Generating Formal Models with the LLVM Symbolic Simulator

Galois, Inc. | 421 SW 6th Avenue, Suite 300 | Portland, OR 97204

Introduction

This document describes how to use the Galois LLVM Symbolic Simulator's command-line interface, `lss`, to generate a formal model of a cryptographic algorithm compiled to LLVM from C, and to compare that model against a reference specification to verify that the LLVM version is correct. This brings benefits such as allowing programmers to experiment with efficient, customized implementations of an algorithm while retaining confidence that the changes do not affect the overall functionality.

We assume some knowledge of C, a basic understanding of LLVM, and a passing familiarity with cryptography. However, we do not assume familiarity with symbolic simulation, formal modeling, or theorem proving. We assume that the user has installed the LLVM 3.0 or 3.1 toolchain (available from `http://llvm.org/releases`). Additionally, we assume that the user has installed the Cryptol tool set and the ABC logic synthesis system from UC Berkeley if she wishes to complete the equivalence checking portion of the tutorial. Installation and configuration of those tools is outside the scope of this tutorial.

In the examples of interaction with the simulator and other tools, lines beginning with a hash mark (`#`) or short text followed by an angle bracket (such as `abc 01>`) indicate command-line prompts, and the following text is input provided by the user. All other monospaced text is the output of the associated tool.

Setting up the Environment

Ensure that `clang`, `llvm-dis`, `llvm-ld`, `lvm-link`, etc., are in your path; these tools are part of the standard LLVM distribution. Similarly, make sure that the `lss` executable bundled with this document is in your path.

All source code used by this tutorial can be found within the `tutorial/code` subdirectory of the release. There is a Makefile in that directory that can be used to compile everything with `clang`, link things together, and the `check` target can be used to start the equivalence checking process.

Symbolic Simulation

The LLVM Symbolic Simulator takes the place of a typical post-compilation execution environment, but makes it possible to reason about the behavior of programs on a wide range of potential inputs, rather than a fixed set of test vectors. A standard post-compilation execution environment for, e.g., clang-compiled programs, runs programs on concrete input values, producing concrete results. Symbolic simulation works similarly, but allows inputs to take the form of symbolic variables that represent arbitrary, unknown values. The result is then a mathematical formula that describes the output of the program in terms of the symbolic input variables.

Given a formula representing a program's output, we can then either evaluate that formula with specific values in place of the symbolic input variables to get concrete output values, or compare the formula to another, using known mathematical transformations to prove that the two are equivalent.

One downside of symbolic simulation, however, is that it cannot easily handle interaction with the outside world. If a program simply takes an input, performs some computation, and produces some output, symbolic simulation can reliably construct a model of that computation. Cryptographic algorithms typically fall into this category. In cases where a program does some computation, produces some output, reads some more input, and continues the computation based on the new information, we either need a model of the outside world, or to assume that the input could be completely arbitrary, and reason about what the program would do for any possible input.

The LLVM Symbolic Simulator can evaluate simple output methods, such as the ubiquitous `printf()`. If the value printed depends on the value of a symbolic input variable, a bit-level textual representation is shown with '?' marks to denote symbolic bits.

The LLVM Symbolic Simulator provides a set of special functions for performing operations on symbolic values, and for emitting the formal model representing symbolic values of interest. The rest of this tutorial will demonstrate how to use these methods to generate a formal model of a simple AES128 implementation, and then compare this model to a reference specification.

Supplying Symbolic Input to AES128

In the code subdirectory of the directory containing this tutorial, there are some files and directories of note. The file `aes128BlockEncrypt_driver.c` contains the driver code that sets up the symbolic inputs to the AES128 block encrypt function; `aes128BlockEncrypt.[ch]` contains the block encrypt function implementation; the `sym-api` subdirectory contains copies of the ‘`sigfuns API'' header and implementation files; finally, theref`‘ subdirectory includes the cryptol reference specification and equivalence checking script.

Let's start with `aes128BlockEncrypt_driver.c`. This code creates a simple wrapper around the `aes128BlockEncrypt` function. We will use this source file as a running example, and step through what each line means in the context of symbolic simulation.

Note that the driver includes `sym-api.h` to get access to the special functions used to interact with the symbolic simulator.

