CryptoLine: A Tutorial

Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang

1. Introduction

CryptoLine is a verification tool chain for cryptographic assembly programs. It contains the CryptoLine model checker and tools for building models from executable binary codes. CryptoLine is designed for verifying algebraic properties in cryptographic programs. It has been used to verify cryptographic assembly programs in OpenSSL and blst, PQClean, and pqm4.

In this tutorial, we explain notable features of CryptoLine through two running examples from x86_64 implementations for NIST P-256 curve in OpenSSL. Specifically, NIST P-256 curve uses the finite field $\mathbb{Z}_{p256}$ where $p256$ is

`0xffffffff00000001 0000000000000000 00000000ffffffff ffffffffffffffff`.

We will verify the addition (`ecp_nistz256_add`) and Montegomery multiplication (`ecp_nistz256_mul_montx`) over the field $\mathbb{Z}_{p256}$ from `crypto/ec/asm` in OpenSSL. All CryptoLine codes can be found in `examples/openss/ecp_nistz256/x86_64` from the CryptoLine distribution.

2. CryptoLine Overview

To verify cryptographic programs with CryptoLine, a verifier has to construct program models written in the CryptoLine language. Such program models could be written manually. Manual construction nevertheless could be tedious or even deviant from real cryptographic programs. To help verifiers, CryptoLine provides a Python script `itrace.py` to extract traces from running cryptographic programs in gdb. Verifiers will obtain traces of cryptographic programs as executed on hardware. Since traces are extracted from gdb, they are sequences of assembly instructions from the underlying hardware architecture. To convert such traces to CryptoLine models, CryptoLine provides another Python script `to_zdsl.py` to help verifiers translate assembly instructions to CryptoLine commands. Through `itrace.py` and `to_zdsl.py`, accurate CryptoLine models can be constructed rather easily. They are indispensable in practice.

Based on the CryptoLine models generated by `to_zdsl.py`, verifiers need to annotate models with input assumptions (or *pre-conditions*) and output requirements (or *post-conditions*). Additional annotations are often required to guide CryptoLine verification engines as well. After necessary annotations are added, the CryptoLine verification tool will prove if all post-conditions must hold under pre-conditions automatically. If CryptoLine fails to prove post-conditions, hints can be found in CryptoLine log files. Verifiers can decide whether more annotations are needed or bugs are found from the hints.

The CryptoLine verification tool employs two engines for proving different properties about CryptoLine models. The SMT-based engine calls an external SMT QFBV (Satisfiability Modulo Quantifier-Free Bit-Vector Theory) solver to prove range properties. The CAS-based engine calls an external CAS (Computer Algebra System) to prove algebraic properties. Generally, the SMT-based engine is automatic but unsuitable for complex non-linear algebraic properties. The CAS-based engine on the other hand is much better for algebraic properties but requires more annotations. Verifiers need to choose which engine to use by their discretion.

3. Installing CryptoLine

CryptoLine is an open-sourced project available at `https://github.com/fmlab-iis/cryptoline`. To download its source code, type

```
$ git clone https://github.com/fmlab-iis/cryptoline.git
```

CryptoLine is written in OCaml and requires the OCaml package manager opam, external SMT solvers, and CAS's. Use the following commands to install the opam package manager, the SMT solver boolector, and the CAS Singular in ubuntu:

```
$ sudo apt-get install opam boolector singular-ui
```

Additional OCaml libraries are needed to compile CryptoLine. To initialize opam and install these libraries, use the following commands:

```
$ opam init --disable-sandboxing    # initialize opam
$ eval $(opam env)                  # set up environment variables
$ opam install dune lwt_ppx zarith  # install additional OCaml packages
```

Finally, go to the CryptoLine directory and compile it with the following commands:

```
$ cd cryptoline
$ dune build
```

The built CryptoLine binaries are in the `_build/_default` directory. To make a symbolic link for convenience and check if everything works fine, try

```
$ ln -s _build/default/cv.exe
$ cv.exe -v -isafety examples/openssl/ecp_nistz256/ecp_nistz256_mul_mont.cl
```

If you see messages similar to the following, you are all set!

```
Parsing Cryptoline file:                [OK]            0.002074 seconds
Checking well-formedness:               [OK]            0.000732 seconds
Transforming to SSA form:               [OK]            0.000278 seconds
Normalizing specification:              [OK]            0.000017 seconds
Rewriting assignments:                  [OK]            0.000229 seconds
Verifying program safety:
         Cut 0
             Round 1 (32 safety conditions, timeout = 300 seconds)
                 Safety condition #3    [OK]
                 Safety condition #4    [OK]
                 Safety condition #0    [OK]
                 ...
                 Safety condition #28   [OK]
                 Safety condition #31   [OK]
         Overall                        [OK]            5.185277 seconds
Verifying range specification:          [OK]            2.155957 seconds
Rewriting value-preserved casting:      [OK]            0.000023 seconds
Verifying algebraic specification:      [OK]            0.107180 seconds
Verification result:                    [OK]            7.452392 seconds
```

4. Running Examples from OpenSSL

The CryptoLine verification tool is a model checker. That is, it checks specified properties about models. To verify cryptographic programs with CryptoLine, we need to write a model for the program and specify properties about the model. In this tutorial, we will verify the x86_64 assembly subroutines `ecp_nistz256_add` and `__ecp_nistz256_mul_montx` from `ecp_nistz256-x86_64.pl` in crypto/ec/asm from OpenSSL. The subroutine ecp_nistz256_add takes two inputs a and b from the field $\mathbb{Z}_{p256}$, computes their sum $c \equiv a + b \mod p256$, and stores c in memory. The API for __ecp_nistz256_mul_montx takes two inputs a, b from $\mathbb{Z}_{p256}$, computes their Montgomery product $c \equiv a \times b \times 2^{-256} \mod p256$, and stores c in memory. They are used by OpenSSL on x86_64 by default. We will see important features of CryptoLine in these running examples.

