MultiOtter: Multiprocess Symbolic Execution

Jonathan Turpie, Elnatan Reisner, Jeffrey S. Foster, Michael Hicks
Computer Science Department
University of Maryland
College Park, MD
{yudi, elnatan, jfoster, mwh}@cs.umd.edu

ABSTRACT

Symbolic execution can be an effective technique for exploring large numbers of program paths, but it has generally been applied to programs running in isolation, whose inputs are files or command-line arguments. Programs that take inputs from other programs—servers, for example—have been beyond the reach of symbolic execution. To address this, we developed a multiprocess symbolic executor called MultiOtter, along with an implementation of many of the POSIX functions, such as socket and select, that interactive programs usually rely on. However, that is just a first step. Next, we must determine what symbolic inputs to feed to an interactive program to make multiprocess symbolic execution effective. Providing completely unconstrained symbolic values causes symbolic execution to spend too much time exploring uninteresting paths, such as paths to handle invalid inputs. MultiOtter allows us to generate inputs that conform to a context-free grammar, similar to previous work, but it also enables new input generation capabilities because we can now run arbitrary programs concurrently with the program being studied. As examples, we symbolically executed a key-value store server, redis, and an FTP server, vsftpd, each with a variety of inputs, including symbolic versions of tests from redis's test suite and wget as a client for vsftpd. We report the coverage provided by symbolic execution with various forms of symbolic input, showing that different testing goals require different degrees of symbolic inputs.

1. INTRODUCTION

Symbolic execution [10] is a powerful testing technique for systematically exploring paths through a program. However, it has largely been used to study programs whose symbolic inputs come from files or the command line. To test programs that communicate with other programs would require a faithful model of system resources such as sockets and pipes, and such modeling is not typically performed.[1] Furthermore, system models alone would not be enough: while we could imagine implementing means to open and communicate via pipes and sockets, servers require clients to be active, creating and connecting to sockets or responding adaptively to exchanges of information. Therefore, there must be a programmatic way to specify, from the point of view of the server, how possible clients may interact with it symbolically.

To support symbolic execution of interactive programs like servers, we propose using *multiprocess* symbolic execution: we run a client program in parallel with a server, and we provide a realistic POSIX library that the two use to communicate. The POSIX library provides the functionality that servers rely on to communicate with clients, and by running programs in parallel, we can have a client that creates and connects to sockets.

Not only does this approach put interacting programs within reach of symbolic execution, but it also allows us, by carefully selecting the client program, to precisely control how we inject symbolic data into the server. The data can range from purely symbolic (i.e., totally unconstrained) data, if we want a brute force exploration of the program; to data constrained using a context-free grammar for avoiding erroneous inputs, as has been explored in previous work [5, 11]; to very targeted inputs which may consist of, for example, concrete tests converted into symbolic tests, or real client programs that are themselves fed symbolic input.

One key point for multiprocess symbolic execution is that all processes must share the same constraints on symbolic values. For example, if we generate symbolic data in a client and send this data to the server, the server's code might examine these symbolic values and cause symbolic execution to branch, following each possible path. To maintain consistency between server and client, the client code must execute under the new constraints resulting from branches taken in the server's code—assumptions made about symbolic values in one process's code must be visible to other processes.

We implemented multiprocess symbolic execution by extending the symbolic executor Otter [14] to support multiple processes—we call this extension *MultiOtter*—and we implemented a large piece of the POSIX specification through which multiple processes can communicate. We use newlib [13] as our C standard library, and we implemented several POSIX components, such as a file system, socket functions, and the ability to fork new processes. Our POSIX implementation, limited to the parts needed by our experiments, is written almost entirely in C, but with low-level functionality such as forking and blocking handled specially by MultiOtter. Using our POSIX implementation is quite easy; adding a small preamble to a server's code and linking to our POSIX model is all that is needed to run a client in parallel with the server. Execution then proceeds as usual: the programs create sockets and communicate over them using read/write or send/recv, the server can test for connections using select, etc.

We demonstrate MultiOtter's utility with a sequence of

[1] Cloud9, a symbolic execution framework developed concurrently with the present work, does model these resources; we defer discussion of it to Section 5.

experiments. First, we studied redis, a key-value store server. MultiOtter made it easy to run a simple client program in parallel with redis to feed it purely symbolic data. However, doing so only covered paths corresponding to invalid inputs. Gracefully handling invalid inputs is certainly important, but so is handling *valid* inputs. For our second experiment, to reach code for handling redis's main functionality, we created clients that first generate input conforming to context-free grammars and then pass the data to the server. These inputs led to far fewer error paths than the purely symbolic input, and they successfully issued many commands to the server. Our final experiment with redis fully exercised the ability to run an arbitrary program as a client, going beyond context-free grammars: we modified some tests from redis's test suite by making input values symbolic, effectively yielding a symbolic test suite. All the symbolic tests passed, but we discovered that symbolic execution did not cover any code beyond that covered by the original test suite. Nonetheless, the symbolic test suite gives much stronger assurance of correctness than the test suite itself provides: each concrete test demonstrates that one input-output pair is correct, but a symbolic test shows that *all* input-output pairs of that form are correct.

Next, we studied vsftpd, a security-minded FTP server. As with redis, we began by feeding the server purely symbolic data and then input constrained by a grammar. In this case, neither form of input was effective at covering anything beyond paths handling invalid inputs—vsftpd's parsing of its input created a large number of paths that handle erroneous inputs, and the grammar we used was not focused enough to steer execution toward paths for valid inputs. We next ran vsftpd with an actual FTP client, wget. Running a single concrete input, wget covered many more lines than the earlier experiments, but of course only a single path. Providing wget with symbolic input led to a moderate increase in both lines and paths executed. Finally, we returned to grammars, but this time restricted to the commands wget might issue; doing so afforded some improvement over the previous grammar.

In summary,

- We introduce MultiOtter, a new symbolic execution engine that can execute several processes in parallel within a symbolic environment.
- We implemented a POSIX model that can be symbolically executed by MultiOtter to allow programs to use POSIX system calls to interact.
- We used MultiOtter to run a number of clients against the redis and vsftpd servers and, for several methods of providing symbolic input, compared the tradeoffs between how many (and which) lines and paths were executed. We found that, under a time limit, purely symbolic inputs explore only paths for invalid inputs and therefore cover relatively few lines; grammars have widely varying effectiveness, depending on both the program and the grammar itself; and highly constrained inputs (symbolic tests or real client programs) can cover a significant number of lines, but they explore relatively few paths.

Thus, multiprocess symbolic execution is needed for handling interactive programs, and the choice of how to supply symbolic inputs must take into account the goal of symbolic testing.

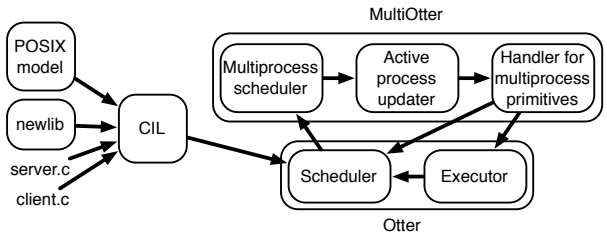

Figure 1: The architecture of MultiOtter.

2. MULTIOTTER

The overall architecture of MultiOtter is shown in Figure 1. The core of the system is Otter [14], a single-process symbolic executor. We now briefly describe Otter, followed by our POSIX implementation and, finally, MultiOtter itself.

