18733: Applied Cryptography S17

Mini Project

due April 19, 2017, 11:59pm EST

1 SMT Solvers

1.1 Satisfiability Modulo Theory

SMT (Satisfiability Modulo Theories) is a decision problem for logical formulas expressed in first order logic. SMT takes in a set of constraints, logical relationships between variables and determines if there exists a set of values for all of the variables in the system that simultaneously satisfy all of the constraints.

The SMT problem is related to the SAT (Satisfiability or Boolean Satisfiability) problem which is to decide if a boolean formula is satisfiable. An example of a SAT problem is:

$$\phi(a, b) := a \land (a \lor b),$$

where a, b is a boolean variables. This boolean formula ϕ is satisfiable by $a = \top, b = \bot$.

An important case of the SAT problem is 3-SAT which is the problem of determining satisfiability of a formula in the conjunctive normal form (CNF) where each clause is limited to three literals. 3-SAT is a complexity class that comes up frequently in many famous landmark constructions of cryptography such as Blum's proof of a Bounded Non-Interactive Zero Knowledge Proof System. It is known that the 3-SAT problem is NP-Complete.

SAT problems decide if given boolean formula is satisfiable, and we can think of SMT as a generalization of SAT where we allow the literals to be richer data types such as bit vectors instead of boolean values. Boolean values can be thought of as a single bit, or a bit vector of length 1 which generalizes to bit vectors of arbitrary length. An example of of a SMT problem is the following:

$$\phi(x) := (x > 5) \land (x < 10)$$

This problem is obviously satisfiable by the values $x = 6, 7, 8, 9$.

1.2 Z3

Z3 is an SMT solver from Microsoft Research and is available at: https://github.com/Z3Prover/z3

I have provided examples of how to interact with Z3 using python in `mini-project_template.py`. For completeness I will cover some of these cases here.

To start using Z3, create a solver object as follows:

```
s = Solver()
```

Next, we will create some variables. For this project all of the variables we will be working with will be bit vectors. The following creates 3 variables which are bit vectors of size 32.

```
x, y, z = BitVecs('x, y, z', 32)
```

Once we have some variables, we need to inform the solver between the relationships between these variables. For example, suppose we are interested in the case where $z = 10$, $y > z$ and $z < x < y$.

```
s.add(z == 10)
s.add(y > z)
s.add(x < y)
```

This specifies the SMT problem:

$$(z = 10) \land (y > z) \land (x < y)$$

At this point we might be interested in knowing if of model even has a solution. To check this we can execute the following:

```
print(s.check())
```

To which the response will be in $\{sat, unsat\}$ depending on if the equations are satisfiable or unsatisfiable. Of course in this case, the model is satisfiable with values such as $z = 10, y = 12, x = 11$.

Note that several constraints can be specified on the same line, for example, we can do the following:

```
s.add(z == 10)
s.add(z < x < y)
```

Which produces the SMT problem:

$$(z = 10) \land (z < x < y)$$

The following is a list of all the important operations that you will need to know how to do in this assignment:

```
s.add(x == 123) #Setting variable equal to constant
s.add(x == 0x1234) #Setting variable equal to hexadecimal constant
s.add(x != 123) #Constraining a variable to not be a particular value
s.add(x == y) #Setting a variable equal to another variable

s.add(x > y)
s.add(x < y)
s.add(x >= y)
s.add(x <= y) #Standard inequalities
```

```
s.add(x == z & y) #Logical bitwise AND
s.add(x == z | y) #Logical bitwise OR
s.add(x == z ^ y) #Logical bitwise XOR

s.add(x == RotateRight(y, 10)) #Rotate the bits of y right by 10 with replacement
s.add(x == RotateLeft(y, 10)) #Rotate the bits of y left by 10 with replacement
s.add(x == LShR(y, 10)) #Logical Shift Right of bits of y by 10
s.add(x == y >> 10) #Arithmetic Shift Right of bits of y by 10
s.add(x == y << 10) #Logical Shift Left of bits of y by 10

s.add(x == y + z) #Addition
s.add(x == y - z) #Subtraction
s.add(x == y * z) #Multiplication
```

1.3 Toy Example

Defined below is the C code for a function called `toy_hash`, and we are assuming here that unsigned long is a 64 bit data type. `toy_hash` is the type of hash function often seen in capture the flag (CTF) games designed to be too difficult to analyze and find collisions for by hand, but trivial enough to be done rapidly with the aid of tools.

```
unsigned long toy_hash(unsigned long input)
{
   C1 = 0x5D7E0D1F2E0F1F84;
   C2 = 0x388D76AEE8CB1500;
   C3 = 0xD2E9EE7E83C4285B;

   input *= C1;
   input = _rotr(input, input & 0xF);
   input = input ^ C2;
   input = _rotl(input, input & 0xF);
   output = input + C3;

   return output;
}
```

