Fuzzing Program Models with Satisfiability Modulo Theories (SMT) Solvers

WILSON NGUYEN, Stanford University

Abstract: A framework was developed to model programs for the purpose of finding inputs that would crash the programs. The framework was developed in the SMT-LIB v2 input and output language [1] for SMT solvers. Programs were limited in scope to those solely contain integers, booleans, integer operations, random access arrays, array index retrievals, if and else statements, and for-loop constructs. Once programs were modeled, assertions would be created to constrain the scope of possible inputs and assert that the program models would evaluate to a 'Crash' type. The Z3 theorem prover by Microsoft Research [11] was used to check the satisfiability of these assertions and return a model satisfying all of them. The model returned would contain inputs that would crash the modeled-program. The framework is currently successful at finding inputs that cause crashes by undefined integer operations and array index out of bounds retrievals.

CS Concepts: • **Programming Languages → SMT Solving**; *Z3 theorem prover*; • **Program Modeling → Fuzzing**; crashes

KEYWORDS
Fuzzing, SMT, Satisfiability Modulo Theories, Z3, Crashes

1 INTRODUCTION

Fuzzing is a software testing technique [6] meant to reveal inputs that cause the software being tested to crash, leak memory, or unintentionally modify protected program memory. Currently, popular fuzzing engines such as SAGE and Triton [3] use a mixture of randomized inputs and constraint solving to reveal crash-causing inputs. The framework discussed in this paper solely uses constraint solving to find these crash-causing inputs. Fuzzing assists software developers by finding security bugs in their code and language designers by addressing the memory and logical safety of tested programming languages.

Satisfiability Modulo Theories or SMT problems [7] are constraint satisfaction problems modeled in logical formulas incorporating theories expressed in first order logic. Theories used in this framework include Ints [9], ArraysEx [10], and Core theory (basic Boolean operators and functions) [8]. SMT solvers take in a set of assertions (logical formulas/constraints) and attempt to find a model that satisfies all of these assertions. This relates to the constraint-programming paradigm [5] in which relations between constants and variables are expressed as constraints.

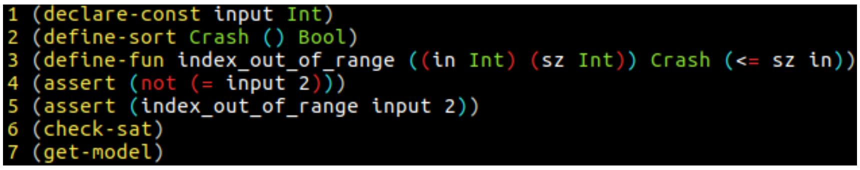
2 PROCESS AND EXPERIMENTATION DETAILS

2.1 Learning SMT-LIB v2 and Z3, Basic fuzzing

Before this project, I had no experience with SMT, Z3, or fuzzing. Thus, I had to learn the strategies related to constraint solving, fuzzing ideals, and the SMT-LIB v2 language used to interact with SMT solvers (specifically Z3). After the initial portion of the rise4fun Z3 tutorial [2], I was able to develop this very crude, simple fuzzer for finding an index out of range (code on following page).

My initial approach was to define a datatype called *Crash,* which was based on a boolean. A *Crash* could be either true or false. The constant *input* was an unassigned integer. The function *index_out_of_range* was defined to take in two integers: *in,* an index, *sz*, the size of the array.

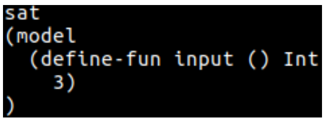
index_out_of_range would return a *Crash* type which would be true if the size of the array was less than or equal to the index and false otherwise. I created two assertions: the first was a formula that would be true if *input* was not equal to 2; the second was the function *index_out_of_range* called on the arguments *input* and 2. Thus, the input to the program cannot be an integer that has a value of 2 and the function must return a *Crash* which is true.



```
(declare-const input Int)
(define-sort Crash () Bool)
(define-fun index_out_of_range ((in Int) (sz Int)) Crash (<= sz in))
(assert (not (= input 2)))
(assert (index_out_of_range input 2))
(check-sat)
(get-model)
```

A first crude fuzzing attempt.

Checking satisfiability guarantees that there exists a model where all assertions could be satisfied (the logical formulas in the assertions all evaluate to true). The command *get-model* returns a model that satisfies all the assertions; in this case, the model below defines *input* as a function that takes in no arguments and returns an integer value of 3. Everything in SMT-LIB v2 is a function[1]. Thus, constants are just syntactic sugar for functions. In context, Z3 found an index, which was 3, that would cause an out of bounds array access, on an array of size 2.

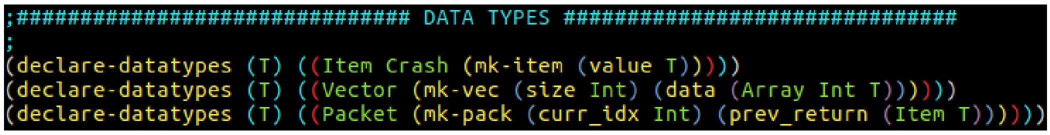


```
sat
(model
  (define-fun input () Int
    3)
)
```

Z3 output in SMT-LIB v2

2.2 Defining Datatypes

After completing the rise4fun Z3 tutorial[2] and reading a significant portion of the documentation for SMT-LIB, I began to write the framework for what would ultimately become the core of my SMT fuzzer. Outlined in the following sections is the streamlined framework (many hours of debugging, searching, and restarting has led to this). The first step was to create datatypes that would become the foundational types for input, output, and information transfer in the framework.



```
;############################### DATA TYPES ###############################
;
(declare-datatypes (T) ((Item Crash (mk-item (value T)))))
(declare-datatypes (T) ((Vector (mk-vec (size Int) (data (Array Int T))))))
(declare-datatypes (T) ((Packet (mk-pack (curr_idx Int) (prev_return (Item T))))))
```