2.1 Background: Symbolic execution and Otter

Symbolic execution [10] is a technique for systematically exploring paths through a program by introducing *symbolic values* into the program. A symbolic executor tracks these values as they flow through the program and conceptually forks execution if a conditional depends on a symbolic value, exploring both branches. The *path condition* is the conjunction of all the conditions corresponding to the branches chosen along a given path; it gives the precise conditions, expressed as constraints on the symbolic values, under which that particular path would be taken. For the remainder of this paper, we assume familiarity with symbolic execution, and we refer the reader to the literature (e.g. [10, 2, 14]) for more background.

Otter [14] is a symbolic executor that, given a C program as input, parses it into an intermediate form using CIL [12] and then interprets the program. Program variables can be assigned fresh symbolic values by invoking the __SYMBOLIC function; this is generally done in a driver that then invokes the program's main function. Otter operates over a worklist of what we call *jobs*, which are snapshots of execution comprising a program counter, a path condition, and the state of memory. At each step of execution, Otter selects a job to advance, executes the next instruction, and stores the results back into the worklist (or reports them, if the program exited or an error occurred).

2.2 POSIX model

Otter is a pure (or static) symbolic executor, similar to KLEE [2] but somewhat different from dynamic symbolic executors [6, 7, 17, 15]. As such, it requires an implementation of all library functions used by the program being studied. In previous work [2, 14], the set of available library functions was relatively limited. In particular, support for network functions was brittle or nonexistent.

We have implemented a more complete system model including both the C library functions and a simulation in C of other POSIX functions. The C library implementation is provided by using CIL to compile newlib [13] against our POSIX implementation.

We wrote an in-memory file system called OtterFS as a

backend for many POSIX functions. OtterFS is written in C and makes use of the blocking and shared memory primitives discussed below. Unlike typical in-memory file systems, OtterFS does not attempt to simulate the layout of a physical drive, but rather just the file system structure and data contents. These design choices aim to simplify the symbolic execution of programs that use OtterFS. OtterFS includes support for some POSIX devices including /dev/null, /dev/tty, /dev/console (which is hard-linked to /dev/tty), and /dev/zero. Reading from /dev/tty produces symbolic data, and writes to it are logged by MultiOtter. It is possible to add other devices by implementing read and write functions for them and adding them to the list of devices. OtterFS supports traditional Unix permissions, but it only supports two users: root and non-root.

In OtterFS, sockets are implemented as a wrapper over a pair of pipes (which are, in turn, implemented as special fixed-sized files). The socket functions keep track of the state of the socket and enforce the expected behavior of each system call. OtterFS does not simulate network errors or other abnormal environment behaviors, assuming that the TCP layer absorbs all such errors. Most of the rest of the socket implementation is stubs of the network layers that are not being simulated. For example, there is no need to use socket lingering, since all communication is instantaneous, but the ability to get and set the socket linger is necessary to execute programs that attempt to use the linger feature.

2.3 Multiprocess symbolic execution

Since Otter handles only single processes, MultiOtter must manage metadata associated with multiple processes, which it does by extending jobs to *multijobs*. A multijob comprises a job (the running job), a set of idle processes, and a set of memory locations shared among all processes. Since multijobs form a subclass of jobs, MultiOtter can reuse large portions of Otter's functionality, simply providing Otter with a multijob wherever it expects a job.

Idle processes are each represented by a program counter and local memory. Recall that the running job has a program counter, memory, *and* a path condition. This path condition is shared among *all* processes in a multijob. Correct behavior depends on the path condition being shared, as this example illustrates:

```
if (fork()) { if (x) send(x); else send(1); } /* Parent */
else { y = recv(); assert(y); } /* Child */
```

Clearly, the assertion should not fail, regardless of the value of x: when x is 0, the parent sends 1, and when x is sent, it is nonzero. Consider what happens if x holds the symbolic value α. When the parent takes the true branch for **if**(x), it assumes α is nonzero and sends α to the child, which then asserts α is nonzero. If the path condition were not shared, the child would consider α an unconstrained symbolic value, and the assertion would fail. For the assertion to succeed, the child must share the parent's path condition.

Throughout execution, MultiOtter maintains a worklist of multijobs and, as depicted in Figure 1 and described in Figure 2, MultiOtter iteratively selects a multijob from its worklist, chooses a process from this multijob to execute, sets the chosen process to be the running job, and either handles a multiprocess-specific function call or passes the job to Otter. We now consider each step in more detail.

The first step in MultiOtter's main loop is scheduling a

```
1 while worklist is not empty:
2       multijob ← pick_multijob(worklist)
3       process ← pick_process(multijob)
4       multijob ← set_running_process(multijob, process)
5       if multijob is calling a multiprocess primitive:
6             results ← handle_call(multijob)
7       else:
8             results ← Otter(multijob)
9       worklist ← worklist ∪ results
```

Figure 2: MultiOtter pseudocode.

Function name	Description
otter_fork	Fork a new process
alloc_shared	Allocate shared memory
free_shared	Free shared memory
begin_atomic	Begin atomic section
end_atomic	End atomic section
block	Block on shared memory

Table 1: Multiprocess primitive functions.

job to execute, which occurs in two parts. First, on line 2 in Figure 2, MultiOtter chooses which multijob to explore next. For this, MultiOtter reuses Otter's job scheduler. Otter allows many different scheduling strategies; we use its version of generational search [7] although exploring the performance of other strategies may be interesting future work. In generational search, a given multijob runs until all its processes terminate. When a multijob branches (for example, at **if** statements that depend on symbolic values), one branch is selected randomly and followed, and the others are stored in the worklist as the next *generation*. When the multijob terminates, a new multijob is randomly selected from the oldest generation. This strategy gains the benefit of depth-first search's deep exploration (so later parts of a program can be covered) but the generations provide some breadth.

Next, on line 3, MultiOtter chooses which process in the multijob to run. For this, MultiOtter uses a preemptive, round-robin scheduler. This allows other processes to continue running even if one process is in an infinite loop. MultiOtter's scheduler also determines if any process that had blocked (see below) should be awakened.

After selecting a process to run, MultiOtter makes the chosen process the multijob's active job on line 4. This involves moving the previously active job to the list of idle processes, adjoining the multijob's shared memory to the process's local memory, and attaching the path condition to the process.

On line 5, MultiOtter inspects the chosen process to see if it is about to execute a function that is a multiprocess primitive—a function that only makes sense in the context of multiprocess execution and cannot be implemented in C. Since these are relevant only to multiprocess execution, MultiOtter must handle such functions itself; it cannot delegate them to Otter. A list of MultiOtter's primitives is given in Table 1. These are analogous to system calls in real systems, and our POSIX implementation makes use of them; in fact, these functions are *only* called through OtterFS and POSIX functions, not from programs directly.

The most basic functionality multiprocess symbolic execution needs is the ability to create new processes through

calls to fork. Our implementation of fork includes some C code (to update the open-file table) followed by a call to the MultiOtter primitive otter_fork. This call causes MultiOtter to create a new process within the current multijob. As required by the semantics of fork, the child's memory is a copy of the parent's, but modifications to non-shared memory in a child process do not alter the values in the parent process, and vice versa. Otter (and, hence, MultiOtter) has a purely functional representation of memory, so this copy-on-write semantics was quite easy to implement.

