Symbolic Execution with Function Abstraction

A thesis submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Harshvardhan Sharma

to the
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
July, 2016

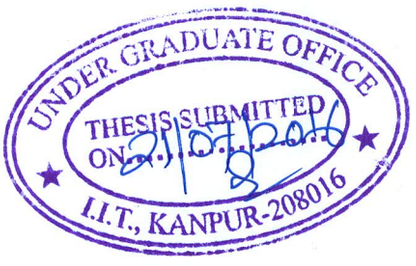
CERTIFICATE

It is certified that the work contained in the thesis titled **Symbolic Execution with Function Abstraction**, by **Harshvardhan Sharma**, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sandeep Shukla
Department of Computer Science & Engineering
IIT Kanpur

Dr. Subhajit Roy
Department of Computer Science & Engineering
IIT Kanpur

July, 2016



ABSTRACT

Name of student: **Harshvardhan Sharma** Roll no: **11907299**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Symbolic Execution with Function Abstraction**

Names of Thesis Supervisors: **Dr. Sandeep Shukla, Dr. Subhajit Roy**

Month and year of thesis submission: **July, 2016**

Symbolic Execution has proven to be an effective technique for identifying bugs in software. It involves running the program on "symbolic" inputs which are initially unconstrained. The executor builds constraints for different program paths and uses an SMT (Satisfiability Modulo Theories) solver to generate concrete inputs for which the program will take those paths. However, the efficacy of symbolic execution is limited by the problem of path explosion.

In this thesis, we try to overcome this problem by selectively abstracting function calls during execution. If we encounter any bugs in our over-approximate execution, we refine the abstractions and attempt to generate concrete inputs that will trigger the bug. In this case, we do no worse than standard symbolic execution. In the absence of bugs on the abstract path, however, we avoid analysis of a potentially unbounded number of program paths.

To my parents

Acknowledgements

I would like to express my profound gratitude to my advisors, Prof. Sandeep Shukla and Prof. Subhajit Roy for introducing me to the area of program analysis and providing their valuable guidance on every aspect of this thesis.

I am indebted to my family and friends for their support during the five years of my stay at IIT Kanpur.

I would also like to thank the authors of KLEE for making their work open-source so that others can benefit from their research.

Contents

List of Tables **xiii**

List of Figures **xv**

1 Introduction **1**

2 Algorithm **5**

2.1 Symbolic Execution without Abstraction 5

2.2 Symbolic Execution with Abstraction 6

2.2.1 Preprocessing 8

2.2.2 Abstraction 9

2.2.3 Backtracking 11

2.2.4 Refinement 11

2.3 Example 12

2.3.1 Preprocessing 12

2.3.2 Execution 12

3 Implementation and Experiments **19**

4 Related Work **23**

4.1 Symbolic Execution 23

4.2 Path Summarization 23

4.3 Path Merging 24

4.4 Abstract Symbolic Execution 24

4.5 Other Similar Techniques 25

5 Conclusion **27**

5.1 Scope for further work 27

References **29**

Appendices **33**

A Appendix **35**

A.1 factor.c 35

A.2 adpcm.c 38

List of Tables

3.1 Experimental results on testme.c 20

3.2 Experimental results on factor.c 20

3.3 Experimental results on adpcm.c 21

List of Figures

2.1 Algorithm: Symbolic Execution with Function Abstraction 7

2.2 Algorithm: Check if a function can be abstracted 10

2.3 Algorithm: Perform abstraction of function f for state s 10

2.4 Backtrack on encountering error in abstract state s 11

2.5 Test program: testme.c 13

2.6 Abstract functions 14

2.7 Program state on line 54 for choice = 1 with abstraction 15

2.8 Program state on line 54 for choice = 2 with abstraction 15

2.9 Program state on line 54 for choice = 2 after refinement 15

2.10 Program state on one of the program paths on line 54 for $\neg(choice = 1) \wedge \neg(choice = 2)$. False positive. 16

2.11 Program state on one of the program paths on line 54 for $\neg(choice = 1) \wedge \neg(choice = 2)$. No error. 17

2.12 Program state on one of the program paths on line 54 for $\neg(choice = 1) \wedge \neg(choice = 2)$. No error. 17

Chapter 1

Introduction

Automated program analysis aims to verify the correctness or safety of programs and generate tests which provide high coverage of the program's different paths and uncover bugs. Generation of such tests has traditionally been done via manual or random methods which do not perform well and do not scale to large programs. Symbolic execution has emerged as a popular and effective technique in the last decade to explore program paths and discover bugs in software. Symbolic execution tools have been used to find crashes in popular and well-tested programs in both academia and industry [1] [2] [3] [4]. The technique has also been adopted by the information security community to automatically find vulnerabilities in real world software and generate exploits for them [5] [6].

Symbolic execution consists of running the target program on *symbolic* values rather than concrete inputs. These symbolic values are initially unconstrained. The execution engine performs operations on these symbolic values as the program progresses. When a branch instruction is encountered whose condition depends on symbolic values, the program state is forked into two with each state maintaining the constraints on the symbolic inputs encountered on its path. On path termination due to an error or program exit, the *path constraints* are used to generate concrete inputs using a Satisfiability Modulo Theories (SMT) solver [7]. The program will follow this path when run concretely with the generated inputs. If the program ended in an error, we get a concrete value on which the error can be reproduced.

On the other hand, if the program exited normally, we are assured that the class of inputs which satisfies the path constraints is safe.

Unlike many static analysis techniques, symbolic execution does not produce any false positives if there are no abstractions. Another advantage of using symbolic execution is that it exhibits continuous progress, as partial results are also useful. However, it suffers from the problem of *path explosion*; the number of paths to be analyzed increases exponentially with increase in the number of branch instructions in the program.

In this thesis, we propose an algorithm to restrict the number of paths spawned by symbolic execution. We do this by over-approximating the behaviour of some functions in the program and avoiding their execution. To do this, we first identify functions which do not have a direct bearing on the program property we are interested in. For example, if we are analyzing the program for heap memory access violations, we can afford some imprecision in functions that do not directly or indirectly affect accesses to the heap. We can also be assured that no errors will be encountered within these functions, so we will not miss any bugs by skipping their execution. These functions are abstracted, *i.e.*, the memory locations that they could potentially modify and their return values are treated as unconstrained. If such an abstract path terminates without an error, we save execution of a large number of states that would have been created by branching in the abstract function and its callees. If, however, an error is encountered, we fall back to complete execution of the function and try to generate a concrete input to verify that the error was not a false positive.

We evaluate our algorithm on a set of test programs. The preliminary experimental results show that our technique is indeed helpful in keeping path explosion under check.

Summary

The major contributions of this thesis are:

- We propose an algorithm for abstraction of function calls during symbolic execution that relies solely on aliasing information. Further, we describe how to refine the abstraction in case of errors to get concrete inputs and to retain precision in our analysis.
- We have developed a tool AF-KLEE (Abstract Functions-KLEE) that implements our algorithm. This tool is an extension of KLEE, a symbolic virtual machine built on top of the LLVM compiler infrastructure.
- We evaluate our tool on several test programs and compare its performance with KLEE to show its effectiveness.

Chapter 2

Algorithm

We first describe the standard symbolic execution algorithm here before specifying our modifications to it in Section 2.2.

2.1 Symbolic Execution without Abstraction

The pseudocode for symbolic execution is shown in Algorithm 2.1, except the lines 10-15 and 24-26 (marked with *) which have been added for function abstraction and refinement respectively. The symbolic execution engine maintains a set of active states $\mathcal{S}$ for execution. Each state has its corresponding path conditions and symbolic store. It is initialized with the program's entry point and a blank symbolic store (line 4). A search strategy, such as breadth first search, depth first search or generational search [8] picks the next state for execution (line 6) . The execution engine updates the symbolic store after symbolically executing each instruction. For example, if x is a symbolic variable represented by X in the symbolic store and the instruction $y = x + 2$ is executed, the relation $Y = X + 2$ is added to the symbolic store and the updated state is inserted in the set of active states (lines 33-34).

On a branch instruction involving a symbolic variable, the state is forked into two new states, each carrying the conditions for its path (lines 17-20). As an example, consider that state s with path conditions Π encounters the instruction

$if(y > 0)$

This will create states s_t and s_f corresponding to the true and false branches respectively. Their path constraints will be
$\Pi_t = \Pi \wedge (X + 2 > 0)$ and $\Pi_f = \Pi \wedge \neg(X + 2 > 0)$
Depending on the implementation, an SMT solver will be used at this point to determine if both the conditions are satisfiable. If it is not the case, the state with the unsatisfiable conditions is removed.

On path termination due to an error (lines 21-23) or program exit (lines 30-31), the state will be removed and the solver will be invoked to create concrete inputs which satisfy the path constraints.
The drawback of this approach is that the number of states increases exponentially with the number of branches. Loops with symbolic constraints on termination, recursive functions and multiple sequential conditional statements will result in a state-space explosion.

In concolic testing, or dynamic symbolic execution, the execution engine also keeps track of concrete values and the runtime memory layout of the program. The initial state is seeded with random concrete values of the symbolic variables. The advantage of having the concrete values is that when an instruction is encountered for which symbolic constraints can't be evaluated, the concrete values can be used instead and execution can continue. This can happen when an operation is not supported by the SMT theory being used (for example, multiplication when the solver only supports linear arithmetic) or an external function call whose definition is not available.

2.2 Symbolic Execution with Abstraction

On a high level, our algorithm for symbolic execution with function abstraction has the following parts.

- **Preprocessing:** instrument the program to insert abstract versions of functions and other metadata that will aid our analysis

Figure 2.1: Algorithm: Symbolic Execution with Function Abstraction

1: **Data**:
2:
- Program location s: Stores the program counter. We augment it with additional information including:
  - abstract: whether some function on the path has been abstracted
  - lastConcreteState: pointer to last concrete state on the path
  - encounteredError: whether an abstract state descending from this state encountered an error
- Symbolic store Γ: Stores symbolic variables and associated formulae
- Path predicate Π for the current path
- Set of active states $\mathcal{S}$. Each element in this set is a 3-tuple of program location, path predicate and symbolic store.

