Testing for Buffer Overflows with Length Abstraction

Ru-Gang Xu[1] Patrice Godefroid[2] Rupak Majumdar[1]
[1]Department of Computer Science, University of California, Los Angeles, USA
[2]Microsoft Research, Redmond, USA
{rxu, rupak}@cs.ucla.edu, pg@microsoft.com

ABSTRACT

We present Splat, a tool for automatically generating inputs that lead to memory safety violations in C programs. Splat performs directed random testing of the code, guided by symbolic execution. However, instead of representing the entire contents of an input buffer symbolically, Splat tracks only a prefix of the buffer symbolically, and a *symbolic length* that may exceed the size of the symbolic prefix. The part of the buffer beyond the symbolic prefix is filled with concrete random inputs. The use of symbolic buffer lengths makes it possible to compactly summarize the behavior of standard buffer manipulation functions, such as string library functions, leading to a more scalable search for possible memory errors. While reasoning only about prefixes of buffer contents makes the search theoretically incomplete, we experimentally demonstrate that the symbolic length abstraction is both scalable and sufficient to uncover many real buffer overflows in C programs. In experiments on a set of benchmarks developed independently to evaluate buffer overflow checkers, Splat was able to detect buffer overflows quickly, sometimes several orders of magnitude faster than when symbolically representing entire buffers. Splat was also able to find two previously unknown buffer overflows in a heavily-tested storage system.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Verification

Keywords: buffer overflows, length abstraction, underapproximation, directed testing, testing C programs

1. INTRODUCTION

Memory safety violations can lead to severe security vulnerabilities. Software containing these errors can lead to denial of service or loss of control to an attacker, costing billions of dollars in damage [3]. Although many techniques and tools have been developed for finding such errors, none have been shown to be 100% effective on realistic code [24]. Proving the absence of these errors using static analysis usually leads to many false warnings due to the lack of precise reasoning about bit operations, pointer arithmetic and arithmetic overflow. Finding inputs leading to such errors using random or systematic testing is also difficult due to the large input space.

Recent advances in test generation using symbolic execution provide an interesting sweet spot between static analysis and random testing. Automatic test input generation aims at finding a set of inputs whose execution covers as many program paths and statements as possible. One such comprehensive method is to generate tests that cover all paths in a program with a *bounded input* using *dynamic* symbolic execution [1, 6]. Obviously, enumerating all feasible program paths becomes problematic as the size and complexity of the program increases. Path explosion can be alleviated by performing test generation compositionally [5], but full path coverage is still problematic for very large applications within a limited search period, say, one night.

In this paper, we argue for a lighter-weight approach to buffer overflow detection using dynamic test generation. Our insight and belief is that tracking all symbolic values contained in input buffers is often more precise than necessary for finding buffer overflows. This extra precision results in a large input space that cannot be searched completely in a reasonable amount of time.

Therefore, we propose a test input generation algorithm that tracks and symbolically reasons about *lengths* of input buffers and strings. This is done by extending symbolic execution with respect to buffer contents to also include a buffer length and a string length (if the buffer contains a string). In order to speed up the search and make it more tractable, symbolic execution only tracks the influence of data values stored in *prefixes* of input buffers, instead of full buffers. We show using code coverage data obtained from preliminary experiments that this underapproximation has the effect of pruning the search space in a *uniform* manner.

As our technique explores an underapproximation of the input space, some errors may be missed. However, this underapproximation finds a wide class of common memory safety violations. In many errors of this type, the length of the input is important while the contents are not. Furthermore, the underapproximation may be tuned to be more precise by making the symbolic input prefix longer. The user can initially use a short prefix and gradually increase the prefix as their testing budget allows. In the limit, all input becomes symbolic, allowing for completeness, but at higher test generation cost.

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

```
unsigned int strlen
             (char *s) {
    char *ptr = s;
    unsigned int cnt = 0;
    while (*ptr != '\0') {
        ++ptr;
        ++cnt;
    }
    return cnt;
}
```

```
void t1(char *s) {
    unsigned int i;
    int A[5];
    i = strlen(s);
    if(i > 4)
         return;
    A[i+1] = 0;
}
```

```
void t2(char *s) {
     unsigned int i, tmp;
     int A[5];
     i = strlen(s);
     if(i > 4)
          return;
     tmp = i+1;
     assert(tmp>=0 && tmp<5);
     A[tmp] = 0;
}
```

Figure 1: Buffer overflow due to an off-by-one error in `t1`. Memory checks are added to `t1` with an assert as shown in `t2`.

Figure 1 shows a possible buffer overflow in procedure `t1` resulting from an off-by-one error when accessing buffer `A`. The input is a string s where the string length i is an index to a buffer of size 5. There is a check to see if the input i is within the bounds of the buffer but the check does not consider that i is incremented before the buffer access. If the input is 4, the bounds check passes, but the illegal write to $A[4+1]$ results in a memory safety violation.

Assuming code for the function `strlen` is available, a directed testing tool [6] will attempt to exercise all feasible program paths and find $n+1$ unique explicit paths for inputs s of length less or equal to n. Indeed, covering all possible execution paths of the `strlen` function for an input size bounded by n requires $n+1$ input tests of different length. In this case, if n is greater than 4, the off-by-one error in accessing the buffer is found simply because all string lengths are enumerated up to n.

Note that this off-by-one error only depends on the size of the input string s, not on its content. To eliminate the redundant inputs due to the unfoldings of the loop in `strlen`, we track only the abstract length of the input string s and instrument the string library, including functions like `strlen`, with additional code that updates abstract lengths.[1] `strlen` would thus be replaced by a function that returns a symbolic length. Then, program variable i would be assigned to this symbolic length. Using directed testing, this would result in two paths to be covered, each satisfying $(i > 4)$ or $(i \leq 4)$ respectively. A decision procedure can produce satisfying assignments for these constraints that allow these paths to be executed.

However, it can happen that neither of the two paths hit the bug (e.g. the two inputs $i = 5$ and $i = 0$). To remedy this, we must include implicit paths by providing instrumentation that tracks allocated memory and adds the appropriate checks before each memory dereference (e.g., see [7]). Function `t2` in Figure 1 shows these checks in the form of an assertion. At each such assertion, we then try to solve for an input that violates the assertion. In this case, the path constraint with the assertion is $(i \leq 4) \wedge (i+1 \geq 5)$ resulting in a symbolic input length $(i = 4)$.

We have implemented this combination of techniques in Splat, a tool for finding memory safety violations in C programs. In the next section, we illustrate further the key features of Splat with a more realistic example. In Section 3, we recall basic notions of directed test generation. In Section 4, we describe the implementation of Splat, and specify how to carry out symbolic execution with symbolic lengths. Section 5 discusses results of experiments with a large set of benchmarks. Our experiments validate the choice of length abstractions, showing that Splat can efficiently find buffer overflows in many programs for which complete symbolic searches do not. The paper ends with a discussion of related work.

[1] An alternative would be to generate such function summaries automatically for bounded domains, using techniques as described in [5].