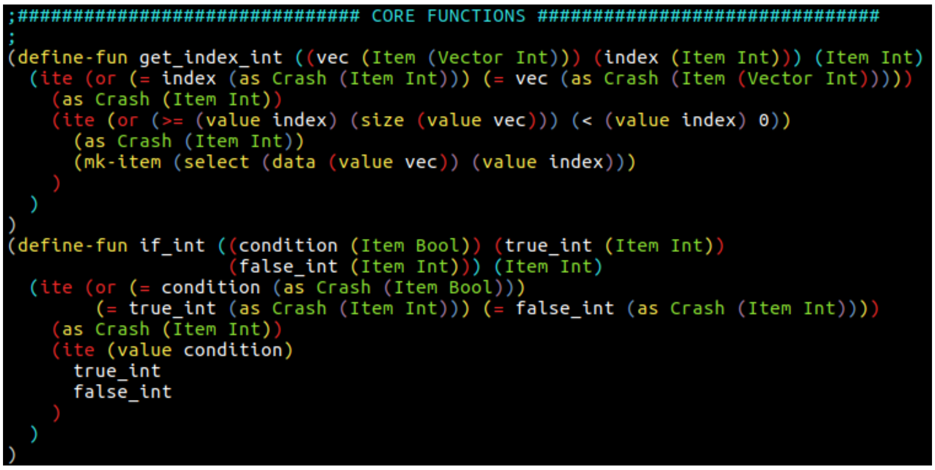
Declaration of Item, Vector, and Packet datatypes.

Inspired by exceptions and some-types, I envisioned that all functions would receive as input either a crash or a valid value. If a function received a crash, then it would in return output a crash that would cause surrounding functions to return crashes (carrying crashes up the execution chain). Thus, I defined an *Item* type that was either a *Crash* or contained a *value* of type *T*. The symbol *T* is a placeholder for a type; therefore, *Items* can contain values of any type. Since everything in SMT-LIB is a function, *Arrays* are simply functions that take in an integer (the index) and return a value (the value at the index). Since I was testing out of bounds access, I needed a datatype that contained both *Array* functionality and a limited size. Thus, I defined the type *Vector,* which contained both an integer *size*, and *data* an *Array Int T*. Thus, data would extract a function that takes in an integer and returns a value of type *T*. Finally, in order to model for-loops, I needed a datatype that would package up the current index and the value returned by the previous iteration. Thus, the datatype *Packet* contains an integer *curr_idx* and an *Item T* retrieved by *prev_return*.

Note: In these datatype declarations, value, size, data, curr_idx, prev_return are selector functions that extract values out of the datatypes and mk-item, mk-vec, mk-pack are constructors.[2]

2.3 Core Functions

I decided that the essential expressions to include in the framework were index access, if/else statements, and for-loops. Thus, I defined functions that would emulate these and propagate crashes when necessary.

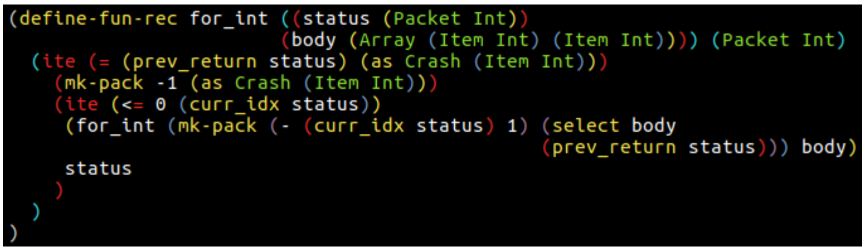


```
;################################# CORE FUNCTIONS #################################
;
(define-fun get_index_int ((vec (Item (Vector Int))) (index (Item Int))) (Item Int)
  (ite (or (= index (as Crash (Item Int))) (= vec (as Crash (Item (Vector Int)))))
    (as Crash (Item Int))
    (ite (or (>= (value index) (size (value vec))) (< (value index) 0))
      (as Crash (Item Int))
      (mk-item (select (data (value vec)) (value index)))
    )
  )
)
(define-fun if_int ((condition (Item Bool)) (true_int (Item Int))
                    (false_int (Item Int))) (Item Int)
  (ite (or (= condition (as Crash (Item Bool)))
        (= true_int (as Crash (Item Int))) (= false_int (as Crash (Item Int))))
    (as Crash (Item Int))
    (ite (value condition)
      true_int
      false_int
    )
  )
)
```

Model for array index retrieval and if/else statements

The function *get_index_int* takes in as arguments an *Item (Vector Int)* assigned to *vec* and an *Item Int* assigned to *index* and returns an *Item Int*. The command *ite (boolean expression) (expression a) (expression b)* returns *expression a* if the *boolean expression* is true and *expression b* otherwise. Thus, *get_index_int* returns a *Crash* when vec or index are *Crash*, or when the value of *index* is greater than or equal to the size of *vec*, or when the value of *index* is less than zero. If *get_index_int* does not return a *Crash*, it returns an *Item Int*, which is the value at the arg *index*.

The function *if_int* takes in as arguments an *Item Bool* as condition, *Item Int* as *true_int*, and *Item Int* as *false_int.* If any of the input parameters are *Crash*, then the if/else statement will return a crash. Otherwise, *if_int* will return the appropriate *Item Int: true_int or false_int.*

