVE-2017-15020	B	Determinate	ARA	Translated	Simple	✓
	CVE-2017-9747	B	Determinate	Cloned	Translated	Simple	✓
	CVE-2017-12799	I	Determinate	—	Trivial	Simple	✓
	CVE-2017-6965	I	Determinate	—	Failed	—	✗
	CVE-2017-9752	I	Determinate	—	Translated	Simple	✓
	CVE-2017-14745	C	Determinate	—	Failed	—	✗
autotrace	CVE-2017-9151	B	Indeterminate	—	—	—	✗
	CVE-2017-9153	B	Indeterminate	—	—	—	✗
	CVE-2017-9156	B	Determinate	ARA	Trivial	Simple	✓
	CVE-2017-9157	B	Determinate	ARA	Trivial	Simple	✓
	CVE-2017-9168	B	Determinate	Failed	—	—	✗
	CVE-2017-9191	B	Determinate	ARA	Failed	—	✗
	CVE-2017-9161	C	Determinate	—	Trivial	Simple	✓
	CVE-2017-9183	C	Determinate	—	Trivial	Complex	✓
	CVE-2017-9197	C	Determinate	—	Trivial	Complex	✓
	CVE-2017-9198	C	Determinate	—	Trivial	Complex	✓
	CVE-2017-9199	C	Determinate	—	Trivial	Complex	✓
	CVE-2017-9200	C	Determinate	—	Trivial	Complex	✓

TABLE III: Patch generation by Senx

Because both the buffer range and the access range start at `inbuf`, Senx synthesizes the patch predicate as `spp*imagewidth > scanlinesizein`. Senx then finds that `cpContig2SeparateByRow` contains error handling code, which has a label `bad`, and generates the patch as below. As the buffer allocation and overflow occur in the same function, Senx puts the patch immediately before the buffer allocation.

```
if (spp*imagewidth > scanlinesizein)
  goto bad;
```

The official patch invokes the same error handling and is placed at the same location as Senx's patch. However, the official patch checks that "`(bps != 8)`". From further analysis, we find that both patches are equivalent, though the human-generated patch relies on the semantics of the libtiff format, while Senx's patch directly checks that the loop cannot exceed the size of the allocated buffer.

libarchive-CVE-2016-5844. This integer overflow in the ISO parser in libarchive can result in a denial of service via a specially crafted ISO file. The overflow happens in function `choose_volume` when it multiplies a block index, which is a 32-bit integer, with a constant number. This can exceed the maximum value that can be represented by a 32-bit integer and overflow into a negative number, which is then used as a file offset.

Senx detects the integer overflow when it runs libarchive's ISO parser with a crafted ISO file. It generates an expression of the overflown value as the product of 2048 and `vd→location`. Further Senx detects that the overflown value is assigned to a 64-bit variable `skipsize`, thus classifying this as a repairable integer overflow. Senx patches the vulnerability by casting the 32-bit value to a 64-bit value before multiplying:

```
- skipsize = LOGICAL_BLOCK_SIZE * vd->location;
+ skipsize = 2048 * (int64_t)vd->location;
```

The official patch is essentially identical to the patch generated by Senx. The only difference is that the official patch uses the constant `LOGICAL_BLOCK_SIZE` rather than its equivalent value 2048 in the multiplication.

D. Comparison with Other Work

To illustrate whether Senx can address the limitations of state-of-art APR tools, we evaluate the effectiveness of Senx against SemFix [34] and Angelix [30]. Because SemFix and Angelix require considerable effort for each application and vulnerability, we were only able to make this comparison on 2 vulnerabilities for one application, autotrace. We use all four built-in test inputs for autotrace, and 50 randomly generated inputs for examples and one vulnerable-triggering input for each vulnerability.

In both cases Semfix and Angelix are unable to generate a patch either because they are unable to locate an existing program constructs to change to pass both positive test inputs and negative test inputs, or they are unable to synthesize a guard statement to prevent the vulnerabilities from being triggered. Senx, on the other hand, is able to generate working patches for both vulnerabilities.

E. Applicability

We evaluate how applicable loop cloning, access range analysis and expression translation are across a larger dataset. To generate such a dataset, we extract all loops that access memory buffers and the allocations of these buffers from the 11 programs in GNU Coreutils, regardless of whether they contain vulnerabilities or not. We then apply Senx's loop analysis to all loops and find that loop cloning can be applied to 88% of the loops and access range analysis can be applied to 46% of the loops. This is in line with our results from the vulnerabilities. For the sake of space, we describe the details of these experiments in the Appendix.