3: **Input**: Program entry point s_0.
4: $\mathcal{S} \leftarrow \{(s_0, \top, \emptyset)\}$ ▷ Initial state
5: **while** $\mathcal{S} \neq \emptyset$ and !timeout **do**
6: $(s, \Pi, \Gamma) \leftarrow$selectState($\mathcal{S}$)
7: $\mathcal{S} \leftarrow \mathcal{S} - \{(s, \Pi, \Gamma)\}$
8: $inst \leftarrow s$.programCounter
9: *// Carry out abstraction at runtime if this is a call instruction*
10: **if** $inst$.type == Call **then** *
11: $f \leftarrow inst$.calledFunction *
12: **if isAbstractable**(s, f) **then** *
13: **Abstract**$((s, \Pi, \Gamma), f)$ *
14: **end if** *
15: **end if** *
16: **switch** $inst$.type **do**
17: **case** ConditionalBranchInst
18: $e \leftarrow inst$.condition
19: $(s', \Pi \wedge e, \Gamma), (s'', \Pi \wedge \neg e, \Gamma) \leftarrow$concolicExecute($inst$) ▷ Fork the state
20: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(s', \Pi \wedge e, \Gamma), (s'', \Pi \wedge \neg e, \Gamma)\}$
21: **case** Assertion
22: **if** isSat$(\Pi \wedge \neg inst.condition)$ **then**
23: reportBug(s)
24: **if** $s.abstract$ **then** *
25: **BacktrackAndRefine**$(s, \mathcal{S})$ *
26: **end if** *
27: **else**
28: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(s.next, \Pi, \Gamma)\}$
29: **end if**
30: **case** Halt
31: continue
32: **case** Default
33: $(s', \Pi', \Gamma') \leftarrow$concolicExecute($inst$)
34: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(s', \Pi', \Gamma')\}$
35: **end switch**
36: **end while**

- **Abstraction:** selectively call abstract functions at runtime, after preserving the concrete state of the program
- **Symbolic Execution:** explore the concrete and abstract states for potential errors as in standard dynamic symbolic execution
- **Backtracking:** discard the abstract states and activate the corresponding saved concrete state in case a bug is encountered in an abstract state
- **Refinement:** Perform symbolic execution from the concrete state to verify that the bug exists

2.2.1 Preprocessing

Before proceeding with abstraction based symbolic execution, there is a requirement of suitable summaries for the functions in the program. These can be obtained in several ways including:

- Using an automatic summarization tool such as FunFrog [9]
- Performing pointer alias analysis to get over-approximate function behavior [10]
- Manual annotations

In our implementation, we use manual annotations along with aliasing information to generate abstractions. We determine the mod-ref behavior of the function and treat the modified memory locations as unconstrained. This is done in a series of LLVM passes mentioned below. Some of these steps are specific to dynamic memory allocation/deallocation related analyses but can be easily adapted to verify other properties.

1. Instrument the program to wrap all calls to memory management functions (such as free) to also modify a special global variable called the *error bit*.

2. Perform alias analysis to identify functions which directly modify the *error bit*. Then perform a bottom up traversal of the program's call graph to propagate this information to caller functions. This is done using the $GlobalsModRef$ pass in LLVM. [11]

3. We disable abstractions for every function which modifies the *error bit*. Section 2.3 shows examples of abstract functions.

2.2.2 Abstraction

During symbolic execution of the program, we do the following when we encounter a function call (Algorithm 2.1, lines 10-15).

1. We check if the function can be abstracted (Algorithm 2.2). If the call cannot modify *error bit* and an abstract version of the function is available in the module, then with a probability p decided by the user, we call the abstract function instead. All future calls on this path to the same function are also abstracted. By randomizing the abstraction process, we ensure that

    - We do not get very weak conditions on path termination due to too many abstractions. This may lead to a large number of false positives, thus limiting the efficacy of the lighweight abstract executions.

    - We get some coverage of the original function and can check for bugs in it. This may not be true if we are only looking for a specific class of bugs and the abstraction guarantees that the function doesn't have such bugs.

2. If this is the first abstraction to be done along the current path, we make a copy of the program state at this point. This will be referred to as the *lastConcreteState*. (Algorithm 2.3)

Figure 2.2: Algorithm: Check if a function can be abstracted

```
1: function isAbstractable(s, f)
2:     if s.encounteredError then return false
3:     if modifiesErrorBit(f) then return false
                                          ▷ using points-to information
4:     if !abstractVersionPresent(f) then return false
5:     for v in modifiedValues(f) do
6:         // Do not abstract if a modified variable is used for dereferencing, either
       as a pointer or an array or struct offset
7:         if usedForDereference(v) then return false
8:     end for
9:     if rand() % N then return true          ▷ randomize abstraction process
10:    else return false
11: end function
```

Figure 2.3: Algorithm: Perform abstraction of function f for state s

```
1: procedure Abstract((s, Π, Γ), f)
2:     if ! s.abstract then
3:         (s', Π', Γ') ←copy(s, Π, Γ)
4:         s.lastConcreteState ←(s', Π', Γ')
5:         s.abstract ←true
6:     // Replace all calls to f by its symbolic version for s and all its descendants
7:     alias(s, f, abstractVersion(f))
8: end procedure
```

Figure 2.4: Backtrack on encountering error in abstract state s

```
1: procedure BacktrackAndRefine(s, S))
2:     concState ←s.lastConcreteState
3:     concState.encounteredError ←true
4:     S ←S ∪ {concState}
5:     S ←S − {p ∈ S : p.abstract and p.lastConcreteState = concState}
6: end procedure
```

2.2.3 Backtracking

When an abstract state ends in an error (Algorithm 2.1, Lines 24-26), we do the following (Algorithm 2.4).

1. Add *lastConcreteState* corresponding to the abstract state to the list of active states in the symbolic execution engine. We disable all abstractions on *lastConcreteState*'s path.

2. Since the states emanating from *lastConcreteState* will now be executed concretely, we remove all abstract states which descended from it.

2.2.4 Refinement

Finally, to generate a concrete input and to retain precision, we perform standard concolic testing on the *lastConcreteState* made active after the error on the abstract state. If the error state is found to be reachable, we generate a concrete test case and report the error. It may also happen that the error state is never reached. This can be because of the following reasons

- The reported error was a false positive, in which case the refinement has been useful.

- Concolic testing failed to uncover the error. Since the number of paths originating from *lastConcreteState* is, in general, not bounded, it is to be expected that dynamic symbolic execution may fail to find the path that ends in an error. In this case, we do no worse than the vanilla concolic testing strategy.

The user can still inspect the error report from the abstract state and manually check the feasibility of the error path.

Section 2.3 shows examples of both these cases.

2.3 Example

We will now illustrate the working of our algorithm by describing how the analysis of Code 2.5 will proceed. The program has a double free bug on the code path (…44, 45, …, 46, …, 47, …, 54). The *main* function has a symbolic *choice* variable which decides whether this bug is triggered. There is another potential double free on path (…49, 50, 51, 32, 33, 52, 54). However, this path has a control flow dependency on the global variable x and is not reachable. The accompanying figures represent abstract functions with purple color and concrete calls with green color. Only relevant variables have been included in the symbolic and concrete stores for brevity.

2.3.1 Preprocessing

The preprocessing step adds the abstract functions in Code 2.6 to the program. The functions *conditional_free* and *unconditional_free* are not abstracted because they modify the *error bit*. Note that the concrete functions are not removed and all call sites are still unaltered.

2.3.2 Execution

On line 40 the symbolic store contains $\{choice, n\}$ and the path predicate is *true*. It is then split into two states, with conditions $choice = 1$ and $\neg(choice = 1)$ respectively. The second branch is further split into $\neg(choice = 1) \wedge (choice = 2)$ and $\neg(choice = 1) \wedge \neg(choice = 2)$. We will analyze these three paths separately.

1. **choice = 1**

   Symbolic Execution: In this case the execution will enter the factorial

Figure 2.5: Test program: testme.c

```
1  int x, y, z;
2  unsigned factorial (unsigned n)
3  {
4      z = 2;
5      unsigned factorial = 1;
6      for (unsigned i = 1; i <= n; i++) {
7          factorial *= i;
8      }
9      return factorial;
10 }
11 unsigned bitsum (unsigned n)
12 {
13     y = 1;
14     unsigned sum = 0;
15     while (n) {
16         sum += n & 1;
17         n >>= 1;
18     }
19     return sum;
20 }
21 unsigned change_x ()
22 {
23     x += y;
24     return 0;
25 }
26 void unconditional_free(unsigned *p)
27 {
28     free(p); // will lead to double free on line 54
29 }
30 void conditional_free(unsigned *p)
31 {
32     if (x > 5)
33         free(p); // will lead to double free on line 54
34 }
35 int main(int argc, char **argv)
36 {
37     unsigned choice, n;
38     make_symbolic(choice);
39     unsigned *p = (unsigned *)malloc(sizeof(unsigned));
40     make_symbolic(n);
41     if (choice == 1) { // no error
42         factorial(n);
43     }
44     else if (choice == 2) { // error
45         bitsum(n);
46         change_x();
47         unconditional_free(p);
48     }
49     else { // false positive if change_x() is abstracted
50         factorial(n);
51         change_x();
52         conditional_free(p);
53     }
54     free(p);
55     return 0;
56 }
```

Figure 2.6: Abstract functions

```
1  unsigned _abstract_factorial (unsigned n)
2  {
3      int ret_fact;
4      make_symbolic(ret_fact);
5      make_symbolic(z);
6      return ret_fact;
7  }
8  unsigned _abstract_bitsum (unsigned n)
9  {
10     int ret_bitsum;
11     make_symbolic(ret_bitsum);
12     make_symbolic(y);
13     return ret_bitsum;
14 }
15 unsigned _abstract_change_x ()
16 {
17     int ret_x;
18     make_symbolic(ret_x);
19     make_symbolic(x);
20     return ret_x;
21 }
```

function with an unconstrained value of n. As a result, the execution engine will spawn an unbounded number of states and symbolic execution will never terminate.

Abstract Symbolic Execution: Here, the call to *factorial* can be abstracted. It is replaced by *_abstract_factorial* which marks z and its return value as unconstrained. The program then calls *free* on p (line 54) and ends without an error (Fig 2.7). Thus we are able to verify that all the paths corresponding to the branch $choice = 1$ are free from double free errors by performing only one abstract execution.

2. **choice = 2**

   Symbolic Execution: The execution will generate 33 states within the *bitsum* function, one for each loop iteration. All these states will end in a double free error on line 54 which will be correctly reported.