MultiOtter also needs a method of managing shared memory. We created two primitives for this purpose: alloc_shared to create shared memory and free_shared to free it. OtterFS makes heavy use of shared memory. MultiOtter's model presumes all processes are on a single machine, so all files and other file-like objects, such as sockets, are stored in shared memory, as is the system-wide open-file table.

The final set of primitives we created are for synchronization among processes: begin_atomic and end_atomic create atomic sections, and block accepts any number of pointers to shared memory and suspends the calling process until one of the designated shared memory regions is modified. The process scheduler will not preempt a program while it is in an atomic section. Calling block releases the process scheduler to schedule other processes. OtterFS uses these primitives for many calls that access the file system, for blocking reads and writes, and for select.

As an example, Figure 3 illustrates how OtterFS uses synchronization primitives for blocking reads and writes on pipes, which are required by POSIX to be performed atomically if possible. A read can only occur if there is data in the pipe's buffer; otherwise, it must block until there is data. A similar situation occurs when writing to a pipe. Additionally, if the write is larger than the free space in the pipe's buffer, data is written into any space that is free, but the write must block until there is space to continue writing the rest of the data. begin_atomic and end_atomic are used to make sure that other processes modify neither the data in the pipe nor the pipe's internal metadata while the read or write is occurring. The calls to begin_atomic immediately after the calls to block ensure that validation of the pipe's state and use of the pipe are performed within a single atomic section.

Finally, if the running job is not calling a multiprocess primitive, MultiOtter simply delegates the instruction to Otter in line 8. Since Otter presumes its input is a job and not a multijob, it manipulates only the running job—not the inactive processes. If shared memory or the path condition change, MultiOtter makes these changes visible to the other processes as each one is selected to run. This includes the case where Otter branches on a symbolic value and returns multiple jobs: the entire multijob is duplicated, including the waiting processes, and each process will thenceforth run under its multijob's path condition. (Like memory, this duplication is implemented functionally and hence is quite straightforward and efficient.)

```
int read_pipe(pipe_t *pipe, char *buf, size_t num) {
  /* Prevent other processes from changing the pipe. */
  begin_atomic();
  /* Block until there is data to read. */
  while(pipe_is_empty(pipe)) {
    block(pipe); /* implies end_atomic */
    begin_atomic();
  }
  /* Read as much as possible */
  size_t how_many = min(num, available_data(pipe));
  _read_pipe_internal(pipe, buf, how_many);
  end_atomic();
  return how_many;
}

int write_pipe(pipe_t *pipe, char *buf, size_t num) {
  begin_atomic();
  /* Write in a loop, blocking if more space is needed. */
  size_t still_to_write = num;
  while (still_to_write > 0) {
    /* Block until there is free space to write into. */
    while(pipe_is_full(pipe)) {
      block(pipe); /* implies end_atomic */
      begin_atomic();
    }
    /* Write as much as possible */
    size_t how_many = min(still_to_write, free_space(pipe));
    _write_pipe_internal(pipe, buf, how_many);
    still_to_write -= how_many;
    buf += how_many;
  }
  end_atomic();
  return num;
}
```

Figure 3: Using synchronization primitives with pipes in OtterFS.

3. CONTEXT-FREE GRAMMARS

Previous work [5, 11] has explored how to control symbolic execution's search using context-free grammars, preventing the exploration of the many possible parsing-error paths through programs that require structured input. We explored the effectiveness of this technique for programs that read their inputs over the network, adapting an approach used by the authors of the string-constraint solver HAMPI [9]: we transform a context-free grammar into a set of C functions that produce a symbolic string conforming to the grammar. For our experiments, we wrote a small client program that connects to the server, produces a string using the generated C code, and sends the string over a socket to the server. This allows MultiOtter to explore only those server inputs which conform to the grammar.

The generated code uses two different techniques to create symbolic strings. First, we encode symbolic strings of a fixed length by fixed-sized arrays of symbolic bytes. Grammars introduce a symbolic string of length exactly X by using the nonterminal[2] $stringX$. Second, Otter supports if-then-else values—symbolic values of the form $ite(i, t, e)$, which equals t if i is nonzero and e otherwise. We create these using the ?: operator when a nonterminal can expand to any one of several string literals, producing a symbolic string representing any one of the literals. Note that we use these techniques because Otter does not directly support fully symbolic strings, i.e., variables that range over strings of arbitrary size and contents.

Pseudocode describing our CFG-to-C transformation is shown in Figure 4, and code implementing it is given in Fig-

[2]We could call these terminals, but our transformation treats them more like nonterminals; hence our terminology.

for each nonterminal N (other than *stringX*):
    emit [`char *generate_`N`(void) {`]
    if N has more than one rhs:
        emit [`int choice; __SYMBOLIC(&choice);`]
    counter ← 0
    for each rhs R of N that is not a string literal:
        s ← R with each nonterminal M replaced by `generate_`M`()`
        if R is the last non-string-literal rhs and no rhs of N is a string literal:
            emit [`return concat(`s`);`]
        else:
            emit [
                `if (choice ==` *counter*`)`
                    `return concat(`s`);`
                `else`
            ]
            increment *counter*
    if some rhs of N is a string literal:
        emit [`return (`]
        for each rhs R of N that is a string literal:
            if R is the last string-literal rhs: emit [R]
            else: emit [`choice ==` *counter* `?` R `:`]
            increment *counter*
        emit [`);`]
    emit [`}`]

Figure 4: Grammar-constraint code generation pseudocode. An *rhs* is the list of terminals and nonterminals on the right-hand side of a production, e.g., $N \to rhs_1|rhs_2$.

ure 12 in Appendix B. For brevity, two pieces of our transformation are omitted from these figures. First, an initial preprocessing step adds auxiliary nonterminals to remove syntactic sugar such as Kleene star. Second, *stringX* nonterminals are handled specially, but we omit the (straightforward) process of producing the C functions corresponding to these nonterminals.

Example grammar. Consider the following CFG specifying a tiny subset of FTP, where nonterminals are italicized and terminals are shown in red.

start → `USER␣` *string3* `\nPASS␣` *string3* `\n` *cmd**;
cmd → `HELP\n` | `LIST\n` | `QUIT\n`;

Figure 5 shows a simplified version of the C code produced by our transformation. The preprocessing step introduces a new nonterminal

cmd_star → ϵ | *cmd cmd_star*

and replaces *cmd*∗ with *cmd_star* in *start*'s right-hand side. Then, one function is produced for each nonterminal.

We produce a generate_string$X$ function for each *stringX* nonterminal used in the given grammar. As generate_string3 demonstrates, these use the Otter primitives __ASSUME and __SYMBOLIC. __ASSUME(exp) adds the constraint that expression exp is nonzero, and __SYMBOLIC(&v) assigns a fresh symbolic value to variable v.