VIII. Related Work

A. Automatic Patch Generation

Leveraging Fix Patterns. By observing common human-developer generated patches, PAR generates patches using fix patterns such as altering method parameters, adding a null checker, calling another method with the same arguments, and adding an array bound checker [20]. Senx differs from PAR in two aspects. First, PAR is unable to generate a patch when the correct variables or methods needed to synthesize a patch are not accessible at the faulty function or method. Second, PAR uses a trial-and-error approach that tries out not only each fixing pattern upon a given bug, but also variables or methods that are accessible at the faulty function or method to synthesize a patch. On the contrary, Senx employs a guided approach that identifies the type of the given bug and chooses a corresponding patch model to generate the patch for the bug and systematically finds the correct variables to synthesize the patch based on semantic information provided by a patch model.

LeakFix fixes memory leak bugs by adding a memory deallocation statement for a leaked memory allocation at the correct program location [16]. By abstracting a program into a CFG containing only program statements related to memory allocation, deallocation, and usage, LeakFix transforms the problem of finding a fix for a memory leak into searching for a CFG edge where a memory deallocation statement can be added to fix the memory leak without introducing new bugs.

Using Program Mutations. GenProg is a pioneering work that induces program mutations, i.e. genetic programming, to generate patches [55]. Leveraging test suites, it focuses on program code that is executed for negative test cases but not for positive test cases and utilizes program mutations to produce modifications to a program. As a feedback to its program mutation algorithm, it considers the weighted sum of the positive test cases and negative test cases that the modified program passes. Treating all the results of program mutations as a search space, its successor improves the scalability by changing to use patches instead of abstract syntax trees to represent modifications and exploiting search space parallelism [23].

Using SMT Solvers. SemFix [34] and Angelix [30] use constraint solving to find the needed expression to replace incorrect expressions used in a program. By executing a target program symbolically with both inputs triggering a defect and inputs not triggering the defect, they identify the constraints that the target program must satisfy to process both kinds of inputs correctly. They then synthesize a patch using component-based program synthesis, which combines components such as variables, constants, and arithmetic operations to synthesize a symbolic expression that can make the target program satisfy the identified constraints.

Learning from Correct Code. Prophet learns from existing correct patches [27]. It uses a parameterized log-linear probabilistic model on two features extracted from the abstract trees of each patch: 1) the way the patch modifies the original program and 2) the relationships between how the values accessed by the patch are used by the original program and by the patched program. With the probabilistic model, it ranks candidate patches that it generated for a defect by the probabilities of their correctness. Finally it uses test suites to test correctness of the candidate patches. Like other generate-and-validate automatic patch generation techniques, its effectiveness depends on the quality of the test suites.

B. Mitigating Security Vulnerabilities

Fortifying Programs. One way to prevent vulnerabilities from being exploited is by fortifying programs to make them more robust to malicious inputs. Software Fault Isolation (SFI) instruments checks before memory operations to ensure that they cannot corrupt memory [18], [31], [52], [57]. Control Flow Integrity (CFI) learns valid control flow transfers of a program and validates control flow transfers to prevent execution of exploit code [13], [51], [58]. Alternatively, some techniques modify the layout or permissions of critical memory regions to detect or prevent exploits [9], [12], [43], [45], [49], [50].

By contrast, Talos introduces the notion of Security Workarounds for Rapid Response (SWRR), which steers program execution away from a vulnerability to error handling code, and instruments SWRRs to programs to prevent malicious inputs from triggering vulnerabilities [19].

While these techniques prevent exploit of vulnerabilities at the cost of extra runtime overhead or loss of program functionality, Senx generates patches to fix vulnerabilities without imposing such a cost.

Filtering Inputs. Some techniques detect and simply filter out malicious inputs [4], [11], [28], [47], [53]. Among them, Bouncer combines static analysis and symbolic analysis to infer the constraints to exploit a vulnerability and generates an input filter to drop such malicious inputs [11]. Shields models a vulnerability as a protocol state machine and constructs network filters based on it [53].

While most of these techniques identify malicious inputs by the semantics or syntax of the malicious inputs, some of them focus on the semantics of vulnerabilities [4], [28]. Similar to these techniques, Senx also uses the semantics of vulnerabilities to synthesize patches. However, Senx has a different goal of generating patches to fix vulnerabilities.