   Abstract Symbolic Execution: In this case, the function *bitsum* may be abstracted. In that case, there will be no forks and we will hit an error on line 54 (Fig 2.8). the program will then go back to line 45 and execute the concrete

Figure 2.7: Program state on line 54 for choice = 1 with abstraction

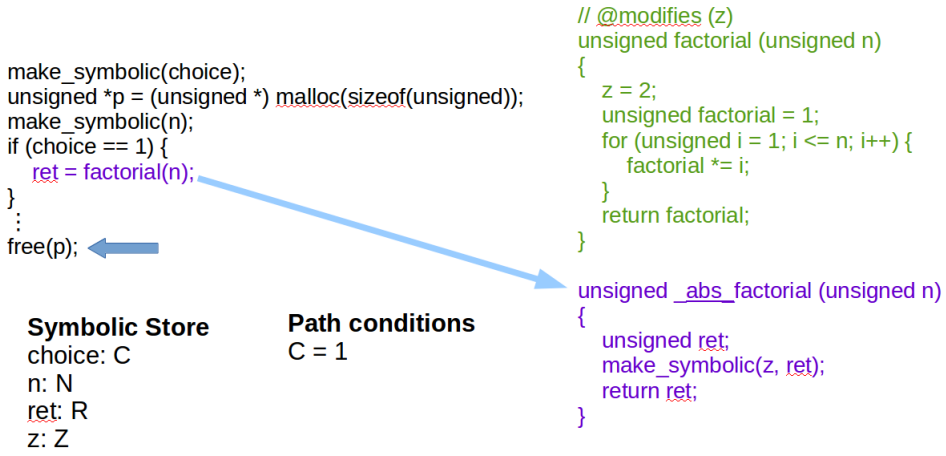
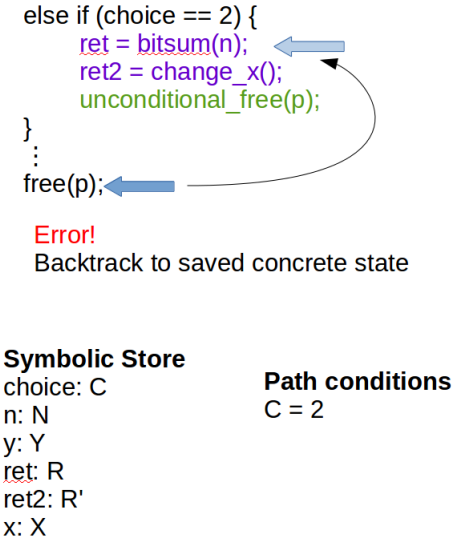


Figure 2.8: Program state on line 54 for choice = 2 with abstraction



function *bitsum*. The execution will then continue as in the previous case and verify that the error was not a false positive (Fig 2.9).

3. $\neg(\textbf{choice} = \textbf{1}) \wedge \neg(\textbf{choice} = \textbf{2})$

   Symbolic Execution: Similar to the case when *choice* = 1, this case will spawn an infinite number of states within *factorial*. The execution will not stop. However, none of the states will report a false positive.

   Abstract Symbolic Execution: In this case, the behaviour depends significantly on which functions are abstracted.

Figure 2.9: Program state on line 54 for choice = 2 after refinement

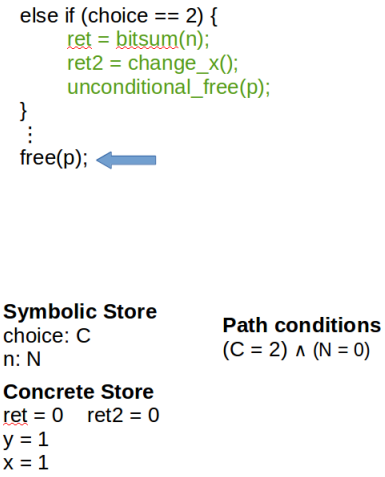
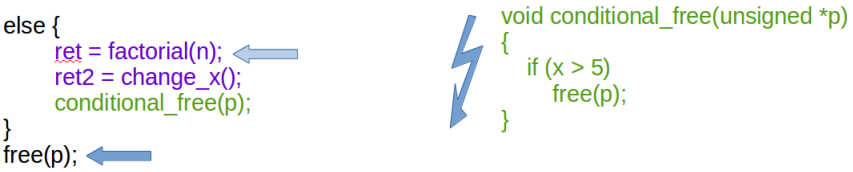


Figure 2.10: Program state on one of the program paths on line 54 for $\neg(choice = 1) \wedge \neg(choice = 2)$. False positive.



- If both *factorial* and *change_x* are abstracted, a false error report will be generated. The concrete execution will start from line 50 and will spawn an infinite number of states. Thus, we will not be able to verify or discard the abstract bug report. (Fig 2.10)

- If *factorial* is abstracted but *change_x* is not, we verify that there are no double free errors by executing only one abstract path (Fig 2.11).

- If *factorial* is not abstracted but *change_x* is, there will be a false error report (Fig 2.12). Concrete execution will start from line 51 and classify the abstract error report as a false positive.

Figure 2.11: Program state on one of the program paths on line 54 for $\neg(choice = 1) \wedge \neg(choice = 2)$. No error.

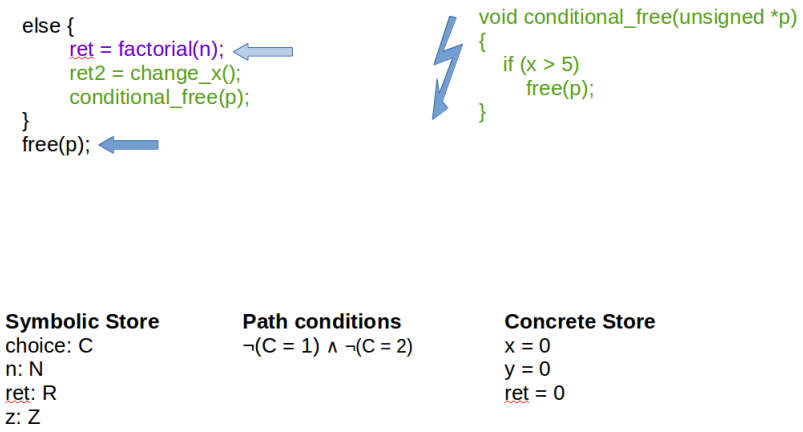
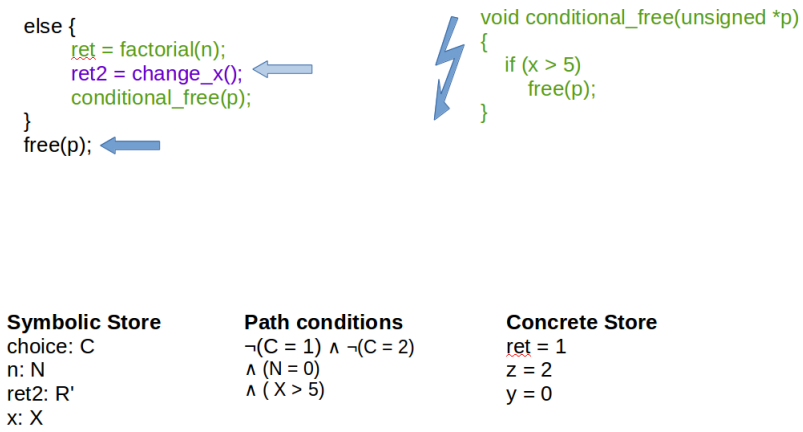


Figure 2.12: Program state on one of the program paths on line 54 for $\neg(choice = 1) \wedge \neg(choice = 2)$. No error.



This case shows how randomization is helpful in the abstraction process. Abstracting only one of the functions gives a better result than abstracting both of them. In general, a high degree of abstraction can lead to weak conditions causing multiple false positives.

Summary

In this chapter, we described an algorithm to perform abstraction of function calls. This reduces the number of paths to be explored by symbolic execution significantly in most cases. The number of paths we save can even be infinite e.g. in case of a loop where the number of iterations is dependent on a symbolic value. We also described how we retain precision and avoid reporting false positives by backtracking to the concrete state.

Chapter 3

Implementation and Experiments

We have developed AF-KLEE, a tool that implements our algorithm for symbolic execution with function abstraction within KLEE [2], a symbolic execution engine, which in turn is built on top of the LLVM compiler infrastructure [12].

The instrumentation and alias analysis of the program being tested is done by AF-KLEE in a series of LLVM passes which leverage the information provided by the core LLVM infrastructure. It uses the following additional LLVM passes:

- Data Layout Pass, to get an accurate representation of the memory layout of the program. This improves the precision of alias analysis passes.
- Globals Mod-Ref Alias Analysis, to identify which functions modify the error bit, described in Section 2.2.1.
- Argument Promotion Pass, to change upto 10 pass-by-reference arguments for every function to pass-by-value if their value is not changed by the function. This simplifies reasoning about the function's mod-ref behaviour. We do not abstract a function if it retains any pointer arguments after this pass.

Abstractions of functions can also be provided manually by adding them to the program source.

Another small but important implementation detail is that we disable all function inlining. Function inlining is disadvantageous for AF-KLEE as inlined functions can't be abstracted.

Table 3.1: Experimental results on testme.c

	Time	**Paths explored**	**Solver queries**
KLEE	Time Out (5 min)	3747	2093
AF-KLEE	0.7 sec	36 (3 abstract)	38

Table 3.2: Experimental results on factor.c

	Time	**Paths explored**	**Solver queries**
KLEE	Time Out (1 hour)	14974	37084
AF-KLEE	51 min	44711 (44590 abstract)	1279

We compared AF-KLEE with KLEE on the following test programs. The experiments were run on an Intel Core i7 (4770, 3.4 Gz) machine with 16GB RAM running Ubuntu 14.04.

testme.c

This is the code sample shown in Code 2.5. We ran it with a timeout of 5 minutes. KLEE does not terminate within the given time limit, which is expected because the number of states to be explored is not bounded. The results are shown in Table 3.1. AF-KLEE was able to detect the double-free bug and verify the absence of bugs on other paths by executing only 36 paths.

factor.c

This program performs integer factorization. It uses a linked list for storing the factors and makes use of the Wheel factorization algorithm [13] to find the prime factors. This is a simplified version of the *factor* utility which is a part of GNU Coreutils [14]. The source code of the modified version is given in Appendix A.1. We ran both the tools on this program with a timeout of 1 hour. As shown in the results in Table 3.2, AF-KLEE finishes execution on the program in 51 minutes and thus verifies that it does not contain heap memory access violations.

