Automation in Cryptology

Ashish Tiwari

Abstract

Inspired by the recent work on sketching-based synthesis of programs [SLRBE05, SLTB+06], we have developed a language for specifying sketches, or partially specified programs. We have also developed a tool that automatically completes the sketch so that the complete program satisfies some desired post-condition. We used the language and the tool to synthesize several padding-based encryption schemes. In this report, we describe the language, the synthesis technique, and the case study of synthesizing padding-based schemes.

Terms: Synthesis, Sketching, Encryption, SMT solving, Exists-Forall constraint solving

1 Background

The goal of the project is to develop new verification and synthesis techniques for programs that model cryptographic protocols. In the first year, verification approaches based on abstract interpretation and constraint solving were explored. Incomplete, but fast, procedures for proving and falsifying security are important for creating effective synthesis tools.

In the second year, we have focused on the synthesis of cryptographic protocols. Inspired by the recent work on sketching-based synthesis of programs [SLRBE05, SLTB+06], we have developed a language for specifying sketches that are automatically completed by our tool to satisfy some desired post-condition. We used the language and the tool to synthesize several padding-based encryption schemes. Our results here are reminiscent of the results obtained by the synthesis tool Zoocrypt. The main challenge is encoding security properties as post-conditions. We discuss this further in the report below. We also used the tool to explore the possibility of synthesizing oblivious transfer protocols.

We report on the language for specifying sketches, the tool for completing the sketches, and the case studies below.

2 A Language for Specifying Sketches

A sketch is an incomplete program. In this section, we describe a language for specifying sketches. In particular, the language allows for encoding problems where the goal is to synthesize a program given a library of pre-defined functions [GJTV11].

Our language is designed so that it can be used to write a concrete program, as well as, a completely unknown program constructed using a library of pre-defined functions.

A program sketch `Prgm_sketch` has the following syntax:

`Prgm_sketch` ::== (*program_name* `Comment` $*$ `Declarations Library Parameters Blocks Post`)

The terminal *program_name* is just a string. A comment, `comment` is of the form (**comment** *string*). The remaining blocks of the language are described below.

- *The library block.* This takes the form

  (**library** [(*function_name arity*)] $*$)

  The final synthesized program should be constructed using only the functions in the library.

- *The declarations block.* The declarations block defines the library functions and the type of the data elements in the program. It takes the following form:

  (**decls** [`yices_definition`] $*$)

  where `yices_definition` is a valid definition (of a function or type) in the SMT solver Yices [SRI].

- *Parameter Declarations.* Parameters that are used to specify the sketch are also explicitly listed using the syntax:

  (**parameters** [*parameter_name*] $*$)

  where *parameter_name* is simply a string.

- *Program Sketch Block.* Program sketch is a collection of single blocks and it is specified using the following syntax:

  (**blocks** [`single_block`] $*$)

  where `single_block` specifies the sketch corresponding to a block of straight-line code using the following syntax:

  (*label* `param_or_num` (`rhs_option` $*$))

where
(a) *label* is a string that names the straight-line code block,
(b) `param_or_num` is a parameter name or a number that represents the number of lines of code in the block, and
(c) `rhs_option` represents a possible choice of the function name (from the library) and arguments that can occur on the right-hand side expression on each line in the block. The syntax for `rhs_option` is as follows:

$$(\textit{func\_name}\ \texttt{block\_label\_list}+\ )$$

where *func_name* is a string naming a function from the library followed by finite number (equal to the arity of the function) of lists of block-labels. A `block_label_list` is simply a list of block labels:

$$([\textit{block\_label}\ \mid\ -]+)$$

For example, $(f\ \ (L1)\ \ (L2))$ is a possible `rhs_option`, and its meaning is that (one possibility for) the rhs-expression of this block is an expression of the form $f(x, y)$ where x is the value computed on some line in block $L1$ and y is the value computed on some line in block $L2$. The meaning of $(f\ \ (L1\ \ -)\ \ (L2))$ would be that the first argument of f can be a value computed in block $L1$ or any value computed in the current block prior to this line.
For example, a block specified by

$$(L1\ \ na\ \ ((\textit{oplus}\ \ (lm\ \ lr)\ \ (-))\ \ (G\ \ (lr\ \ -))))$$

corresponds to straight-line program of length na, where each of the na lines has an expression of the form $\textit{oplus}(x, y)$ or $G(z)$, where x is some value computed in block lm or block lr, y is some value computed in this block prior to current line, and z is some value computed in block lr or in current block prior to current line.
Inputs to a program are assigned a line in the program. Hence, we can also have $(\textit{input}\ m)$ as a possible `rhs_option`.