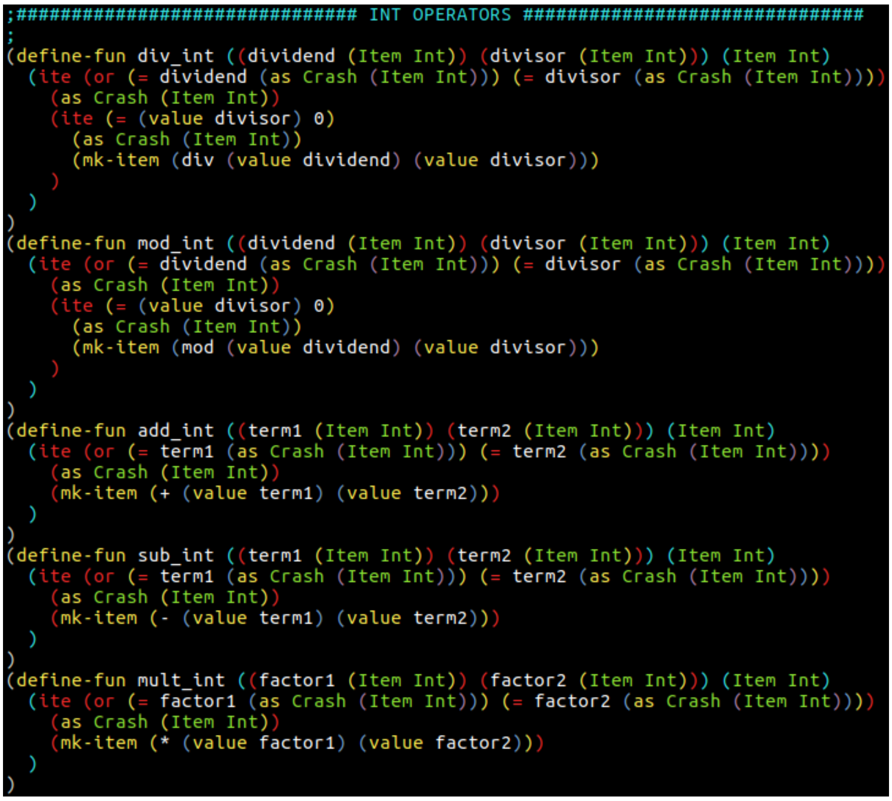


```
(define-fun-rec for_int ((status (Packet Int))
                         (body (Array (Item Int) (Item Int)))) (Packet Int)
  (ite (= (prev_return status) (as Crash (Item Int)))
    (mk-pack -1 (as Crash (Item Int)))
    (ite (<= 0 (curr_idx status))
     (for_int (mk-pack (- (curr_idx status) 1) (select body
                                                 (prev_return status))) body)
     status
    )
  )
)
```

There were three individual implementations of the for-loop function each with their own tradeoffs (recursive function theories are on the very limit of what Z3 can handle). The one presented in this section was the most generic (other versions are described in the results discussion section below). Conceptually, I modeled a for-loop as a recursive function that takes in as arguments a status (containing the current index and the return value of the previous iteration) and a function that serves as the body of the for-loop. Since SMT-LIB[1] does not have theories that support higher-order functions, I decided to use the theories of Arrays[10] to emulate a lambda function as an argument (since arrays are just syntactic sugar for a function). The *for_int* propagates a *Crash* when the previous iteration returned a *Crash*. Otherwise, if the current index is greater than or equal to zero, recurse further with a new status that contains 1 minus the current index and the value returned by the body function evaluated on the value of the previous iteration. Otherwise, if the current index is less than zero, return the last status.

2.4 Integer Operations

Up until this stage, the only crashes modeled are those caused by index out of bounds during array retrievals. To further the model, I wanted to build on the theory of Integers. *Crash* needs to be propagated up the chain during integer operations. Additionally, division or mod with a dividend of zero should result in a crash.



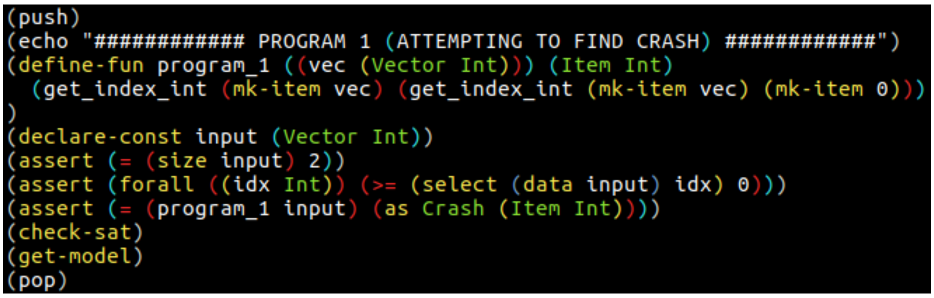
```
;################################ INT OPERATORS ################################
;
(define-fun div_int ((dividend (Item Int)) (divisor (Item Int))) (Item Int)
  (ite (or (= dividend (as Crash (Item Int))) (= divisor (as Crash (Item Int))))
    (as Crash (Item Int))
    (ite (= (value divisor) 0)
      (as Crash (Item Int))
      (mk-item (div (value dividend) (value divisor)))
    )
  )
)
(define-fun mod_int ((dividend (Item Int)) (divisor (Item Int))) (Item Int)
  (ite (or (= dividend (as Crash (Item Int))) (= divisor (as Crash (Item Int))))
    (as Crash (Item Int))
    (ite (= (value divisor) 0)
      (as Crash (Item Int))
      (mk-item (mod (value dividend) (value divisor)))
    )
  )
)
(define-fun add_int ((term1 (Item Int)) (term2 (Item Int))) (Item Int)
  (ite (or (= term1 (as Crash (Item Int))) (= term2 (as Crash (Item Int))))
    (as Crash (Item Int))
    (mk-item (+ (value term1) (value term2)))
  )
)
(define-fun sub_int ((term1 (Item Int)) (term2 (Item Int))) (Item Int)
  (ite (or (= term1 (as Crash (Item Int))) (= term2 (as Crash (Item Int))))
    (as Crash (Item Int))
    (mk-item (- (value term1) (value term2)))
  )
)
(define-fun mult_int ((factor1 (Item Int)) (factor2 (Item Int))) (Item Int)
  (ite (or (= factor1 (as Crash (Item Int))) (= factor2 (as Crash (Item Int))))
    (as Crash (Item Int))
    (mk-item (* (value factor1) (value factor2)))
  )
)
```

Integer operator definitions

Here I defined the integer operations as expected. All of these functions return a *Crash* if any of the input *Item Int* s are of the *Crash* type. Additionally for the case of mod and division, if the dividend item has the value of 0, then it should return a crash. Otherwise, the functions return what is expected from their operation.