For all other nonterminals, we produce a function that

```
char *generate_string3(void) {
    char *str = malloc(4), c;
    __SYMBOLIC(&c); __ASSUME(c); str[0] = c;
    __SYMBOLIC(&c); __ASSUME(c); str[1] = c;
    __SYMBOLIC(&c); __ASSUME(c); str[2] = c;
    str[3] = 0;
    return str;
}

char *generate_start(void) {
    return concat("USER ", generate_string3(), "\nPASS ",
                  generate_string3(), "\n", generate_cmd_star());
}

char *generate_cmd_star(void) {
    int choice; __SYMBOLIC(&choice);
    if (choice == 0)
        return concat(generate_cmd(), generate_cmd_star());
    else
        return (""); // Empty string
}

char *generate_cmd(void) {
    int choice; __SYMBOLIC(&choice);
    return (choice == 0 ? "HELP\n" :
            choice == 1 ? "LIST\n" :
            "QUIT\n");
}
```

Figure 5: Grammar-constraint code example.

makes a symbolic choice of how to expand the nonterminal. Expanding to anything other than a string literal involves calling functions to generate substrings and then concatenating the results. Expanding to a string literal simply returns that string, but if a nonterminal can expand to any one of several string literals, they are all grouped together using the ?: operator, as discussed above.

Example execution. We now consider an execution starting from generate_start. When calling this function, Otter first generates two symbolic strings of length 3 (for the username and password), then it generates the string for *cmd**, and finally it concatenates the results. As it generates *cmd**, it will branch on the variable choice. In the choice != 0 case, it generates the empty string, and the final result is

USER $\alpha_0\alpha_1\alpha_2$\nPASS $\alpha_3\alpha_4\alpha_5$\n

In the choice == 0 case, it calls generate_cmd, where Otter does *not* branch on the ?: operator but instead generates a symbolic if-then-else pointer which can refer to any of the three possible strings. After that, Otter branches on choice in the recursive call to generate_cmd_star. In the choice == 0 case, this process repeats, but in the choice != 0 case, the final string is

USER $\alpha_0\alpha_1\alpha_2$\nPASS $\alpha_3\alpha_4\alpha_5$\n$\beta_0\beta_1\beta_2\beta_3\beta_4$

where the α_is are constrained to be non-null and the β_is are if-then-else values that depend on the symbolic value of choice from generate_cmd. For example, if that symbolic value is γ, the value of β_0 is

$ite(\gamma = 0, \texttt{'H'}, ite(\gamma = 1, \texttt{'L'}, \texttt{'Q'}))$.

Our grammar-constraint encoding differs from HAMPI's [9] in several ways. HAMPI requires a maximum string length, while our encoding does not require one—one can be given, but if not, Otter will continue exploring ever-larger strings in the grammar until its execution is terminated by the user or by a timeout. Also, HAMPI performs some simplifications of the given CFG, but our transformation uses the CFG as given. Finally, HAMPI's encoding uses only arrays of symbolic bytes; it does not use if-then-else values. We leave exploring the ramifications of these different encodings to future work.

As we will see in Section 4.1, using context-free grammars to approximate non–context-free languages can be problematic. Note, though, that because our transformation produces C code that is "executable" by our symbolic executor, we are not truly limited to context-free grammars. We can hand-modify the generated code to produce non–context-free behavior, if desired. We have not explored this possibility but it is something we hope to consider in the future.

4. EXPERIMENTS

To validate the practicality of our symbolic executor and POSIX model, we symbolically executed two widely used server programs—a key-value store server, redis, and an FTP server, vsftpd. Also, to explore the impact of varying degrees of constraints on symbolic input, we used several clients that provided different types of input: purely symbolic input, input conforming to a context-free grammar, and highly constrained input based on an existing test suite or client program. We ran our experiments on a Mac Pro with two 2.26 GHz quad-core Xeon processors (note, though, that MultiOtter is single-threaded) and 16 GB of RAM. Each execution was given a timeout of 3 hours.

The metrics we considered were line coverage and number of paths explored. Line coverage is a useful baseline because executing code is obviously necessary for finding a bug in that code, but counting the number of paths is a much stronger way of determining how much of the program has truly been exercised. Counting paths in multiprocess symbolic execution requires clarification: we are actually counting path *conditions*. Thus, each multijob counts as a single path regardless of the number of processes. Furthermore, we ignored paths that were created as a result of branching but were never selected for execution by the scheduler before the timeout. Recall that we used a search strategy that executes each multijob to completion before switching to another multijob; thus, every path was a complete path, except for the one that was executing when the timeout expired.

To enable us to attach a client process to a server, we added a small preamble to each server's main function; this preamble calls fork to create the client process. The client process creates a socket and connects to the server before writing data to the server. However, the client would receive an error indicating a refused connection if it tried to connect before the server was ready. Therefore, we also added, at the point where the server calls listen, a single assignment setting a sentinel value. Our block primitive delays the client's execution until this value is set. Note that this initialization, which is needed even for feeding purely symbolic data to the server, directly relies on our multiprocess symbolic execution framework.

Experiment		Line cov	Paths
1	pure symbolic	11.78%	717
2	overapprox. grammar	17.58%	164
	underapprox. grammar	23.84%	481
3	test suite	25.45%	70
	symbolic test suite	25.45%	83

Table 2: Experimental results – redis.

We also made some other minor modifications to our subject programs to facilitate our experiments. For example, for redis, we decreased the number of objects that it allocates at startup, and we circumvented its wrappers around malloc and free. Both of these modifications lighten the load on MultiOtter without affecting program semantics in our experiments. For vsftpd, we modified its mmap-based memory allocator to use malloc.

4.1 redis

redis is a key-value store server written in C and consisting of approximately 10,000 lines of code. We used version 2.2.0-rc4 in our experiments. The core commands that redis supports are various reading, writing, and updating commands for a map from strings to strings, or from strings to one of several data structures containing strings. Aside from this functionality, redis has other features, such as commands for clients to subscribe to named channels and publish messages over those channels, or to perform a set of actions atomically in case multiple clients are accessing the database at the same time.

redis's main use of our POSIX model is in its use of sockets: creating a socket to listen for connections, using select to wait for either client commands or new connections, and reading and writing over a socket with each client. It also has an event loop that regularly calls gettimeofday; we implemented a simple notion of time wherein each successive call to gettimeofday returns a value one microsecond later than the previous call. Making time symbolic would allow us to search for errors related to timing issues[3], but it also greatly expands the space of possible executions, so we leave this for future work.

We executed redis under MultiOtter in a series of experiments, the results of which are shown in Table 2. For each experiment, the table shows the percentage of line coverage attained and the number of paths explored. It is clear from the table that exploring more paths does not mean covering more lines of code, but it is important to recall that each path represents a different class of inputs—so there is some merit to covering many paths, even at the expense of line coverage. Our experiments are an investigation into how to achieve different tradeoffs: unconstrained input explores many paths but does not cover many lines, highly constrained input explores few paths but many lines, and moderately constrained input strikes a balance between those extremes.

[3]In one initial experiment, for example, our tick was one *second*, rather than one microsecond, and this exposed a potential starvation problem: redis schedules one specific function for execution every 100 milliseconds. If the function takes longer than that to run, redis will invoke the function again, rather than servicing waiting clients.

Experiment 1: Purely symbolic input. As our first experiment, to exercise redis and MultiOtter at a basic level and to see how well a straightforward but naive approach would fare, we provided a client that feeds 100 unconstrained symbolic bytes of data to the server. While this client explored over 700 paths, only 12% of the code was covered, and none of the paths corresponded to a well-formed command being issued.