Table 3.3: Experimental results on adpcm.c

	Time	**Paths explored**	**Solver queries**
KLEE	Time Out (1 hour)	301	748
AF-KLEE	Time Out (1 hour)	323640 (all abstract)	222

adpcm.c

This program is a slightly modified version of the *adpcm* code from the WCET benchmarks [15]. The original program has been changed to make use of dynamic memory. The modified code is given in Appendix A.2. The results in Table 3.3 show that AF-KLEE covers orders of magnitude more paths than KLEE, though all of them are abstract. This is because complex functions like *sin* and *cos* were abstracted.

Chapter 4

Related Work

4.1 Symbolic Execution

The basic ideas behind symbolic execution were developed in the 1970s by Boyer *et al* [16] and King [17]. The technique has attracted significant attention in the past decade due to the advent of efficient SMT solvers which make symbolic execution feasible in practice. KLEE [2] was among the first symbolic execution engines that scaled to real world code and demonstrated the strength of symbolic execution by finding multiple bugs in the well-tested GNU Coreutils. SAGE [8] [1] has been used to find a large number of bugs in Microsoft's software products. Other successful symbolic execution engines include CUTE [18], Cloud9 [19] and S2E [20].

4.2 Path Summarization

Godefroid *et al* summarize functions as first order logic formulae during execution and avoid repetitive executions for inputs that satisfy the same path conditions [21]. They have extended their work to do the summarization and composition on a demand-driven basis [22].

Khurshid *et al* provide alternate abstract representations for the state of some C++ STL containers [23] to aid symbolic execution. This technique does not generalize to arbitrary functions.

Other works [24] [25] automatically generate function summaries and can be used to complement our technique. However, these techniques require that the behaviour of the function be completely known, which may not be the case with real world software that makes use of external library functions and system calls.

Li *et al* perform lazy symbolic execution with abstraction and sub-space search [26]. They define a declarative language to provide summaries for functions as logic formulae. These summaries are used for symbolic execution. On every path termination, functions are expanded to get a trace which satisfies the overall path conditions. We use a similar strategy of lazy execution. However, our approach aims to quickly verify interprocedural paths for correctness. We do not check for feasibility if no errors are encountered. Our approach can also be scaled to larger programs with more precise alias analysis eliminating the need for manual annotations.

Lal *et al* perform variable abstraction to get over-approximate program behaviour in their reachability solver, Corral followed by hierarchical refinement [27]. We take a similar approach for abstraction and refinement in our algorithm.

4.3 Path Merging

Avgerinos *et al* [3] alternate between static and dynamic symbolic execution to get the benefits of both techniques with impressive results. They identify program parts which can be analysed effectively with static symbolic execution and statically generate path conditions which encompass multiple concrete paths. They then merge these paths and continue with dynamic symbolic execution.

Kuznetsov *et al* also propose a merging strategy to combine several paths during dynamic symbolic execution [28]. These techniques are orthogonal to our algorithm.

4.4 Abstract Symbolic Execution

Anand *et al* [29] perform shape analysis of data structures such as linked lists and arrays and avoid execution of a state which is subsumed by another as defined by

their abstraction.

Albarghouthi *et al* [30] use predicate abstraction to check for state subsumption in concurrent programs.

Rungta *et al* [31] perform abstraction in the form of backward program slicing to perform guided symbolic execution in concurrent programs.

4.5 Other Similar Techniques

Livshits *et al* perform alias analysis to find abstractions of C functions which are then used to find security property violations in a static manner [10]. Their technique may report false positives due to imprecision in the aliasing information and may miss some errors due to unsound assumptions. We use a similar technique for abstraction but our points-to analysis is sound. Additionally, we perform concolic execution to verify the bugs.

Under-constrained symbolic execution [32] [33] removes all preconditions on function parameters to perform unit testing. It has been used for regression testing by comparing two versions of the same program and checking for crashes which occur in only one of them. This technique, however, suffers from false positives.

Chapter 5

Conclusion

In this thesis, we presented function abstraction as a technique for addressing the scalability issues of symbolic execution. Our proposed algorithm is capable of analyzing a large number of potential program paths in a single execution. The preliminary results obtained by running our tool, AF-KLEE on simple test programs are promising. This leads us to believe that function abstraction can be used as a viable method for aiding symbolic execution analyses, especially when looking for a specific class of bugs in programs where functional correctness is not a concern. This is often the case with security-related program analysis.

5.1 Scope for further work

AF-KLEE is still a proof of concept and there is scope for improvement. Our aim is to extend it so that it can scale to large programs with little manual intervention and handle a larger class of bugs.

The abstraction process can be made completely automated with more precise alias analysis information. It can also be integrated with external summarization tools which use static analysis techniques like model checking and abstract interpretation to generate more precise summaries.

Our tool can also be extended to look for other classes of memory corruption bugs and not just heap memory allocation / deallocation related issues. Our algorithm and implementation are generic enough to support such extensions.

Changes can also be made to the post-refinement concrete execution to make it more targeted so that it can create concrete crash-inducing inputs in a shorter time.

References

[1] Ella Bounimova, Patrice Godefroid, and David Molnar. "Billions and billions of constraints: Whitebox fuzz testing in production". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 122–131.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing symbolic execution with veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 1083–1094.

[4] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. "EXE: automatically generating inputs of death". In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), p. 10.

[5] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. "Towards automatic generation of vulnerability-based signatures". In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE. 2006, 15–pp.

[6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. "Automatic Exploit Generation". In: *Commun. ACM* 57.2 (Feb. 2014), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/2560217.2560219. URL: http://doi.acm.org/10.1145/2560217.2560219.

[7] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories." In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[8] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[9] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. "FunFrog: bounded model checking with interpolation-based function summarization". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2012, pp. 203–207.

[10] V. Benjamin Livshits and Monica S. Lam. "Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs". In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-11. Helsinki, Finland: ACM, 2003, pp. 317–326. ISBN: 1-58113-743-5. DOI: 10.1145/940071.940114. URL: http://doi.acm.org/10.1145/940071.940114.

[11] *LLVM Alias Analysis Infrastructure*. http://llvm.org/docs/AliasAnalysis.html.

[12] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[13] *Wheel Factorization*. http://primes.utm.edu/glossary/page.php?sort=WheelFactorization.

[14] *GNU Coreutils*. https://www.gnu.org/software/coreutils/.

[15] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. "The Mälardalen WCET benchmarks: Past, present and future". In: *OASIcs-OpenAccess Series in Informatics*. Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[16] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT&Mdash;a Formal System for Testing and Debugging Programs by Symbolic Execution". In: *Proceedings of the International Conference on Reliable Software*. Los Angeles, California: ACM, 1975, pp. 234–245. DOI: 10.1145/800027.808445. URL: http://doi.acm.org/10.1145/800027.808445.

[17] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: http://doi.acm.org/10.1145/360248.360252.

[18] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 263–272.

[19] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. "Cloud9: a software testing service". In: *ACM SIGOPS Operating Systems Review* 43.4 (2010), pp. 5–10.

[20] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems". In: *SIGPLAN Not.* 47.4 (Mar. 2011), pp. 265–278. ISSN: 0362-1340. DOI: 10.1145/2248487.1950396. URL: http://doi.acm.org/10.1145/2248487.1950396.

[21] Patrice Godefroid. "Compositional Dynamic Test Generation". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: ACM, 2007, pp. 47–54. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190226. URL: http://doi.acm.org/10.1145/1190216.1190226.

[22] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. "Demand-Driven Compositional Symbolic Execution". In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_28. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_28.

[23] Sarfraz Khurshid and Yuk Lai Suen. “Generalizing symbolic execution to library classes”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 31. 1. ACM. 2005, pp. 103–110.

[24] Denis Gopan and Thomas Reps. “Low-level library analysis and summarization”. In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 68–81.

[25] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. “Automatic predicate abstraction of C programs”. In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 203–213.

[26] Guodong Li and Indradeep Ghosh. “Lazy Symbolic Execution through Abstraction and Sub-space Search”. In: *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*. Ed. by Valeria Bertacco and Axel Legay. Cham: Springer International Publishing, 2013, pp. 295–310. ISBN: 978-3-319-03077-7. DOI: 10.1007/978-3-319-03077-7_20. URL: http://dx.doi.org/10.1007/978-3-319-03077-7_20.

[27] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. “A Solver for Reachability Modulo Theories”. In: *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 427–443. ISBN: 978-3-642-31424-7. DOI: 10.1007/978-3-642-31424-7_32. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_32.

[28] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. “Efficient state merging in symbolic execution”. In: *Acm Sigplan Notices* 47.6 (2012), pp. 193–204.

[29] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. “Symbolic execution with abstraction”. In: *International Journal on Software Tools for Technology Transfer* 11.1 (2009), pp. 53–67. ISSN: 1433-2787. DOI: 10.1007/s10009-008-0090-1. URL: http://dx.doi.org/10.1007/s10009-008-0090-1.

[30] Aws Albarghouthi, Arie Gurfinkel, Ou Wei, and Marsha Chechik. “Abstract Analysis of Symbolic Executions”. In: *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 495–510. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_43. URL: http://dx.doi.org/10.1007/978-3-642-14295-6_43.

[31] Neha Rungta, Eric G. Mercer, and Willem Visser. “Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution”. In: *Model Checking Software: 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*. Ed. by Corina S. Păsăreanu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–191. ISBN: 978-3-642-02652-2. DOI: 10.1007/978-3-642-02652-2_16. URL: http://dx.doi.org/10.1007/978-3-642-02652-2_16.

[32] Dawson Engler and Daniel Dunbar. “Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA ’07. London, United Kingdom: ACM, 2007, pp. 1–4. ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273464. URL: http://doi.acm.org/10.1145/1273463.1273464.

[33] David A Ramos and Dawson Engler. “Under-constrained symbolic execution: correctness checking for real code”. In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 49–64.