2. EXAMPLE

Although Splat can find general memory safety violations, we introduce and motivate our technique by examining how we can find buffer overflows in C programs. Figure 2 illustrates a buffer overflow that was present in WuFTP, an ftp server. In this example, the string functions are the standard `string.h` functions. A string is stored in some fixed sized buffer as an array of non-zero 8-bit characters followed by a string terminator represented by a 0 byte. A buffer overflow occurs when a string is copied into some buffer that is too small. This is detected by Splat because copying a character beyond the end of the buffer results in an illegal write. Even though this example is small, it challenges most static analysis and automated test generation tools. Specifically, the path sensitivity and pointer arithmetic causes static tools to report many false alarms, while the large buffers create scalability problems for test generation tools. We demonstrate how Splat overcomes these problems.

Testing Algorithm. Our algorithm for detecting buffer overflows implemented in Splat combines two ideas: first, systematically searching for test inputs using directed testing techniques [6, 17], and second, tracking buffers and strings *partially symbolically*. We quickly recall the directed test generation algorithm, and then explain the partial symbolic representation of strings.

The systematic search for test inputs runs the program on symbolic values representing the input in addition to concrete inputs. The program is instrumented to additionally maintain a *valid range* for each pointer. The valid range denotes the set of addresses that can be safely accessed by the pointer. For example, for a pointer into a buffer, the valid range is between the start and end of the buffer.

Splat maintains a *symbolic state* that maps concrete addresses to symbolic expressions, and a *path constraint* that stores the sequence of conditionals executed, as well as a sequence of symbolic constraints representing the predicates in the conditionals. At each memory dereference, if the dereferenced address is a symbolic expression, Splat constructs a symbolic constraint such that any satisfying assignment to this constraint will ensure that after executing the current path, the address being dereferenced will point outside the valid range. Thus, finding a satisfying assignment indicates a memory safety violation, and this satisfying assign-

```
01: void lookup(char *resolved) {
02:  char *wbuf = "blah";
03:  if (resolved[0] == '/' &&
04:        resolved[1] == '\0')
05:         rootd = 1;
06:  else rootd = 0;
07:
08:  if (strlen(resolved) + strlen(wbuf) +
09:      rootd + 1 > 1024) return;
10:  if (rootd == 0)
11:      strcat(resolved, "/");
12:  strcat(resolved, wbuf);
13:
14: }
15:
16: void test() {
17:  char resolved[1024];
18:  input(resolved,5);
19:  lookup(resolved);
20:  exit(0);
21: }
```

Figure 2: Buffer overflow due to off-by-one error

ment provides an input to the program that demonstrates the bug. This test is then generated and run to confirm the bug. If there is no satisfying assignment, the systematic search continues by generating a new input by modifying the path constraint and finding a satisfying assignment for this modified constraint. The new input is guaranteed to have a different execution path from all previous runs. Splat terminates when no new execution path can be found or when a bug is found.

Fully and Partially Symbolic Representations. In Figure 2, there is an off-by-one error that causes a buffer overflow in the `strcat` function on line 12. If `resolved` is equal to a non-root directory, then an extra "/" is added (lines 10–11). The length check on line 8–9 is incorrect, and `rootd` should be `!rootd`. The bug is exposed when `lookup` is called with a pathname that results in a resolved pathname length of exactly 1024. Since most static analyzers treat buffers and pointer arithmetic conservatively, they are likely to generate many false positives for any code of this form, and identifying this particular bug within this large set of false positives may be difficult.

Normally, directed testing tools would track 1024 symbolic variables: one for each character in the input (call this the *fully symbolic* representation). Unfortunately, introducing such a large number of symbolic values results in a large number of paths and a large set of symbolic constraints that stresses the capacity of the underlying constraint solver. Thus, this bug is difficult to find for directed testing tools using a fully symbolic representation. For example, running Cute [17] on the program of Figure 2 took 2 hours and generated 1019 paths before finding the error. Excess paths are created when running through the various string manipulation functions (such as `strlen`) because those functions are not summarized.

Our algorithm for buffer overflows is based on the following observations. First, for many buffer overflows (including this one), most of the actual content of the buffer is not relevant, what is relevant is the *length* of the string stored in the buffer, and *some small prefix* of that string. Therefore, instead of the exact fully symbolic representation, we use a *partially symbolic* representation that tracks a few characters of the stored string and its length symbolically while filling the rest of the buffer randomly.

Second, many strings are manipulated as an abstract datatype using the standard `string.h` header functions. Once we introduce this partially symbolic representation, we can precisely abstract the behavior of many header functions instead of stepping through them. For example, the `strlen` function can be abstracted to simply return the symbolic length of the string. This can drastically reduce the number of paths to be explored in the directed search.

With these two optimizations, our algorithm can reach all branches and detect all memory safety violations in `lookup` while exploring only a few paths.

In general, of course, partial symbolic representations can miss paths, but our approach allows the tester to iteratively increment the size of the symbolic prefix.

Running Splat. We demonstrate our technique step by step. The `test` function is the test harness and the starting point for Splat. The `input(p,k)` function specifies the input, where `p` is the address of the buffer storing the input and `k` is the number of symbolic entries in the input (*i.e.,* the symbolic prefix). In this example, we chose the symbolic prefix to be 5 characters. In general, the user can set a short prefix length and gradually increase the length as their test budget allows. To allow large input strings to be generated, a symbolic string length is associated with the input. If the symbolic length exceeds the symbolic prefix, characters beyond the prefix are randomly generated.

Thus, the test harness constructs the following inputs for Figure 2. First, it fills the character buffer `resolved` with random characters (each of size 8 bits) followed by the string terminator character. Of these random characters, the first 5 are tracked. At the same time, it constructs an integer representing the length of the string that is also tracked symbolically. Finally, it calls the `lookup` function with the buffer `resolved` (line 19). Such a test harness could be automatically constructed by static analysis of the C code [6] with some default symbolic prefix length parameter.

During the first run, the input string will be of length 5 with randomly chosen non-zero characters in the first five entries and a string terminator in the last entry. Let's say for this run, we randomly generate `resolved =` "a1weq". We introduce 5 symbolic values representing the first 5 elements of `resolved`: $\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4$ and a symbolic value β representing the string length. We instrument the `strlen` function to return the symbolic length of a string. Thus, when called with `resolved`, `strlen` returns the symbolic value β as the length. When the program is executed with this input, it does not take the then branch at line 3 nor the then branch at line 8. Executing to line 9, we have $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge \neg(\beta + 5 > 1024)$ as the path constraint.

Notice that there is no predicate representing the branch at line 10. This is because the variable `rootd` is not symbolic, hence the conditional `rootd == 0` evaluates directly to `true` and so is not included in the path constraint (see [6]). At line 11, we update the symbolic state of `resolved` by updating the string length to $\beta + 1$. At this point, we have to check whether the call to `strcat` at line 11 can cause a buffer overflow. To check this, we ask whether there is a satisfying assignment for the extended path constraint $\neg(\alpha_0 = '/' \wedge$

$\alpha_1 = 0) \wedge \neg(\beta+5 > 1024) \wedge (\beta+1 \geq 1024)$. Since the above constraint is unsatisfiable, there is no buffer overflow (yet).

