Using Safety Properties to Generate Vulnerability Patches

ZHEN HUANG, DAVID LIE, GANG TAN, AND TRENT JAEGER

Zhen Huang is an Assistant Professor in the School of Computing at DePaul University. He earned his BASc from Wuhan University, and his MS and PhD from the University of Toronto. He works on computer systems with an emphasis on software security. zhen.huang@depaul.edu

David Lie received his BASc from the University of Toronto in 1998, and his MS and PhD from Stanford University in 2001 and 2004, respectively. He is currently a Professor in the Department of Electrical and Computer Engineering at the University of Toronto. He also holds appointments in the Department of Computer Science, the Faculty of Law, and is a research lead with the Schwartz Riesman Institute for Technology and Society. He was the recipient of a best paper award at SOSP for this work. David is also a recipient of the MRI Early Researcher Award and the Connaught Global Challenge Award. David has served on various program committees including OSDI, USENIX Security, IEEE Security & Privacy, NDSS, and CCS. Currently, his interests are focused on securing mobile platforms, cloud computing security, and bridging the divide between technology and policy. lie@eecg.toronto.edu

Automatic Program Repair (APR) methods attempt to fix vulnerabilities in programs comprehensively and without introducing new defects. Senx uses novel safety properties to generate patches, and it succeeds in generating patches for 32 of 42 real-world vulnerabilities. We explain how Senx works, compare it to other APR methods, and demonstrate why Senx is better at repairing source code.

Fixing security vulnerabilities in a timely manner is critical to protect users from attacks that exploit vulnerabilities. Unfortunately, a recent study shows that the average time to release software patches for vulnerabilities is 52 days, and the bottleneck lies in creating software patches [1].

Automatic Program Repair (APR) tools aim to automatically provide patches that fix vulnerabilities. Most of them rely on a set of positive/negative example inputs to produce a patch that makes the vulnerable program behave correctly according to these example inputs [4, 6, 7]. The patched program must pass the positive example inputs but raise errors on the negative example inputs. But obtaining a complete set of example inputs is often difficult, and the patched program may behave incorrectly on other inputs, or the vulnerability may still be exploited by other inputs [8]. We refer to this traditional method as "example-based."

We propose a different approach called "property-based" APR that relies on vulnerability-specific, program-independent, human-specified safety properties. A safety property specifies the condition on which a type of vulnerability cannot be triggered. For example, a safety property for buffer overflow vulnerabilities can be that a program should never have access beyond the bounds of a buffer.

Our property-based approach has three major advantages: 1) a small set of safety properties can be defined once and applied on numerous programs without the need to specify anything pertaining to each of the programs; 2) the properties are precise and complete by nature so they work for all possible inputs; 3) it leverages a specific vulnerability's context to generate a customized and efficient patch for the vulnerability, as opposed to the nonspecific and often inefficient patches generated by previous methods [5].

Property-based APR faces several outstanding challenges. First, it must identify the correct property to enforce for a given vulnerability because the properties are vulnerability-specific. Second, our goal is to generate source code patches that can be easily adopted by developers; as a result, the safety properties must be expressed using program entities such as variables. Third, the generated patches should affect program execution if and only if a safety property is violated. Finally, the generated patches should incur minimum performance overhead.

To address these challenges, we have designed Senx to automatically generate source code patches for security vulnerabilities using safety properties. We demonstrate the effectiveness of Senx using three important classes of vulnerabilities: buffer overflows, bad casts, and integer overflows. Our evaluation demonstrates that Senx is able to produce correct patches for over 76% of the vulnerabilities. And we believe that, in principle, Senx can generate patches for any class of vulnerabilities for which a safety property can be specified.