Appendices

Appendix A

Appendix

A.1 factor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#include "klee/klee.h"

#define MAX_N_FACTORS (sizeof (uintmax_t) * 8)

unsigned char wheel_tab[] = {1,2,2,4,2,4,2,4,6,2,6,4,2,4,6,6,2,6,4,2,6,4,6,8,4,
    2,4,2,4,14,4,6,2,10,2,6,6,4,2,4,6,2,10,2,4,2,12,10,2,4,2,4,6,2,6,4,6,6,6,2,
    6,4,2,6,4,6,8,4,2,4,6,8,6,10,2,4,6,2,6,6,4,2,4,6,2,6,4,2,6,10,2,10,2,4,2,4,
    6,8,4,2,4,12,2,6,4,2,6,4,6,12,2,4,2,4,8,6,4,6,2,4,6,2,6,10,2,4,6,2,6,4,2,4,
    2,10,2,10,2,4,6,6,2,6,6,4,6,6,2,6,4,2,6,4,6,8,4,2,6,4,8,6,4,6,2,4,6,8,6,4,
    2,10,2,6,4,2,4,2,10,2,10,2,4,2,4,8,6,4,2,4,6,6,2,6,4,8,4,6,8,4,2,4,2,4,8,6,
    4,6,6,6,2,6,6,4,2,4,6,2,6,4,2,4,2,10,2,10,2,6,4,6,2,6,4,2,4,6,6,8,4,2,6,10,
    8,4,2,4,2,4,8,10,6,2,4,8,6,6,4,2,4,6,2,6,4,6,2,10,2,10,2,4,2,4,6,2,6,4,2,4,
    6,6,2,6,6,6,4,6,8,4,2,4,2,4,8,6,4,8,4,6,2,6,6,4,2,4,6,8,4,2,4,2,10,2,10,2,4,
    2,4,6,2,10,2,4,6,8,6,4,2,6,4,6,8,4,6,2,4,8,6,4,6,2,4,6,2,6,6,4,6,6,2,6,6,4,
    2,10,2,10,2,4,2,4,6,2,6,4,2,10,6,2,6,4,2,6,4,6,8,4,2,4,2,12,6,4,6,2,4,6,2,
    12,4,2,4,8,6,4,2,4,2,10,2,10,6,2,4,6,2,6,4,2,4,6,6,2,6,4,2,10,6,8,6,4,2,4,8,
    6,4,6,2,4,6,2,6,6,6,4,6,2,6,4,2,4,2,10,12,2,4,2,10,2,6,4,2,4,6,6,2,10,2,6,4,
```

```
22      14,4,2,4,2,4,8,6,4,6,2,4,6,2,6,6,4,2,4,6,2,6,4,2,4,12,2,12};
23
24  #define WHEEL_SIZE 5
25  #define WHEEL_START (wheel_tab + WHEEL_SIZE)
26  #define WHEEL_END (wheel_tab + (sizeof wheel_tab / sizeof wheel_tab[0]))
27
28  unsigned char *w = wheel_tab;
29
30  struct factor {
31      uint64_t val;
32      struct factor *next;
33  };
34
35  uint64_t next_factor(uint64_t *nptr, uint64_t *dptr)
36  {
37      uint64_t q;
38      do {
39          q  = *nptr / *dptr;
40          if (*nptr == q * *dptr)
41          {
42              *nptr = q;
43              return *dptr;
44          }
45          *dptr += *(w++);
46          if (w == WHEEL_END)
47              w = WHEEL_START;
48      } while (*dptr <= q);
49      return 0;
50  }
51
52  uint64_t _symbolic_next_factor(uint64_t *nptr, uint64_t *dptr)
53  {
54      uint64_t ret;
55      klee_make_symbolic(&ret, sizeof(uint64_t), "ret");
56      return ret;
```

```
57 }
58
59 void add(uint64_t f, struct factor **factors)
60 {
61     if (!*factors) {
62         *factors = (struct factor *) malloc(sizeof(struct factor));
63         (*factors)->val = f;
64         (*factors)->next = NULL;
65         return;
66     }
67     struct factor *n, *node = *factors;
68     while (node->next != NULL) {
69         node = node->next;
70     }
71     n = (struct factor *) malloc(sizeof(struct factor));
72     n->val = f;
73     n->next = NULL;
74     node->next = n;
75 }
76
77 void factorize(uint64_t n, struct factor **factors)
78 {
79     if (n <= 1) return;
80     uint64_t i, d = 2;
81     for (i = 0; i < MAX_N_FACTORS; i++) {
82         uint64_t factor = next_factor(&n, &d);
83         if (!factor) break;
84         add(factor, factors);
85     }
86     if (n != 1) add(n, factors);
87 }
88
89 void print_factors(struct factor *factors)
90 {
91     struct factor *prev;
```

```
92      while (factors) {
93          printf("%lu␣", factors->val);
94          prev = factors;
95          factors = factors->next;
96          free(prev);
97      }
98      printf("\n");
99  }
100
101 int main(int argc, char **argv)
102 {
103     struct factor *factors = NULL;
104     if (argc != 2) {
105         fprintf(stderr, "Usage:␣%s␣number\n", argv[0]);
106     }
107     errno = 0;
108     uint64_t n = strtoull(argv[1], NULL, 10);
109     if (errno)
110         exit(1);
111     factorize(n, &factors);
112     print_factors(factors);
113 }
```

A.2 adpcm.c

```
1  /* $Id: adpcm.c,v 1.7 2005/06/15 07:27:31 ael01 Exp $ */
2  /**************************************************************************/
3  /*                                                                        */
4  /*   SNU-RT Benchmark Suite for Worst Case Timing Analysis                */
5  /*   =====================================================                */
6  /*                              Collected and Modified by S.-S. Lim       */
7  /*                                                   sslim@archi.snu.ac.kr */
8  /*                                                Real-Time Research Group */
9  /*                                               Seoul National University */
```

```
10 /*                                                                      */
11 /*                                                                      */
12 /*       < Features > - restrictions for our experimental environment   */
13 /*                                                                      */
14 /*         1. Completely structured.                                    */
15 /*              - There are no unconditional jumps.                     */
16 /*              - There are no exit from loop bodies.                   */
17 /*                (There are no 'break' or 'return' in loop bodies)     */
18 /*         2. No 'switch' statements.                                   */
19 /*         3. No 'do..while' statements.                                */
20 /*         4. Expressions are restricted.                               */
21 /*              - There are no multiple expressions joined by 'or',     */
22 /*               'and' operations.                                      */
23 /*         5. No library calls.                                         */
24 /*              - All the functions needed are implemented in the       */
25 /*                source file.                                          */
26 /*         6. Printouts removed (Jan G)                                 */
27 /*                                                                      */
28 /*                                                                      */
29 /*                                                                      */
30 /************************************************************************/
31 /*                                                                      */
32 /*  FILE: adpcm.c                                                       */
33 /*  SOURCE : C Algorithms for Real-Time DSP by P. M. Embree             */
34 /*                                                                      */
35 /*  DESCRIPTION :                                                       */
36 /*                                                                      */
37 /*    CCITT G.722 ADPCM (Adaptive Differential Pulse Code Modulation)   */
38 /*    algorithm.                                                        */
39 /*    16khz sample rate data is stored in the array test_data[SIZE].    */
40 /*    Results are stored in the array compressed[SIZE] and result[SIZE].*/
41 /*    Execution time is determined by the constant SIZE (default value  */
42 /*    is 2000).                                                         */
43 /*                                                                      */
44 /*  REMARK :                                                            */
```

```
45 /*                                                                      */
46 /*  EXECUTION TIME :                                                    */
47 /*                                                                      */
48 /*                                                                      */
49 /************************************************************************/
50
51 /* To be able to run with printouts
52 #include <stdio.h> */
53 #include <stdlib.h>
54 #include "klee/klee.h"
55
56 /* common sampling rate for sound cards on IBM/PC */
57 #define SAMPLE_RATE 11025
58
59 #define PI 3141
60 #define SIZE 3
61 #define IN_END 4
62
63 /* COMPLEX STRUCTURE */
64
65 typedef struct {
66     int real, imag;
67 } COMPLEX;
68
69 /* function prototypes for fft and filter functions */
70 void fft(COMPLEX *,int);
71 int fir_filter(int input,int *coef,int n,int *history);
72 int iir_filter(int input,int *coef,int n,int *history);
73 int gaussian(void);
74 int my_abs(int n);
75
76 void setup_codec(int),key_down(),int_enable(),int_disable();
77 int flags(int);
78
79 int getinput(void);
```

```
80  void sendout(int),flush();
81
82  int encode(int,int);
83  void decode(int);
84  int filtez(int *bpl,int *dlt);
85  void upzero(int dlt,int *dlti,int *bli);
86  int filtep(int rlt1,int al1,int rlt2,int al2);
87  int quantl(int el,int detl);
88  /* int invqxl(int il,int detl,int *code_table,int mode); */
89  int logscl(int il,int nbl);
90  int scalel(int nbl,int shift_constant);
91  int uppol2(int al1,int al2,int plt,int plt1,int plt2);
92  int uppol1(int al1,int apl2,int plt,int plt1);
93  /* int invqah(int ih,int deth); */
94  int logsch(int ih,int nbh);
95  void reset();
96  int my_fabs(int n);
97  int my_cos(int n);
98  int my_sin(int n);
99
100 /* G722 C code */
101
102 /* variables for transimit quadrature mirror filter here */
103 int tqmf[24];
104
105 /* QMF filter coefficients:
106    scaled by a factor of 4 compared to G722 CCITT recommendation */
107 int h[24] = {
108     12,   -44,   -44,   212,    48,  -624,   128,  1448,
109     -840, -3220,  3804, 15504, 15504,  3804, -3220,  -840,
110     1448,   128,  -624,    48,   212,   -44,   -44,    12
111 };
112
113 int xl,xh;
114
```