- *The Post block.* A postcondition for the program can be specified in the post block as follows:

$$(\textbf{ensure}\ \ \textit{yices\_formula})$$

where *yices_formula* is a quantifier-free Yices formula over program variables, where a program variable is specified as follows:

$$(\textbf{output}\ \ \textit{block\_label}\ \ \texttt{param\_or\_num})$$

which denotes the program variable assigned on line specified by the line number in the specified block.

A concrete straight-line program can be written using blocks of length 1 in which there is just one option for the right-hand side expression. An arbitrary straight-line program of length n can be written as

$$(L1\ \ n\ \ ((f1\ \ (-)\ \ (-))\ \ (f2\ \ (-)\ \ (-))\ldots(fn\ \ (-)\ \ (-))))$$

When performing synthesis, finding one program from the set of all n line programs can be difficult. The language above allows the user to constrain the class of programs.

Types. To further constrain the class of valid programs conforming to a sketch, we introduced a notion of types in the sketching language. Each program variable in the sketch is assigned two values – its usual data value, and another value that we call the "type" value. Each library function f has two instances: one instance specifies how it modifies the data values of the variables, and the second instance specifies a constraint on the types of the inputs and outputs of the function. We will illustrate the use of types in the case study section below.

3 From Sketch to $\exists\forall$ Formula

The existence of a program that conforms to a given sketch and that satisfies the given post condition can be formulates as an $\exists\forall$ formula. The $\exists\forall$ formula says that there is a well-formed program such that forall values of the inputs, the output of the program satisfies the post condition.

We have developed a tool that parses a given sketch and creates a $\exists\forall$ Yices formula. Furthermore, the tool then calls the $\exists\forall$ solver of Yices to solve the formula. If there is a solution, the tool outputs the model for the exists variables, which can be used to obtain the concrete program. Additionally, the tool can also search for alternate solutions for the same sketch.

4 Padding-based Encryption Schemes

Inspired by the success of the tool Zoocrypt in synthesizing padding-based encryption schemes, we used our synthesis tool for exploring the same space.

The sketch in Figure 1 shows the sketch we used. The library for constructing the padding scheme consists of two unary hash functions, G and H, a binary *xor* function (called *oplus* in Figure 1), a slight variant of *xor* called *oplusr*, and the identity function. Padding with

```
(oaep_sketch

 (decls ...)

 (parameters na nb)

 (library (G 1) (H 1) (oplusr 2) (oplus 2) (identity 1))

 (blocks
  (lm 1 ((input m::(bool-to-bv false false false false true))))
  (lr 1 ((input r::(bool-to-bv false false false true false))))
  (l1 na ( (oplusr (lm lr) (-)) (G (lr -)) (H (lm -))))
  (l2 2  ( (identity (l1 lr) ) ))
  (l3 nb ( (oplus (l2 -) (l2 -)) (H (l2 -)) (G (l2 -)) )))

 (ensure (and (= (output lm 1) (output l3 nb))
              (isrand (type l2 1)) (isrand (type l2 2))))
 )
```

Figure 1: Sketch used for synthesizing various padding-based encryption schemes. Declarations are shown in Figure 2.

0 is not modeled explicitly. It is added as a post-processing step to make the hash functions applicable on its arguments.