2.5 Program Models

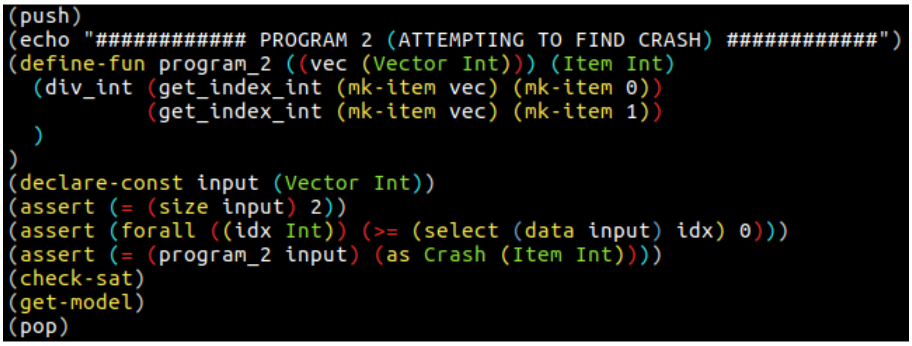
As suggested by Varun (CS242 TA), I decided to limit the scope of programs to those only composed of the datatypes, core functions, and operations defined above. Additionally, programs are set to take in an arbitrary length vector (unless restricted by an assertion) and eventually return a single integer. Thus, programs are defined as functions that take in a *Vector Int* and return an *Item Int,* which could be a *Crash* type. Below is the first program definition.



```
(push)
(echo "############ PROGRAM 1 (ATTEMPTING TO FIND CRASH) ############")
(define-fun program_1 ((vec (Vector Int))) (Item Int)
  (get_index_int (mk-item vec) (get_index_int (mk-item vec) (mk-item 0)))
)
(declare-const input (Vector Int))
(assert (= (size input) 2))
(assert (forall ((idx Int)) (>= (select (data input) idx) 0)))
(assert (= (program_1 input) (as Crash (Item Int))))
(check-sat)
(get-model)
(pop)
```

Program 1.

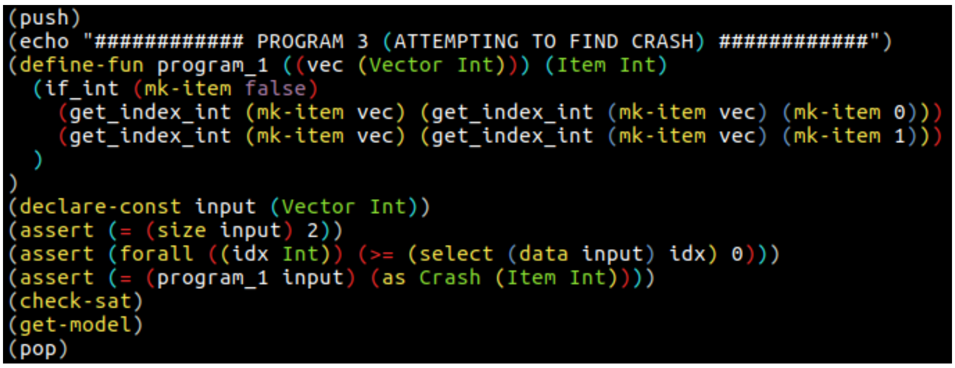
Z3 maintains a global stack of all user declared formulas and assertions [2]. In order to prevent definition and declaration conflicts, I decided to model each program within its own stack scope with the push and pop SMT-LIB commands. The function *program_1* models a simple program that returns the integer found at the index, which is the integer at index 0. In a more familiar syntax, *program_1* attempts to model *vec[vec[0]]*. The input to the program is a constant declared as *input* of type *Vector Int*. Constants are functions that are not predefined (thus, Z3 must find a model containing a definition for this *input*). I added a couple of additional assertions to limit the scope of the input, just to test cases where the limited scope is necessary (if the program has a restricted input set). In this case, the *input* must have a size of 2 and all elements in the *Vector* must be greater than or equal to zero. The final assumption is that the return of *program_1* called on *input* is equal to *Crash*. The commands *check-sat* and *get-model* check to see if all assertions can be satisfied and retrieve a model that does so.



```
(push)
(echo "############ PROGRAM 2 (ATTEMPTING TO FIND CRASH) ############")
(define-fun program_2 ((vec (Vector Int))) (Item Int)
  (div_int (get_index_int (mk-item vec) (mk-item 0))
           (get_index_int (mk-item vec) (mk-item 1))
  )
)
(declare-const input (Vector Int))
(assert (= (size input) 2))
(assert (forall ((idx Int)) (>= (select (data input) idx) 0)))
(assert (= (program_2 input) (as Crash (Item Int))))
(check-sat)
(get-model)
(pop)
```

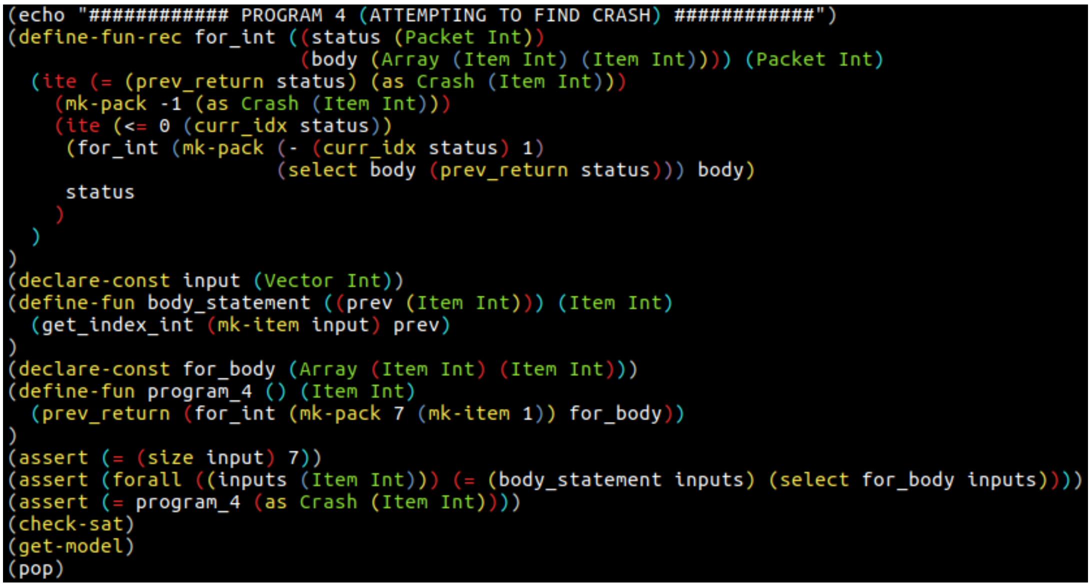
Program 2.