Dr. Tan is a Professor in the Computer Science and Engineering Department at Pennsylvania State University. He obtained his BE in computer science from Tsinghua University, and his PhD in computer science from Princeton University. His research interests are computer security, formal methods, and programming languages. He currently serves on the DARPA ISAT study group. He has also received multiple awards, including a James F. Will Career Development Professorship from 2016 to 2019, an NSF CAREER Award, two Google Research Awards, a Distinguished Reviewer Award at the 2018 IEEE Symposium on Security and Privacy, a Ruth and Joel Spira Excellence in Teaching Award at Penn State, and some best paper awards at academic conferences.
gtan@cse.psu.edu

Trent Jaeger is a Professor in the Computer Science and Engineering Department at Pennsylvania State University. Trent's primary research interests are systems and software security. He has published over 150 research papers and the book *Operating Systems Security*, which has been taught in universities worldwide. Trent has made significant contributions to the Linux community, including mandatory access control, integrity measurement, process tracing, and namespace services. Trent currently serves the computer security research community on the Executive Committee of ACM SIGSAC as Past Chair, as Steering Committee Chair of NDSS, on editorial boards of *Communications of the ACM* and *IEEE Security & Privacy*, and on the Academic Advisory Board of the UK's Cyber Body of Knowledge project.
tjaeger@cse.psu.edu

Example-Based versus Property-Based

We now discuss the limitations of state-of-the-art APR tools. We use the program in Listing 1 as the target program, which is adopted from a real-world buffer overflow vulnerability CVE-2012-0947 in a popular media stream processing library. The program takes a string and its length as input, and outputs the reversed string. It outputs "" if an error occurs. Similar to the real vulnerability, two functions are used, one to allocate the output buffer, and the other to process the input string.

The buffer overflow happens when the size, specified from the command line, is smaller than the actual length of the input string. To fix the buffer overflow, a check can be added to ensure that the actual length of the string is smaller than the allocated size of the buffer into which it is copied. Note that the buffer size is only known to main; so the check should be added at line 19 to compare size against `strlen(argv[2])`. While a human developer can easily add this check, which indeed was in the official patch for the vulnerability, it presents challenges for state-of-the-art APR tools.

```
 1 char* rev(const char *inp, char *out) {
 2    // reverse a string
 3    //  inp is the input string
 4    //  out is an output buffer
 5    if (inp != NULL) {
 6        int i, len = strlen(inp);
 7        // Failed to check if (len + 1 <= size_of_out)
 8        for (i = 0; i < len; i ++)
 9            out[i] = inp[len - i];
10        out[i] = '\0';
11        return out;
12    }  else
13        return "###";
14 }
15
16 void main(int argc, char *argv[]) {
17    int size = atoi(argv[1]) + 1;
18    char *out = (char *)malloc(size);
19    // patch: if (strlen(argv[2]) + 1 > size) exit(1);
20    printf("%s\n", rev(argv[2], out));
21 }
```

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

Example-based approaches. Many APR tools rely on example inputs to fix vulnerabilities. For example, SemFix and Angelix use test inputs to find path constraints needed to generate fixes [4, 6]. Table 1 presents typical test inputs needed to use such tools to fix the buffer overflow for our example in Listing 1.

This approach has two drawbacks. First, the generated path constraints are often based on the concrete values used in the test inputs instead of the relationships between program variables. Given the test inputs in Table 1, SemFix and Angelix would wrongly infer that the value of `argv[1]` is not correlated with whether tests are positive or negative, based on the fact that it has the same values in both positive and negative test inputs.

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

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

Second, the approach is highly sensitive to the completeness of test inputs. Because the length of the input string is smaller than 3 for positive tests whereas the length is not smaller than 3 for negative tests, SemFix and Angelix would incorrectly derive that `strlen(argv[2]) < 3` needs to be added to the program to fix the buffer overflow. The incorrect patch is generated due to the missing of a positive test input with `strlen(argv[2]) > 2`) in the test suite. This illustrates that example-based tools can easily fail when tests are missing in the test suite, which is notoriously hard to make complete.

Property-based approaches. AutoPaG creates patches using a predicate similar to a safety property [3]. But it handles only one vulnerability type, buffer overflows, so it cannot generate a correct patch if the vulnerability is of any other type. Moreover, it would fail to produce a patch if the safety property needs to be enforced in a location other than the function in which the vulnerability occurs. As in our example, the patch should be placed in the `main` function, but the buffer overflow occurs in the `rev` function. Lastly, the patch it generates can incur high performance overhead because it would add the patch to check the buffer size inside the `for` loop on line 8 due to the fact that the buffer overflow occurs within the loop.

