Senx: Semantically Correct Patch Generation for Security Vulnerabilities

Zhen Huang
University of Toronto
z.huang@mail.utoronto.ca

David Lie
University of Toronto
lie@eecg.tornto.edu

***Abstract*—Security vulnerabilities are among the most critical software defects in existence. As such, they require patches that are correct and quickly deployed. This motivates an automatic patch generation method that emphasizes semantic correctness and wide applicability. To address these challenges, we propose Senx, which uses three novel patch generation techniques to create patches for buffer overflow, integer overflow and bad memory offset vulnerabilities. Senx uses *loop cloning* and *access range analysis* to analyze loops involved in these vulnerabilities. For vulnerabilities that span multiple functions, Senx uses *symbolic translation* to translate symbolic expressions and place them in a function scope where all values are available to create the patch. This enables Senx to patch vulnerabilities with complex loops and interprocedural dependencies that previous semantics-based patch generation systems could not handle.**

We have implemented a prototype called Senx using this approach. Our evaluation shows that the patches generated by Senx successfully fix 33 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. All patches that Senx produces are correct, and Senx identifies cases where its analysis will fall short and instead of producing an incorrect patch that does not fix the vulnerability, correctly aborts patch generation.

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 the process of fixing security vulnerabilities [23]. As a result, an entire line of research inquiry into automated patch generation has arisen to try to address this challenge [21], [24], [28], [29], [31], [34]–[36], [43], [44].

Automatic patch generation approaches broadly break down into two categories: search-based and semantics-based. Search-based approaches try various arbitrary code changes and use a battery of test cases to check whether any of the changes succeeded in fixing the bug [24], [28], [44]. Because search-based techniques can generate unconstrained code changes, they are applicable to a wide variety of bugs, but the correctness of the generated patches can be uneven, and depend heavily on the comprehensiveness of the test cases [32], [36]. In contrast, semantics-based techniques use code analysis (i.e. static analysis and symbolic execution) to

```
1 char *foo_malloc(x,y) {
2   return (char *)malloc(x * y + 1);
3 }
4
5 // print a flattened array in
6 // 2-dimensional format
7 int foo(char *input) {
8   // input format: RRCCDDD....DDD
9   //  RR: number of rows
10  //  CC: number of columns
11  //  DDD...DDD: flattened array data
12  //
13  // benign input: 0203123456
14  // output:
15  //     1 2 3
16  //     4 5 6
17
18  int size = strlen(input);
19  char *p = input;
20  int rows = extract_int(p);
21  p +=2;
22  size -= 2;
23  int cols = extract_int(p);
24  p +=2;
25  size -=2;
26+ if ((double)(cols+1)*(size/cols) >
27+      rows * (cols + 1) + 1)
28+   return -1;
29  char *output = foo_malloc(rows, cols + 1);
30  if (!output)
31    return -1;
32  bar(p, size, cols, output);
33  printf("%s\n", output);
34  free(output);
35  return 0;
36 }
37
38 void bar(char *src,int size,int cols,char *dest) {
39  char *p = dest;char *q = src;
40  while (q < src + size)  {
41    for (unsigned j = 0; j < cols; j++)
42      *(p++) = *(q++);
43    *(p++) = '\n';
44  }
45  *p = '\0';
46 }
```

Listing 1. A buffer overflow CVE-2012-0947 with a patch (prefixed with '+').

produce patches that attempt to address the underlying bug rather than change the code just enough to satisfy the test cases [31], [34]. As a result, semantics-based techniques are more likely to produce correct patches, but are less applicable, since they are constrained to cases where the code analysis is able to analyze the defective code.

When used to fix security vulnerabilities, the requirements that patches work correctly is even more pressing, as falsely believing that a vulnerability has been addressed when in fact it has not, may lead a user to disable other mitigating protections, such as removing configuration workarounds or firewall rules. In this paper, we propose Senx, which aims to create semantically correct patches for common security vulnerabilities, such as buffer overflows, integer overflows and memory corruption due to bad-offset calculations using semantics-based analyses. However, many of these vulnerabilities involve complex code structures that code analysis techniques traditionally find challenging. For example, buffer overflows often involve complex loop structures. In addition, formulating a check to test a memory access is within the memory-range of the data-structure, may require the patch generation system to synthesize interprocedural code if the allocation of the memory and the faulty memory access occur in different functions.

To concretely illustrate these challenges, consider the buffer overflow vulnerability CVE-2012-0947 [13] from libav in Listing 1. The vulnerability arises because the code copies user-provided data into a buffer whose allocation size is computed based on the number of rows and columns specified in the input, but the amount of data copied is based on the size of the data provided. For reference, the correct patch is provided on lines 26-29, which consists of checking if the amount of data to be copied (`((cols+1)×(size/cols))`) by the nested loop in `bar` is greater than size of the buffer (`rows×(cols+1)+1`) allocated by `foo_malloc`, in which case the patch returns an error to `foo`'s caller. To generate this patch, Senx must correctly identify the pointer `p` used to write to the buffer `dest` in `bar` and infer the memory access range of `p` from the nested loops. Further, Senx must detect that the allocation of `dest` is actually performed in another function `foo_malloc` and symbolically compute its allocation size. Finally, Senx must identify `foo` as the common caller of both `bar` and `foo_malloc` and translate the symbolic expression for both the memory access range of `p` from `bar` and the allocation size of `dest` from `foo_malloc` into the scope of `foo` so that the patch can be generated.

Senx accomplishes this with the introduction of three novel patch generation techniques. First, rather than try to statically analyze arbitrarily complex loops, Senx attempts to clone the loop code to be used in the patch predicate, but slicing out any code that may have side effects from the cloned loop so that the loop only computes the memory access range of the pointers in the loop. We call this technique *loop cloning*. For loops where code that has side effects that cannot be safely removed – for example, the loop execution depends on the result of a function call, which Senx cannot safely slice out – Senx falls back on a symbolic analysis technique we developed, called *access range analysis*. Finally, to identify and place the patch in a function scope where all symbolic expressions required in the predicate are available, Senx uses *symbolic translation*, which uses the equivalence between function arguments and parameters to generate a set of equivalent expressions. This enables Senx to generate patches where the defective code is spread across multiple functions. This overcomes a limitation of all previous semantics-based patch generate systems that we are aware of, which can only generate patches for defects that are enclosed entirely within in a single function [24], [28], [31], [34].