At line 12, we update the string length of `resolved` to be $\beta + 5$ and again check for a possible overflow. This time, we check if there is a satisfying assignment for the extended path constraint $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge \neg(\beta + 5 > 1024) \wedge (\beta + 5 \geq 1024)$. This constraint is satisfiable and a solution is $\beta = 1019$ and $\alpha_0\alpha_1\alpha_2\alpha_3\alpha_4$ = "a1weq". This indicates a potential buffer overflow. Next we generate an input string with a prefix of "a1weq" but of length 1019 by filling the non-symbolic suffix with random non-zero characters. This new test case causes the `resolved` array to overflow.

Suppose now we fix the bug by replacing `rootd` in line 9 with `!rootd` and rerun Splat. The first run with "a1weq" passes all memory violation checks. We create a new test case by negating the last branch predicate, proceeding in a depth-first order. Our path constraint currently is $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta+6 \leq 1024)$. We solve for a new test case satisfying $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta+6 > 1024)$, getting a 1019 length string with "a1weq" prefix. After the next run, we get the path constraint $\neg(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta+6 > 1024)$. We recognize that both branches of the last conditional statement have already been explored so we negate the first condition and solve for the path constraint $(\alpha_0 = '/' \wedge \alpha_1 = 0)$, getting the string "/" as the next input. We dropped the $(\beta + 6 > 1024)$ constraint because by negating an early branch predicate we can no longer guarantee that we will hit the later branch. The third run with "/" as input has $(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 \leq 1024)$ as its path constraint. Again we search alternative path constraints depth first and negate the last branch, getting the new path constraint $(\alpha_0 = '/' \wedge \alpha_1 = 0) \wedge (\beta + 6 > 1024)$. However, β represents the symbolic length of the input and the second character α_1 is the string terminator so there is no satisfying assignment for this constraint. Then, Splat terminates after exploring all three paths in `lookup`.

Summary. Splat is composed of three main ingredients: (1) instrumentation at every memory access to detect memory safety violations (buffer overflows), (2) directed testing, and (3) partially symbolic representations with symbolic tracking of string length and symbolic summaries of string library functions. Memory safety violations are found by tracking (de)allocations of memory and insuring all dereferences stay within their respected bounds. The systematic search of directed testing insures all explicit paths will be explored. Combining (1) and (2) checks each memory dereference along all explicit paths, that is all implicit paths leading to possible memory safety violations are explored. In addition, (3) scales Splat to realistic programs that rely on the C string library by reducing the burden on the symbolic path exploration. In the experimental section, we validate our choice of partially symbolic representations by showing that, despite the lightweight nature of the abstraction, it is sufficient to find many buffer overflows in real benchmarks.

3. DIRECTED TESTING

We recall the directed random test generation algorithm [6, 17], describing it for a simple imperative language with a memory model similar to the C programming language's. Specifically, we model dynamic memory allocation on the heap and limited pointer arithmetic. For clarity of exposition, we omit function calls and the stack.

Syntax. The operations of the language consist of labeled statements $\ell : s$. Labels correspond to instruction addresses. A statement is either (1) a normal termination statement halt or the abnormal program termination statement abort, (2) an *input statement* $\mathsf{input}(l, k)$ that copies an input buffer of size k into a buffer starting at address l of size at least k, (3) an assignment $l := e$ where l is an an address, and e is a side-effect free expression, (4) a conditional statement $\mathsf{if}(e)\mathsf{goto}\ \ell$ where e is a side-effect free expression and ℓ is a program label, (5) a heap allocation $l := \mathsf{malloc}(k)$ where after the statement l points to newly allocated memory of size k, (6) heap deletion $\mathsf{free}(e)$ where after the statement, memory pointed by address e is removed. For all labeled statements, we assume $\ell + 1$ is a valid label. For a labeled conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ we assume both ℓ' and $\ell + 1$ are valid program labels.

Expressions are constructed from constants and addresses using arithmetic operations, boolean operations, pointer equality, the sizeof function, or the dereference operator $*$. Boolean operators are evaluated to integers in the C style. A value is either an integer (default value: 0) or an address (default value: `NULL`). A non-null address can be dereferenced with the $*$ operator which returns the value stored at the address. Addresses $\langle b, \mathsf{offs}\rangle$ have a base address b and an integer offset offs. Addresses point to some entry in a buffer. A size k *buffer* $A = \langle v_0 \dots v_{k-1}\rangle$ is an ordered list of values. v_i can be retrieved by dereferencing an address $\langle b, i\rangle$ where b is the base address of the buffer. The operator $\mathsf{sizeof}(a)$, where a is an address, returns the size of the buffer pointed by a, that is the number of elements from a to the end of the allocated buffer.

A program $P = \langle S, \ell_0\rangle$ is a tuple containing a list of labeled statements S, and a starting label ℓ_0 where execution starts.

Semantics. A *memory* M is a partial mapping from addresses to values. A *concrete state* $\langle M, \ell\rangle$ consists of a memory M and a program label ℓ.

Given a program $P = \langle S, \ell_0\rangle$, execution starts from the starting label ℓ_0 with initial memory M_0. For convenience, we write $M[b \mapsto A]$ to update a buffer A by adding (if b is fresh) or modifying (if b was previously mapped) the mappings $\{(\langle b, k\rangle, v_k) \mid v_k \in A\}$ while keeping all other addresses m' mapped to $M(m')$. We write $M[\langle b, \mathsf{offs}\rangle \mapsto v]$ to update a single address $\langle b, \mathsf{offs}\rangle$ with a value v.

Expressions are evaluated as usual with the standard order of precedence. Address expressions $\langle b, e\rangle$ containing some base address or address b and expression e evaluates to $\langle b, i\rangle$ where e evaluates to an integer value i under the current state $\langle M, \ell\rangle$. Address expressions $\langle\langle b, e\rangle, e'\rangle$ evaluate to $\langle b, e + e'\rangle$. A dereference $*a$ evaluates to v if $(a, v) \in M$. If $a = \mathtt{NULL}$ or $M(a)$ is undefined, then the program terminates abnormally due to a memory safety violation.

For an assignment statement $\ell : l := e$, the left-hand side l is an address where the result is to be stored. The expression e is evaluated to a concrete value v in the context of the current state $\langle M, \ell\rangle$, the memory is updated to $M[l \mapsto v]$ and the new program label pc is set to $\ell + 1$. An input statement $\ell : \mathsf{input}(l, k)$ is interpreted by updating the memory M to the memory $M[b \mapsto A]$ where b is the base address of the address l, and A is a buffer of randomly chosen integers of size k, and the new label is set to $\ell + 1$. For a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, the expression e is evaluated in the current state $\langle M, \ell\rangle$. If the evaluated value is zero, the new program label is ℓ', otherwise the new label is $\ell + 1$. In either case, the memory remain unchanged. For an allocation statement $\ell : l := \mathsf{malloc}(k)$, a new base address b is created where $M[b \mapsto A]$ and A is a buffer of 0s of size k and $\langle b, 0\rangle$ is stored in address l. For the deletion statement $\mathsf{free}(\langle b, \mathsf{offs}\rangle)$, if M contains a mapping from address $\langle b, \mathsf{offs}\rangle$, all addresses with the base address b are removed otherwise the program terminates abnormally with a memory safety violation. At the halt statement, execution terminates normally. At the abort statement, execution terminates abnormally.