Safety Properties

To generate a patch that fixes a vulnerability, Senx requires an input to trigger the vulnerability. The input can be a proof-of-concept exploit or an input generated by a fuzzer. With this input, Senx generates a patch that will enforce the safety property violated by the vulnerability.

A Senx patch can have one of two forms: 1) a check-and-error patch that inserts a check to detect if a safety property no longer holds and raises an error to direct program execution away from the path where the vulnerability resides; 2) a repair patch that modifies existing code to prevent a safety property from being violated.

Each safety property corresponds to a particular vulnerability class and is an abstract Boolean expression that will be mapped to concrete variables in a program. We describe below the three types of safety properties that Senx currently supports.

Sequential buffer overflows. A sequential buffer overflow occurs when a sequence of memory accesses traversing a buffer crosses from a memory location inside the buffer to a memory location outside of the buffer. The Senx safety property for buffer overflows defines two abstract objects: a memory access and a buffer. The term *buffer* refers to any bounded memory region, which may include structs, objects, or arrays. The term *memory access* corresponds to an array access or pointer dereference occurring inside a loop. This safety property covers both the case when the memory access exceeds the upper range of the buffer and the case when the memory access falls below the lower range (sometimes called a *buffer underflow*).

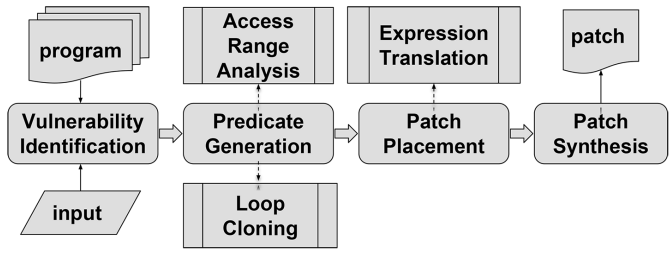

Figure 1: Workflow of Senx: each rounded rectangle represents a step in Senx's patch generation; each rectangle with vertical bars represents a component of Senx.

Bad casts. A harmful memory access can result from an offset from a base pointer beyond the upper bound of the buffer the base pointer is pointing to. This type of vulnerability may occur for several reasons, but it commonly occurs when a pointer is cast to a type that is incompatible with the object the pointer points to. The safety property for bad casts can prevent both bad casts for simple structs and objects, as well as nested structs and objects.

Integer overflows. An integer overflow takes place when a variable is assigned a value larger or smaller than what can be represented by the type of the variable. An integer overflow can lead to a vulnerability when the result of the overflow is then used in operations such as allocating a buffer, producing a buffer that is far smaller than expected. Consequently, the safety property for integer overflows checks that value used in certain operations is not the result of an integer overflow.

For our prototype, we have started with these three vulnerability classes. Nonetheless, they represent a good percentage of CVE vulnerabilities. Based on our informal analysis of the vulnerabilities published in `CVE Details` in 2018, the most popular vulnerability categories are denial of service, code execution, and overflow. By examining 100 randomly chosen CVE reports for each of the three vulnerability categories, we find that 25% of CVE vulnerabilities are buffer overflows, bad casts, or integer overflows. We believe the principles behind Senx can be extended to other vulnerability classes, and we plan to do so as our future work.

Senx

Senx aims to generate source code patches that can be easily verified and adopted by developers. As shown in Figure 1, Senx generates patches in four major steps: vulnerability identification, predicate generation, patch placement, and patch synthesis.

Vulnerability Identification

In vulnerability identification, Senx runs a program with an input that can trigger a vulnerability and outputs the violated

safety property, the *vulnerability point* (the program location where the safety property is violated), and the source code expressions for the execution trace. Senx runs the program using concolic execution to generate the execution trace corresponding to the vulnerability-triggering input. Senx records the execution trace as source code expressions, which conform to the syntax of the programming language of the target program, for synthesizing a source code patch. To support complex data types such as nested C/C++ structs, references to structs, and arrays with pointers, Senx records the relationships between data objects and the way data objects are referenced. This way Senx can recover the full expression for a data object such as `foo→f.bar[10]`.