Experiment 2: Grammar-constrained input. Next, we used a grammar to try to limit the commands to valid inputs. For simplicity, we restricted keys and values to be exactly one character long. Longer keys and values would be unlikely to exercise additional program behavior; hence, they would do little but complicate our experiments, especially because redis computes hashes of keys, and hashes often cause trouble for constraint solvers [4]. One wrinkle in using context-free grammars to describe redis's input language is that it is not context-free: it includes several commands that accept a variable number of arguments, and the number of arguments is itself a part of the command. To deal with this, we used two different grammars: one underapproximating the actual input language by requiring each command to have a fixed number of arguments, and one overapproximating by allowing malformed inputs that specify the wrong number of arguments.[4] The grammars are given in Appendix A.

As shown in Table 2, both grammars get better line coverage than the purely symbolic input does, despite exploring fewer paths. However, the overapproximation gets less coverage and explores fewer paths than the underapproximation. This is due to more complicated symbolic expressions than the underapproximate grammar generates, which result in more and slower calls to the constraint solver. We suspect these constraints come from converting the symbolic value representing the number of arguments into an integer. Another factor, albeit of less importance, in the overapproximate grammar's lower coverage is that many paths corresponding to partially valid input have multiple offshoots that all hit the same error-handling code. This is code that the underapproximate grammar cannot execute so, given enough time, the overapproximate grammar should get higher coverage. However, exploring these paths reduces the rate at which the overapproximate grammar executes paths for valid commands, which results in lower line coverage given our time limit.

As another way of comparing these experiments, Figure 6 shows the number of lines covered by each symbolic execution as a function of time. All three forms of input covered a large number of lines quickly—this is mostly code to initialize the server and read an input. After this, the three diverge. The pure symbolic client executed many paths that all hit the same three errors; it finally found a different error around 5,000 seconds and then a fifth around 9,000 seconds. In contrast, both of the grammars periodically execute a new command, yielding new coverage. The overapproximate grammar covers new lines more slowly than the underapproximate grammar because, as noted above, it runs more slowly in general due to more complicated symbolic expressions.

[4]Previous work on grammar-based constraints for symbolic execution [5] noted that approximating input languages is sometimes necessary, but it did not explore the impact of these approximations.

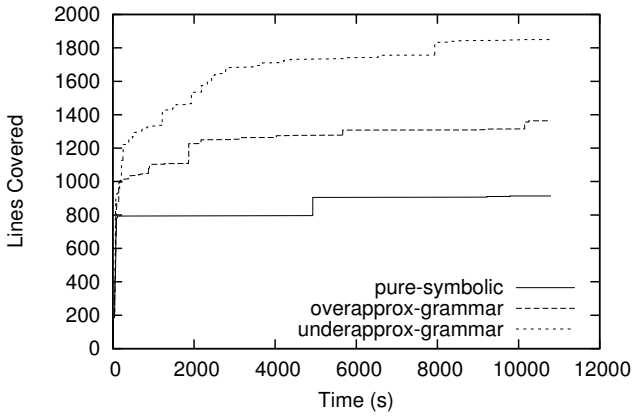

Figure 6: Number of lines covered over time for redis.

For both grammars, we found that a large proportion of the multijobs left in MultiOtter's worklist when the timeout expired had never finished generating inputs conforming to the grammar; relatively few were branches off of code in redis itself. Perhaps other ways of encoding grammar constraints would behave differently, but we leave this investigation to future work. Regardless, our experiments clearly confirm previous results showing that constraining inputs to context-free grammars is helpful in exploring more of the program by eliminating invalid inputs [5, 11].

Experiment 3: Symbolic tests. Finally, we converted a portion of redis's test suite to symbolic tests to see the effects of further constraining the symbolic input to the server. redis's test suite as a whole consists of 845 tests and achieves 58% line coverage, as measured by lcov. We focused on just a subset of this test suite: 70 tests that exercise redis's list data structure, which on their own achieve 25% line coverage of redis's code base. The tests are written in Tcl. We modified the existing test harness so that executing the tests did not interact with the server, but instead printed out a trace of the commands to be issued and assertions to be checked by MultiOtter. We modified these traces to make the keys and most of the values used in the tests be symbolic. We then attached each of these tests (or groups of tests, in cases where several tests had to be run together to behave properly) as client programs to the server.

One benefit of using a test suite is that its assertions provided semantic properties for MultiOtter to test. (In the earlier experiments, in contrast, MultiOtter only tested the server for basic safety violations such as null dereferences or buffer overflows.) Making the keys and values *purely* symbolic caused some of the assertions to fail because symbolic values were sometimes interpreted as whitespace rather than data. However, constraining the keys and values to be letters—that is, symbolic bytes in the range `[A-Za-z]`—caused all of the test suite's assertions to pass.

Interestingly, the concrete tests and the symbolic tests covered all the same lines and, aside from line coverage, the symbolic tests did not explore very many paths—in all, the symbolic tests explored 83 paths, compared to the 70 concrete tests (one path per test). This shows that the concrete

Experiment		Line cov	Paths
1	pure symbolic	21.24%	85
2	grammar	21.50%	172
3	wget (concrete)	32.75%	1
	wget + symbolic	34.07%	28
	reduced grammar	25.51%	143

Table 3: Experimental results – vsftpd.

tests are actually quite general. A good test is meant to demonstrate not just that one *particular* input is handled properly but that a *class* of inputs is, and our experiment demonstrates that this is the case for redis's test suite. The tests depended on the specific values only when a symbolic value was compared against a value that we left concrete in the tests, or when the hash of one symbolic value was compared against another. However, one drawback to our findings is that they suggest that, at least for redis, more work is required to get better coverage. Simply replacing concrete values with symbolic values only goes so far; more sophisticated tests with more intricate assertions seem to be necessary.

4.2 vsftpd

vsftpd is an FTP server, also written in C and about 10,000 lines of code, designed for security and speed. We used version 2.0.7 of vsftpd and configured it to run in standalone mode (as opposed to being forked on demand by inetd). As with redis, we executed vsftpd under MultiOtter in a series of experiments, measuring line coverage and paths explored. We initialized OtterFS so the server makes available one file with a single-character filename. We enabled anonymous access and disabled use of chroot and file locking, which OtterFS does not fully support. We disabled passive mode FTP to highlight the effectiveness of using an actual client program over a grammar—the client program can interpret the failed activation of passive mode and switch to active mode. The results of all experiments are shown in Table 3.

Experiment 1: Purely symbolic input. Similar to our first experiment with redis, we began with a client that connects to the server and sends it 100 unconstrained symbolic bytes. The majority of the code covered was executed before vsftpd was ready to accept connections. The stream of symbolic bytes did not do very well: it explored few paths and covered few lines, and none of the paths explored in the allotted time successfully logged in.

Investigating this, we found that, upon receiving input, vsftpd checks whether each character is a special character (such as a null, newline, or space) before checking to see whether it received a valid FTP command. When MultiOtter takes the branch corresponding to detecting one of the special characters at any one of these checks, it takes time before MultiOtter eventually backtracks to try the other, more useful, branch. This fruitless exploration is further exacerbated because, when vsftpd hits an error, it does not immediately exit—it prints an error message, such as "invalid command", and continues reading the remainder of the input. So, when executing vsftpd, MultiOtter does this as well, only backtracking when all 100 input bytes are exhausted. Given our time limit, MultiOtter failed to explore any path where USER, the command initiating login, appeared as part of vsftpd's input.