Program 2: To demonstrate the ability of the fuzzer to find crashes related to integer operations, I model the program: *vec[0]/vec[1]* with similar assertions on the input.



```
(push)
(echo "############ PROGRAM 3 (ATTEMPTING TO FIND CRASH) ############")
(define-fun program_1 ((vec (Vector Int))) (Item Int)
  (if_int (mk-item false)
    (get_index_int (mk-item vec) (get_index_int (mk-item vec) (mk-item 0)))
    (get_index_int (mk-item vec) (get_index_int (mk-item vec) (mk-item 1)))
  )
)
(declare-const input (Vector Int))
(assert (= (size input) 2))
(assert (forall ((idx Int)) (>= (select (data input) idx) 0)))
(assert (= (program_1 input) (as Crash (Item Int))))
(check-sat)
(get-model)
(pop)
```

Program 3: To test the functionality of if statements, I modeled the program: *if false: vec[vec[0]] else: vec[vec[1]].*



```
(echo "############ PROGRAM 4 (ATTEMPTING TO FIND CRASH) ############")
(define-fun-rec for_int ((status (Packet Int))
                         (body (Array (Item Int) (Item Int)))) (Packet Int)
  (ite (= (prev_return status) (as Crash (Item Int)))
    (mk-pack -1 (as Crash (Item Int)))
    (ite (<= 0 (curr_idx status))
     (for_int (mk-pack (- (curr_idx status) 1)
                       (select body (prev_return status))) body)
     status
    )
  )
)
(declare-const input (Vector Int))
(define-fun body_statement ((prev (Item Int))) (Item Int)
  (get_index_int (mk-item input) prev)
)
(declare-const for_body (Array (Item Int) (Item Int)))
(define-fun program_4 () (Item Int)
  (prev_return (for_int (mk-pack 7 (mk-item 1)) for_body))
)
(assert (= (size input) 7))
(assert (forall ((inputs (Item Int))) (= (body_statement inputs) (select for_body inputs))))
(assert (= program_4 (as Crash (Item Int))))
(check-sat)
(get-model)
(pop)
```

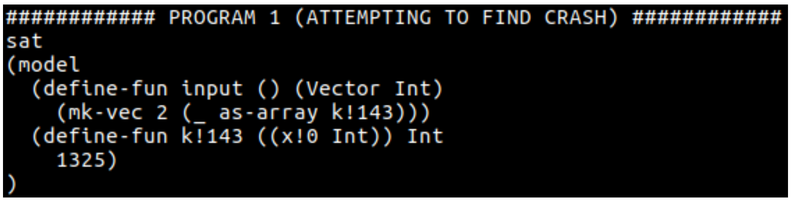
Program 4.

Finally, to test the functionality of for-loops, I created *program_4*. Used here is one of the three different implementations of for-loops (later versions discussed in results and discussion below). First, I declare the constant *input* as a *Vector Int* with a later assertion that it must have a size of seven. The function *body_statement* contains the expressions that are contained in the for-loop body. In this case, the body models this: *vec[prev]* where prev is the *Item Int* returned by the previous iteration. Since Z3 does not support higher-order functions (it just does not have those theories yet) and has trouble with function macros, I had to declare a constant *for_body* that is an *Array (Item Int) (Item Int)* and assert that for all possible indexes, the return of *for_body* would be equal to the return of *body_statement.* This was meant to imitate the functionality of lambda functions in SMT-LIB. Finally, *program_4* is defined as the last return of the recursive call to *for_int* that starts off with a status of current index 7 and a previous return value of 1. The body of the for-loop is *for_body*. The final assertion is that *program_4* returns a *Crash*.

3 RESULTS AND DISCUSSION

3.1 Results of SMT Solving (Z3) - **SUCCESS**

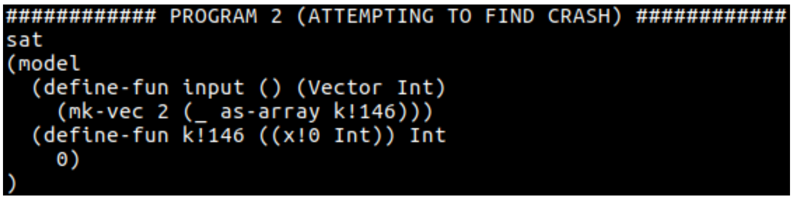
Every collection of assertions, declarations, and definitions (in the push and pop stack scopes) were both satisfiable and produced valid models that would have resulted in crashes. The output of Z3 below states that a vector of size 2 that contains an array that always returns 1325 will crash the program. Thus, *program_1: vec[vec[0]]* will crash if the input vector given is *[1325, 1325]*.



```
############ PROGRAM 1 (ATTEMPTING TO FIND CRASH) ############
sat
(model
  (define-fun input () (Vector Int)
    (mk-vec 2 (_ as-array k!143)))
  (define-fun k!143 ((x!0 Int)) Int
    1325)
)
```

Z3 output on Program_1

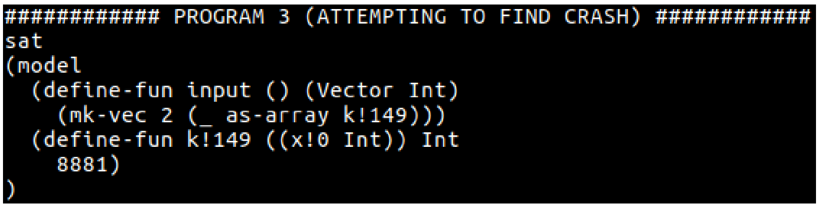
The output of Z3 states that an input vector of *[0,0]* will crash *program_2: vec[0]/vec[1].*



```
############ PROGRAM 2 (ATTEMPTING TO FIND CRASH) ############
sat
(model
  (define-fun input () (Vector Int)
    (mk-vec 2 (_ as-array k!146)))
  (define-fun k!146 ((x!0 Int)) Int
    0)
)
```

Z3 output on Program_2

The output of Z3 for *program_3* states that an input vector of *[8881, 8881]* will crash the program *if false: vec[vec[0]] else: vec[vec[1]].* It is important to note that Z3 did not have to produce a model containing both elements in the vector having the same value (as in all the previous models outputted). As can be seen in the for-loop program (discussed below), Z3 returns the first model that it finds that satisfies all the assertions.