Directed Random Test Generation. Splat performs symbolic execution of the program together with concrete execution, similar to other directed testing algorithms [1,6, 17]. It maintains a *symbolic memory map* μ, and a *symbolic path constraint* ξ in addition to the concrete state. These are updated during the course of execution. The symbolic memory map μ is a mapping from concrete memory base addresses b to a pair $\langle \varphi, \gamma\rangle$ where $\varphi = \langle \alpha_0, \ldots, \alpha_{k-1}\rangle$ is a list of symbolic expressions representing k values of the concrete buffer A, and γ is a symbolic expression representing the size of A.

At every statement $\ell : \mathsf{input}(l, k)$, the symbolic memory map μ introduces a mapping $\mu[b \mapsto \langle \varphi, \alpha_k\rangle]$ from the base address b of l to a tuple of k fresh symbolic values $\varphi = \langle \alpha_0, \ldots, \alpha_{k-1}\rangle$ and a fresh symbolic value α_k representing the length of the buffer. We refer to the k symbolic values as the *symbolic prefix*. At every assignment $\ell : l := e$, the symbolic memory map is updated to map the address l to $\mu(e)$, the symbolic expression obtained by replacing all dereferences and sizeof operations in e with symbolic expressions that are stored in μ. Specifically, we replace all dereferences $*\langle b, \mathsf{offs}\rangle$ to the offs-th symbolic expression in φ from the pair $\mu(b)$ if it exists, otherwise we replace with the concrete value $M(\langle b, \mathsf{offs}\rangle)$. Note that the address $\langle b, \mathsf{offs}\rangle$ is concrete here, but when we check whether a dereference is valid, the offset maybe symbolic. We replace $\mathsf{sizeof}(\langle b, \mathsf{offs}\rangle)$ with $(\gamma - \mathsf{offs})$ where γ is the symbolic length of the buffer obtained from the last element of the tuple $\mu(b)$ if it exists, otherwise we replace with the concrete size of the buffer.

The path constraint ξ is initially *true*. At every conditional statement $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, if the execution takes the then branch, the symbolic constraint ξ is updated to $\xi \wedge \mu(e)$ and if the execution takes the else branch, the symbolic constraint ξ is updated to $\xi \wedge \neg\mu(e)$. Thus, ξ denotes a logic formula over the symbolic input values that the concrete inputs are required to satisfy to execute the path executed so far. For each memory dereference $*\langle b, e\rangle$, we check if $\xi \wedge \mu(e \geq \mathsf{sizeof}(*\langle b, 0\rangle))$ or $\xi \wedge \mu(e < 0)$ is satisfiable. If either are, we generate a test case satisfying that constraint. If the new test case violates memory safety, we report the error.

Given an execution, Splat generates a new testcase by selecting a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ along the path that was executed such that (1) the current execution took the "then" (respectively, "else") branch of the conditional, and (2) the "else" (respectively, "then") branch of this conditional is uncovered. Let ξ_ℓ be the symbolic constraint just before executing this conditional and ξ_e be the constraint generated by the execution of this conditional. We find a satisfying assignment for the constraint $\xi_\ell \wedge \neg\xi_e$. The property of a satisfying assignment is that if these inputs are provided at each input statement, then the new execution will follow the old execution up to the location ℓ, but then take the conditional branch opposite to the one taken by the old execution, thus ensuring that the other branch is covered. The satisfying assignment is used to define a new input for the next run of the program.

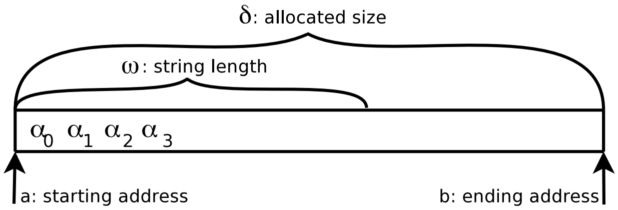

Figure 3: Memory nodes contain possibly symbolic representations of string length and size of allocated memory.

p = malloc(s)	$\rightarrow$	$create(p, p+s, \mu(s))$
int i	$\rightarrow$	$create(\&i, \&i + sz(i), null)$
*p	$\rightarrow$	$mn = find(p)$
		$assert(p \leq mn.\delta + mn.a)$
		$assert(p \geq mn.a)$
free(p)	$\rightarrow$	$mn = find(p)$
		$assert(p \leq mn.\delta + mn.a)$
		$assert(p \geq mn.a)$
		$delete(mn)$

Figure 4: Tracking memory for heap and stack allocations, and checking pointer dereferences.

4. IMPLEMENTATION

We implemented Splat for testing C programs. Splat consists of three parts: a source-to-source instrumenter, a library for tracking memory allocations, deallocations, and accesses, and a library for symbolic execution, constraint solving, and coverage tracking.

The instrumenter takes the source code of the program and adds calls to the runtime library that tracks memory allocation and memory dereferences. It also adds the calls that run the symbolic execution in parallel with the concrete execution.

4.1 Symbolic State

Aside from the concrete state (*i.e.,* the heap and stack), Splat tracks symbolic values, allocated memory sizes, and string lengths. Tracking concrete allocated memory sizes allows the detection of memory safety violations, while tracking symbolic values and string lengths allow the generation of new inputs that result in memory safety violations.

For each allocated buffer, Splat internally maintains a *memory node* that represents the state of the buffer. The structure of a memory node is shown in Figure 3. A memory node tracks the starting address a and ending address b of the buffer, the symbolic size δ of the buffer, the symbolic contents $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{k-1}$ (with k being the preset constant for the length of the symbolic prefix), and (in case the node refers to a string) the symbolic string length ω. Because there exists only one symbolic length ω per memory node, we can only track one string that starts at the beginning of the buffer. In our experiments, we have found no need to track multiple strings per buffer. Memory nodes are kept sorted by address ranges using a splay tree data structure [19]. As an optimization, if the symbolic content stored in a buffer is a constant, it is not stored and we rely on the concrete value.

Memory nodes are created whenever a local variable is allocated on the stack (e.g., when a function is called) or when memory is allocated in the heap. Figure 4 shows the instrumentation that is added to the program to create memory nodes and to check memory dereferences for possible errors. The function $create(a, b, sz)$ creates a new memory node that starts at the concrete address a and ends at the concrete address b with a (possibly symbolic) size sz. Notice that for allocations, the compiler may choose to allocate more memory for alignment but we track the most conservative allocation. In case the allocated memory is a string, the field ω of the memory node is set to a fresh symbolic constant. Recall that μ is the symbolic memory map, and $\mu(s)$ returns a possibly symbolic expression by evaluating s in the symbolic state. The function $find(p)$ finds the memory node associated with an address p in the splay tree, if one exists. The function $delete(mn)$ deletes the memory node mn from the splay tree. Memory nodes are removed from the splay tree when memory is freed on the heap or when stack variables go out of scope on a function return.

Symbolic Execution. Symbolic expressions arise from inputs. Following our description of test generation in section 3, we define an input through the `input(ptr, k)` function where `ptr` points to the beginning of an input buffer and `k` is the number of elements in the input. This adds mappings to fresh symbolic values in the contents of the memory node that `ptr` points to. As these elements are accessed and modified, other memory nodes are updated with symbolic expressions.