Because Senx creates patches for security vulnerabilities, Senx places a higher emphasis on correctness than previous systems. Unlike previous patch generation systems that rely on test cases to determine the correctness of the patch, Senx explicitly identifies instances where its analysis may be incorrect and aborts patch generation in those cases. In our experiments, Senx generates 33 correct patches out of 42 vulnerabilities, and in the remaining 9 cases, correctly detects that it will not be able to generate a patch and aborts instead of generating a patch that does not actually fix the vulnerability. We call this property of Senx's patches *semantic correctness*.

Our main contributions in this paper are:

- We describe the design of Senx, an automatic patch generation system for buffer overfow, integer overflow and bad-offset vulnerabilities. Senx makes use of three novel techniques: loop cloning, access range analysis and symbolic translation.
- We describe the implementation of a Senx prototype on top of the LLVM [39] framework and leveraging KLEE [17] as the symbolic execution engine. Senx implements its own symbolic expression ISA that is optimized for translation back into source code, so that source code patches can be easily generated. In addition, Senx's symbolic expressions have support for complex expressions involving array indicies and C/C++ structs and classes. We plan to make Senx open-source so that others may build on our work or deploy it to automatically create security patches.
- We evaluate Senx on a corpus of 42 vulnerabilities across 11 popular applications, including PHP interpretor, 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 33 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.

II. Overview

In this section, we first present the definitions relevant to the problem that our approach addresses. We then give an

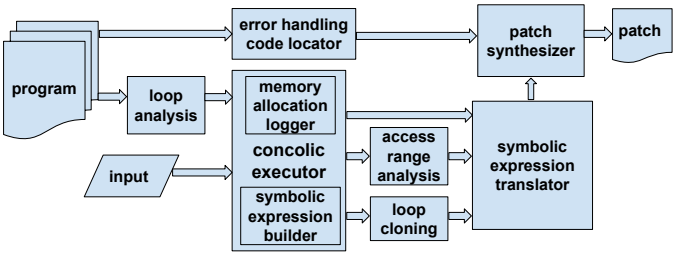

Fig. 1. Workflow of Senx.

overview of our approach and illustrate it by going through the example code. Finally, we discuss the limitations of Senx.

A. Problem Definition

The goal of Senx is to synthesize sematically correct source code patches for integer overflow, buffer overflow, and bad offset vulnerabilities. Figure 1 shows the workflow of our approach. It takes as input the source code of an application and an input that triggers a vulnerability in the application. The input does not need to exploit the vulnerability, but has to trigger the vulnerability in such a way that it is detectable (it crashes the program for example). Given this, Senx will gather information about the program by executing it concolically, identify and classify the vulnerability if it is one of types it can handle, analyze the appropriate code and finally generate a patch. At any time in the process, if Senx detects that a stage can't generate correct results, it will abort and not generate a patch. We begin by precisely defining the types of vulnerabilities Senx can currently handle below, as well as what a Senx patch is and what semantically correct means.

Integer overflow. An integer overflow occurs when an integer is assigned a value larger or smaller than can be represented. This manifests as a large value that is increased and becomes a small value, or a small value that is decreased and becomes a large value. Integer overflows usually become vulnerabilities when the integer is subsequently used as an index into an array, which enables an attacker to corrupt or access arbitrary memory locations.

Buffer overflow. A buffer overflow occurs when series of memory accesses traversing a buffer in a loop crosses from an allocated buffer to a memory location outside of the buffer. We use the term *buffer* in the broad sense to refer to either a bounded memory region (such as a struct or class object) or an array. The memory access can be the result of an array dereference or pointer dereference.

Bad Offset. A bad offset vulnerability occurs when a memory access is computed off a base pointer, and exceeds the upper bound of the object the base pointer points to. Some causes of bad offset vulnerabilities include an incorrect calculation of an array index or a casts of a pointer to the the incorrect object type. A bad offset vulnerability may also be called a *non-linear buffer overflows* in the literature.

Patch. Senx generates patches in one of two forms: a) a single `if` statement that checks a *predicate*, which evaluates to true if and only if the vulnerability is about to be triggered or b) an added *data type cast* to an evaluation such as addition or multiplication that could otherwise lead to an integer overflow. In the first form, if the predicate evaluates to true, control is transferred to error handling code that avoids executing the vulnerable code, treats the current input as an error, and returns the application back to a known state. In the second form, the cast avoids integer overflows by ensuring that the evaluation is performed using the correct data type.

Semantic correctness. We say a patch is semantically correct if it prevents the execution of the vulnerable code if and only if inputs that will trigger the vulnerability is given to the program. In other words, the patch predicate must only evaluate to true on vulnerability-triggering inputs while evaluating to false on all other inputs. Senx's correctness assumes that the off-the-shelf pointer alias analyses it uses is precise, as it depends on knowing the values of variables at every point along the execution path. While Senx could use other pointer analyses, all static pointer analyses have the potential to be imprecise under some circumstances, which have the potential to cause Senx's analysis to be incorrect. In our evaluation of 42 vulnerabilities, we find that the pointer analysis used by our Senx prototype to be adequate, and Senx does not produce any incorrect patches. We further discuss this limitation in Section II-D.

B. Patch Generation

Senx generates a patch in the following 4 steps. First, based on a concolic execution of the program with the vulnerability-triggering input, Senx classifies the vulnerability into one of the 3 categories it supports: an integer overflow, a buffer overflow, or a bad offset. Second, based on the type of vulnerability, Senx either generates the patch predicate that will detect if the input given to the program will trigger the vulnerability or identifies the correct data type to cast to a evaluation to avoid integer overflow. Third, Senx finds a location in the source code where error handling code exists and the patch predicate can be evaluated and translates the patch predicate if needed. Finally, Senx synthesizes a patch that checks the patch predicate and calls the error handling code if the predicate evaluates to true. Senx uses Talos [23] to find and select the error handling code to call. In each of steps one to three, Senx will not generate a patch if it cannot guarantee semantic-correctness. For the fourth step, semantic correctness is not involved since the error handling code is only triggered if the predicate evaluates to true. We now describe each of the four steps in more detail using the example from Listing 1.

Vulnerability classification. Senx first classifies the vulnerability using a concolic execution of the program with a vulnerability-triggering input. During the concolic execution, the *symbolic expression builder* of Senx generates symbolic expressions for program variables. Using these expressions, Senx can then assemble them into more complex symbolic expressions that represent the size of memory allocations

and the location of memory accesses. In this way, Senx can symbolically represent the upper and lower bounds of all memory objects.