Note that redis did not exhibit this behavior. When redis encounters an error, such as those triggered by the purely symbolic input, it discards the rest of its input, so MultiOtter quickly backtracks. Also, redis does not print an error message upon encountering these errors, so MultiOtter does not spend time, as it does with vsftpd, executing code to send messages to the client. This explains the large difference in number of paths explored by the purely symbolic client for the two server programs.

Experiment 2: Grammar-constrained input. Second, we used a grammar, shown in Figure 10 in Appendix A, that represents all the commands that could be sent to a basic FTP server. The grammar constrains the input to issue a USER command, followed by a symbolic username, a PASS command with a symbolic password, an arbitrary series of other control commands, and finally QUIT. Despite this structure, the grammar covered very few additional lines, compared to the purely symbolic input. The same problem occurred here as before: even when constrained to begin with a USER command, finding a valid user name (the shortest is FTP, a standard anonymous login) proved too much for MultiOtter, given the time limit.

The additional lines covered in this experiment handled the commands USER, PASS, and QUIT, which are supplied concretely by the grammar. (QUIT executes successfully, but USER and PASS simply lead to errors.) This experiment explored more *paths* than the purely symbolic input because the paths were generally shorter: the grammar generated inputs that were shorter than the 100 bytes used in the first experiment.

Experiment 3: A real client – wget. Third, we used a real client program, wget. As a baseline, we ran wget through MultiOtter with a concrete input instructing it to download the planted file on the file system. This resulted in a huge improvement in coverage over the previous experiments: while wget missed some pre-login code executed in the previous experiments, it issued all the commands to log in and download the file.

In fact, having an active client such as wget is critical to downloading a file. Unlike redis, vsftpd expects its client to create additional sockets, beyond the one over which the initial connection is established. These sockets transmit data rather than commands and responses. This means that clients that only provide data to the server but do not actively create sockets—such as the clients used in the first two experiments—are limited to executing commands that communicate only over the initial socket. Fully exercising an FTP server's functionality would therefore be very difficult without an active client program executing in parallel with the server.

Due to its reliance on POSIX, such a client/server interaction was beyond the reach of symbolic execution in the past. However, the experiment just described was a *concrete* execution, using MultiOtter only because it is directly comparable to the other experiments. To make use of the *symbolic* executor, we next ran wget with a partially symbolic input that set the file name symbolic. This resulted in covering 1.32% lines beyond those covered with a con-

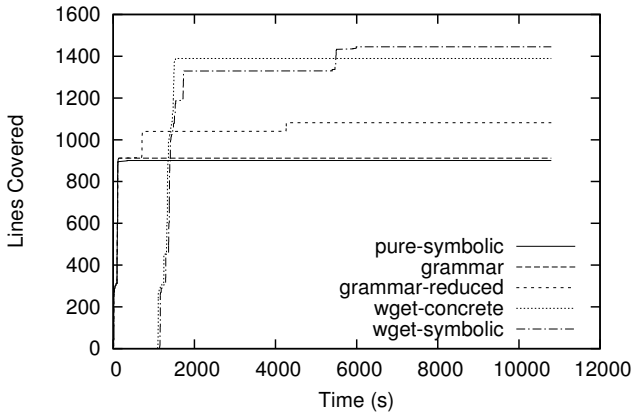

Figure 7: Number of lines covered over time for vsftpd.

crete input. While this is not a large increase in coverage, it does cover all lines covered by the concrete test while also covering many more paths through that code.

Finally, to more directly compare the grammar to wget, we constructed a reduced grammar, shown in Figure 11 in Appendix A, that only contains commands that wget could execute. The reduced grammar covered 29 lines (0.7%) that were not covered by the symbolic wget; these lines correspond to error conditions that wget does not hit because wget always supplies valid inputs. Symbolic wget covered 392 lines (9.2%) that were not covered by the reduced grammar; these correspond to the execution of a successful file download.

Comparing the two grammars, the reduced grammar covered all the lines that the full grammar covered plus an additional 4% of vsftpd's lines. These extra lines correspond to successfully logging in and to issuing a CWD command (albeit for an invalid directory).

As a final analysis of our experiments, Figure 7 shows the number of lines covered as a function of time, for each experiment. In each case, most of the new lines were covered at the beginning of execution. Some of these lines correspond to vsftpd's initialization code, which is the same in each run. However, after that, the coverage from the symbolic input and the full grammar stalled because they repeatedly issued invalid commands. The reduced grammar also mostly issued invalid commands, but there are two points at which its coverage increased: it logged in and issued a command at 700 seconds, and then it issued an invalid command (while logged in, which covers different code than pre-login invalid commands cover) at 4,000 seconds. The wget runs exhibit a sharp delay before achieving any coverage. This is due to a long pause while Otter initializes several large tables of data that wget uses for parsing inputs. We believe we can eliminate this pause with a small optimization in Otter, but despite this delay, the wget runs quickly surpass the others in lines covered.

4.3 Discussion

For both redis and vsftpd, we found in agreement with earlier findings [5, 11] that, given a reasonable timeframe, purely symbolic inputs only cover paths for invalid inputs. Also, highly constrained inputs are able to explore long, interesting program paths, although they tend not to explore very many paths.

Constraining inputs to context-free grammars was much more variable. First, grammars were much more effective in improving coverage beyond purely symbolic input with redis than with vsftpd. This is because of a difference between how the two servers' inputs are delimited. FTP messages are self-delimited, so vsftpd must check each input character to see if it is a control character, and this leads to a large amount of branching. redis's protocol, on the other hand, specifies the lengths of arguments; in this context, any character is legal so checking for control characters—and branching as a result—is unnecessary for reading input. Second, for each server, there was a noticeable difference between the performance of the two grammars we used. In both cases, constraining the grammar more led to higher line coverage, and also to more paths covered (for redis) or only slightly fewer (for vsftpd). Previous work [5, 11] discussing context-free grammars as a way to constrain symbolic execution did not emphasize how important the details of the grammar are to its effectiveness. Third, for vsftpd, we noted that there is a fundamental limitation when using context-free grammars as input: vsftpd expects the client to create new sockets in order to, for example, download a file. A client that simply supplies a stream of input data will not fill this role and will not be able to explore pieces of the server's functionality that depend on an active client.

Thus, context-free grammars can sometimes be used to strike a balance between purely symbolic input (potentially exploring many paths but few lines) and highly constrained input (many lines but few paths). However, care must be taken to craft an appropriate grammar, and if a server truly requires an active client to interact with, grammars will not be effective.

5. RELATED WORK

There is no shortage of work on symbolic execution [8, 16, 6, 7, 2, 17, 15], but previous work has focused on programs in isolation. Even works that study concurrent programs [8, 18] study a single multithreaded program at a time, rather than multiple communicating programs. There have even been papers [15, 18, 14] presenting results of symbolically executed server programs, but these papers had to work around the need for a client program. One paper [15] involved extracting code that was known to contain bugs, rather than executing the program in its entirety. In our previous work [14], we implemented mocked version of POSIX functions and carefully initialized the file system so that calls to read and recv provided the necessary data to the server. These manual modifications were tedious and error-prone, and they had to be separately done for each program being studied. MultiOtter and its POSIX library obviate the need for this extra labor.