The symbolic updates occur as described in Section 3. However we need to take into account some specific features of C: memory allocation and string manipulations. We replace the functions in `string.h` with our string library that is aware of symbolic lengths and memory nodes. The `string.h` functions are modified to update the symbolic length of strings. This is similar to how CSSV [2] symbolically executes string manipulation functions. Figure 5 show the updates for some widely used string manipulation functions.

The `strlen(s)` function makes the contents of the return value to be the symbolic string length. The symbolic length is not simply the field ω of the memory node that holds the string, because `s` may not point to the beginning of the memory node a. The function `strcpy(d,s)` copies a string `s` to `d`, so the symbolic length of string `d` is updated with the length of string `s`. Again, offsets with relation to the starting address of the memory nodes are added to take account of d or s not being at the starting addresses of their respective memory nodes. The function `strcat(d,s)` appends the string `s` to the end of the string `d` so after the operation the symbolic length of string `d` is the sum of the length of `d` and the length of `s`.

The function `sprintf(d,c,`$s_1...s_n$`)` creates a string at `d` from a format string c and some string parameters $s_1...s_n$. We do not write all cases for the update of the length of d, because there are many cases for calculating the length of a C format string. Instead, we show a simplified update

$$
\begin{array}{lll}
\mathrm{l = strlen(s)} & \rightarrow & mn = find(s) \\
 & & \mu[\&l \mapsto mn.\omega - mn.a + s] \\
 & & \\
\mathrm{strcpy(d,s)} & \rightarrow & mn_d = find(d); mn_s = find(s) \\
 & & mn_d.\omega = d - mn_d.a + mn_s.\omega \\
 & & \qquad\qquad -s + mn_s.a \\
 & & assert(mn_d.\omega < mn_d.\delta) \\
 & & \\
\mathrm{strcat(d,s)} & \rightarrow & mn_d = find(d); mn_s = find(s) \\
 & & mn_d.\omega = mn_d.\omega + mn_s.\omega \\
 & & \qquad\qquad -s + mn_s.a \\
 & & assert(mn_d.\omega < mn_d.\delta) \\
 & & \\
\mathrm{sprintf}(d, c, s_1...s_n) & \rightarrow & \forall 1 \leq i \leq n.\ mn_i = find(s_i) \\
 & & mn_d = find(d) \\
 & & mn_d.\omega = d - mn_d.a + strlen(c) \\
 & & \qquad\qquad + \sum_{i=1}^{n} mn_i.\delta - i + mn_i.a \\
 & & assert(mn_d.\omega < mn_d.\delta)
\end{array}
$$

Figure 5: Tracking memory for string operations.

where length of d is the sum of the string length of c plus the lengths of the parameters.

4.2 Test Generation

As discussed in Section 3, Splat explores all paths by iteratively finding satisfying assignments for new path constraints representing unexplored paths. In the implementation, each path is a sequence of integers representing branches taken. Each branch id is mapped to a symbolic expression (if is not a constant) representing the predicate associated with taking or not taking the branch. A trie [12] stores all previously explored paths and whether negation of a branch is unsatisfiable or has already been explored. This allows Splat to use different search strategies. For example, a depth-first search only requires storing one path in the trie. The search terminates when all paths are either unsatisfiable or have been explored.

Memory Safety. The splay tree of memory nodes track all allocated memory in the heap and on the stack. For each pointer dereference `*p`, we should be able to find, by calling $find$(`p`), the memory node that contains the pointer in the splay tree. If we do not find a memory node, we have a memory safety violation. This approach is similar to Valgrind's Memcheck [15]. However, unlike Memcheck, since the return address on the stack is not part of any memory node, we can always detect buffer overflows that overwrite the return address. Note that if we do not track symbolic state, Splat can be used as a runtime checker for memory violations.

In addition, we explicitly add assertions about well-formedness of memory in the code using the function $assert(e)$. If during test generation, we can find a satisfying assignment for the conjunction of the path constraints with the symbolic expression $\neg e$, we have found a potential error. Since symbolic execution can be imprecise, such a satisfying assignment is subsequently run as a new input to confirm the bug and exhibit a concrete execution trace to the user.

For example in Figure 4, when we dereference a pointer, we generate the assertion $assert(p \leq mn.a + mn.\delta)$, where mn is the memory node for p. Failure of this assertion indicates an input for which the pointer p points beyond its memory node (and hence a memory error). This is a stricter approach to memory safety by insuring the dereference occurs in the memory node pointed to by the referent [10]. The *referent* refers to the valid address in the expression to be dereferenced. For example in `*(ptr + 5)`, `ptr` is the referent. If `ptr` points to a character buffer of size 3, `*(ptr + 5)` will always be a violation of the referent notion of memory safety. However, `ptr + 5` may still be a valid address; if the buffer is on the stack, `ptr + 5` can point to some other variable allocated on the stack.

Similar checks are performed for string operations as seen in Figure 5. Whenever a string is copied into another buffer, a check is made to see if the string length will exceed the size of the buffer. Whenever `sprintf` is called, the length of the generated string is checked to see if it fits in the destination buffer.

4.3 Constraint Solving

We generate new inputs by finding satisfying assignments for path constraints and constraints representing memory violations. We use STP, a bit accurate SAT based decision procedure [4]. This allows us to deal with widely-used bit operators and arithmetic overflow. In our experience, arithmetic overflow has been crucial in generating many memory safety violations.

A satisfying assignment for a symbolic length may go beyond the symbolic prefix. Concrete buffer entries beyond the symbolic prefix are randomly chosen characters (excluding the string terminator). Further we add additional constraints that make the symbolic length consistent with the symbolic prefix. The occurrence of the string terminator in the symbolic prefix affects the symbolic length. Given a symbolic string length α_k for the symbolic prefix $\alpha_0\alpha_1\ldots\alpha_{k-1}$, we have the added constraints $\alpha_i = 0 \Rightarrow \alpha_k = i$ for $0 \leq i < k$.

Figure 6 shows a buffer overflow bug originating in the Bind DNS server that demonstrates the need for a bit-accurate constraint solver. The bug is caused by an arithmetic overflow in line 34. If `dlen - n` < 0, a huge amount would be copied. This example requires the analysis to understand bit operators, pointer arithmetic, and fixed-sized integers. To test this example, we create a symbolic buffer with 100 symbolic values of size 1 byte each. Since we fill the `msg` buffer entirely with symbolic values, the symbolic string length is not tracked. Note that we could have made the two strings within the message have symbolic lengths and saved many extra executions, but that would require knowledge of the internals of `rrextract`.

Instead of listing all runs, we examine a run that reaches line 34. At line 34, suppose for the given run $n = 13$, `cp` is 0x40232504, `eom` is 0x40232568, $\mathsf{dlen} = \alpha_i << 8 \mid \alpha_{i+1}$ and $\mathsf{type} = \alpha_i << 8 \mid \alpha_{i+1}$. We want to find a satisfying assignment to $(\alpha_i << 8 \mid \alpha_{i+1} - 13 >_{unsigned} 1024)$ in conjunction with the path constraint $(\alpha_j << 8 \mid \alpha_{j+1} = T\_NXT) \wedge (0x40232504 + \alpha_i << 8 \mid \alpha_{i+1} \leq_{unsigned} 0x40232568)$. Note that we must distinguish between the unsigned less-than and the signed less-than operators. Given a decision procedure for bit-vectors, a satisfying assignment can be found: for example $\mathsf{dlen} = 12$ results in a 2GB `memcpy`. In this example, if the underlying constraint solver was not bit accurate, the error would be missed.