Based on the information collected from the concolic execution, it performs the following classification. If it detects that any variable had an integer overflow, then it classifies the vulnerability as an integer overflow. If it is not an integer overflow, Senx inspects the symbols collected during symbolic execution to see if an out-of-bounds memory access outside of an allocated region occurred. If the out-of-bounds access occurs in a loop and is dependent on the number of iterations of the loop, it is classified as a buffer overflow. Otherwise, Senx checks if the out-of-bounds access is an offset of a base pointer, and classifies it as a bad offset. If the vulnerability does not meet any of these definitions, then Senx does not generate a patch because it cannot guarantee the correctness of the patch.

In Listing 1, Senx detects that the faulty memory access occurs when the pointer `p` is dereferenced and written to on line 42. Since the access occurs inside a loop, and `p` is incremented on each loop iteration, Senx classifies this vulnerability as a buffer overflow.

Buffer Overflow. For buffer overflows, Senx generates patches that check a predicate that checks whether the memory access can exceed the upper range of the buffer. While Senx could try to evaluate this predicate inside the loop, there are two reasons it does not do this. First, evaluating the predicate inside the loop every time would increase the overhead of the check. Second, it might not be possible to evaluate the predicate inside the loop because the size of the buffer cannot be computed. For example, in Listing 1, the size of buffer `dest` cannot be computed from the variables inside the scope of `bar`.

Instead, Senx generates a predicate that checks if the memory access range of the loop can exceed the size of the buffer. To do this, Senx must compute the number of loop iterations, the amount the memory access is incremented per iteration and the size of the buffer. The number of iterations is computed using one of two methods that Senx supports: *loop cloning* and *access range analysis*, which we discuss in more detail in Section II-C. The other two components: the increment per iteration and size of the buffer are both computed symbolically from the Senx's concolic execution. Senx requires that the increment must be the same for each iteration, or predicate generation will fail and Senx will not produce a patch.

Senx may not be able to correctly generate a patch in two cases. First, Senx must convince itself that it's symbolic analysis is complete. Because the symbolic representation of the predicate is computed from a concolic execution of the program along the vulnerable path, Senx conducts a static analysis to ensure that the reaching definition of each variable is unique to the executed path. If not, the predicate it computes will not be accurate for the other paths and Senx aborts. Second, the computation for some of the variables in the predicate may not be expressible in Senx's symbolic theory. For example, the number of loop iterations may not be symbolically representable by Senx. In those cases, Senx also aborts and does not generate a patch. This check is also applied for bad offset and integer overflow vulnerabilities, described below.

In our example, Senx generates a predicate for a buffer overflow, which checks whether the number of loop iterations, `(cols+1)*(size/cols)`, can exceed the size of the buffer `p` points to. Since the allocation for the buffer is in another function, Senx performs interprocedural analysis of arguments and parameters, to realize that in this execution, `p` points to `dest`, which is an alias `output`, and whose size is `x*y+1` as defined by its allocation in `foo_malloc`.

Bad Offset. For bad offset vulnerabilities, the predicate checks whether the out-of-bounds memory access exceeds the upper bound of the base object it is accessing, which Senx extracts from the memory access' symbolic representation.

Integer Overflow. Senx classifies integer overflows into two types: repairable and unrepairable. In a repairable integer overflow, the data type of the result variable of an evaluation mismatches the data type of the evaluation, and using the data type of the result variable for the evaluation would have avoided the integer overflow. In this case, a cast of the data type of the result variable before the evaluation eliminates the overflow. Senx repairs such vulnerabilities by adding in such a cast. In an unrepairable integer overflow, the data type of the result variable matches the data type of the evaluation. In this case, the vulnerability can only be addressed by detecting that the overflow is imminent and invoking error handling code.

Predicate Placement. For the first form of the patches, Senx then needs to select a location to place the predicate. Ideally, Senx tries to place the predicate in the same function scope as the vulnerability. However, as mentioned earlier, sometimes the predicate cannot be evaluated at the location where a vulnerability occurs and must be placed at a point in the program where all required symbols are in scope. For example, bcause the size of `dest` is `x*y + 1` in the scope of `foo_malloc`, it cannot be evaluated in `bar`. As a result, Senx searches up the call stack for a context where all symbols in the predicate are available and contains error handling code, and finds that `foo` is a common parent to both functions and that the necessary symbols can be translated using *symbolic translation* into the context of `foo`.

Error Handling Code. Senx uses Talos to locate existing error handling code in an application [23]. Within a function, Senx places the patch in the earliest point of the function where the predicate can be evaluated. It relies on the error handling code located by Talos to release any resource aquired before the evaluation of the predicate.

C. Loop Analysis

Buffer overflows involve memory accesses that are mappable to a loop induction variable that represents the number of iterations the loop executes for. To determine whether a loop will overflow the buffer, Senx needs to symbolically compute

the number of loop iterations a loop will execute for, map that to the access range and compare that with the size of the buffer being accessed.

In general, statically extracting the number of loop iterations is undecidable as it is reducible to the halting problem. As a result, Senx uses a dynamic method, which we call *loop cloning* to extract the number of iterations. The idea behind loop cloning is to use the same logic the program uses to execute the loop to compute the number of iterations. It performs a program slice of the loop to keep all the loop logic, but remove any instructions that may have side effects (i.e. they modify memory or make system calls), and includes the cloned loop in the predicate to compute the number of iterations.

However, loop cloning cannot be used in cases where instructions that have side effects cannot be sliced away from the logic that computes the number of iterations. For example, the exit condition of a loop maybe depend on a function that can make system calls. As a result, such loops cannot be cloned. In such cases, Senx falls back to *access range analysis*. Access range analysis attempts to statically normalize the loop and then extract the number of iterations from the normalized form of the loop. Both loop cloning and access range analysis are described in more detail in Section III.

In our example, Senx recognizes that the vulnerable access on line 42 occurs in a nested loop. However, one of the loop induction variables `q` is involved in a write to memory because its value is assigned to `p`. Because Senx's slicing must output code as well, it currently can only slice out code at the granularity of a line of code, so this loop cannot be cloned. As a result, Senx performs access range analysis, which computes an access range for the loops as `(cols + 1)*(size/cols)`. The details of this operation on Listing 1 are given in Section III-C. Finally, Senx will generate a predicate which compares this with the size of the buffer into which `p` is writing.