Several symbolic execution systems [6, 7, 3, 2] interact with the underlying system during symbolic execution by making external calls using concrete values. This means the symbolic executor's model of, for example, the operating system does not need to be complete for the symbolic executor to work. On the other hand, making concrete external calls limits the reach of symbolic execution and can also lead to an imprecise or inconsistent view of program state. Despite interacting with the environment, though, these systems still focus only on studying a single program at a time. The en-

vironment they envision is essentially static, consisting of library or OS functions rather than another executing program. Perhaps one could use these systems to symbolically execute programs which communicate via the network by performing the network calls concretely and having a client program running "on the other side". This has not been attempted, however, and there are potential stumbling blocks, primarily due to the fact that the other program would execute concretely. Chipounov, Kuznetsov, and Candea [3] discuss ways of overcoming some of these problems. They also consider tradeoffs that come from various levels of fidelity in modeling the environment, but it is unclear how these considerations translate when the environment is another program.

Our experiments with grammar-constrained input were directly inspired by previous work demonstrating how to prevent symbolic execution from "getting lost" in uninteresting program paths [5, 11]. However, this work focused on a single program in isolation. We presented results showing that grammar-based constraints can be effective for interacting programs, too, but only if the interaction between the programs is data-only; we also demonstrated the impact of approximating context-sensitive languages with context-free grammars.

Loop-extended symbolic execution [15] adds support for better handling loops, which, among other benefits, helps prevent symbolic execution from spending too much time exploring paths for invalid inputs. That work also matches symbolic inputs to features of the input grammar such as lengths of arguments. Combining loop-extended symbolic execution and multiprocess symbolic execution remains an interesting avenue for future work.

A system very similar to MultiOtter, called Cloud9 [1], was developed concurrently to our work. As we did, the authors enhanced a symbolic executor—they used KLEE [2]—with multiprocess support. They developed a POSIX model very similar to ours, although they support threads and they allow for simulating network anomalies (e.g., packet fragmentation), which we do not. Another major component of Cloud9, orthogonal to symbolic execution of communicating programs, is distributing symbolic execution over a cluster of computers to exploit the inherent parallelizability of symbolic execution. Their paper discusses using existing test cases as a model for symbolic tests, but their experiments focus on their distributed architecture and how it, and their POSIX model, extend the reach of symbolic execution with purely symbolic inputs. Our work studies in more detail how different forms of symbolic input affect what parts of the program will execute.

6. CONCLUSION

We built a multiprocess symbolic execution framework along with a POSIX model that many real communicating programs use. We showed that this framework allows us to symbolically execute server programs, but that we must be careful about how we provide input if we wish to cover more than error-handling code. Context-free grammars provide some benefit in this regard, but they are of limited effectiveness when they only approximate the program's actual input language, or when the program requires more than simply data as input—such as establishing socket connections at arbitrary points in the program's execution. Using a real client program, or modifying concrete tests to make symbolic tests, gives more control over what paths get symbolically executed, yielding higher line coverage but exploring fewer paths. Thus, depending on the testing goal, each of these forms of symbolic input can be useful.

7. REFERENCES

[1] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth ACM SIGOPS/EuroSys Conference on Computer Systems*, EuroSys '11, pages 1–15, Salzburg, Austria, 2011. ACM.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[3] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 265–278, New York, NY, USA, 2011. ACM.

[4] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[5] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215, 2008.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. Internet Society, 2008.

[8] S. Khurshid, S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. *TACAS*, pages 553–568, 2003.

[9] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[11] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 134–143, New York, NY, USA, 2007. ACM.

[12] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[13] The Newlib Homepage, 2011. `http://sourceware.org/newlib/`.

[14] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In

Proceedings of the 32nd International Conference on Software Engineering (ICSE), pages 445–454, Cape Town, South Africa, May 2010.

[15] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236, New York, NY, USA, 2009. ACM.

[16] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE-13*, pages 263–272, 2005.

[17] N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334. ACM, 2010.

APPENDIX

A. CONTEXT-FREE GRAMMARS

We present here the context-free grammars used in our experiments. redis's input language is documented at `http://redis.io/topics/protocol`, and its commands are documented at `http://redis.io/commands`. Figures 8 and 9 show the overapproximate grammar used for redis. For the underapproximate grammar, we expanded all occurrences of '+' twice and we set the number of arguments (*digit* in the figures) accordingly. For example, the MGET and MSET commands in the underapproximate grammar were

*3\r\n$4\r\nMGET\r\n key key
*5\r\n$4\r\nMSET\r\n key value key value

Also, for the sort command, we omitted all optional arguments, leaving

*2\r\n$4\r\nSORT\r\n key

Next, Figures 10 and 11 show the grammars used to construct vsftpd's inputs.

These grammars rely on several nonterminals that our grammar-constraint code generator automatically introduces:

- string$N$, for any N, produces a symbolic string of length N
- digit generates a symbolic string of length 1 constrained to be in the range [0−9]
- letter generates a symbolic string of length 1 constrained to be in either [A−Z] or [a−z].

start → *command*+;

command →
| *generic_command*
| *string_command*
| *list_command*
| *set_command*
| *connection_command*
;

string_command →
| *2\r\n$3\r\nGET\r\n key*
| *3\r\n$3\r\nSET\r\n key value*
| *3\r\n$5\r\nSETNX\r\n key value*
| *4\r\n$5\r\nSETEX\r\n key seconds value*
| *3\r\n$6\r\nAPPEND\r\n key value*
| *2\r\n$6\r\nSTRLEN\r\n key*
| *4\r\n$6\r\nSETBIT\r\n key offset* $1\r\n (0|1) \r\n
| *3\r\n$6\r\nGETBIT\r\n key offset*
| *4\r\n$8\r\nSETRANGE\r\n key offset value*
| *4\r\n$8\r\nGETRANGE\r\n key begin end*
| *2\r\n$4\r\nINCR\r\n key*
| *2\r\n$4\r\nDECR\r\n key*
| * *digit \r\n$4\r\nMGET\r\n key*+
| *3\r\n$6\r\nINCRBY\r\n key increment*
| *3\r\n$6\r\nDECRBY\r\n key decrement*
| *3\r\n$6\r\nGETSET\r\n key value*
| * *digit \r\n$4\r\nMSET\r\n* (*key value*)+
| * *digit \r\n$6\r\nMSETNX\r\n* (*key value*)+
;

generic_command →
| * *digit \r\n$3\r\nDEL\r\n key*+
| *2\r\n$6\r\nEXISTS\r\n key*
| *1\r\n$9\r\nRANDOMKEY\r\n*
| *3\r\n$4\r\nMOVE\r\n key db*
| *3\r\n$6\r\nRENAME\r\n key key*
| *3\r\n$8\r\nRENAMENX\r\n key key*
| *3\r\n$6\r\nEXPIRE\r\n key seconds*
| *3\r\n$8\r\nEXPIREAT\r\n key timestamp*
| *2\r\n$4\r\nKEYS\r\n pattern*
| *2\r\n$4\r\nTYPE\r\n key*
| * *digit \r\n$4\r\nSORT\r\n key* [$2\r\nBY\r\n *pattern*] [$5\r\nLIMIT\r\n *offset count*] ($3\r\nGET\r\n ($1\r\n#\r\n|*pattern*))* [$3\r\nASC\r\n|$4\r\nDESC\r\n] [$5\r\nALPHA\r\n] [$5\r\nSTORE\r\n *key*]
| *2\r\n$3\r\nTTL\r\n key*
| *2\r\n$7\r\nPERSIST\r\n key*
;