The first two variable declarations in the `main` function are those with the most relevance to symbolic simulation.

```
SWord32 *pt  = lss_fresh_array_uint32(4, 0x8899aabbUL);
SWord32 *key = lss_fresh_array_uint32(4, 0x08090a0bUL);
```

These declarations each create a new array with entirely symbolic contents, intended to be used as the plaintext and key inputs to the block encrypt function. Their size is fixed (4 elements of type `uint32_t`, or 128 bits), but each element is a symbolic term representing an arbitrary 32-bit unsigned value. The second parameter to the `lss_fresh_array_uint32` function is the initial value for each element if this code is executed in a *concrete* context (i.e., not via `lss`). We can ignore it for our purposes here.

The next declaration is standard C, and creates an uninitialized 128-bit array for holding the ciphertext result. This does not hold symbolic values at the time of declaration, but the values stored inside it will be symbolic if they depend on the values in `pt` or `key`, as we will expect them to.

Next, calculation of the AES128 ciphertext occurs in a typical fashion, by calling the block encrypt function and passing both in parameters and out parameters by pointer.

```
aes128BlockEncrypt(pt, key, ct);
```

The next and final statement does the work of creating a formal model from the AES128 block encrypt function. This function instructs the symbolic simulator to generate a formula that describes how the elements of `ct` depend on the elements of `pt` and `key`, and then writes that formula to a file called `aes.aig`:

```
lss_write_aiger_array_uint32(ct, 4, "aes.aig");
```

Ultimately, we want to find a formal model that describes the output of the AES128 block encrypt function, in terms of whatever symbolic inputs it happens to depend on. In this case, this includes every byte of the input key and plaintext. However, for some algorithms, it could include only a subset of the symbolic variables in the program.

The formal model that the simulator generates takes the form of an And-Inverter Graph (AIG), which is a way of representing a boolean function purely in terms of the logical operations `and'' and`not". The simplicity of this representation makes the models easy to reason about, and to compare to models from other sources. However, the same simplicity means that the model files can be very large in comparison to the input source code.

Running the Simulator

To generate a formal model from the example described in the previous section, we can use the `lss` command, which forms the command-line front end of the LLVM Symbolic Simulator. At minimum, it needs to know where to find a fully-linked LLVM bitcode containing a `main` function. Typing `make` inside the code subdirectory will produce a file called `aes.bc` that meets these criteria.

The following command will then run `lss` to create a formal model:

```
# lss aes.bc
```

This will result in a file called `aes.aig` that can be further analyzed using a variety of tools, including the Galois Cryptol tool set and the ABC logic synthesis system from UC Berkeley.

Viewing the Intermediate Representations

When working with input C source and linked LLVM bitcodes, it can be usefult to inspect two underlying intermediate representations: LLVM itself, and LLVM-Sym, the language to which LLVM programs are translated by `lss`. For example, let's say we've compiled the AES128 driver code by hand:

```
clang -emit-llvm -I./sym-api -c aes128BlockEncrypt_driver.c \
  -o aes128BlockEncrypt_driver.bc
```

To see the LLVM assembly langugage representation of this program, one can use `llvm-dis`, which produces a `.ll` file containing the disassembled bitcode:

```
llvm-dis aes128BlockEncrypt_driver.bc
```

Similarly, to view the disassembly after it has been transformed into the LLVM-Sym representation, the `--xlate` option may be supplied to `lss`. This option causes the LLVM-Sym representation to be displayed to stdout; the user may redirect output when convenient:

```
lss --xlate aes128BlockEncrypt_driver.bc > aes128BlockEncrypt_driver.xlate
```

When viewing debugging output from `lss`, program locations are currently shown in reference to the LLVM-Sym representation, so it is sometimes useful to view that representation alongside `lss` feedback.

Verifying the Formal Model Using Cryptol

One easy way to verify an LLVM implementation against a reference specification is via the Cryptol tool set. Cryptol is a domain-specific language created by Galois for the purpose of writing high-level but precise specifications of cryptographic algorithms [@cryptol]. The Cryptol tool set has built-in support for checking the equivalence of different Cryptol implementations, as well as comparing Cryptol implementations to external formal models.

This tutorial comes with a handful of Cryptol files, most notably `Rijndael.cry` and `equivAES.cry`. The former is a Cryptol specification of the Rijndael cipher. In particular, it contains the function `blockEncrypt` which should have equivalent functionality to the `aes128BlockEncrypt` function in our C source. Well, nearly equivalent: we write a small wrapper around this function, as can be seen in `equivAES.cry` that reorders the bytes of the inputs and outputs as needed to the form expected by the `blockEncrypt` function. This essentially makes the calling convention and data layout assumptions of both functions identical before attempting to show equivalence.