D. Limitations

Senx does not generate a patch if it cannot guarantee soundness. To summarize, there are 4 conditions under which Senx will not generate a patch: 1) the symbolic expression builder determines there may be other reaching definitions to variables involved in the predicate, 2) predicate generation fails because some variables cannot represented symbolically, usually because Senx is unable to determine the number of iterations for a complex loop, 3) Senx is unable to place the patch at a location where all variables in the predicate are in scope. In practice, these limitations arise in roughly 20% of cases – our evaluation shows that Senx is successful in patching 33 of the 42 vulnerabilities we evaluated in Section V.

Senx's other limitation is that to determine whether the symbolic variables over which it computes predicates have aliases, Senx uses the alias analysis provided by LLVM [7] to check whether any memory writes along the execution path from where a patch is to be placed until where a vulnerability manifests, e.g. buffer overflow occurs, can alias with any variables involved in a patch predicate. As a result, the pointer analyses need only be precise along this short path, and in many cases there are not even any pointers being dereference along that path at all. As a result, while imprecision in pointer analysis has the potential to cause Senx to return incorrect patches, in practice it does not appear to be a major factor. We note that Senx similarly also assumes that variables are not prone to races, and similar to pointer analysis, that the reaching definitions it computes statically hold at runtime.

III. Design

We now describe the major components of Senx in detail. We first describe the *symbolic expression builder* which builds symbolic expressions along the concolic execution of a target program. We then describre Senx's two loop analysis techniques: *access range analysis* and *loop cloning*, which generate symbolic expressions to represent the access range of the pointer for simple loops and complex loops respectively. Finally, we describe *symbolic translation*, which translates the symbolic expressions denoting the access range and the buffer range into a common scope.

A. Symbolic Expression Builder

We leverage concolic execution to build symbolic expressions used for synthesizing a patch for a target program. While we base our concolic execution engine on KLEE [17], we do not use the symbolic representation that KLEE uses as it is heavily tied to maximizing path exploration, and does not store enough information to easily translate expressions back into source code to construct patches. As a result, we design our own symbolic representation as a sequence of pseudo instructions defined in Table I.

The instructions include `Load` and `Store` memory access instructions, `BinOp` binary operations such as arithmetic operations and `CmpOp` comparison operations such as $>$ and $\geq$, `StructOp` struct operations that access a field of a struct, `ArrayOp` array operations that access an element of an array, `Allocate` for local variable allocation, `Branch` for unconditional and conditional branches, `Call` for function calls and `Ret` for function returns. Each instruction can have an optional label denoted as `label`. The concolic execution uses a program counter that points to the current instruction, which is referred to as `PC` in the table. For each instruction presented in Column "Instruction", the concolic execution interprets it using the semantic operation indicated in Column "Semantic". The results of these operations are stored in Single Static Assignment (SSA) form such that each instruction instance has a unique variable associated with it. The execution makes not distinction between registers and memory.

Symbolic expressions are generated using the rules corresponding to each instruction in Column "Rule to Build Symbolic Expression", the symbolic expression builder builds one or more symbolic expressions for the instruction or any relevant instruction. Each symbolic expression is of the form LHS := RHS, where LHS and RHS are the left side and right

TABLE I
RULES FOR BUILDING SENX SYMBOLIC EXPRESSIONS.

Instruction	Semantic	Rule to Build Symbolic Expression	Description
val = Load var	$val \leftarrow var$	RHS := getExpr(var)	read from var
val = Load $*p$	$val \leftarrow *p$	RHS := makeDeref(getExpr(p))	read via pointer p
Store $var, val1$	$p \leftarrow val1$	RHS := getExpr($val1$), LHS := var	write $val1$ to var
Store $*p, val1$	$*p \leftarrow val1$	RHS := getExpr($val1$), LHS := makeDeref(p)	write via pointer p
val = GetElement $var, field$	$val \leftarrow$ StructOp($var, field$)	RHS := makeStructOp($var, field$)	read from a struct field
val = GetElement $var, index$	$val \leftarrow$ ArrayOp($var, index$)	RHS := makeArrayOp($var, index$)	read from an array element
val = BinOp $val1, val2$	$val \leftarrow$ BinOp($val1, val2$)	RHS := makeBinOp(getExpr($val1$), getExpr($val2$))	binary operations
val = CmpOp $val1, val2$	$val \leftarrow$ CmpOp($val1, val2$)	RHS := makeCmpOp(getExpr($val1$), getExpr($val2$))	comparisons
val = Allocate $size$	$val \leftarrow$ Allocate($size$)	RHS := getName(val)	allocate a local variable
Branch $label$	$PC \leftarrow label$	N/A	unconditional branch
Branch $cond, label1, label2$	$PC \leftarrow label1$ if $cond$	N/A	conditional branch
	$PC \leftarrow label2$ if $\neg cond$	N/A	
val = Call $f(a, \ldots)$	$val \leftarrow f(a, \ldots)$	RHS := makeCall(getName(f), getExpr(a), ...)	call function f with $a, \ldots$
Ret $val1$	$val \leftarrow val1$ && $caller.val \leftarrow val1$	RHS := getExpr($val1$) && $caller$.RHS := getExpr($val1$)	return *val1* to caller

side of an assignment respectively. An explicit LHS is used only for Store instructions. The LHS for all other instructions is the SSA value associated with each instruction. The concolic execution maintains a call stack so that each *Ret* instruction sets a value in its caller, denoted by `caller`, with the return value.

The symbolic expression builder uses several helper functions as described in Table IV in the Appendix. Each of these functions generate a symbolic expression according to their description. For example, `makeDeref("p")` returns "$\star$p", where $*$ represents pointer dereference. In keeping with SSA, the symbolic expressions generated for an instruction are stored along with the instruction. In this way, the symbolic expressions associated with an instruction can easily be retrieved by referring to the instruction.

Complex Data Types. Because the patch generated by Senx is in the form of the source code of a target program, the symbolic expressions must conform to the proper language syntax of the program.

Symbolic expressions for simple data types such as char, integer, or float, are generated in a rather straightforward way. However, symbolic expressions for complex data types such as C/C++ structs and arrays are more challenging. 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.