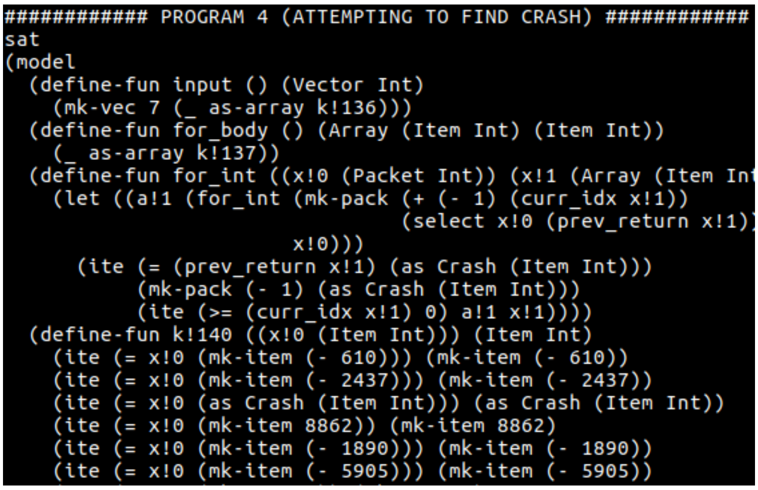
```
############ PROGRAM 3 (ATTEMPTING TO FIND CRASH) ############
sat
(model
  (define-fun input () (Vector Int)
    (mk-vec 2 (_ as-array k!149)))
  (define-fun k!149 ((x!0 Int)) Int
    8881)
)
```

Z3 output on Program_3

Apologies for the awkward space here. The photo would not fit on this page.

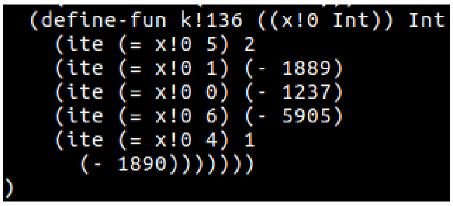
3.2 For-loops - Z3 / SMT Limits

Since there were two constants in *program_4*, *for_body* and *input*, which were *Array (Item Int) (Item Int)* and V*ector,* respectively. Z3 produced a model containing definitions for both.



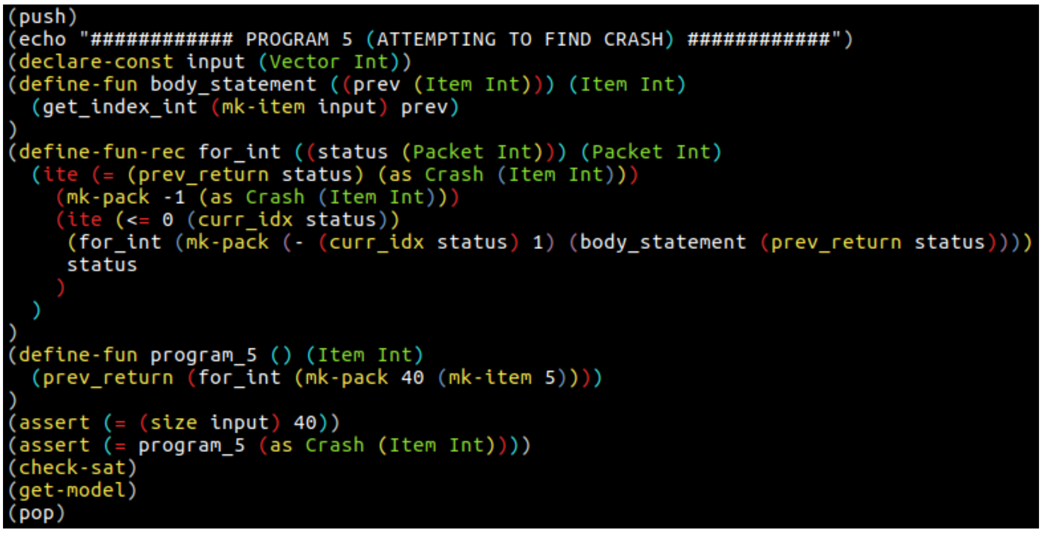
```
############ PROGRAM 4 (ATTEMPTING TO FIND CRASH) ############
sat
(model
  (define-fun input () (Vector Int)
    (mk-vec 7 (_ as-array k!136)))
  (define-fun for_body () (Array (Item Int) (Item Int))
    (_ as-array k!137))
  (define-fun for_int ((x!0 (Packet Int)) (x!1 (Array (Item In
    (let ((a!1 (for_int (mk-pack (+ (- 1) (curr_idx x!1))
                                 (select x!0 (prev_return x!1)
                        x!0)))
      (ite (= (prev_return x!1) (as Crash (Item Int)))
           (mk-pack (- 1) (as Crash (Item Int)))
           (ite (>= (curr_idx x!1) 0) a!1 x!1))))
  (define-fun k!140 ((x!0 (Item Int))) (Item Int)
    (ite (= x!0 (mk-item (- 610))) (mk-item (- 610))
    (ite (= x!0 (mk-item (- 2437))) (mk-item (- 2437))
    (ite (= x!0 (as Crash (Item Int))) (as Crash (Item Int))
    (ite (= x!0 (mk-item 8862)) (mk-item 8862)
    (ite (= x!0 (mk-item (- 1890))) (mk-item (- 1890))
    (ite (= x!0 (mk-item (- 5905))) (mk-item (- 5905))
```

– 90 lines omitted for clarity –



```
  (define-fun k!136 ((x!0 Int)) Int
    (ite (= x!0 5) 2
    (ite (= x!0 1) (- 1889)
    (ite (= x!0 0) (- 1237)
    (ite (= x!0 6) (- 5905)
    (ite (= x!0 4) 1
      (- 1890)))))))
)
```