Figure 8: Overapproximate redis grammar, part 1.

list_command →
| **3\r\n$5\r\n* (L|R) *PUSH\r\n key value*
| **3\r\n$6\r\n* (L|R) *PUSHX\r\n key value*
| **5\r\n$7\r\nLINSERT\r\n key*
(*$6\r\nBEFORE*|*$5\r\nAFTER*) *\r\n pivot value*
| **2\r\n$4\r\n* (L|R) *POP\r\n key*
| *** *digit \r\n$5\r\nB* (L|R) *POP\r\n key+ timeout*
| **4\r\n$10\r\nBRPOPLPUSH\r\n key key timeout*
| **2\r\n$4\r\nLLEN\r\n key*
| **3\r\n$6\r\nLINDEX\r\n key index*
| **4\r\n$4\r\nLSET\r\n key index value*
| **4\r\n$6\r\nLRANGE\r\n key begin end*
| **4\r\n$5\r\nLTRIM\r\n key begin end*
| **4\r\n$4\r\nLREM\r\n key count value*
| **3\r\n$9\r\nRPOPLPUSH\r\n key key*
;

set_command →
| **3\r\n$4\r\nS* (ADD|REM) *\r\n key member*
| **4\r\n$5\r\nSMOVE\r\n key key member*
| **3\r\n$9\r\nSISMEMBER\r\n key member*
| **2\r\n$5\r\nSCARD\r\n key*
| **2\r\n$4\r\nSPOP\r\n key*
| **2\r\n$11\r\nSRANDMEMBER\r\n key*
| *** *digit \r\n$6\r\nSINTER\r\n key+*
| *** *digit \r\n$11\r\nSINTERSTORE\r\n key key+*
| *** *digit \r\n$6\r\nSUNION\r\n key+*
| *** *digit \r\n$11\r\nSUNIONSTORE\r\n key key+*
| *** *digit \r\n$5\r\nSDIFF\r\n key+*
| *** *digit \r\n$10\r\nSDIFFSTORE\r\n key key+*
| **2\r\n$8\r\nSMEMBERS\r\n key*
;

connection_command →
| **2\r\n$6\r\nSELECT\r\n db*
| **2\r\n$4\r\nAUTH\r\n password*
| **1\r\n$4\r\nPING\r\n*
| **2\r\n$4\r\nECHO\r\n message*
| **1\r\n$4\r\nQUIT\r\n*
;

one_char_arg → *$1\r\n string1 \r\n*;
one_digit_arg → *$1\r\n digit \r\n*;

key → *one_char_arg*;
value → *one_char_arg*;
pattern → *one_char_arg*;
message → *one_char_arg*;
member → *one_char_arg*;
password → *one_char_arg*;

offset → *one_digit_arg*;
count → *one_digit_arg*;
seconds → *one_digit_arg*;
decrement → *one_digit_arg*;
increment → *one_digit_arg*;
timeout → *one_digit_arg*;
begin → *one_digit_arg*;
end → *one_digit_arg*;
db → *one_digit_arg*;
index → *one_digit_arg*;
pivot → *one_digit_arg*;
timestamp → *one_digit_arg*;

Figure 9: Overapproximate redis grammar, part 2.

start → USER␣ *username* \nPASS␣ *password* \n (*cmd* \n)*
QUIT\n;

cmd →
no_arg_cmd
| *upload_cmd* ␣ *filename*
| REST␣ *int* \n (APPE | STOR | RETR) ␣ *filename*
| RNFR␣ *filename* \nRNTO␣ *filename*
| *dir_cmd* ␣ *dirname*
| *file_cmd* ␣ *filename*
| MODE␣ (S | B | C)
| STRU␣ (F | R | P)
| TYPE␣ ((A␣ | E␣) (N | T | C) | I | L␣ *digit*)
| PORT␣ *int* , *int* , *int* , *int* , *int* , *int*
;

no_arg_cmd → NOOP | CDUP | PWD | HELP | SYST | STAT
| SITE | ABOR | LIST | NLST | PASV;
upload_cmd → APPE | STOR | STOU;
dir_cmd → CWD | MKD | RMD | LIST | NLST | STAT | SMNT;
file_cmd → DELE | RETR | LIST | NLST | STAT;

dirname → [/] *filename* (/ *filename*)*;

filename → *letter*+;
username → *string3*;
password → *string2*;
int → *digit*+;

Figure 10: Full FTP grammar.

start → USER␣ *username* \nPASS␣ *password* \n (*cmd* \n)*
QUIT\n;

cmd →
no_arg_cmd
| REST␣ *int* \nRETR␣ *filename*
| *dir_cmd* ␣ *dirname*
| *file_cmd* ␣ *filename*
| TYPE␣ ((A␣ | E␣) (N | T | C) | I | L␣ *digit*)
| PORT␣ *int* , *int* , *int* , *int* , *int* , *int*
;

no_arg_cmd → PWD | SYST | LIST | PASV;
dir_cmd → CWD | LIST;
file_cmd → RETR | LIST;

dirname → [/] *filename* (/ *filename*)*;

filename → *letter*+;
username → *string3*;
password → *string2*;
int → *digit*+;

Figure 11: Reduced FTP grammar.

B. CONTEXT-FREE GRAMMAR CONSTRAINT GENERATION CODE

Figure 12 shows OCaml code that implements the pseudocode presented in Figure 4. Our actual implementation is somewhat more complicated still because it frees memory allocated by concat, but Figure 12 does not free this memory.

```
type term_or_nonterm = T of string | N of string
type rhs = term_or_nonterm list
type productions = { name : string; rhs_list : rhs list; }
type grammar = productions list

let get_terminals rhs_list =
  let rec split ts others = function
    | [] → ts, others
    | [T t]::rhs_list → split (t::ts) others rhs_list
    | rhs::rhs_list → split ts (rhs::others) rhs_list
  in split [] [] rhs_list

let iteri f lst =
  ignore (List.fold_left (fun i x → f i x; succ i) 0 lst)

let print_term_or_nonterm = function
  | T t → printf "\"%s\"" t
  | N n → printf "generate_%s()" n

let print_rhs = function
  | [] → failwith "Empty rhs"
  | c::cs →
    printf "return concat(";
    print_term_or_nonterm c;
    List.iter (fun c → printf ", "; print_term_or_nonterm c) cs;
    printf ", 0);\n"

let print_rhs_wrapped n rhs =
  printf "if (choice == %d) {\n" n;
  print_rhs rhs;
  printf "} else "

let print_terminal_expansions start t ts =
  printf "return ";
  iteri (fun i t → printf "choice == %d ? \"%s\" :" (start + i) t) ts;
  printf "\"%s\";\n" t

let print_definition { name; rhs_list } =
  printf "char *generate_%s()(void) {\n" name;
  printf "int choice; __SYMBOLIC(&choice);\n";
  let terminals, others = get_terminals rhs_list in
  begin match terminals with
    | terminal :: terminals →
      iteri (fun n prod → print_rhs_wrapped n prod) others;
      print_terminal_expansions (List.length others) terminal terminals
    | [] → match others with
      | [] → failwith "No productions"
      | p :: ps →
          iteri (fun n p → print_rhs_wrapped n p) ps;
          print_rhs p
  end;
  printf "}\n"

let print_grammar = List.iter print_definition
```

Figure 12: Grammar-constraint code generation.