To address the challenge, we include the `GetElement` instruction, which reads a field from a struct or an element from an array, in Senx's symbolic instruction set. The symbolic expression builder leverages the `GetElement` instructions and debug symbols that describe the ordered list of struct fields to construct symbolic expressions denoting access to complex data types including arrays and structs. `GetElement` is overloaded, but since the symbolic expression builder maintains the data type of each variable, it calls the appropriate version based on the type passed in `var`. To generate valid C/C++ code for a symblic expression, it retrieves the variable expression associated with `var`. If `var` is an array, it uses the helper function `makeArrayOp`, which recursivly generates code associated with the `index` argument. If `var` is a struct, it calls the helper function `makeStructOp` , which returns the name of the field in the struct. To determine whether an access to the struct is via a pointer or directly to an object, it checks witherh `var` is a result of a `Load` instructino or not, and generates the symbolic expression accordingly.

In order to build symbolic expression for complex data access involving nested complex data types, both `makeArrayOp` and `makeStrutOp` use the symbolic expression for the variable `var`, which can be the result of a previous `Load` instruction or `GetElement` instruction. In this way, symbolic expressions for complex data access such as `foo`$\rightarrow$`f.bar[10]`, where `foo` is a pointer to a struct that has a field `f` and `bar` is an array belonging to `f`, can be constructed.

B. Loop Cloning

To generate a predicate for a buffer overflow vulnerability, Senx must compare the memory range a loop may read or write to with the size of the buffer being read or written. The latter is extracted by the symbolic expression builder, so we focus on loop cloning and access range analysis to describe how the memory range of a loop is calculated. Both loop cloning and access range analysis are functions in Senx that take as input a function `F` in the 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 converted into source code and compared with the allocated buffer range in the predicate.

The key idea of loop cloning is to produce new code that can be called safely at runtime to return the access range without causing any side-effects, i.e. changing program state or affecting program input/output. The new code is constructed from existing code, referred to as *cloning*, and will be called at a location where the buffer range is available so that the access range returned by the new code can be compared against the buffer range.