5. EVALUATION

We demonstrate Splat on several programs. We ran Splat on benchmarks representing real exploits [24] and found all bugs except two. Because the benchmarks were not full programs, we also ran Splat on a module in the Snort intrusion detection system, the WuFTP server, and NVDS, a well-tested flash-based memory system. All tested programs except NVDS had known memory safety bugs. For the case studies, Splat found all known bugs and 2 unknown bugs in NVDS.

Table 1 shows the results. All experiments were performed on a 2.33GHz Intel Core 2 Duo with 2GB of RAM. Each experiment contains both the program containing the bug and the program with the bug fixed. The numbering of each benchmark corresponds to the same numbering as the paper [24] describing the benchmarks. We ran Splat on both buggy and fixed versions, because enumeration of the buggy program stops when the bug is found and depending on the location of the bug, only a fraction of paths are enumerated. Table 1 describes the bugs, the time spent to find the bugs in the buggy program, and the time spent enumerating paths in the fixed program.

For each program, the representation of the input string was chosen with the shortest possible symbolic prefix that can find the bug. The maximum size of the input and the size of the symbolic prefix are shown in Table 1. If the size of the symbolic prefix equals the maximum size, then the input was fully symbolic. A symbolic prefix of size zero was successful in finding bugs in 6 out of the 14 benchmarks. A symbolic prefix of 10 characters was successful in finding bugs for the WuFTP case study. The other benchmarks did not use the `string.h` library thus requiring the input to be fully symbolic.

For programs that utilized the `string.h` library, we demonstrated how the length abstraction allows directed testing to scale to larger more complex programs. We show that the length abstraction allows directed testing to find errors in Bind 4, WuFTP 2, and the WuFTP case study that could not be found with a fully symbolic input within our given testing budget. We also show that Splat with the length abstraction enumerates fewer paths without sacrificing branch coverage.

5.1 Finding Memory Safety Violations

Real Exploit Benchmarks. The first 14 rows of Table 1 are real exploit benchmarks [24]. These benchmarks are small stripped down versions of Bind, Sendmail, and WuFTP, specifically designed to test buffer overflow detection tools. These benchmarks were independently developed to be small but realistic and representative of known buffer overflows. They have been shown to substantially challenge dynamic and static buffer overflow detection tools [23,24]. In these thorough evaluations, four static detection tools were no better than randomly guessing buffer overflow warnings for programs with or without such errors. Only one static tool (Polyspace) was marginally better but produced 1 warning for every 12 lines of code. Splat successfully found errors in all benchmarks except two, without reporting any false warnings.

The original benchmarks contained inputs that would result in finding the exploit. These inputs were removed and replaced with symbolic inputs. For the 6 benchmarks that exclusively used the `string.h` library, we represented the

```
GETSHORT (s, cp) { \
   register u_char *t_cp = (u_char *)(cp); \
   (s) = ((u_int16_t)t_cp[0] << 8) \
    | ((u_int16_t)t_cp[1]) \
    ; \
   (cp) += INT16SZ; \
}

GETLONG (l, cp) { \
   register u_char *t_cp = (u_char *)(cp); \
   (l) = ((u_int32_t)t_cp[0] << 24) \
    | ((u_int32_t)t_cp[1] << 16) \
    | ((u_int32_t)t_cp[2] << 8) \
    | ((u_int32_t)t_cp[3]) \
    ; \
   (cp) += INT32SZ; \
}
```

```
void rrextract(char *msg, int msglen) {
 int len, n;
 short type, dlen;
 char *eom, *cp, expanded;
 char data[MAXDATA*2];
 eom = msg + msglen; cp = msg;
 n = strlen (cp); if (n > 15) return;
 cp += n;  len += n;
 GETSHORT(dlen, cp);
 cp += 2;  len += 2;
 if (cp + dlen > eom) return;
 GETSHORT(type, cp);
 cp += 2;  len += 2;
 if (type !=  T_NXT) return;
 n = strlen(cp); if (n > 15) return;
 cp += n;  cp1 = data;
 memcpy(cp1, cp, dlen - n);  // overflow
 cp += (dlen - n);  cp1 += (dlen - n);
}
```

Figure 6: Buffer overflow due to arithmetic overflow

whole input with only a symbolic length. For all other benchmarks, we represented all characters of the input with 100 symbolic characters.

The Bind programs represent several buffer overflows in the Bind DNS Server. Bind 1 contained a memcpy that had an arithmetic expression in the size argument that could overflow. Bind 2 and 3 had memcpy size arguments that were improperly bound-checked. Bind 4 contained a sprintf without a bounds check. Sendmail represent bugs in the Sendmail email server. Sendmail 1 did not increment a counter as it processed the "<" character. Sendmail 2 contains a copy to a fixed sized buffer without a bounds check. Sendmail 3 has an index that is not reset after reading a return character. Sendmail 4 does not check the size if a return character is read. Sendmail 5 contains an improper bounds check for sequences of "/". Sendmail 6 contains an arithmetic underflow. Sendmail 7 allows an arbitrary size to be passed as a bound for `strncpy`. WuFTP represent bugs in the WuFTP ftp server. WuFTP 1 and WuFTP 3 contain unchecked strcpy or strcat functions. WuFTP 2 contains an incorrect bounds calculation as shown in Figure 2.

All benchmarks finish quickly except Sendmail 1 and 5 which timeout. To reach the bug, Sendmail 5 requires a long string of "/" characters of some particular length while Sendmail 1 requires repeated occurrences of the pattern "<>". Therefore, Sendmail 1 and 5 require the input to be fully symbolic. In either case, the buggy input was difficult to find because both benchmarks require finding a particular long input from millions of inputs that all lead to different paths.

Snort. We tested the "Back Orifice" rootkit detector module in the Snort intrusion system that had a known buffer overflow. Snort modules have well-defined inputs that describe a packet. Splat can model this packet as a symbolic buffer. The bug occurs because the length of the packet field is not checked and later used as a bound on a while-loop that reads the contents of the packet. Splat quickly finds this buffer overflow.

WuFTP. We tested a version of the WuFTP server with a buffer overflow in the pathname normalization function. The example in Section 2 is a simplified version of that bug. Although WuFTP processes packets, the contents of the packets are strings that are interpreted as FTP commands. We test WuFTP by replacing the packet with a symbolic string with a 10 character symbolic prefix and a symbolic length. To skip the parser, we make several keywords in the string concrete and others symbolic. These keywords are defined by the underlying grammar. The details of construction were presented without the memory and string lengths fully tracked in a previous study [14]. Running Splat on each of these strings finds the error after 240 seconds.

NVDS. NVDS is a non-volatile storage system for flash memory that had been a target of substantial random differential testing [9]. We tested NVDS on an emulated system in RAM. Testing was different from the previous experiments, because NVDS did not just accept a string as an input. To test NVDS, we formatted the emulated flash, wrote to it three times and read from it once. The parameters for the writes and reads were symbolic. We found overflows in the memory emulating the flash that resulted from an arithmetic overflow in the bound checking in both the write and read functions.