The sketch in Figure 1 has two inputs – the message m in block lm line 1, and a random number r in block lr line 1. This is followed by a straight-line code block $l1$ of length na that constructs the padding scheme. It is allowed to use the hash functions and the xor function. Two of the values computed in block $l1$ (including the random number r) are picked in block $l2$ to be concatenated, encrypted and sent onb the network. The block $l3$ decodes the messages received from block $l2$. Thee decoding block is of length nb and it can use the hash functions and the xor function.

The main challenge in applying program synthesis techniques to cryptography lies in encoding the security conditions. Here, we use types to capture desired properties of a padding-based encryption scheme.

With each data value, we associate an independent "type" value. In the example in Figure 1, the "type" associated to a data value is a bitvector of length 5:
(a) The first bit keeps information about the size of the data value –

since we have hash functions mapping bitvectors of one size to another, we keep a bit to store the size of the data value.
(b) The second bit is set if the data value is essentially the same as a random value in its domain. It is difficult to carry forward this information precisely, so we use conservative typing rules to update the value of the second type-bit during each operation.
(c) The third bit is set if the top function application is the hash function G. This information is used to update the "israndom" second bit of the type.
(d) The fourth bit is set if the top function application is the hash function H. It is used for the same purpose as the previous bit.
(e) The fifth bit is set if the top function application is the *xor* function. Again, this information is used for the same purpose as the previous two type-bits.

Using the data value and the type value, we state the desired post condition as shown in the **ensure** formula; namely,
(a) the result of decoding (written as `(output l3 nb)`, the value on line `nb` in block `l3`) should be equal to the message `m` (written as `(output lm 1)`, the value on line 1 in block `lm`), and
(b) the two values that are transmitted, namely the value on lines 1 and 2 in block `l2`, should essentially be random – that is, the second bit of their respective type values should be set. The type value of line 1 in block `l2` is written as `(type l2 1)`..

The declarations part of the sketch in Figure 1 is shown in Figure 2. The declarations include
(a) the domain of data value of the variables; in this example, we use bitvectors of length 5 as the domain of messages, etc. The choice of length is arbitrary: larger bitlengths would mean more computational resources would be required to solve the synthesis problem, but smaller bitlengths could lead to synthesis of schemes that do not work for arbitrary sizes.
(b) the domain of type value of the variables; in this example, we again use bitvectors of length 5 as the domain of the "type" value of program variables. The meaning of the 5 bits was explained before.
(c) the concrete function definitions that update the data value of the variables; in this example, we have used bitwise exclusive-or as the definition of `oplus` and `oplusr`, bitvector rotate right by 2 as the definition of `G` and bitvector rotate left by 3 as the definition of `H`.
(d) the type constraints induced by function applications; in this example, the definitions `tG`, `tH`, `toplus`, `toplusr`, and `tidentity` list the constraints on the types of the inputs and outputs of the five functions in the library.

All declarations are written in valid Yices syntax. Currently, the $\exists\forall$ solver for Yices works well on Boolean and bitvectors, and hence all data values are bitvectors and functions are mapped to functions

```
(decls
 (define-type typ (bitvector 5))
 (define-type word (bitvector 5))
 (define fG::(-> word word) (lambda (x::word) (bv-rotate-right x 2)))
 (define fH::(-> word word) (lambda (x::word) (bv-rotate-left x 3)))
 (define foplus::(-> word word word) (lambda (x::word y::word) (bv-xor x y)))
 (define foplusr::(-> word word word) (lambda (x::word y::word) (bv-xor x y)))
 (define fidentity::(-> word word) (lambda (x::word) x))
 (define ism::(-> typ bool) (lambda (x::typ) (bit x 0)))
 (define isr::(-> typ bool) (lambda (x::typ) (not (bit x 0))))
 (define isrand::(-> typ bool) (lambda (x::typ) (bit x 1)))
 (define istopg::(-> typ bool) (lambda (x::typ) (bit x 2)))
 (define istoph::(-> typ bool) (lambda (x::typ) (bit x 3)))
 (define istopx::(-> typ bool) (lambda (x::typ) (bit x 4)))
 (define tG::(-> typ typ bool) (lambda (x::typ y::typ)
  (and (isr x) (ism y) (<=> (isrand x) (isrand y)) (not (istoph x))
    (istopg y) (not (istoph y)) (not (istopx y)))))
 (define tH::(-> typ typ bool) (lambda (x::typ y::typ)
   (and (ism x) (isr y) (<=> (isrand x) (isrand y)) (not (istopg x))
    (istoph y) (not (istopg y)) (not (istopx y)))))
 (define toplusr::(-> typ typ typ bool) (lambda (x::typ y::typ z::typ)
   (and (or (and (ism x) (ism y) (ism z)) (and (isr x) (isr y) (isr z)))
    (not (istopg z)) (not (istoph z)) (istopx z) (not (istopx y))
    (or (istopg x) (istoph x) (istopg y) (istoph y)) (not (istopx x))
    (<=> (and (or (isrand x) (isrand y)) (/= x y)) (isrand z)))))
 (define toplus::(-> typ typ typ bool) (lambda (x::typ y::typ z::typ)
   (and (or (and (ism x) (ism y) (ism z)) (and (isr x) (isr y) (isr z)))
     (not (istopg z)) (not (istoph z)) (istopx z)
     (or (istopg x) (istoph x) (istopg y) (istoph y))
     (<=> (and (or (isrand x) (isrand y)) (/= x y)) (isrand z)))))
 (define tidentity::(-> typ typ bool) (lambda (x::typ y::typ) (= x y)))
)
```