Rectifying Inputs. Alternative to filtering inputs, some techniques rectify malicious inputs to prevent them from triggering vulnerabilities. With taint analysis, SOAP learns constraints on input by observing program executions with benign inputs. From the constraints that it has learned, it identifies input that violates the constraints and tries to change the input to make it satisfy the constraints. By doing so, it not only renders the input harmless but also allows the desired data in the rectified input to be correctly processed [25].

Based on the observation that exploit code embedded in inputs is often fragile to any slight changes, A2C encodes inputs with a one-time dictionary and decodes them only when the program execution goes beyond the paths likely to have vulnerabilities in order to disable the embedded exploit code [21].

IX. Conclusion

This paper presents the design and implementation of Senx, a system that uses human-specified safety properties to generates patches for buffer overflow, bad cast, and integer overflow vulnerabilities. Senx uses three novel program analysis techniques introduced in this paper: loop cloning, access range analysis and expression translation. In addition, Senx utilizes an expression representation that facilitates the translation of expressions extracted from symbolic execution back into C/C++ source code. Enabled by these techniques, Senx generates patches correctly for 76% of the 42 real-world vulnerabilities. Senx's main limitations are limited precision in alias analysis for loop cloning, and the inability to converge expressions to find a location in the program where all necessary variables are in scope.

Acknowledgement

This research was supported in part by an NSERC Discovery Grant (RGPIN 2018-05931) and a Canada Research Chair (950-228402).

References

[1] GiNaC is Not a CAS. http://www.ginac.de/, 2018. Accessed: May, 2018.

[2] http://blogs.gentoo.org/ago, 2018. Accessed: May, 2018.

[3] http://github.com/asarubbo/poc, 2018. Accessed: May, 2018.

[4] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), SP '06, IEEE Computer Society, pp. 2–16.

[5] http://lists.gnu.org/archive/html/bug-coreutils/, 2018. Accessed: May, 2018.

[6] http://bugzilla.maptools.org/, 2018. Accessed: May, 2018.

[7] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[8] http://llvm.org/docs/Passes.html#loop-simplify-canonicalize-natural-loops, 2018. Accessed: May, 2018.

[9] Chen, P., Xu, J., Hu, Z., Xing, X., Zhu, M., Mao, B., and Liu, P. What you see is not what you get! thwarting just-in-time rop with chameleon. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (June 2017), pp. 451–462.

[10] http://cve.mitre.org, 2018. Accessed: May, 2018.

[11] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: Securing software by blocking bad input. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 117–130.

[12] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium* (Jan. 1998), pp. 63–78.

[13] Criswell, J., Dautenhahn, N., and Adve, V. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 292–307.

[14] Cullmann, C., and Martin, F. Data-Flow Based Detection of Loop Bounds. In *Workshop on Worst-Case Execution Time* (2007).

[15] http://www.cvedetails.com, 2018. Accessed: May, 2018.

[16] Gao, Q., Xiong, Y., Mi, Y., Zhang, L., Yang, W., Zhou, Z., Xie, B., and Mei, H. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 459–470.

[17] Gopan, D., and Reps, T. Low-level library analysis and summarization. In *Computer Aided Verification* (Berlin, Heidelberg, 2007), W. Damm and H. Hermanns, Eds., Springer Berlin Heidelberg, pp. 68–81.

[18] Huang, W., Huang, Z., Miyani, D., and Lie, D. Lmp: Light-weighted memory protection with hardware assistance. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications* (New York, NY, USA, 2016), ACSAC '16, ACM, pp. 460–470.

[19] Huang, Z., D'Angelo, M., Miyani, D., and Lie, D. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)* (May 2016), pp. 618–635.

[20] Kim, D., Nam, J., Song, J., and Kim, S. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 802–811.

[21] Kwon, Y., Saltaformaggio, B., Kim, I. L., Lee, K. H., Zhang, X., and Xu, D. A2c: Self destructing exploit executions via input perturbation. In *Proceedings of NDSS'17* (2017), Internet Society.

[22] Lattner, C., Lenharth, A., and Adve, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 278–289.

[23] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 2012 International Conference on Software Engineering* (June 2012), pp. 3–13.

[24] Lin, Z., Jiang, X., Xu, D., Mao, B., and Xie, L. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2007), ASIACCS '07, ACM, pp. 329–340.

[25] Long, F., Ganesh, V., Carbin, M., Sidiroglou, S., and Rinard, M. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 80–90.

[26] Long, F., and Rinard, M. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 166–178.

[27] Long, F., and Rinard, M. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, ACM, pp. 298–312.

[28] Long, F., Sidiroglou-Douskos, S., Kim, D., and Rinard, M. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2014), POPL '14, ACM, pp. 439–452.