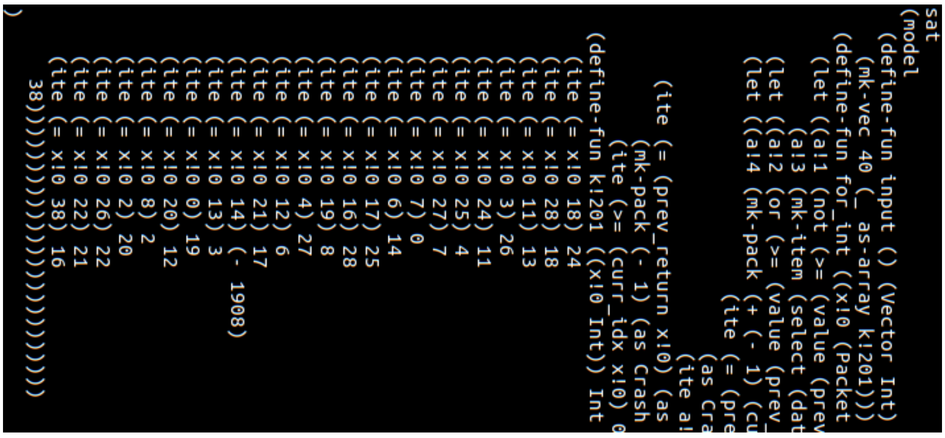
The resulting model identified the vector *[-1237, -1889, -1890, -1890, 1, 2, -5905]* as the input that would cause a crash. I attempted to test iterations of higher than seven and found that for an iteration count of higher than 11, Z3 was unable to find a satisfying model. Z3 would either hang indefinitely (as much as I could tell) or respond with a message saying that it was unknown whether the assertions were satisfiable. Recursive functions are on the very limit of the theories that Z3 can handle. Additionally, my use of Array theory [10] to create pseudo-lambda functions that could be passed in as arguments significantly increased the computation required, because Z3 needed to generate a definition for the *for_body* constant (although *body_statements* was an already a defined function). I preferred this modeled of a for-loop because *for_int* was generic over any body since the body function could be passed in as a lambda (described earlier). However, a cap of 11 iterations was not satisfying to me. Thus, I decided to move in the direction of a less generic solution that could allow for more iterations.



```
(push)
(echo "############ PROGRAM 5 (ATTEMPTING TO FIND CRASH) ############")
(declare-const input (Vector Int))
(define-fun body_statement ((prev (Item Int))) (Item Int)
  (get_index_int (mk-item input) prev)
)
(define-fun-rec for_int ((status (Packet Int))) (Packet Int)
  (ite (= (prev_return status) (as Crash (Item Int)))
    (mk-pack -1 (as Crash (Item Int)))
    (ite (<= 0 (curr_idx status))
     (for_int (mk-pack (- (curr_idx status) 1) (body_statement (prev_return status))))
     status
    )
  )
)
(define-fun program_5 () (Item Int)
  (prev_return (for_int (mk-pack 40 (mk-item 5))))
)
(assert (= (size input) 40))
(assert (= program_5 (as Crash (Item Int))))
(check-sat)
(get-model)
(pop)
```

for-loop model 2

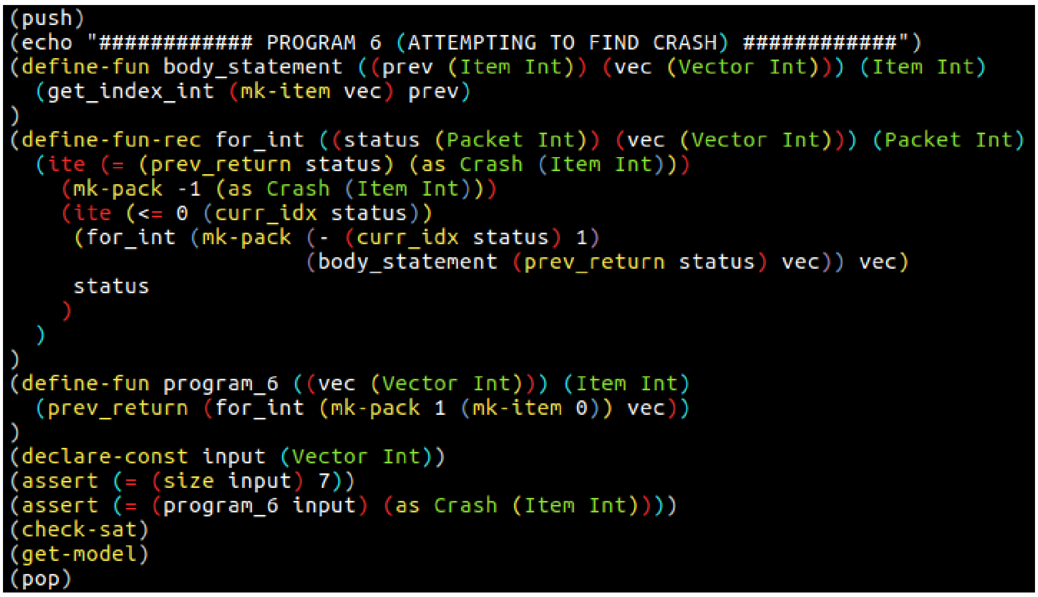
In this model, instead of passing in the body of the for-loop as an *Array*, the function *for_int* directly calls the *body_statement* function. The tradeoff, however, is that a new *for_int* function must be modeled for each for-loop in the program. However, as evident by the result below, Z3 is able to create a model that satisfies all the assertions even at moderate iterations (40 for this test).



```
sat
(model
  (define-fun input () (Vector Int)
    (mk-vec 40 (_ as-array k!201)))
  (define-fun for_int ((x!0 (Packet
    (let ((a!1 (not (>= (value (prev
          (a!3 (mk-item (select (dat
    (let ((a!2 (or (>= (value (prev
    (let ((a!4 (mk-pack (+ (- 1) (cu
                (ite (= (pre
                  (as Cra
                    (ite a!
      (ite (= (prev_return x!0) (as
        (mk-pack (- 1) (as Crash
          (ite (>= (curr_idx x!0) 0
  (define-fun k!201 ((x!0 Int)) Int
    (ite (= x!0 18) 24
    (ite (= x!0 28) 18
    (ite (= x!0 11) 13
    (ite (= x!0 3) 26
    (ite (= x!0 24) 11
    (ite (= x!0 25) 4
    (ite (= x!0 27) 7
    (ite (= x!0 7) 0
    (ite (= x!0 6) 14
    (ite (= x!0 17) 25
    (ite (= x!0 16) 28
    (ite (= x!0 19) 8
    (ite (= x!0 4) 27
    (ite (= x!0 12) 6
    (ite (= x!0 21) 17
    (ite (= x!0 14) (- 1908)
    (ite (= x!0 13) 3
    (ite (= x!0 0) 19
    (ite (= x!0 20) 12
    (ite (= x!0 8) 2
    (ite (= x!0 2) 20
    (ite (= x!0 26) 22
    (ite (= x!0 22) 21
    (ite (= x!0 38) 16
      38)))))))))))))))))))))))))
)
```