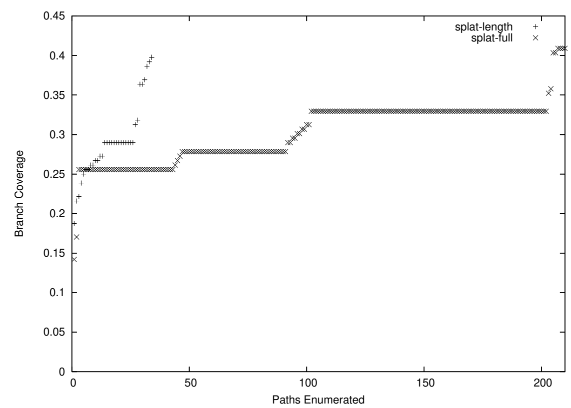

Figure 7: Coverage for Bind 2: Splat-length enumerates 34 unique paths in Bind 2 with a random string and a symbolic length. Splat-full must enumerate 210 paths with a symbolic input of 100 bytes.

5.2 Length Abstractions

	Program	LOC	Prefix	Size	Bug	Buggy	Fixed
Real Exploit Benchmarks	`Bind 1`	2.9K	100	100	size arg of memcpy could overflow	0.02s	0.5s
	`Bind 2`	3.1K	0	2048	size arg of memcpy could be negative	0.1s	1.1s
	`Bind 3`	2.5K	100	100	size arg of memcpy not checked	0.05s	0.1s
	`Bind 4`	2.7K	0	2048	sprintf without bounds check	0.6s	0.6s
	`sendmail 1`	1.9K	100	100	“<><><>....<><><>”	t/o	t/o
	`sendmail 2`	2.4K	0	2048	copy to fixed sized buffer without check	6.8s	t/o
	`sendmail 3`	2.0K	100	100	index not reset	18.6s	1m16s
	`sendmail 4`	2.4K	100	100	no bounds check	0.16s	1m49s
	`sendmail 5`	2.1K	100	100	“/////////. . .//////////////”	t/o	t/o
	`sendmail 6`	2.2K	100	100	arithmetic underflow	0.04s	1m38s
	`sendmail 7`	2.8K	100	100	unchecked size bound	0.05s	18.2s
	`WuFTP 1`	2.4K	0	2048	strcpy without bounds checks	0.3s	1.8s
	`WuFTP 2`	2.6K	0	2048	off-by-one bound check	0.03s	0.03s
	`WuFTP 3`	2.3K	0	2048	strcpy and strcat without bounds checks	0.4s	0.4s
Programs	`snort`	1.7K	100	100	unchecked look bound	13s	33s
	`WuFTP`	36K	10	2048	off-by-one bound checks	4m	4m43s
	`nvds`	12K	80	80	bounds check does not consider arithmetic overflow	2m5s	2hr1m52s

Table 1: Experimental Results: Splat bug finding effectiveness. LOC is lines of code. Prefix is the length of the symbolic prefix in bytes. Size is the maximum length of the input string in bytes. Buggy is time spend finding the bug. Fixed is time spent rerunning the test after fixing the error. t/o means timeout after 2 hours.

	Symbolic Length					Fully Symbolic			
Program	**Prefix**	**Size**	**Buggy**	**Fixed**	**Coverage**	**Size**	**Buggy**	**Fixed**	**Coverage**
`Bind 2`	0	2048	0.1s	1.1s	70/176 = 40%	100	0.61s	32.5s	70/176 = 40%
`Bind 4`	0	2048	0.6s	0.6s	21/38 = 55%	100	t/o	t/o	21/38 = 55%
`sendmail 2`	0	2048	6.8s	t/o	53/90 = 59%	100	3.4s	t/o	53/90 = 59%
`WuFTP 1`	0	2048	0.3s	1.8s	22/50 = 44%	100	2.9s	22.5s	22/50 = 44%
`WuFTP 2`	0	2048	0.03s	0.03s	28/120 = 23%	100	t/o	t/o	46/120 = 38%
`WuFTP 3`	0	2048	0.4s	0.4s	54/140 = 39%	100	0.7s	1.16s	54/140 = 39%
`WuFTP`	10	2048	240s	283s	552/1285 = 43%	1024	t/o	t/o	372/1285 = 29%

Table 2: Experimental Results: Comparing the length abstraction with a fully symbolic input string on programs with string manipulations. Prefix is the length of the symbolic prefix in bytes. Size is the maximum length of the input string in bytes. t/o means timeout after 2 hours for the benchmarks or 24 hours for the case study. Coverage is the branch coverage for the testing fixed programs run until completion or timeout.

For the 6 benchmarks and the 1 case study where string operations were restricted to the C string library, we ran Splat with the input string being represented by a small prefix (10 characters for the WuFTP case study and no prefix for the benchmarks) and a symbolic string length. We compared how Splat performs when the input string was symbolically represented by its length (Splat-length) with how Splat performed with the input string represented by all symbolic characters (Splat-full). Splat-full represents previous work in test generation tools that tries to completely search all paths up to some size input. Splat-full required 100 symbolic characters as the input for the benchmarks and 1024 characters as the input for WuFTP. These sizes were chosen to be slightly greater than the minimum string length that could set off the error, thus giving Splat-full the best chance possible in finding the error. In our experiments, Splat-length performed faster than Splat-full in finding errors as seen in Table 2 — showing the effectiveness of using a string length abstraction.

In Bind 4 and WuFTP 2, the fully symbolic string technique could not find the bug within 2 hours while Splat-length found both errors in less than a second. In the WuFTP case study, Splat-full could not find the error within 24 hours. In all 3 tests, Splat-full was stuck in generating input not relevant to reaching the error.

For example, Bind 4 contains many `sprintf` of the form, where `buf` is a fixed 1000 byte buffer and all other arguments are inputs:

```
sprintf(buf, "%s: query(%s) %s (%s:%s)",
        sysloginfo, queryname, complaint, dname,
        inet_ntoa(data_inaddr(a_rr->d_data)));
```

Suppose `sysloginfo`, `queryname`, `complaint`, and `dname` are all string inputs with maximum length of 250 while

```
01 char pathspace[ MAXPATHLEN ];
02 char old_mapped_path[ MAXPATHLEN ];
03 char mapped_path[ MAXPATHLEN ] = "/";
04 int mapping_chdir(char *orig_path) {
05   char *path = &pathspace[0];
06   strcpy( old_mapped_path, mapped_path );
07   strcpy( path, orig_path );
08      . . .
09 }
```

Figure 8: WuFTP 1: strcpy at line 07 can overflow

`(data_inaddr(a_rr->d_data))` was a 32-bit integer input. To check for an overflow, Splat-length solves the constraint:

$$\begin{aligned} strlen(sysloginfo) + strlen(queryname) \\ +strlen(complaint) + strlen(dname) \\ +15 + 15 + 1 > 1000 \end{aligned}$$

The constraint has been simplified to include the string length of the address representing the 32-bit integer as 15, the length of the format string as 15, string terminator as 1 and the size of `buf` as 1000.

