Coursework 03
Detection of Software Vulnerabilities:
Static Analysis (Part I)

Introduction

This coursework introduces students to the basic concepts of software verification and validation concerning security aspects. In particular, this coursework provides five theoretical and practical exercises: (i) describe the syntax and semantics of the most common SMT background theories and their application to verify security properties in software systems; (ii) check the satisfiability of SMT equations using state-of-the-art solvers based on the theory of bit-vectors; (iii) analyze the main advantages and disadvantages of BMC techniques concerning soundness and completeness in order to verify security properties; (iv) build SMT formulas extracted from C programs to check for buffer overflow and memory leak; lastly, (v) develop a precise memory model for software verification by taking into account memory alignment.

Learning Objectives

By the end of this lab you will be able to:

- Introduce software verification and validation.
- Understand soundness and completeness concerning detection techniques.
- Emphasize the difference between static analysis, testing/simulation, and debugging.
- Explain bounded model checking of software.
- Explain precise memory model for software verification.

1) **(Satisfiability Modulo Theories)** Describe the syntax and semantics of the background theories used by SMT solvers, which are relevant for verifying security properties in software systems, as described by the SMT-LIB.[1]

2) **(Solving SMT equations)** Check the satisfiability of these propositional equations. You must use the theory of bit-vector (QF_BF) implemented in the Z3 SMT solver.[2] Once you write the equations in QF_BF, you must check them using the commands: `(check-sat)` and `(get-model)` [1]. Here you must present both the equations and output of the SMT solver Z3.

a) $(\neg a \vee \neg b) \wedge (\neg b \vee c) \wedge b$
b) $(((d \wedge c) \vee (p \wedge \neg ((c \wedge \neg (d))))) \equiv ((c \wedge d) \vee (p \wedge c) \vee (p \wedge \neg (d))))$
c) $(((a) \wedge ((b) \rightarrow \neg (a)) \equiv \neg (b))$

[1] http://smtlib.cs.uiowa.edu/,

http://smtlib.cs.uiowa.edu/language.shtml,

http://smtlib.cs.uiowa.edu/theories.shtml

[2] https://github.com/Z3Prover/z3

3) **(Soundness and Completeness)** What are the main advantages and disadvantages of the BMC technique? How can we ensure the soundness and completeness of the BMC technique? There exist some research papers that discuss these topics [2,3].

4) **(C/C++ API of Z3)** Write a program using the C/C++ API of Z3 to verify the following C programs to check for *buffer overflow* and *memory leak*. Note that you must build two sets of formulas *C* and *P* and then check for the satisfiability of the resulting equation C ∧ ¬P using the SMT solver Z3.

a) Buffer overflow.

```
int main() {
  int a[2], i, x, *p;
  p=a;
  if (x==0)
    a[i]=0;
  else
    a[i+1]=1;
  assert(*(p+2)==1);
}
```

b) Memory leak.

```
#include <stdlib.h>
int main() {
  char *p = malloc(5);
  char *q = malloc(5);
  p=q;
  free(p);
  p = malloc(5);
  free(p);
  return 0;
}
```

5) **(Aligned Memory Model)** Derive the equations C and P from the following program C. You must enforce the C alignment rules when writing the resulting formula C ∧ ¬ P [4]. You must check the resulting formula using the SMT solver Z3.

```
#include <stdint.h>
struct foo {
  uint16_t bar[2];
  uint8_t baz;
};
int main() {
  struct foo qux;
  qux.bar[0] = 10;
```

```
  qux.bar[1] = 20;
  qux.baz = 'C';
  struct foo *quux = &qux;
  quux++;
  quux->baz = 'D';
  return 0;
}
```

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Question 1) Has the student described the syntax and semantics of the background theories that are relevant for verifying security properties?	
The student has described at least three background theories from the SMT-lib and explained how relevant they are to check for security properties.	(2)
The student has described some background theories from the SMT-lib. He/she has also explained, with some limitations, how relevant they are to check security properties.	(1)
No attempt has been made.	(0)

Question 2) Has the student encoded all three propositional formulae into bit-vectors? Has he/she also checked for their satisfiability using an SMT solver?	
The student has encoded all three propositional formulae into bit-vectors. He/she can check for their satisfiability using an SMT solver.	(2)
The student has encoded one or two propositional formulae into bit-vectors. He/she also has some limitations to check the satisfiability of the SMT formulae.	(1)
No attempt has been made.	(0)

Question 3) Has the student described the advantages and disadvantages of the BMC technique by addressing soundness and completeness aspects?	
The student has described the advantages and disadvantages of the BMC technique. In particular, he/she has addressed these discussions by focusing on soundness and completeness aspects.	(2)
The student demonstrates limited capability to explain the advantages and disadvantages of the BMC technique. He/she does not correctly explain soundness and completeness aspects.	(1)
No attempt has been made.	(0)

Question 4) Has the student written the SMT formulae to check for buffer overflow and memory leaks using the C/C++ API of the SMT solver Z3?	
The student has written the SMT formulae to check for buffer overflow and memory leaks using the C/C++ API of the SMT solver Z3.	(2)
The student demonstrates limited capability to encode the SMT formulae to check for buffer overflow and memory leaks. He/she does not correctly encode those properties using the C/C++ API of the SMT solver Z3.	(1)
No attempt has been made.	(0)

Question 5) Has the student encoded the C alignment rules for the given program?	
The student has encoded the C alignment rules for the given program.	(2)
The student has encoded the constraints and properties, but he/she does not consider C alignment rules for the given program.	(1)
No attempt has been made.	(0)

References:

[1] Tutorial https://rise4fun.com/z3/tutorial/guide.

[2] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, James Worrell: Linear Completeness Thresholds for Bounded Model Checking. CAV 2011: 557-572

[3] Lucas C. Cordeiro, Bernd Fischer, João Marques-Silva: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Trans. Software Eng. 38(4): 957-974 (2012).

[4] Jeremy Morse, Mikhail Ramalho, Lucas C. Cordeiro, Denis A. Nicole, Bernd Fischer: ESBMC 1.22 - (Competition Contribution). TACAS 2014: 405-407.