```
115 /* variables for receive quadrature mirror filter here */
116 int accumc[11],accumd[11];
117
118 /* outputs of decode() */
119 int xout1,xout2;
120
121 int xs,xd;
122
123 /* variables for encoder (hi and lo) here */
124
125 int il,szl,spl,sl,el;
126
127 int qq4_code4_table[16] = {
128     0,  -20456,  -12896,   -8968,   -6288,   -4240,   -2584,   -1200,
129     20456,   12896,    8968,    6288,    4240,    2584,    1200,       0
130 };
131
132 int qq5_code5_table[32] = {
133     -280,    -280,  -23352,  -17560,  -14120,  -11664,   -9752,   -8184,
134     -6864,   -5712,   -4696,   -3784,   -2960,   -2208,   -1520,    -880,
135     23352,   17560,   14120,   11664,    9752,    8184,    6864,    5712,
136     4696,    3784,    2960,    2208,    1520,     880,     280,    -280
137 };
138
139 int qq6_code6_table[64] = {
140     -136,    -136,    -136,    -136,  -24808,  -21904,  -19008,  -16704,
141     -14984,  -13512,  -12280,  -11192,  -10232,   -9360,   -8576,   -7856,
142     -7192,   -6576,   -6000,   -5456,   -4944,   -4464,   -4008,   -3576,
143     -3168,   -2776,   -2400,   -2032,   -1688,   -1360,   -1040,    -728,
144     24808,   21904,   19008,   16704,   14984,   13512,   12280,   11192,
145     10232,    9360,    8576,    7856,    7192,    6576,    6000,    5456,
146     4944,    4464,    4008,    3576,    3168,    2776,    2400,    2032,
147     1688,    1360,    1040,     728,     432,     136,    -432,    -136
148 };
149
```

```
150 int delay_bpl[6];
151
152 int delay_dltx[6];
153
154 int wl_code_table[16] = {
155     -60,  3042,  1198,   538,   334,   172,    58,   -30,
156     3042,  1198,   538,   334,   172,    58,   -30,   -60
157 };
158
159 int wl_table[8] = {
160     -60,   -30,    58,   172,   334,   538,  1198,  3042
161 };
162
163 int ilb_table[32] = {
164     2048,  2093,  2139,  2186,  2233,  2282,  2332,  2383,
165     2435,  2489,  2543,  2599,  2656,  2714,  2774,  2834,
166     2896,  2960,  3025,  3091,  3158,  3228,  3298,  3371,
167     3444,  3520,  3597,  3676,  3756,  3838,  3922,  4008
168 };
169
170 int         nbl;                  /* delay line */
171 int         al1,al2;
172 int         plt,plt1,plt2;
173 int         rs;
174 int         dlt;
175 int         rlt,rlt1,rlt2;
176
177 /* decision levels - pre-multiplied by 8, 0 to indicate end */
178 int decis_levl[30] = {
179     280,   576,   880,  1200,  1520,  1864,  2208,  2584,
180     2960,  3376,  3784,  4240,  4696,  5200,  5712,  6288,
181     6864,  7520,  8184,  8968,  9752, 10712, 11664, 12896,
182     14120, 15840, 17560, 20456, 23352, 32767
183 };
184
```

```
185 int         detl;
186
187 /* quantization table 31 long to make quantl look-up easier,
188    last entry is for mil=30 case when wd is max */
189 int quant26bt_pos[31] = {
190     61,    60,    59,    58,    57,    56,    55,    54,
191     53,    52,    51,    50,    49,    48,    47,    46,
192     45,    44,    43,    42,    41,    40,    39,    38,
193     37,    36,    35,    34,    33,    32,    32
194 };
195
196 /* quantization table 31 long to make quantl look-up easier,
197    last entry is for mil=30 case when wd is max */
198 int quant26bt_neg[31] = {
199     63,    62,    31,    30,    29,    28,    27,    26,
200     25,    24,    23,    22,    21,    20,    19,    18,
201     17,    16,    15,    14,    13,    12,    11,    10,
202     9,     8,     7,     6,     5,     4,     4
203 };
204
205
206 int         deth;
207 int         sh;         /* this comes from adaptive predictor */
208 int         eh;
209
210 int qq2_code2_table[4] = {
211     -7408,   -1616,   7408,  1616
212 };
213
214 int wh_code_table[4] = {
215     798,   -214,    798,   -214
216 };
217
218
219 int         dh,ih;
```

```
220 int         nbh,szh;
221 int         sph,ph,yh,rh;
222
223 int         delay_dhx[6];
224
225 int         delay_bph[6];
226
227 int         ah1,ah2;
228 int         ph1,ph2;
229 int         rh1,rh2;
230
231 /* variables for decoder here */
232 int         ilr,yl,rl;
233 int         dec_deth,dec_detl,dec_dlt;
234
235 int         dec_del_bpl[6];
236
237 int         dec_del_dltx[6];
238
239 int     dec_plt,dec_plt1,dec_plt2;
240 int     dec_szl,dec_spl,dec_sl;
241 int     dec_rlt1,dec_rlt2,dec_rlt;
242 int     dec_al1,dec_al2;
243 int     dl;
244 int     dec_nbl,dec_yh,dec_dh,dec_nbh;
245
246 /* variables used in filtez */
247 int         dec_del_bph[6];
248
249 int         dec_del_dhx[6];
250
251 int         dec_szh;
252 /* variables used in filtep */
253 int         dec_rh1,dec_rh2;
254 int         dec_ah1,dec_ah2;
```

```
255 int         dec_ph,dec_sph;
256
257 int     dec_sh,dec_rh;
258
259 int     dec_ph1,dec_ph2;
260
261 /* G722 encode function two ints in, one 8 bit output */
262
263 /* put input samples in xin1 = first value, xin2 = second value */
264 /* returns il and ih stored together */
265
266 /* MAX: 1 */
267 int my_abs(int n)
268 {
269     int m;
270
271     if (n >= 0) m = n;
272     else m = -n;
273     return m;
274 }
275
276 int _symbolic_my_abs(int n)
277 {
278     int ret;
279     klee_make_symbolic(&ret, sizeof(int), "ret");
280     return ret;
281 }
282
283 /* MAX: 1 */
284 int my_fabs(int n)
285 {
286     int f;
287
288     if (n >= 0) f = n;
289     else f = -n;
```

```
290     return f;
291 }
292 
293 int _symbolic_my_fabs(int n)
294 {
295     int ret;
296     klee_make_symbolic(&ret, sizeof(int), "ret");
297     return ret;
298 }
299 
300 int my_sin(int rad)
301 {
302     int diff;
303     int app=0;
304 
305     int inc = 1;
306 
307     /* MAX dependent on rad's value, say 50 */
308     while (rad > 2*PI)
309         rad -= 2*PI;
310     /* MAX dependent on rad's value, say 50 */
311     while (rad < -2*PI)
312         rad += 2*PI;
313     diff = rad;
314     app = diff;
315     diff = (diff * (-(rad*rad))) /
316         ((2 * inc) * (2 * inc + 1));
317     app = app + diff;
318     inc++;
319     /* REALLY: while(my_fabs(diff) >= 0.00001) { */
320     /* MAX: 1000 */
321     while(my_fabs(diff) >= 1) {
322         diff = (diff * (-(rad*rad))) /
323             ((2 * inc) * (2 * inc + 1));
324         app = app + diff;
```

```
325         inc++;
326     }
327 
328     return app;
329 }
330 
331 int _symbolic_my_sin(int rad)
332 {
333     int ret;
334     klee_make_symbolic(&ret, sizeof(int), "ret");
335     return ret;
336 }
337 
338 int my_cos(int rad)
339 {
340     return (my_sin (PI / 2 - rad));
341 }
342 
343 int _symbolic_my_cos(int rad)
344 {
345     int ret;
346     klee_make_symbolic(&ret, sizeof(int), "ret");
347     return ret;
348 }
349 
350 /* MAX: 1 */
351 int encode(int xin1,int xin2)
352 {
353     int i;
354     int *h_ptr,*tqmf_ptr,*tqmf_ptr1;
355     long int xa,xb;
356     int decis;
357 
358     /* transmit quadrature mirror filters implemented here */
359     h_ptr = h;
```

```
360     tqmf_ptr = tqmf;
361     xa = (long)(*tqmf_ptr++) * (*h_ptr++);
362     xb = (long)(*tqmf_ptr++) * (*h_ptr++);
363     /* main multiply accumulate loop for samples and coefficients */
364     /* MAX: 10 */
365     for(i = 0 ; i < 10 ; i++) {
366         xa += (long)(*tqmf_ptr++) * (*h_ptr++);
367         xb += (long)(*tqmf_ptr++) * (*h_ptr++);
368     }
369     /* final mult/accumulate */
370     xa += (long)(*tqmf_ptr++) * (*h_ptr++);
371     xb += (long)(*tqmf_ptr) * (*h_ptr++);
372
373     /* update delay line tqmf */
374     tqmf_ptr1 = tqmf_ptr - 2;
375     /* MAX: 22 */
376     for(i = 0 ; i < 22 ; i++) *tqmf_ptr-- = *tqmf_ptr1--;
377     *tqmf_ptr-- = xin1;
378     *tqmf_ptr = xin2;
379
380     /* scale outputs */
381     xl = (xa + xb) >> 15;
382     xh = (xa - xb) >> 15;
383
384     /* end of quadrature mirror filter code */
385
386     /* starting with lower sub band encoder */
387
388     /* filtez - compute predictor output section - zero section */
389     szl = filtez(delay_bpl,delay_dltx);
390
391     /* filtep - compute predictor output signal (pole section) */
392     spl = filtep(rlt1,al1,rlt2,al2);
393
394     /* compute the predictor output value in the lower sub_band encoder */
```

```
395     sl = szl + spl;
396     el = xl - sl;
397
398     /* quantl: quantize the difference signal */
399     il = quantl(el,detl);
400
401     /* invqxl: computes quantized difference signal */
402     /* for invqbl, truncate by 2 lsbs, so mode = 3 */
403     dlt = ((long)detl*qq4_code4_table[il >> 2]) >> 15;
404
405     /* logscl: updates logarithmic quant. scale factor in low sub band */
406     nbl = logscl(il,nbl);
407
408     /* scalel: compute the quantizer scale factor in the lower sub band */
409     /* calling parameters nbl and 8 (constant such that scalel can be scaleh) */
410     detl = scalel(nbl,8);
411
412     /* parrec - simple addition to compute recontructed signal for adaptive pred */
413     plt = dlt + szl;
414
415     /* upzero: update zero section predictor coefficients (sixth order)*/
416     /* calling parameters: dlt, dlt1, dlt2, ..., dlt6 from dlt */
417     /*  bpli (linear_buffer in which all six values are delayed */
418     /* return params:      updated bpli, delayed dltx */
419     upzero(dlt,delay_dltx,delay_bpl);
420
421     /* uppol2- update second predictor coefficient apl2 and delay it as al2 */
422     /* calling parameters: al1, al2, plt, plt1, plt2 */
423     al2 = uppol2(al1,al2,plt,plt1,plt2);
424
425     /* uppol1 :update first predictor coefficient apl1 and delay it as al1 */
426     /* calling parameters: al1, apl2, plt, plt1 */
427     al1 = uppol1(al1,al2,plt,plt1);
428
429     /* recons : compute recontructed signal for adaptive predictor */
```