Because the patch must be inserted into a function where both the access range and buffer range are available, loop

```
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 2. 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 3. A cloned and sliced loop that no longer contain 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.

cloning first searches on the call chain that leads to `F` to find such function. The search starts from the immediate caller of `F` and stops at the first function F_p in which the buffer range is available. Note that this means that loop cloning cannot be applied to the case when both buffer allocation and buffer access exist in the same function.

If no such function can be found, Senx will not be able to generate a patch. If such function is found, loop cloning then clones each function F_i along the call chain from `F` until F_p into the new code that returns the access range. As a result, each F_i is either a direct or indirect caller of `F` or `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 F_i is cloned using the following steps.

1) Loop cloning clones the entire code of F_i into F_i`_clone`.
2) Using program slicing, it removes all statements that are not needed in order to computer the access range or pass the access range to F_p. If F_i is `F`, it retains statements on which the execution of `inst` is dependent. If 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 F_i`_clone` to `void` and removes any return statement in F_i`_clone`.
4) It adds two output parameters `start` and `end` to F_i`_clone`. If 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 F_i is a caller of `F`, it changes the call statement to include the two output parameters in the list of call arguments.

After cloning each F_i, loop cloning inserts a call to the last cloned function into 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 2, which presents a loop adopted from a real buffer overflow vulnerability CVE-2007-1887 [11] 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 3.

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 buffer range is available. Because `udf_decode` directly calls `decode`, it needs to clone `decode` only.

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 generate a predicate that will check whether `in` can exceed the end of the buffer it is pointing 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.

C. Access Range Analysis

Access range analysis takes as input a function f and a memory access instruction $inst$ in f, and outputs the range of the memory access $[A_1, A_n]$ as a pair of symbolic expressions.

Using LLVM's built-in loop canonicalization functionality [18], access range analysis computes the access range of normalized 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 fairly straight forward. 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 enclosing $inst$ starting with the innermost loop and iterating to the outermost, accumulating increments and decrements on the loop induction variables including the pointer used by $inst$.

Since the loop in `bar` of Listing 1 can be normalized, we use it as an example of how Algorithm 1 can be applied to a nested loop. So f is `bar` and $inst$ is the memory write using pointer `p` at line 42. For each loop, it first retrieves the loop iterator variable and the bounds of it by calling helper function `find_loop_bounds`, and the list of induction variables of the loop along with the $update$ to each of them, which we refer to as the fixed amount that is increased or decreased to an induction variable on each iteration of the loop, 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 41-42.

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 a symbolic 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 symbolic 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 22 in Algorithm 1.

Once this is done, it moves on to analyze the next loop enclosing $inst$, which in Listing 1 is the while loop enclosing the inner for loop. As a consequence, we will have $iter$ = `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 39 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)` by adding the initial value `dest` to the accumulated update of `p`, `(cols+1)*(size/cols)` from acc. Hence it returns [`dest`, `dest+(cols+1)*(size/cols)`] as the symbolic expressions denoting the access range $[A_1, A_n]$.

D. Symbolic Translation

When generating predicates, sometimes the buffer allocation and size is computed in one function scope, while the memory

Algorithm 1 Finding the access range of a memory access.

Input: f: a function
$inst$: a memory access instruction
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**

access range or bad offset is computed in a different function scope. However, since the patches Senx generates are source code patches, the predicate of the patch must be evaluated in a single function scope. Symbolic Translation solves this problem by translating a symbolic expression exp_s from the scope of a source function f_s to an equivalent symbolic expression exp_d in a scope of a destination function f_d. Senx uses symbolic 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, symbolic translation 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, symbolic translation can iteratively translate expressions that are passed to function invocations across edges in the call graph. Formally, symbolic translation can converge the comparison between a expression exp_a , the symbolic memory access location in f_a and exp_s, 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 substitution, although the use of such kind of variables is not very common. We refer to both function parameters and these kind of variables collectively as nonlocal variables. And we refer to an expression consists of only nonlocal variables as a nonlocal expression.

The low-level implementation of symbolic translation in Senx consists of two functions. One, `translate_se_to_scopes`, identifies all candidate functions along the call stack of a function to translate a particular expression to. For example, in Listing 1, it would translate the arguments to `malloc` at line 2 to the scope of it's caller `foo` at line 29, and repeatedly do this for `foo`'s caller. `translate_se_to_scopes` relies on a helper function `make_nonlocal_expr`, which for each scope, translates a local expression into an equivalent expression that consists only of references to nonlocal expressions (i.e. global variables or function parameters). Together, these two functions produce equivalent symbolic expressions for every caller in a function's call stack. For the sake of space, we leave the detailed description of these two functions and how they work on Listing 1 to Appendix Section VII-B.

IV. Implementation

We have implemented Senx as an extension of the KLEE concolic execution engine [17]. Like KLEE, Senx works on C/C++ programs that are compiled into LLVM bitcode [39].

We only re-use the LLVM bitcode execution portion of KLEE, and as described in Section III-A, implement our own symbolic execution engine. The reason is that KLEE's symbolic execution engine is optimized for path coverage, while Senx only needs to execute the vulnerable path. Similarly, KLEE's symbolic expression builder is tightly tied to its exploration engine and is not designed for easy translation back into C/C++ source code. As a result, we felt it would be easier to start with a fresh design for Senx's symbolic expression buildler.

For simplicity and ease of debugging, we represent our symbolic expressions as text strings. To support arithmetic operations and simple math functions on symbolic expressions, we leverage GiNaC, a C++ library designed to provide support for symbolic manipulations of algebra expressions [22].

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 [18]. We extend LLVMSlicer [15] for loop cloning. To locate error handling code, we use Talos [23].

Our memory allocation logger uses KLEE to interpose on memory allocations and stores the call stack for each memory allocation. The concolic executor extends KLEE to detect integer overflows and incorporates the existing memory fault detection in KLEE to trigger our patch generation. Senx also leverages LLVM's built-in pointer analysis [7].

Senx is implemented with 2,543 lines of C/C++ source code, not including the Talos component used to identify error handling code. Half of the source code is used to implement symbolic expression builder, which forms the foundation of other components of Senx.

V. Evaluation

First, we evaluate the effectiveness of Senx in fixing real-world vulnerabilities. Second, we manually examine 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. Last, we measure the applicability of loop cloning, access range analysis, and symbolic translation using a larger dataset.

A. Experiment Setup

We build a corpus of vulnerabilities for Senx to attempt to patch by searching online vulnerability databases [5], [6], [8], software bug report databases, developers' mailing groups [3], [10], [12], and exploit databases [9]. 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 `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

TABLE II
APPLICATIONS FOR TESTING REAL-WORLD VULNERABILITIES.

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

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 [1], [2], [4], [9], [14], [16].

From this, we construct a corpus of 42 real-world buffer overflow, integer overflow and bad-offset vulnerabilities to evaluate the effectiveness of Senx in patching vulnerabilities. The vulnerabilities are drawn 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 consit of 19 buffer overflows, 13 integer overflows, and 10 bad-offset vulnerabilities.