[29] Mechtaev, S., Yi, J., and Roychoudhury, A. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 448–458.

[30] Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 691–701.

[31] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., and Gan, E. RockSalt: better, faster, stronger SFI for the x86. In *Proceedings of the 2012 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 395–404.

[32] Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S. Softbound: highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)* (2009), pp. 245–258.

[33] http://nvd.nist.gov, 2018. Accessed: May, 2018.

[34] Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 772–781.

[35] http://www.exploit-db.com, 2018. Accessed: May, 2018.

[36] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 87–102.

[37] http://bugs.php.net, 2018. Accessed: May, 2018.

[38] http://www.php-security.org/MOPB/MOPB-41-2007.html, 2018. Accessed: May, 2018.

[39] Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 254–265.

[40] Qi, Z., Long, F., Achour, S., and Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2015), ISSTA 2015, ACM, pp. 24–36.

[41] http://bugzilla.redhat.com, 2018. Accessed: May, 2018.

[42] http://www.openwall.com/lists/oss-security/2012/05/03/4, 2018.

[43] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)* (Oct. 2004), pp. 298–307.

[44] http://www.mail-archive.com/sqlite-users@mailinglists.sqlite.org/, 2018.

[45] http://www.angelfire.com/sk/stackshield, 2018.

[46] http://github.com/jirislaby/LLVMSlicer, 2018.

[47] Süsskraut, M., and Fetzer, C. Robustness and security hardening of COTS software libraries. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings* (2007), pp. 61–71.

[48] http://llvm.org/, 2018.

[49] `http://pax.grsecurity.net/`, 2018.

[50] Tian, D., Zeng, Q., Wu, D., Liu, P., and Hu, C. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).

[51] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 941–955.

[52] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review* (1994), vol. 27, pp. 203–216.

[53] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2004), SIGCOMM '04, ACM, pp. 193–204.

[54] Weimer, W., Fry, Z. P., and Forrest, S. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Piscataway, NJ, USA, 2013), ASE'13, IEEE Press, pp. 356–366.

[55] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 364–374.

[56] http://sourceware.org/bugzilla/, 2018. Accessed: May, 2018.

[57] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), pp. 79–93.

[58] Zhang, M., and Sekar, R. Control Flow Integrity for COTS Binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 337–352.

Appendix

We measure how often expression translation is able to converge the memory access range and buffer allocation size into a single function scope, and find that it is able to do so in 85% of the cases.

We use 11 programs from the GNU Coreutils as listed in Table IV to evaluate the applicability of our analysis techniques. The most common reasons for Senx's access range analysis to be aborted is that loops cannot be normalized by LLVM. For example, the number of times a loop that parses string input iterates depends on the content of the string. Such a string cannot be symbolically analyzed by access range analysis.

To understand the reasons that can cause expression translation to abort, we try to converge the buffer size and memory access range for the loops that we could successfully analyze and tabulate the results in Table V. The "Access Range" column tabulates the average percentage of functions in the loop's call stack that expression translation could translate the memory access range into and "Buffer Range" tabulates the average percentage of functions in the buffer allocation's call stack that expression translation could translate the buffer allocation size into. Finally "Converged" indicates out of all loops, what percentage could expression translation find a

Program	**Type**	**SLOC**	**LLVM bitcode**
sha512sum	data checksum	581	135KB
pr	text formatting	1,723	194KB
head	text manipulation	761	109KB
dir	directory listing	3,388	418KB
od	file dumping	1,368	237KB
ls	directory listing	3,388	418KB
base64	data encoding	238	91KB
wc	text processing	784	120KB
cat	file concatenating	495	182KB
sort	data sorting	3,251	433KB
printf	format and print data	694	198KB
AVG	N/A	1,516	230KB

TABLE IV: Programs for evaluating applicability.

Program	**Access Range**	**Buffer Range**	**Converged**
pr	100%	10%	100%
head	100%	25%	100%
tr	86%	36%	100%
od	54%	16%	58%
cat	100%	33%	100%
dir	71%	14%	57%
ls	42%	33%	34%
base64	100%	33%	100%
md5sum	100%	33%	100%
sha512sum	97%	80%	97%
sort	91%	10%	90%
AVG.	85%	29%	85%

TABLE V: Convergence of expression translation.

common function scope in which to place the patch. As we can see, it seems that the buffer allocation size frequently takes parameters that are calculated fairly close in the call stack to the allocation point, and those values are not available higher up in the call chain, thus limiting the functions scopes many of these cases could be converged to.