Figure 2: Declarations used in the sketch for synthesizing various padding-based encryption schemes.

$$f(G(r)||(G(r) \oplus m))$$
$$f(r||(G(r) \oplus m))$$
$$f(G(r \oplus H(m))||(G(r \oplus H(m)) \oplus m))$$
$$f((r \oplus H(m))||(G(r \oplus H(m)) \oplus m))$$
$$f((G(r) \oplus m)||(H(G(r) \oplus m) \oplus r))$$

Figure 3: Some automatically synthesized padding-based encryption schemes.

on bitvectors. Note that program synthesis is done using the specified type and function declarations. These interpretations should be picked carefully so that they satisfy (exactly) the algebraic relations the actual functions satisfy. This may not be possible always, in which case one can choose interpretations that are likely to lead to general solutions.

Synthesized encryption schemes

We can now use our tool to synthesize different padding schemes using different values for the two parameters `na` and `nb`. We can use the tool to generate different solutions for the same values of the parameters.

Some example synthesized schemes are shown in Figure 3. Again, we do not show the padding with 0 that is required to make arguments reach the required bitvector length. Note that the OAEP scheme [BR94] is also generated using $na = 4$ and $nb = 4$ and is shown in Figure 3 as the last scheme. But smaller padding-based schemes were also found by the tool. We could potentially run the tool for larger values of na and nb to obtain more such secure schemes.

5 Summary

We presented an approach for synthesizing secure cryptographic protocols. We first defined a sketching language that can be used to specify library functions from which a scheme needs to be generated. The language allows features that can be used to further prune the search space of all valid programs. We translate the synthesis problem in the sketching language to an $\exists\forall$ Yices formula, and use the yices $\exists\forall$ SMT solver to solve the constraint and obtain a possible program. We used our language and the accompanying synthesis tool to synthesize various padding-based encryption schemes.

Our plan is to apply the approach to synthesize more complicated

protocols, such as the protocols for oblivious transfer. Synthesis results similar to the ones reported here have also been obtained using Zoocrypt [BCK+13]. We also plan to release the tool, the sketches, and the synthesized schemes on a public webpage in the near future.

References

[BCK+13] G. Barthe, J. M. Crespo, C. Kunz, B. Schmidt, B. Gregoire, Y. Lakhnech, and S. Zanella-Beguelin. Fully automated analysis of padding-based encryption in the computational model, 2013. `http://www.easycrypt.info/zoocrypt/`.

[BR94] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology, EUROCRYPT*, volume LNCS 950, 1994.

[GJTV11] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. PLDI*, 2011.

[SLRBE05] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[SRI] SRI International. *Yices: An SMT solver*. `http://yices.csl.sri.com/`.