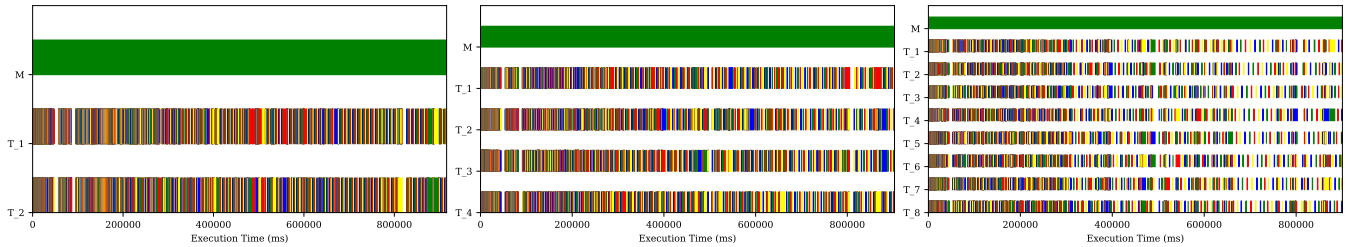

Figure 1: Tcpdump's execution details. *M* represents the execution of the main thread (i.e., running the program's original code and symbolic simulation), and *T_N* represents the executions of Nth constraint solving thread.

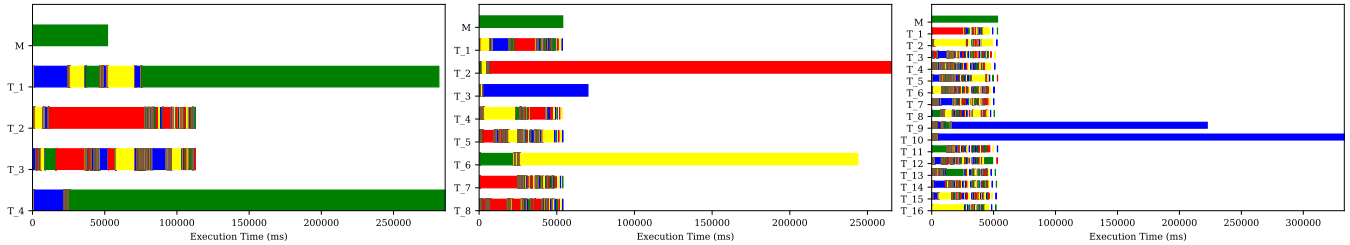

Figure 2: Quickjs's execution details. *M* represents the execution of the main thread (i.e., running the program's original code and symbolic simulation), and *T_N* represents the executions of Nth constraint solving thread.

A INEFFICIENT CONCOLIC EXECUTION

We recorded the execution of each thread and counted the time windows of running the program's original code, the symbolic simulation, and constraint solving. Fig. 1 and Fig. 2 show the execution details of *tcpdump* and *quickjs*, respectively. In both figures, the x-axis represents the timeline, while the y-axis represents the execution details of distinct threads. Each thread's execution consists of multiple rectangles, with each rectangle representing a task executed in the relevant thread. The width of each rectangle indicates the task's execution time. The experiment illustrates the causes of concolic execution performance bottlenecks.

The first reason is that the majority of symbolic constraints constructed during execution are relatively simple. As a result, the time spent on constraint solving is negligible when compared to running the program's original code and symbolic simulation. *Tcpdump* serves as a prime example of this. As illustrated in Fig. 1, the total execution time largely depends on the main thread's execution time. Since each round of constraint solving only takes a brief amount of time, it is not possible to achieve further performance improvement by increasing the number of solving threads. It has been observed that using more threads leads to a sparser distribution of time consumption rectangles, indicating that the solving threads remain idle for most of the time (i.e., wider blank intervals).

The second reason is that some constraints are extremely complex, and solving them takes an extended amount of time. *Quickjs* serves as a prime example of this, with its thread execution details displayed in Fig. 2. As observed, the concolic execution of *Quickjs* spends more time on solving constraints rather than on the main thread, and the total execution time is determined by the constraint solving time consumption. According to the figures, there are two extremely time-consuming constraint solving tasks that significantly reduce the overall efficiency. Moreover, we discovered that when using more threads, these two tasks take even longer than when using fewer threads. This is due to the inefficient state-sharing mechanism of Z3.

B SYMBOLIC MEMORY ADDRESSING EXAMPLE

Listing 1 shows an example of the memory addressing problem in symbolic execution. For the first round of execution, `data` is `0x02000000FFFF02000000FFFF`, which sets both `len1` and `len2` equal to 2. As a result, the program output is "Small data". When the concolic execution is implemented, it generates the constraint $input[0:4] > 0 \wedge input[0:4] < 10 \wedge input[6:10] > 0 \wedge input[6:10] < 10 \wedge input[0:4] + input[6:10] > 5$ at line 15 for the purpose of path exploration. The SMT solver will then work to solve the constraint and generate a new input. Let's assume the new input to be `0x03000000FFFF03000000FFFF`, which is expected to make the program print out "Big data". However, it instead results in `len1` equating to 3 and `len2` equating to 0xFF000000, leading the program to print out "Small data" once again.

Listing 1: An example of symbolic memory addressing.

```
1  char *data = (char *) malloc(100);
2  read(0, data, 100); // 0x02000000FFFF02000000FFFF
3  char *pos = data;
4
5  int len1 = *(int *)pos; // 0x02000000
6  pos += sizeof(int);
7  if (!(len1 > 0 && len1 <10)) exit(-1);
8  pos += len1;
9
10 int len2 = *(int *)pos; // 0x02000000
11 pos += sizeof(int);
12 if (!(len2 > 0 && len2 <10)) exit(-1);
13 pos += len2;
14
15 if (len1 + len2 > 5) printf("Big data\n");
16 else printf("Small data\n");
```