```
430     rlt = sl + dlt;
431
432     /* done with lower sub_band encoder; now implement delays for next time*/
433     rlt2 = rlt1;
434     rlt1 = rlt;
435     plt2 = plt1;
436     plt1 = plt;
437
438     /* high band encode */
439
440     szh = filtez(delay_bph,delay_dhx);
441
442     sph = filtep(rh1,ah1,rh2,ah2);
443
444     /* predic: sh = sph + szh */
445     sh = sph + szh;
446     /* subtra: eh = xh - sh */
447     eh = xh - sh;
448
449     /* quanth - quantization of difference signal for higher sub-band */
450     /* quanth: in-place for speed params: eh, deth (has init. value) */
451     if(eh >= 0) {
452         ih = 3;     /* 2,3 are pos codes */
453     }
454     else {
455         ih = 1;     /* 0,1 are neg codes */
456     }
457     decis = (564L*(long)deth) >> 12L;
458     if(my_abs(eh) > decis) ih--;     /* mih = 2 case */
459
460     /* invqah: compute the quantized difference signal, higher sub-band*/
461     dh = ((long)deth*qq2_code2_table[ih]) >> 15L ;
462
463     /* logsch: update logarithmic quantizer scale factor in hi sub-band*/
464     nbh = logsch(ih,nbh);
```

```
465
466     /* note : scalel and scaleh use same code, different parameters */
467     deth = scalel(nbh,10);
468
469     /* parrec - add pole predictor output to quantized diff. signal */
470     ph = dh + szh;
471
472     /* upzero: update zero section predictor coefficients (sixth order) */
473     /* calling parameters: dh, dhi, bphi */
474     /* return params: updated bphi, delayed dhx */
475     upzero(dh,delay_dhx,delay_bph);
476
477     /* uppol2: update second predictor coef aph2 and delay as ah2 */
478     /* calling params: ah1, ah2, ph, ph1, ph2 */
479     ah2 = uppol2(ah1,ah2,ph,ph1,ph2);
480
481     /* uppol1:  update first predictor coef. aph2 and delay it as ah1 */
482     ah1 = uppol1(ah1,ah2,ph,ph1);
483
484     /* recons for higher sub-band */
485     yh = sh + dh;
486
487     /* done with higher sub-band encoder, now Delay for next time */
488     rh2 = rh1;
489     rh1 = yh;
490     ph2 = ph1;
491     ph1 = ph;
492
493     /* multiplex ih and il to get signals together */
494     return(il | (ih << 6));
495 }
496
497 /* decode function, result in xout1 and xout2 */
498
499 void decode(int input)
```

```
500 {
501     int i;
502     long int xa1,xa2;    /* qmf accumulators */
503     int *h_ptr,*ac_ptr,*ac_ptr1,*ad_ptr,*ad_ptr1;
504
505     /* split transmitted word from input into ilr and ih */
506     ilr = input & 0x3f;
507     ih = input >> 6;
508
509     /* LOWER SUB_BAND DECODER */
510
511     /* filtez: compute predictor output for zero section */
512     dec_szl = filtez(dec_del_bpl,dec_del_dltx);
513
514     /* filtep: compute predictor output signal for pole section */
515     dec_spl = filtep(dec_rlt1,dec_al1,dec_rlt2,dec_al2);
516
517     dec_sl = dec_spl + dec_szl;
518
519     /* invqxl: compute quantized difference signal for adaptive predic */
520     dec_dlt = ((long)dec_detl*qq4_code4_table[ilr >> 2]) >> 15;
521
522     /* invqxl: compute quantized difference signal for decoder output */
523     dl = ((long)dec_detl*qq6_code6_table[il]) >> 15;
524
525     rl = dl + dec_sl;
526
527     /* logscl: quantizer scale factor adaptation in the lower sub-band */
528     dec_nbl = logscl(ilr,dec_nbl);
529
530     /* scalel: computes quantizer scale factor in the lower sub band */
531     dec_detl = scalel(dec_nbl,8);
532
533     /* parrec - add pole predictor output to quantized diff. signal */
534     /* for partially reconstructed signal */
```

```
535     dec_plt = dec_dlt + dec_szl;
536
537     /* upzero: update zero section predictor coefficients */
538     upzero(dec_dlt,dec_del_dltx,dec_del_bpl);
539
540     /* uppol2: update second predictor coefficient apl2 and delay it as al2 */
541     dec_al2 = uppol2(dec_al1,dec_al2,dec_plt,dec_plt1,dec_plt2);
542
543     /* uppol1: update first predictor coef. (pole setion) */
544     dec_al1 = uppol1(dec_al1,dec_al2,dec_plt,dec_plt1);
545
546     /* recons : compute recontructed signal for adaptive predictor */
547     dec_rlt = dec_sl + dec_dlt;
548
549     /* done with lower sub band decoder, implement delays for next time */
550     dec_rlt2 = dec_rlt1;
551     dec_rlt1 = dec_rlt;
552     dec_plt2 = dec_plt1;
553     dec_plt1 = dec_plt;
554
555     /* HIGH SUB-BAND DECODER */
556
557     /* filtez: compute predictor output for zero section */
558     dec_szh = filtez(dec_del_bph,dec_del_dhx);
559
560     /* filtep: compute predictor output signal for pole section */
561     dec_sph = filtep(dec_rh1,dec_ah1,dec_rh2,dec_ah2);
562
563     /* predic:compute the predictor output value in the higher sub_band decoder */
564     dec_sh = dec_sph + dec_szh;
565
566     /* invqah: in-place compute the quantized difference signal */
567     dec_dh = ((long)dec_deth*qq2_code2_table[ih]) >> 15L ;
568
569     /* logsch: update logarithmic quantizer scale factor in hi sub band */
```

```
570     dec_nbh = logsch(ih,dec_nbh);
571
572     /* scalel: compute the quantizer scale factor in the higher sub band */
573     dec_deth = scalel(dec_nbh,10);
574
575     /* parrec: compute partially recontructed signal */
576     dec_ph = dec_dh + dec_szh;
577
578     /* upzero: update zero section predictor coefficients */
579     upzero(dec_dh,dec_del_dhx,dec_del_bph);
580
581     /* uppol2: update second predictor coefficient aph2 and delay it as ah2 */
582     dec_ah2 = uppol2(dec_ah1,dec_ah2,dec_ph,dec_ph1,dec_ph2);
583
584     /* uppol1: update first predictor coef. (pole setion) */
585     dec_ah1 = uppol1(dec_ah1,dec_ah2,dec_ph,dec_ph1);
586
587     /* recons : compute recontructed signal for adaptive predictor */
588     rh = dec_sh + dec_dh;
589
590     /* done with high band decode, implementing delays for next time here */
591     dec_rh2 = dec_rh1;
592     dec_rh1 = rh;
593     dec_ph2 = dec_ph1;
594     dec_ph1 = dec_ph;
595
596     /* end of higher sub_band decoder */
597
598     /* end with receive quadrature mirror filters */
599     xd = rl - rh;
600     xs = rl + rh;
601
602     /* receive quadrature mirror filters implemented here */
603     h_ptr = h;
604     ac_ptr = accumc;
```

```
605     ad_ptr = accumd;
606     xa1 = (long)xd * (*h_ptr++);
607     xa2 = (long)xs * (*h_ptr++);
608     /* main multiply accumulate loop for samples and coefficients */
609     for(i = 0 ; i < 10 ; i++) {
610         xa1 += (long)(*ac_ptr++) * (*h_ptr++);
611         xa2 += (long)(*ad_ptr++) * (*h_ptr++);
612     }
613     /* final mult/accumulate */
614     xa1 += (long)(*ac_ptr) * (*h_ptr++);
615     xa2 += (long)(*ad_ptr) * (*h_ptr++);
616 
617     /* scale by 2^14 */
618     xout1 = xa1 >> 14;
619     xout2 = xa2 >> 14;
620 
621     /* update delay lines */
622     ac_ptr1 = ac_ptr - 1;
623     ad_ptr1 = ad_ptr - 1;
624     for(i = 0 ; i < 10 ; i++) {
625         *ac_ptr-- = *ac_ptr1--;
626         *ad_ptr-- = *ad_ptr1--;
627     }
628     *ac_ptr = xd;
629     *ad_ptr = xs;
630 
631     return;
632 }
633 
634 /* clear all storage locations */
635 
636 void reset()
637 {
638     int i;
639 
```

```
640     detl = dec_detl = 32;   /* reset to min scale factor */
641     deth = dec_deth = 8;
642     nbl = al1 = al2 = plt1 = plt2 = rlt1 = rlt2 = 0;
643     nbh = ah1 = ah2 = ph1 = ph2 = rh1 = rh2 = 0;
644     dec_nbl = dec_al1 = dec_al2 = dec_plt1 = dec_plt2 = dec_rlt1 = dec_rlt2 = 0;
645     dec_nbh = dec_ah1 = dec_ah2 = dec_ph1 = dec_ph2 = dec_rh1 = dec_rh2 = 0;
646
647     for(i = 0 ; i < 6 ; i++) {
648         delay_dltx[i] = 0;
649         delay_dhx[i] = 0;
650         dec_del_dltx[i] = 0;
651         dec_del_dhx[i] = 0;
652     }
653
654     for(i = 0 ; i < 6 ; i++) {
655         delay_bpl[i] = 0;
656         delay_bph[i] = 0;
657         dec_del_bpl[i] = 0;
658         dec_del_bph[i] = 0;
659     }
660
661     for(i = 0 ; i < 23 ; i++) tqmf[i] = 0;
662
663     for(i = 0 ; i < 11 ; i++) {
664         accumc[i] = 0;
665         accumd[i] = 0;
666     }
667     return;
668 }
669
670 /* filtez - compute predictor output signal (zero section) */
671 /* input: bpl1-6 and dlt1-6, output: szl */
672
673 int filtez(int *bpl,int *dlt)
674 {
```