While Splat-length can solve the string lengths and generate an input that will cause the memory violation, a tool without the length abstraction must rely on a complete search. If we fix the integer input, Splat-full requires placing the string terminator at almost all locations in each of the four strings in the worst case. If each string can have a length of 250, there are around 250^4 or approximately 4 billion strings to enumerate. Furthermore, instead of tracking just 4 symbolic constants representing each string length, 1000 symbolic constants must be tracked — leading to a substantial cost increase in constraint solving. In our Bind 4 experiments, we tried to give Splat-full a better chance of finding the error by reducing the input string sizes such that four input strings of length 24 would result in a buffer overflow, but this also led to a time-out as seen in Table 2.

The other experiments involve copying into a buffer without doing a proper check. Figure 8 shows a representative memory violation in WuFTP 1 where the `strcpy` on line 14 may overflow because the input is not bound checked before. Splat-length quickly finds these errors by representing the input with just a symbolic length. However, Splat-full can also find the error by enumerating all string lengths up to the maximum length of the input. Because the 6 benchmarks are small snippets of the real exploit, the Splat-full can terminate and find the error. In real programs, directed testing with a fully symbolic input will unlikely find the bug, because it would be stuck enumerating many irrelevant paths of inputs with varying string lengths.

Since a fully symbolic search may get lost in a large search space, Splat-length also gets better branch coverage, i.e., number of branches explored / total branches, given a limited test budget. As seen in Table 2, in the WuFTP case study, Splat-length's 283 second search had better coverage than the fully symbolic search timing out after 24 hours. Splat-length reaches higher branch coverage faster than Splat-full. Figure 7 shows how branch coverage increases when Splat-length and Splat-full are run on Bind 2. Splat-length only enumerating 34 unique paths results in the same branch coverage as Splat-full enumerating 210 unique paths. As Splat-full is unrolling loops to generate new paths, no new branches are covered. Although it is expected that given substantial resources that Splat-full would get better branch coverage, only in the case of WuFTP 2 did the fully symbolic string approach get better coverage (46/120 versus 28/120). Note that all such experiments are not close to full branch coverage because the benchmarks represent snippets of code from the full program where many paths are unreachable and we do not model all inputs such as networking or configuration options in our WuFTP case study.

6. RELATED WORK

Many approaches to detect memory safety violations statically or dynamically have been proposed. Splat combines ideas from several of those with test input generation.

Runtime Checking. Runtime checkers detect memory safety violations of specific execution runs but require a test input to trigger the violation. For such approaches, there is a trade-off between being able to detect the violation and performance. Valgrind [18] uses one bit for each address to represent if it is allocated or not. If an invalid address is accessed, a memory safety violation is reported. This only detects accesses to unallocated memory, so the return address of a function on the stack can still be overwritten and undetected. Jones and Kelly [10] implemented a memory safety checker in `gcc` that adds instrumentation to check whether an address evaluated from some expression containing an address p still points to the same buffer as p. CRED [16] extends Jones and Kelly by checking bounds only before a memory dereference and focuses only on string operations. If Splat was run without input generation, Splat would be similar to the CRED approach with additional instrumented libraries.

Larson and Austin [13] extends runtime checking by finding errors that may occur along the same control path of the supplied input but with a different input. Their memory model [13] associates each buffer index with a range, and each buffer with a string length and buffer size that are updated symbolically. Splat tracks similar constraints but is more precise because [13] conservatively represents each range and size with an integer instead of a symbolic expression. Also [13] does not perform test generation and may generate false alarms when symbolic execution is imprecise.

Static Analysis. In contrast to dynamic analysis, static analysis runs on all paths of a program and does not require any test inputs. However, they typically generate (many) false positives. CSSV [2] converts string manipulation to integer operations and performs an integer analysis to insure string operations remain within proper bounds. Although false positives were reported to be few, manual summaries are needed for functions and the integer analysis was expensive. Archer [22] tracks linear relationships between variables and automatically generates function summaries by inferring relationships between function parameters by various heuristics. Boon [21] uses a flow insensitive analysis for string manipulations errors which is fast but very imprecise. A comparison between various static tools showed that none were very effective in finding real buffer overflows, either not finding the errors or generating too many false positives [24]. Splat's symbolic execution of string lengths was inspired by static analyses that track only lengths [2, 21, 22].

Directed Testing. The idea of path exploration using both symbolic and concrete execution is from directed testing tools such as DART [6], CUTE [17], EXE [1] and SAGE [8].

Recent papers [1, 7, 11] also suggest to systematically inject assertions in programs during directed test generation in order to detect memory safety violations and other standard programming errors, such as division by zero and integer overflows. Our contribution is to use symbolic length abstraction and symbolic prefixes of input buffers to improve scalability of automatic test generation for buffer overruns.
Underapproximation. To allow Splat to effectively find bugs and finish within a reasonable amount of time, Splat uses an underapproximation that leaves some input suffixes random but tracks the length of the string input symbolically. In the context of test generation, different underapproximations involving heap shapes have been explored in Java Pathfinder [20]. Our experiments with Splat show the new underapproximation using symbolic string lengths and buffer sizes seems to be effective in finding buffer overflows in C programs.

7. CONCLUSION

For any bug-detection technique, there is a tradeoff between cost (how much time it takes) and precision (how accurately it detects errors). In automated state-space exploration, this tradeoff has been explored by varying the search strategy (e.g., depth-first vs. breadth-first) to generate an underapproximation of the state space that still encounters program errors. We believe *length abstractions* capture an interesting property-driven heuristic in approximating the program state space for automatic buffer overflow detection. Our experiments on standard buffer exploit benchmarks [23,24] demonstrate that in many real-life buffer overflow scenarios, just tracking the string length (and a small prefix) is enough to find test inputs that expose the bug. On the other hand, tracking the entire string symbolically, while theoretically complete, fail to terminate or find the bug on the same set of benchmarks.
Acknowledgments: This work was initiated during a visit of the first author to the second author at Bell Laboratories. This research is sponsored in part by the grants NSF CCF-0546170 and NSF CCF-0702743. We thank the authors of [24] for their benchmarks. We thank Alex Groce and Rajeev Joshi for feedback on preliminary versions of Splat.

8. REFERENCES

[1] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: automatically generating inputs of death. In *CCS*, 2006.
[2] N. Dor, M. Rodeh, and S. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.
[3] J. C. Foster, V. Osipov, and N. Bhalla. *Buffer Overflow Attacks*. Syngress, 2005.
[4] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, 2007.
[5] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.
[6] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.
[7] P. Godefroid, M. Y. Levin, and D. Molnar. Active property checking. Technical report, Microsoft, 2007.
[8] P. Godefroid, M.Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
[9] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE*, 2007.
[10] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.
[11] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: amplifying the effectiveness of software testing. In *FSE*, 2007.
[12] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1997.
[13] E. Larson and T. Austin. High coverage detection of input-related security faults. In *USENIX*, 2003.
[14] R. Majumdar and R. Xu. Directed test generation with symbolic grammars. In *ASE*, 2007.
[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.
[16] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.
[17] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, 2005.
[18] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX*, 2005.
[19] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
[20] W. Visser, C. S. Pasareanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, 2006.
[21] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.
[22] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, 2003.
[23] M. Zhivich, T. Leek, and R. Lippmann. Dynamic buffer overflow detection. In *BUGS*, 2005.
[24] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *FSE*, 2004.