Predicate Generation

During predicate generation, Senx takes the violated safety property, which also implies the type of the vulnerability, and the source code expressions generated by vulnerability identification, and outputs a predicate required to prevent the violation of the safety property. Senx maps the violated safety property to concrete expressions over variables, constants, and function calls in the form of the source code of the program.

For buffer overflows, Senx aims to insert the patch before the loop where a set of sequential memory accesses occurred; so it needs to extract expressions that represent the memory access range for the memory accesses. Senx uses two complementary loop analysis techniques: *access range analysis* and *loop cloning*. Both of them take a function *F* in the target program and an instruction *inst* that performs the faulty access in the buffer overflow, and output the symbolic memory access range of *inst*.

Access range analysis. Senx computes the access range of canonicalized loops. It relies on LLVM's built-in loop canonicalization functionality to convert the loop into a standard form. It starts with the innermost loop and iterates to the outermost loop, and accumulates increments and decrements on the loop induction variables.

For each loop, Senx retrieves the loop iterator variable and its bounds and the list of induction variables of the loop and their *update*, the fixed amount that an induction variable is increased or decreased by on each loop iteration. We use the loop in `bar` of Listing 2 to illustrate how access range analysis can be applied to nested loops.

```
 1 char *foo_malloc(x,y) {
 2   return (char *)malloc(x * y + 1);
 3 }
 4
 5 int foo(char *input) {
 6+  if ((double)(cols+1)*(size/cols)+1 >
 7+        rows * (cols+1) + 1)
 8+     return -1;
 9   char *output=foo_malloc(rows,cols+1);
10   if (!output)
11      return -1;
12   bar(p, size, cols, output);
13   return 0;
14 }
15
16 void bar(char *src,int size,int cols,char *dest) {
17   char *p=dest;char *q=src;
18   while (q < src+size)  {
19      for (unsigned j=0;j<cols;j++)
20         *(p++) = *(q++);
21      *(p++) = '\n';
22   }
23   *p = '\0';
24 }
```

Listing 2: A buffer overflow in CVE-2012-0947 with a patch, lines prefixed with "+"

In this example, Senx identifies `j` as the loop iterator variable, whose bounds are `0` and `cols`; it also identifies `j`, `p`, and `q` as induction variables, each of which has an update of `1` for the innermost `for` loop. Senx then symbolically accumulates the update to each induction variable based on the number of loop iterations, which is `cols`. Similarly, Senx finds `q` as the loop iterator variable, with `src` as its lower bound and `src+size` as its upper bound, and `q` and `p` as induction variables, whose accumulated update is `size` and `(cols+1)(size/cols)+1`, respectively, for the `while` loop enclosing the inner `for` loop.

Following the analysis of all the loops enclosing *inst*, Senx performs reaching definition dataflow analysis to find the definition that reaches the beginning of the outermost loop for the pointer *ptr* used by *inst*. In this example, we have *ptr*=`p` whose initial value is `dest` before the `while` loop. By adding the initial value `dest` to the accumulated update of `p`, we will have `dest+(cols+1)(size/cols)+1`. Therefore Senx decides the access range as `[dest,dest+(cols+1)(size/cols)+1]`.

Loop cloning. Senx cannot apply access range analysis to loops that LLVM cannot canonicalize. Instead it uses loop cloning for these loops. At a high level, loop cloning creates new code to compute the number of loop iterations. Senx produces the new code from a clone of the code of the loop in the target program, but removes the code that causes side effects. The new code is used by the generated patch to return the access range. Details on loop cloning can be found in [2].

Function calls. For certain cases, Senx can extract expressions containing function calls. Senx needs to ensure that the generated predicate does not call functions that have side effects. We define three types of side effect: 1) a change to the memory accessible outside of a function; 2) an invocation of a system call that has external impact; 3) an invocation of a function that has any side effect.