All our experiments were conducted on a desktop with 4-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 examine the patch for correctness. To determine if a patch is correct, we apply the three following tests a) we check for semantic equivalence with the official patch released by the vendor if available and semantic correctness by analyzing the code, b) we apply the patch and verify that the vulnerability is no longer triggered by the input and c) we check as best we can that the patch does not interfere with regular operation of the application by using the application to process benign inputs. 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 ① buffer overflow, ② bad-offset, or ③ integer overflow. Column "Symbolic Expressions" shows whether Senx can successfully construct all symbolic expressions that are required to synthesize a patch, as some code constructs may contain expressions outside of the theories Senx supports in its symbolic expression 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 indicies. Finally, "Patched?" summarizes whether the patch generated by Senx fixes a vulnerability. The 9 vulnerabilities where Senx aborts generating a patch are highlighted in red.

Over the 42 vulnerabilities, Senx generates 33 (78.6%) patches, all of which are correct according to our 3 criteria. Of the 14 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 calls 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 (46.2% of cases), which have to compare a buffer allocation with a memory access range. This illustrates that symbolic 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 9 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. 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 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. Patch Case Study

Out of the 33 generated patches, we select 2 patches to describe in detail.

CVE-2017-5225. 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

TABLE III
PATCH GENERATION BY SENX

App.	CVE#	Type	Symbolic Expressions	Loop Analysis	Patch Placement	Data Access	Patched?
sqlite	CVE-2013-7443	③	Determinate	—	Failed	—	✗
	CVE-2017-13685	③	Determinate	—	Trivial	Simple	✓
zziplib	CVE-2017-5976	①	Determinate	Cloned	Translated	Complex	✓
	CVE-2017-5974	③	Determinate	—	Translated	Complex	✓
	CVE-2017-5975	③	Determinate	—	Translated	Complex	✓
Potrace	CVE-2013-7437	②	Determinate	—	Trivial	Complex	✓
libming	CVE-2016-9264	③	Determinate	—	Trivial	Simple	✓
libtiff	CVE-2016-9273	①	Indeterminate	—	—	—	✗
	CVE-2016-9532	①	Determinate	Cloned	Trivial	Complex	✓
	CVE-2017-5225	①	Determinate	ARA	Trivial	Simple	✓
	CVE-2016-10272	①	Determinate	ARA	Translated	Simple	✓
	CVE-2016-10092	③	Determinate	—	Translated	Simple	✓
	CVE-2016-5102	③	Determinate	—	Trivial	Simple	✓
	CVE-2006-2025	②	Determinate	—	Trivial	Complex	✓
libarchive	CVE-2016-5844	②	Determinate	—	Trivial	Complex	✓
jasper	CVE-2016-9387	②	Determinate	—	Trivial	Complex	✓
	CVE-2016-9557	②	Determinate	—	Trivial	Complex	✓
	CVE-2017-5501	②	Determinate	—	Trivial	Complex	✓
ytnef	CVE-2017-9471	①	Determinate	Cloned	Trivial	Simple	✓
	CVE-2017-9472	①	Determinate	Cloned	Trivial	Simple	✓
	CVE-2017-9474	①	Determinate	Failed	—	—	✗
php	CVE-2011-1938	①	Determinate	ARA	Translated	Simple	✓
	CVE-2014-3670	①	Determinate	ARA	Translated	Complex	✓
	CVE-2014-8626	①	Determinate	Cloned	Trivial	Simple	✓
binutils	CVE-2017-15020	①	Determinate	ARA	Translated	Simple	✓
	CVE-2017-9747	①	Determinate	Cloned	Translated	Simple	✓
	CVE-2017-12799	③	Determinate	—	Trivial	Simple	✓
	CVE-2017-6965	③	Determinate	—	Failed	—	✗
	CVE-2017-9752	③	Determinate	—	Translated	Simple	✓
	CVE-2017-14745	②	Determinate	—	Failed	—	✗
autotrace	CVE-2017-9151	①	Indeterminate	—	—	—	✗
	CVE-2017-9153	①	Indeterminate	—	—	—	✗
	CVE-2017-9156	①	Determinate	ARA	Trivial	Simple	✓
	CVE-2017-9157	①	Determinate	ARA	Trivial	Simple	✓
	CVE-2017-9168	①	Determinate	Failed	—	—	✗
	CVE-2017-9191	①	Determinate	ARA	Failed	—	✗
	CVE-2017-9161	②	Determinate	—	Trivial	Simple	✓
	CVE-2017-9183	②	Determinate	—	Trivial	Complex	✓
	CVE-2017-9197	②	Determinate	—	Trivial	Complex	✓
	CVE-2017-9198	②	Determinate	—	Trivial	Complex	✓
	CVE-2017-9199	②	Determinate	—	Trivial	Complex	✓
	CVE-2017-9200	②	Determinate	—	Trivial	Complex	✓

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.

Because both the buffer range and the access range starts 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.

CVE-2016-5844. This integer overflow in the ISO parser in libarchive can result in a denial of service via a spe-

cially 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 a symbolic expression of the overflown value as 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. Applicability

We evaluate how applicable of loop cloning, access range analysis and symbolic 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 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. We measure how often symbolic 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. For the sake of space, we describe the details of these experiments in the Appendix Section VII-C.

VI. Related Work

A. Automatic Patch Generation

Leveraging Fix Patterns. Similar to Senx, some other automatic patch generation APR techniques also leverage fix patterns or models to generate patches.

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 [24]. PAR is able to generate patches for 23% of the 119 bugs from six Java projects. 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.

Focusing on memory leak bugs, LeakFix defines a fix as the only deallocation statement for a memory chunk which must be executed after the allocation statement of the memory chunk and after any use of the memory chunk [21]. By labelling program statements related to memory allocation, deallocation, and usage, and abstracting a program into a CFG containing only those related program statements, LeakFix transforms the problem of finding a fix for a leaked memory chunk into searching for an edge where a pointer expression always points to the memory chunk can be constructed, no execution path covering the edge has a dealloction statement for the memory chunk, and no use of the memory chunk exists on the outgoing path of the edge. LeakFix successfully generates patches for 28% of the 89 reported leaks in the SPEC2000 benchmark.

SPR fixes a defect with transformation schemas [28]. For a defect, it identifies a statement in a target program as a repairing target, then selects a transformation schema, from a list of transformation schemas, to modify the statement by associating an abstract function with it. After that, it repeatedly runs the modified program to discover the correct values that this abstract function should return for both inputs triggering the defect and inputs not triggering the defect. By recording the values of all variables accessible at the identified statement, it tries to find a symbolic expression involving a subset of the variables to act as the abstract function. This symbolic expression is then used to synthesize a patch.

Using Program Mutations.

GenProg is a pioneering work that induces program mutations, i.e. genetic programming, to generate patches [44]. 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 [26].

Based on an analysis on the patches generated by state-of-the-art generate-and-validate APRs, including GenProg [44], RSRepair [35], and AE [43], Kali generates patches that only delete functionality [36]. The analysis finds that the test suites used by those patch generators to determine whether a patch correctly fixes a bug often fail to do so because the test suites do not even check whether the patched program produces correct output. By augmenting the test suites with checks on program output, they find that the vast majority of the patches claimed to be correct by these patch generators are actually

incorrect. This indicates that relying merely on test suites to verify the correctness of automatically generated patches can cause misleading results.

Applying SMT Solver. SemFix uses constraint solving to find the needed symbolic expression to repair the right hand side of an assignment statement or the condition checked by an if statement [34]. By executing a target program symbolically with both inputs triggering a defect and inputs not triggering the defect, it identifies the constraints that the target program must satisfy to process both kinds of inputs correctly. It 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.

Similar to SemFix, Angelix runs the target program symbolically with concrete inputs to discover the constraints that a target program should satisfy to fix a defect and then uses component-based program synthesis to generate a patch [31]. Different from SemFix, it considers more than one statements in a target program so that it is capable of generating a patch that modifies more than one statements in the target program.

Learning from Correct Code. Prophet learns from existing correct patches [29]. 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 APRs, its effectiveness depends on the quality of the test suites.

Unlike Prophet, CodePhage intends to borrow code from the binary code of a donor program and translate it into a source code patch for a defect in a target program [37]. A donor program is a program that reads the same inputs as the target program, and correctly handles both the input that can trigger the defect in the target program and the input that do not trigger the defect. CodePhage searches for a donor program from its list of donor programs. After finding a donor program for the defect, CodePhage searches for a check in the donor program that returns opposite values for the two inputs and considers it as a candidate check. It then runs the donor program to produce symbolic expressions that denote the check as a function that determines the values of some input fields. Finally it searches for locations in the target program where the symbolic expressions can be translated to valid source code as a patch for the target program to check against the same input fields.

The way that CodePhage borrows code from one program and translates the code into another program is analogous to the symbolic translation used by Senx, which instead translates between different scopes of one program.

B. Prevention of Exploiting Security Vulnerabilities

Fortifying Program Code. One way to hinder exploits to vulnerabilities is by fortifying programs to make them more robust to malicious input. Software Fault Isolation (SFI) instruments bounds checks before memory operations to ensure that they cannot corrupt memory [33], [41], [45]. Alternatively, Control Flow Integrity (CFI) learns valid control flow transfers of a program and validates control flow transfers to prevent execution of exploit code [20], [40], [46], [47].