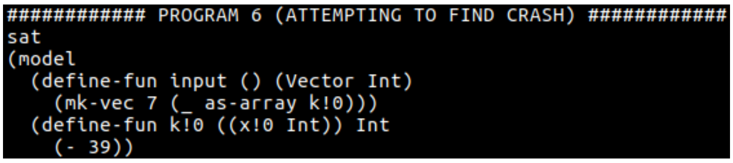
Z3 output for Program_5. Horizontal.

To counteract the lack of generalness in the for-loop model in *program_5*, I attempted to change the model for for-loops and for-bodies to take in a vector as an argument instead of directly accessing the *input* vector in *body_statement*. However, this resulted in an iteration cap of just one (mirroring the cap found in *program_4)*. Possibly, this model hit the limits of the recursive function theory used by Z3. (model is on next page)



```
(push)
(echo "############ PROGRAM 6 (ATTEMPTING TO FIND CRASH) ############")
(define-fun body_statement ((prev (Item Int)) (vec (Vector Int))) (Item Int)
  (get_index_int (mk-item vec) prev)
)
(define-fun-rec for_int ((status (Packet Int)) (vec (Vector Int))) (Packet Int)
  (ite (= (prev_return status) (as Crash (Item Int)))
    (mk-pack -1 (as Crash (Item Int)))
    (ite (<= 0 (curr_idx status))
     (for_int (mk-pack (- (curr_idx status) 1)
                       (body_statement (prev_return status) vec)) vec)
     status
    )
  )
)
(define-fun program_6 ((vec (Vector Int))) (Item Int)
  (prev_return (for_int (mk-pack 1 (mk-item 0)) vec))
)
(declare-const input (Vector Int))
(assert (= (size input) 7))
(assert (= (program_6 input) (as Crash (Item Int))))
(check-sat)
(get-model)
(pop)
```

Program_6 model.



```
############ PROGRAM 6 (ATTEMPTING TO FIND CRASH) ############
sat
(model
  (define-fun input () (Vector Int)
    (mk-vec 7 (_ as-array k!0)))
  (define-fun k!0 ((x!0 Int)) Int
    (- 39))
```

Z3 output for Program_6

4 CONCLUSIONS

4.1 Summary

In summary, I developed a framework to model programs for finding inputs that will cause the programs to crash. For simple programs containing only integer operations, array index retrievals, and if/else statements, Z3 proved the satisfiability of the assertions (in the program models) and successfully generated models containing input vectors that would crash the programs. For-loops provided an even greater challenge to model, since recursive function theories are at the limits of what Z3 can handle and higher-order function theories do not exist in SMT-LIB v2. Three implementations of for-loops provided different trade-offs. The first model of a for-loop in *program_4* was generic over any for-loop body because it could take in a body function as an argument (by using Array theory[10] to imitate higher-order functions and lambdas). However, Z3 could only handle up to 11 iterations with this. A less generic model was used in *program_5* that succeeded to find crashing inputs on moderately high iteration counts (40 iterations) and vector sizes (40 elements). Finally, a third for-loop model was created that failed to reach more than one iteration.

4.2 Future work

Although this was an interesting approach to fuzzing, the limits of this approach were hit quickly. Despite its successes on simple programs, Z3 was unable to generate models that could satisfy more functionally complex programs. For example, very high iterations on for-loops with complex body statements failed to produce satisfying models. This is maybe why successful fuzzers use a mixture of randomized inputs and constraint solving to find crash causing inputs (also known as concolic execution) [3]. Randomized inputs are used to increase code coverage in areas where the constraints are lax and constraint solving for more efficiently reaching areas of code than randomly trying inputs.

REFERENCES

[1] Barrett, Clark, et al. "The SMT-LIB Standard Version 2.6." 18 July 2017, smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.
[2] Bjorner, Nikolaj. "Getting Started with Z3: A Guide." rise4fun, Microsoft Research, 2017, rise4fun.com/Z3/tutorialcontent/guide
[3] "Concolic Testing." Wikipedia, Wikimedia Foundation, 19 Sept. 2017, en.wikipedia.org/wiki/Concolic_testing.
[4] Cok, David R. "The SMT-LIBv2 Language and Tools: A Tutorial." 23 Nov. 2013, smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf.
[5] "Constraint Programming." Wikipedia, Wikimedia Foundation, 26 Nov. 2017, en.wikipedia.org/wiki/Constraint_programming.
[6] "Fuzzing." Wikipedia, Wikimedia Foundation, 5 Dec. 2017, en.wikipedia.org/wiki/Fuzzing.
[7] "Satisfiability modulo Theories." Wikipedia, Wikimedia Foundation, 4 Dec. 2017, en.wikipedia.org/wiki/Satisfiability_modulo_theories.
[8] Tinelli, Cesare. "Core." The Satisfiability Modulo Theories Library, SMT-LIB, 25 Apr. 2015, smtlib.cs.uiowa.edu/theories-Core.shtml.
[9] Tinelli, Cesare. "Ints." The Satisfiability Modulo Theories Library, SMT-LIB, 25 Apr. 2015, smtlib.cs.uiowa.edu/theories-Ints.shtml.
[10] Tinelli, Cesare. "ArraysEx." The Satisfiability Modulo Theories Library, SMT-LIB, 25 Apr. 2015, smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.
[11] "Z3Prover/z3." GitHub, 4 Dec. 2017, github.com/Z3Prover/z3.