Senx uses a flow-sensitive, context-insensitive intraprocedural static analysis to identify the list of functions that do not have

any side effect. Senx initializes the list with functions on a whitelist and then adds each function that has no side effect to the list by analyzing every function of a target program.

Patch Placement

In patch placement, Senx uses the vulnerability point found in vulnerability identification and the predicate generated in predicate generation to find a program location to insert the patch. The patch location must be a point where all necessary variables in the predicate are in the scope. If variables in the predicate are from different scopes, Senx uses *expression translation* to translate the predicate into a new one formed from variables in a common scope. For check-and-error patches, Senx also requires the scope to have some error handling code to call. It uses Talos [1] to find a suitable error handling code.

Expression translation. Senx must produce a patch predicate that can be evaluated in a single function scope, because Senx generates source code patches. In some cases, a target program computes the buffer allocation size in one function scope but the memory access range in a different function scope. As a result, the expression representing the allocation size and the expression representing the memory access range are not valid in a single function scope.

To solve this problem, *expression translation* translates an expression from the scope of a source function to an equivalent expression in the scope of a destination function, without the need to add new function parameters and call arguments. This process is called *converging* the predicate. Expression translation exploits the equivalence between the arguments that are passed to a function by the caller and the function parameters that receive the values of the arguments.

We use the code in Listing 2 to illustrate how it works. To translate the buffer size involved in the buffer overflow, Senx starts with the buffer size expression `xy+1` in the scope of `foo_malloc` and for `x` substitutes `rows` and for `y` substitutes `cols+1` based on the call arguments at line 9. Hence `xy+1` becomes `rows(cols+1)+1` in the scope of `foo`.

Effectiveness of Senx

We evaluate the effectiveness of Senx and the quality of its generated patches using 42 real-world buffer overflow, bad cast, and integer overflow vulnerabilities that are from 11 mature and popular applications. For each vulnerability, we run the corresponding application under Senx with a vulnerability-triggering input. We manually examine the correctness of the generated patch if Senx generates a patch. Otherwise, we examine what caused Senx to abort patch generation. The list of the vulnerabilities and our detailed evaluation are presented in [2].

For the 42 vulnerabilities, Senx generates 32 patches, all of which are correct according to our criteria. Senx applies access range analysis and loop cloning roughly equally for the 13 patched buffer overflows. Senx is unable to apply loop cloning mainly because the loops involve calls to functions that have side effects that Senx cannot remove. Senx must use expression translation to generate 23.8% of the patches because the patches need to be placed in a function different from where the vulnerability occurs. The dominant cause for Senx to abort patch generation is that Senx cannot converge all variables in the patch predicate to a common function scope.

Comparison with other work. We compare the effectiveness of Senx against SemFix [6] and Angelix [4]. Due to the considerable effort required to run SemFix and Angelix, we made the comparison on only two vulnerabilities. Senx generates correct patches for both vulnerabilities, while SemFix and Angelix are unable to generate patches either because they cannot find an existing program construct to change in order to pass both positive test inputs and negative test inputs or because they cannot create a guard statement to prevent the vulnerabilities from being triggered.

Conclusion

Automatic patch generation is a promising solution to rapidly resolve software defects. However, the vast majority of these tools are not well-suited to address software vulnerabilities since they rely on test cases to generate correct patches, whereas it is difficult to have complete test cases for any moderately large target programs. To address software vulnerabilities, we built Senx, a system that uses human-specified safety properties to automatically generate patches. Senx uses three novel program analysis techniques: access range analysis, loop cloning, and expression translation. Evaluation shows that Senx generates patches correctly for 76% of the 42 real-world vulnerabilities.

Acknowledgments

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

References

[1] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pp. 618–635.

[2] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using Safety Properties to Generate Vulnerability Patches," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pp. 539–554.

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

[4] 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 (ICSE '16)*, pp. 691–701.

[5] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: Highly Compatible and Complete Spatial Memory Safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pp. 245–258.

[6] 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 (ICSE '13)*, pp. 772–781.

[7] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pp. 87–102.

[8] 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 (ISSTA 2015)*, pp. 24–36.