By contrast, Talos introduces the notion of Security Workarounds for Rapid Response (SWRR) and applies them to existing programs to stop malicious inputs [23]. By statically analyzing program code, Talos identifies existing error handling code and uses it to synthesize SWRRs, which can be enabled or disabled by users dynamically at runtime to thwart malicious attacks.

Rectifying Inputs. Some techniques rectify malicious program 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 [27].

A2C exploits the observation that exploit code embedded in inputs is often fragile to any slight changes. By encoding inputs with an one-time dictionary and decoding them only when the program execution goes beyond the often vulnerable code, it disables the embedded exploit code and turns it into a program termination [25].

Filtering Inputs. Alternative to rectifying inputs, some techniques detect and simply filter out malicious inputs [19], [30], [38], [42]. 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 [19]. Shields models a vulnerability as a protocol state machine and constructs network filters based on it [42].

VII. Conclusion

We present the design and implementation of Senx, a system that automatically generates patches for buffer overflow, bad offset, and integer overflow vulnerabilities. Senx can synthesize patches in one of two forms.

For a program that manifests a buffer overflow or a bad offset, Senx synthesizes a patch in the first form that uses a predicate to check whether a faulty memory access is about to occur and prevents the faulty memory access by steering the program execution to error handling code, similar to a patch written by human developers. For a program that manifests an integer overflow, Senx can synthesize a patch in the second form that adds a data type cast to avoid the integer overflow or a patch in the first form which checks if the integer overflow is imminent and invokes error handling code.

Senx leverages three novel techniques, symbolic translation, loop cloning, and access range analysis, to construct a patch. Enabled by the three techniques, Senx generates patches correctly for 33 of the 42 real-world vulnerabilities.

REFERENCES

[1] "agostino's blog," http://blogs.gentoo.org/ago.

[2] "agostino's poc repository," http://github.com/asarubbo/poc.

[3] "bug-coreutils Archives," http://lists.gnu.org/archive/html/bug-coreutils/.

[4] "Bugs.MapTools.Org," http://bugzilla.maptools.org/.

[5] "Common Vulnerabilities and Exposures," http://cve.mitre.org.

[6] "CVE Details," http://www.cvedetails.com.

[7] "LLVM Alias Analysis Infrastructure," http://llvm.org/docs/AliasAnalysis.html.

[8] "National Vulnerability Database," http://nvd.nist.gov.

[9] "Offensive Security's Exploit Database Archive," http://www.exploit-db.com.

[10] "PHP Bug Tracking System," http://bugs.php.net.

[11] "PHP5 sqlite_udf_decode_binary() Buffer Overflow Vulnerability," http://www.php-security.org/MOPB/MOPB-41-2007.html.

[12] "Red Hat Bugzilla," http://bugzilla.redhat.com.

[13] "Security issue in libav/ffmpeg," http://www.openwall.com/lists/oss-security/2012/05/03/4.

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

[15] "Static Slicer for LLVM," http://github.com/jirislaby/LLVMSlicer.

[16] "Welcome to Sourceware Bugzilla," http://sourceware.org/bugzilla/.

[17] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[18] Cannoicalize natural loops, http://llvm.org/docs/Passes.html#loop-simplify-canonicalize-natural-loops.

[19] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 117–130. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294274

[20] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 292–307. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.26

[21] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 459–470. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818812

[22] GiNaC is Not a CAS, http://www.ginac.de/.

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

[24] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486893

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

[26] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "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.

[27] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 80–90. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337233

[28] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 166–178. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786811

[29] ——, "Automatic Patch Generation by Learning Correct Code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837617

[30] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 439–452. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535888

[31] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884807

[32] M. Monperrus, "A critical review of automatic patch generation learned from human-written patches: Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 234–242. [Online]. Available: http://dl.acm.org/citation.cfm?id=2568324

[33] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: better, faster, stronger SFI for the x86," in *Proceedings of the 2012 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 395–404. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254111

[34] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486890

[35] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254

[36] Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 24–36. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771791

[37] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 43–54. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737988

[38] M. Süßkraut and C. Fetzer, "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. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/DSN.2007.84

[39] The LLVM Compiler Infrastructure, http://llvm.org/.

[40] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, 1994, pp. 203–216.

[42] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "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*, ser. SIGCOMM '04. New York,

NY, USA: ACM, 2004, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015489

[43] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 356–366. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693094

[44] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070536

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

[46] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[47] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 312–321. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486830

APPENDIX

A. Symbolic Expression Builder Helper Functions

The symbolic expression builder's rules call a number of helper functions that extract and create symbolic expressions to match C/C++ code constructs. These functions are listed in Table IV.

TABLE IV
OPERATIONS PERFORMED BY SYMBOLIC EXPRESSION BUILDER.

Operation	Description
getExpr	get the symbolic expression associated with an instruction or the name of a variable
getName	get the name of a variable
makeDeref	build a symbolic expression to denote dereference
makeBinOp	build a symbolic expression to denote a binary operation including *arithmetic* operations, bitwise *logic* operations, and bitwise *shift* operations
makeCmpOp	build a symbolic expression to denote a comparison including $<, >, =, \neq, \geq, \leq$
makeStructOp	build a symbolic expression to denote an access to a struct field directly or via a pointer
makeArrayOp	build a symbolic expression to denote an access to an array element
makeCall	build a symbolic expression to denote a function call including the name of the function and all the arguments

B. Symbolic Translation Details

Function `translate_se_to_scopes` listed in Algorithm 2 is the core of symbolic translation. It translates a symbolic expression $expr$ to the scope of each function on the call stack $stack$. We illustrate how it works with the code in Listing 1. 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 29], $expr$ ="`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`.

It then iterates each call instruction in $stack$, starting from line 29. For each call instruction, it substitutes the parameters in $expr$ with the arguments used in the call instruction. For line 29, 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 29 exists in function `foo`. After that, it tries to convert "`rows*(cols+1)+1`" into a nonlocal definition in 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, symbolic translation is also able to translate the memory access range expression from the scope of `bar` into the scope of `foo`. Thus, Senx uses symbolic translation to place the patch predicate in `foo`.

C. Applicability of Analysis Techniques

We use 11 programs from the coreutils as listed in Table V 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 symbolic 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 VI. The "Access Range" column tabulates the average percentage of functions in the loop's call stack that symbolic 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 symbolic translation could translate the buffer allocation size into. Finally "Converged" indicates out of

Algorithm 2 Translating a symbolic 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 symbolic 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 a symbolic 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 ≠ ∅ then
6:         for call ∈ 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] ← expr
11:            expr ←make_nonlocal_expr(func, call, expr)
12:            if expr = ∅ then
13:                break
14:            end if
15:        end for
16:    end if
17:    return translated_exprs
18: end procedure
```

TABLE V
PROGRAMS FOR EVALUATING APPLICABILITY.

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

all loops, what percentage could symbolic translation find a 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.

Algorithm 3 Making a nonlocal expression.

Input: f: a function
$inst$: a an instruction in f
$expr$: a symbolic 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 ← ∅
4:     for var ∈ expr do
5:         if ¬ is_var_nonlocal(f, var) then
6:             def ←find_nonlocal_def_for_var(f, inst, var)
7:             if def = ∅ then
8:                 ▷ We cannot find a nonlocal definition for var
9:                 return ∅
10:            else
11:                mapping[var] ← 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
```

TABLE VI
SYMBOLIC TRANSLATION.

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%