```
675     int i;
676     long int zl;
677     zl = (long)(*bpl++) * (*dlt++);
678     /* MAX: 6 */
679     for(i = 1 ; i < 6 ; i++)
680         zl += (long)(*bpl++) * (*dlt++);
681 
682     return((int)(zl >> 14));   /* x2 here */
683 }
684 
685 /* filtep - compute predictor output signal (pole section) */
686 /* input rlt1-2 and al1-2, output spl */
687 
688 int filtep(int rlt1,int al1,int rlt2,int al2)
689 {
690     long int pl,pl2;
691     pl = 2*rlt1;
692     pl = (long)al1*pl;
693     pl2 = 2*rlt2;
694     pl += (long)al2*pl2;
695     return((int)(pl >> 15));
696 }
697 
698 int _symbolic_filtep(int rlt1,int al1,int rlt2,int al2)
699 {
700     int ret;
701     klee_make_symbolic(&ret, sizeof(int), "ret");
702     return ret;
703 }
704 
705 /* quantl - quantize the difference signal in the lower sub-band */
706 int quantl(int el,int detl)
707 {
708     int ril,mil;
709     long int wd,decis;
```

```
710
711     /* abs of difference signal */
712     wd = my_abs(el);
713     /* determine mil based on decision levels and detl gain */
714     /* MAX: 30 */
715     for(mil = 0 ; mil < 30 ; mil++) {
716         decis = (decis_levl[mil]*(long)detl) >> 15L;
717         if(wd <= decis) break;
718     }
719     /* if mil=30 then wd is less than all decision levels */
720     if(el >= 0) ril = quant26bt_pos[mil];
721     else ril = quant26bt_neg[mil];
722     return(ril);
723 }
724
725 int _symbolic_quantl(int el, int detl)
726 {
727     int ret;
728     klee_make_symbolic(&ret, sizeof(int), "ret");
729     return ret;
730 }
731
732 /* invqxl is either invqbl or invqal depending on parameters passed */
733 /* returns dlt, code table is pre-multiplied by 8 */
734
735 /*    int invqxl(int il,int detl,int *code_table,int mode) */
736 /*    { */
737 /*        long int dlt; */
738 /*       dlt = (long)detl*code_table[il >> (mode-1)]; */
739 /*        return((int)(dlt >> 15)); */
740 /*    } */
741
742 /* logscl - update log quantizer scale factor in lower sub-band */
743 /* note that nbl is passed and returned */
744
```

```
745 int logscl(int il,int nbl)
746 {
747     long int wd;
748     wd = ((long)nbl * 127L) >> 7L;   /* leak factor 127/128 */
749     nbl = (int)wd + wl_code_table[il >> 2];
750     if(nbl < 0) nbl = 0;
751     if(nbl > 18432) nbl = 18432;
752     return(nbl);
753 }
754
755 int _symbolic_logscl(int il, int nbl)
756 {
757     int ret;
758     klee_make_symbolic(&ret, sizeof(int), "ret");
759     return ret;
760 }
761
762 /* scalel: compute quantizer scale factor in lower or upper sub-band*/
763
764 int scalel(int nbl,int shift_constant)
765 {
766     int wd1,wd2,wd3;
767     wd1 = (nbl >> 6) & 31;
768     wd2 = nbl >> 11;
769     wd3 = ilb_table[wd1] >> (shift_constant + 1 - wd2);
770     return(wd3 << 3);
771 }
772
773 int _symbolic_scalel(int nbl, int shift_constant)
774 {
775     int ret;
776     klee_make_symbolic(&ret, sizeof(int), "ret");
777     return ret;
778 }
779
```

```
780 /* upzero - inputs: dlt, dlti[0-5], bli[0-5], outputs: updated bli[0-5] */
781 /* also implements delay of bli and update of dlti from dlt */
782
783 void upzero(int dlt,int *dlti,int *bli)
784 {
785     int i,wd2,wd3;
786     /*if dlt is zero, then no sum into bli */
787     if(dlt == 0) {
788         for(i = 0 ; i < 6 ; i++) {
789             bli[i] = (int)((255L*bli[i]) >> 8L); /* leak factor of 255/256 */
790         }
791     }
792     else {
793         for(i = 0 ; i < 6 ; i++) {
794             if((long)dlt*dlti[i] >= 0) wd2 = 128; else wd2 = -128;
795             wd3 = (int)((255L*bli[i]) >> 8L);    /* leak factor of 255/256 */
796             bli[i] = wd2 + wd3;
797         }
798     }
799     /* implement delay line for dlt */
800     dlti[5] = dlti[4];
801     dlti[4] = dlti[3];
802     dlti[3] = dlti[2];
803     dlti[1] = dlti[0];
804     dlti[0] = dlt;
805     return;
806 }
807
808 /* uppol2 - update second predictor coefficient (pole section) */
809 /* inputs: al1, al2, plt, plt1, plt2. outputs: apl2 */
810
811 int uppol2(int al1,int al2,int plt,int plt1,int plt2)
812 {
813     long int wd2,wd4;
814     int apl2;
```

```
815     wd2 = 4L*(long)al1;
816     if((long)plt*plt1 >= 0L) wd2 = -wd2;    /* check same sign */
817     wd2 = wd2 >> 7;                  /* gain of 1/128 */
818     if((long)plt*plt2 >= 0L) {
819         wd4 = wd2 + 128;             /* same sign case */
820     }
821     else {
822         wd4 = wd2 - 128;
823     }
824     apl2 = wd4 + (127L*(long)al2 >> 7L);  /* leak factor of 127/128 */
825 
826     /* apl2 is limited to +-.75 */
827     if(apl2 > 12288) apl2 = 12288;
828     if(apl2 < -12288) apl2 = -12288;
829     return(apl2);
830 }
831 
832 int _symbolic_uppol2(int al1,int al2,int plt,int plt1,int plt2)
833 {
834     int ret;
835     klee_make_symbolic(&ret, sizeof(int), "ret");
836     return ret;
837 }
838 
839 /* uppol1 - update first predictor coefficient (pole section) */
840 /* inputs: al1, apl2, plt, plt1. outputs: apl1 */
841 
842 int uppol1(int al1,int apl2,int plt,int plt1)
843 {
844     long int wd2;
845     int wd3,apl1;
846     wd2 = ((long)al1*255L) >> 8L;   /* leak factor of 255/256 */
847     if((long)plt*plt1 >= 0L) {
848         apl1 = (int)wd2 + 192;      /* same sign case */
849     }
```

```
850     else {
851         apl1 = (int)wd2 - 192;
852     }
853     /* note: wd3= .9375-.75 is always positive */
854     wd3 = 15360 - apl2;             /* limit value */
855     if(apl1 > wd3) apl1 = wd3;
856     if(apl1 < -wd3) apl1 = -wd3;
857     return(apl1);
858 }
859 
860 int _symbolic_uppol1(int al1,int apl2,int plt,int plt1)
861 {
862     int ret;
863     klee_make_symbolic(&ret, sizeof(int), "ret");
864     return ret;
865 }
866 
867 /* INVQAH: inverse adaptive quantizer for the higher sub-band */
868 /* returns dh, code table is pre-multiplied by 8 */
869 
870 /*  int invqah(int ih,int deth) */
871 /*  { */
872 /*        long int rdh; */
873 /*        rdh = ((long)deth*qq2_code2_table[ih]) >> 15L ; */
874 /*        return((int)(rdh )); */
875 /*  } */
876 
877 /* logsch - update log quantizer scale factor in higher sub-band */
878 /* note that nbh is passed and returned */
879 
880 int logsch(int ih,int nbh)
881 {
882     int wd;
883     wd = ((long)nbh * 127L) >> 7L;       /* leak factor 127/128 */
884     nbh = wd + wh_code_table[ih];
```

```
885     if(nbh < 0) nbh = 0;
886     if(nbh > 22528) nbh = 22528;
887     return(nbh);
888 }
889 
890 int _symbolic_logsch(int ih,int nbh)
891 {
892     int ret;
893     klee_make_symbolic(&ret, sizeof(int), "ret");
894     return ret;
895 }
896 
897 #ifndef Seoul_Mate
898 int main(int argc, char **argv)
899 {
900     int i,j,f/*,answer*/;
901     static int *test_data[SIZE*2],*compressed[SIZE],result[SIZE*2];
902 
903     /* reset, initialize required memory */
904     reset();
905 
906     /* read in amplitude and frequency for test data */
907     /*  scanf("%d",&j);
908         scanf("%d",&f); */
909     klee_make_symbolic(&j, sizeof(int), "j");
910     klee_make_symbolic(&f, sizeof(int), "f");
911 
912     /* 16 KHz sample rate */
913     /* XXmain_0, MAX: 2 */
914     /* Since the number of times we loop in my_sin depends on the argument we
915        add the fact: xxmain_0:[]: */
916     for(i = 0 ; i < SIZE ; i++) {
917         test_data[i] = (int *) malloc(sizeof(int));
918         *(test_data[i]) = (int)j*my_cos(f*PI*i);
919     }
```

```
920     test_data[i] = (int *) malloc(sizeof(int));
921     *(test_data[i]) = 0;
922
923
924
925     /* MAX: 2 */
926
927     /*******Antar att test_data[0] = 10 och test_data[1]=-6 frn ovan,          *******
928       och att anropet i forloopen blir encode(test_data[0],test_data[0]);
929       och encode(test_data[1],test_data[1]), eftersom att den annars gr
930      *******over array gransen                                                 *******/
931
932
933     for(i = 0 ; i < IN_END ; i += 2) {
934         compressed[i/2] = (int *) malloc(sizeof(int));
935         *(compressed[i/2]) = encode(*(test_data[i]),*(test_data[i+1]));
936         free(test_data[i]);
937         free(test_data[i+1]);
938     }
939     /* MAX: 2 */
940     for(i = 0 ; i < IN_END ; i += 2) {
941         decode(*(compressed[i/2]));
942         free(compressed[i/2]);
943         result[i] = xout1;
944         result[i+1] = xout2;
945     }
946     /*
947        for( ; j < 32767 ; j++) {
948        i=IN_END-1;
949        printf("\n%4d %4d %4d %4d %4d",j,compressed[i/2] >> 6,
950        compressed[i/2] & 63,result[i],result[i-1]);
951        }
952        */
953     /* print ih, il */
954     /*
```

```
955         for(i = 0 ; i < IN_END/2 ; i++) printf("\n%4d %2d %2d",
956         i,compressed[i] >> 6,compressed[i] & 63);
957         */
958 
959     return result[i]+result[i+1];
960 }
961 #endif
```