We can check that calling this function with `input` = 123456789012345 results in an output of 16469441475088150007. The goal here to find another value, `input'` $\neq$ `input` such that the output is the same. To do this, we will setup a model in Z3 using the following commands:

```
def toy_hash_collision(desired_output, forbidden_input)

  C1 = 0x5D7E0D1F2E0F1F84
  C2 = 0x388D76AEE8CB1500
  C3 = 0xD2E9EE7E83C4285B

  inp, i1, i2, i3, i4, outp = BitVecs('inp, i1, i2, i3, i4, outp', 64)
```

```
s = Solver()
s.add(i1 == inp*C1)
s.add(i2 == RotateRight(i1, i1 & 0xF))
s.add(i3 == i2 ^ C2)
s.add(i4  == RotateLeft(i3, i3 & 0xF))
s.add(outp == i4 + C3)
s.add(outp == desired_output)
s.add(inp != forbidden_input)

print(s.check())
print(s.model())
```

The hash collision function starts by describing the relationship between the input and output that is defined in the `toy_hash` function. This ensures that for any value of variable `inp` that the solver selects, the value of `outp` will be its hash. Some additional constraints are also supplied, for example we are requiring the output to be the particular value that we are interesting in finding a collision for. We also don't want the solver to select the same input value that we originally provided, so we forbid the input from taking that value.

The result of executing `toy_hash_collision` on the correct input is the value 9223495493643788153 which collides with 123456789012345.

1.4 Some Other Thoughts

This idea of using SMT solvers to express the relationship between inputs and outputs of a function forms the basis of an active area of computer science research known as symbolic execution. A module takes in some input which is typically either source code or a binary program and automatically generates the constraints that describe the relationship between inputs and outputs. This can be useful for automatically finding inputs that trigger exploits within the program or for generating test cases that cover a large amount of the control flow paths within the application among other applications.

2 SHA-256

SHA-2 is a class of cryptographic hash functions designed by the NSA. Among the SHA-2 hash functions is a popular one named SHA-256 which produces a 256 bit message digest. SHA-256 in a functional form can be denoted as:

$$\text{SHA-256} : \{0,1\}^* \rightarrow \{0,1\}^{256}$$

Normally, SHA-256 takes inputs of arbitrary length and pads them up to blocks of 64 bytes using a special padding scheme, however for this assignment, I have modified the implementation of SHA-256 in `sha256_template.py` to simply zero-extend the input up to 64 bytes.

Once the input has been padded, it is passed as an argument to the process subroutine that does the actual hashing. This subroutine is called with inputs of 64 bytes, if the value to be hashed is larger than 64 bytes, then the process subroutine will be called several times.

The process subroutine begins by expanding the 64 byte input to a 256 byte vector, by appending 192 bytes that are deterministically generated from the original 64. SHA-256 then iteratively applies a compression function to the 256 byte vector 64 times, on each iteration updating its internal 'hash' value.

Once all of the inputs have been sent through the process function, the internal 'hash' value is the resulting hash of the inputs. This 'hash' value has an initial default value and there is a large hard-coded constant key that is used as a lookup table for SHA-256, not unlike other kinds of lookup tables we have seen in class such as S-Boxes.

For this assignment, we are interested in the version of SHA-256 where we only run it for 18 rounds and where the special padding scheme has been removed in favor of zero-extending. We will denote this special version as SHA-256-18. The full specification of SHA-256-18 can be found in `sha256_template.py`.

3 Certificates

In this assignment, you are given a public-private 2048-bit RSA key pair. You are also given a 'valid' certificate endorsed by Gihyuk's 2048-bit private RSA key. The motivating force behind finding hash collisions will be to perform an attack where you can obtain a valid certificate that allows you to endorse other certificates.

Full specification of certificate as well as the public-private key pairs can be found in `certificates_template.py`. These are not X.509 certificates, they are a bit simpler but are very similar in the information that they contain. I have chosen to go with these custom certificates since the APIs available for interfacing with X.509 certificates do not expose the raw data underneath very well which is needed when using custom non-standard hash functions. Additionally, these certificates are very simple and reduce the complexity on what I want to be a straightforward aspect of the assignment.

Recall the 'rogue certificate' attack that we discussed in class by Sotirov et al. where the attackers obtained a valid certificate for a rogue intermediary CA. This is not only a valid standalone certificate, but also a certificate that is allowed to be used to endorse other certificates, giving the attacker virtually unlimited power. These attacks used a given 'valid' endorsed certificate, and modified it by replacing certain values for some fields of the certificate such that the entire hash remains unchanged. This unchanged hash meant that the signature on the old certificate would also be a signature on the new, more powerful certificate.

If you believe you have formed one of these powerful certificates, you can check your answer by visiting the URL `http://possibility.cylab.cmu.edu/18733/check_cert.php?cert=<url_encoded_cert>` where a URL encoded certificate is obtained by calling urlencodeCertificate with the certificate as an argument. You will want to verify that the certificate granted to the class is valid but not powerful.

Hint: notice that the optional fields `subject_unique_id` and `signer_private_key_str` are not being used, they would be the perfect place to inject some arbitrary characters when creating a rogue certificate.

4 Requirements

1. **[8 points]** Generate two strings s_1, s_2 such that $s_1 \neq s_2$ and SHA-256-18(s_1) = SHA-256-18(s_2) and such that s_1, s_2 are not zero-extended version of each other. Provide these strings as well as any code used to generate them in the assignment that you turn in. Also make sure that these strings are encoded in a readable format such as base 10 or base 16 and not ASCII.

2. **[2 points]** Generate a **rogue CA certificate**. Attach your certificate as well as any code used to generate it and any special output from the server to the assignment that you turn in.