To compare the functionality of the two implementations, we have several options. As mentioned earlier, formal models can be evaluated on concrete inputs, or compared to other formal models using proof techniques to show equivalence for all possible inputs. The contents of `equivAES.cry` show how to compare the formal model of the C implementation against the Cryptol reference specification.

```
...
extern AIG llvm_aes("../aes.aig") : ([4][32], [4][32]) -> [4][32];
theorem MatchesRef : {pt key}. llvm_aes (pt, key) == blockEncryptref_c (pt, key);
blockEncryptref_c : ([4][32], [4][32]) -> [4][32];
blockEncryptref_c (x, y) = ...
...
```

The `extern AIG` line makes the contents of `aes.aig` available as a function called `llvm_aes` that takes two 4x32-bit values as input and produces one 4x32-bit value as output. Finally, the second line states a theorem: that the functions `llvm_aes` and `blockEncryptref_c` should produce the same ciphertext for all possible key and plaintext inputs.

We can load `equivAES.cry` into the Cryptol tool set, yielding the following output:

```
# cryptol equivAES.cry
Cryptol version 1.8.22, Copyright (C) 2004-2011 Galois, Inc.
                                            www.cryptol.net
Type :? for help
Loading "equivAES.cry"..
  Including "Rijndael.cry"..
  Including "Cipher.cry"..
  Including "AES.cry".. Checking types..
  Loading extern aig from "../aes.aig".. Processing.. Done!
*** Auto quickchecking 1 theorem.
*** Checking "MatchesRef" ["equivAES.cry", line 5, col 1]
Checking case 100 of 100 (100.00%)
100 tests passed OK
[Coverage: 0.00%. (100/115792089237316195423570985008687907853269984665640564056...)]
```

By default, the Cryptol interpreter processes every `theorem` declaration by automatically evaluating the associated expression on a series of random values, and ensuring that it always yields “true”. In this case, it tried 100 random key and plaintext values, and the two functions yielded the same output in each case. However, the number of possible inputs is immense, so 100 test cases barely scratches the surface.

To gain a higher degree of confidence that the functions do have the same functionality for all possible inputs, we can attempt to prove their equivalence deductively. From Cryptol’s command line:

```
equivAES> :set symbolic
equivAES> :prove MatchesRef
Q.E.D.
equivAES> :fm blockEncryptref_c "aes-ref.aig"
```

This tells the Cryptol interpreter to switch to symbolic simulation mode (which is one way it can generate formal models from Cryptol functions) and then attempt to prove the theorem named `MatchesRef`. On a reasonably modern machine (as of August 2012), the proof should complete in less than 30 minutes. The output `Q.E.D.` means that the proof was successful.

Finally, the `:fm` command tells the interpreter to generate a formal model of the function `blockEncryptref_c` and store it in `aes-ref.aig`. We can then use this formal model to perform the same proof using an external tool such as ABC, as described next.

Note that the above actions can be performed by running the `check` target of the Makefile in the code subdirectory.

Verifying the Formal Model Using ABC

ABC is a tool for logic synthesis and verification developed by researchers at UC Berkeley [@abc]. It can perform a wide variety of transformations and queries on logic circuits, including those in the AIG form discussed earlier.

As an alternative approach to the equivalence check from the previous section, we can use the `cec` command in ABC to attempt to prove the model generated by the symbolic simulator equivalent to the model generated from the Cryptol specification.

```
# abc
UC Berkeley, ABC 1.01 (compiled Oct 26 2010 13:07:15)
abc 01> cec ./aes.aig ./aes-ref.aig
Networks are equivalent.
abc 01>
```

Generating DIMACS CNF Models

In addition to AIG models, LSS can generate models in DIMACS CNF format for boolean-valued expressions. These models can then be checked for validity using a SAT solver of your choice.

The AES driver in `aes128BlockEncrypt_driver.c` contains the following line:

```
lss_write_cnf(pt[0] != ct[0] &&
              pt[1] != ct[1] &&
              pt[2] != ct[2] &&
              pt[3] != ct[3], "noleaks.cnf");
```

This call instructs LSS to write CNF clauses built from the expression given as the first argument (in this case, an assertion that the plain text and cipher text are different) into the file given as the second argument (`noleaks.cnf`). LSS uses the convention that unsatisfiability of the CNF model corresponds to validity of the given expression.

We can now use a SAT solver to prove that AES will never encrypt plain text into identical cipher text (given sufficient time!):

```
# picosat noleaks.cnf
```