4.1. ecp_nistz256_add**.**

4.1.1. *Model Construction.* The easiest way to construct accurate CryptoLine models is by extracting traces from execution. To do so, we need to build an executable binary for the cryptographic program under verification. Let us write a simple C program which calls the two assembly subroutines.

```
#include <stdint.h>
#define P256_LIMBS 4

typedef uint64_t BN_ULONG;

/* Modular add: res = a+b mod P   */
void ecp_nistz256_add(BN_ULONG res[P256_LIMBS],
                      const BN_ULONG a[P256_LIMBS],
                      const BN_ULONG b[P256_LIMBS]);
/* Montgomery mul: res = a*b*2^-256 mod P */
void ecp_nistz256_mul_mont(BN_ULONG res[P256_LIMBS],
                           const BN_ULONG a[P256_LIMBS],
                           const BN_ULONG b[P256_LIMBS]);

int main (void) {
  BN_ULONG a[P256_LIMBS], b[P256_LIMBS], r[P256_LIMBS];

  /* Modular add: res = a+b mod P   */
  ecp_nistz256_add(r, a, b);
  /* Montgomery mul: res = a*b*2^-256 mod P */
  ecp_nistz256_mul_mont(r, a, b);

  return 0;
}
```

An executable binary can be built with the following command (libcrypto.a is from OpenSSL):

```
$ gcc -o top top.c libcrypto.a
```

CryptoLine provides the Python script itrace.py to extract execution traces from gdb. Write $(CL_HOME) for the root directory of CryptoLine. The execution trace of ecp_nistz256_add is extracted with the following command:

```
$ $CL_HOME/scripts/itrace.py top ecp_nistz256_add ecp_nistz256_add.gas
```

The first argument is the name of the executable binary top, the second argument is the name of the subroutine ecp_nistz256_add, and the third argument is the name of the output file (ecp_nistz256_add.gas). The content of ecp_nistz256_add.gas looks like

```
ecp_nistz256_add:
# %rdi = 0x7fffffffda00
# %rsi = 0x7fffffffd9c0
# %rdx = 0x7fffffffd9e0
# %rcx = 0x7fffffffd9c0
# %r8  = 0x555555580c70
```

```
# %r9  = 0x7ffef3ff00000000
#! -> SP = 0x7fffffffd9b8
push   %r12                        #! EA = L0x7fffffffd9b0; ...
push   %r13                        #! EA = L0x7fffffffd9a8; ...
mov    (%rsi),%r8                  #! EA = L0x7fffffffd9c0; ...
xor    %r13,%r13                   #! PC = 0x55555557c327
mov    0x8(%rsi),%r9               #! EA = L0x7fffffffd9c8; ...
mov    0x10(%rsi),%r10             #! EA = L0x7fffffffd9d0; ...
mov    0x18(%rsi),%r11             #! EA = L0x7fffffffd9d8; ...
lea    -0x33d(%rip),%rsi           # 0x55555557c000 ...
add    (%rdx),%r8                  #! EA = L0x7fffffffd9e0; ...
adc    0x8(%rdx),%r9               #! EA = L0x7fffffffd9e8; ...
         ...
```

The script `itrace.py` shows the register contents when `ecp_nistz256_add` is called. By calling convention, `%rdi`, `%rsi`, and `%rdx` contains the values of the first three arguments to `ecp_nistz256_add`. In this case, we see the pointers `r[]`, `a[]`, and `b[]` are `0x7fffffffda00`, `0x7fffffffd9c0`, and `0x7fffffffd9e0` respectively. For each instruction, `itrace.py` moreover reports the effective address of its argument (`EA`), the value stored in the address (`Value`), and the program counter (`PC`). The `itrace.py` script is necessarily architecture-dependent. It currently supports ARM, MIPS, RISC-V, and x86. It can also connect to remote GDB through a serial port. It has been used to extract traces from ARM Cortex-M4 development boards.

Our next step is to convert x86_64 instructions to corresponding CryptoLine commands. CryptoLine provides the Python script `to_zdsl.py` to convert instructions by writing translation rules. Translation rules are specified in the beginning of execution traces (`ecp_nistz256_add.gas`). They must start with `#`. For `to_zdsl.py`, strings prefixed by `%%` are matched differently. We start with rules for such strings:

```
#! $1c(%rdi) = %%EA
#! (%rdi) = %%EA
#! $1c(%rsi) = %%EA
#! (%rsi) = %%EA
#! $1c(%rdx) = %%EA
#! (%rdx) = %%EA
#! %r$1c = %%r$1c
#! %rax = %%rax
#! %rcx = %%rcx
#! %rdx = %%rdx
```

The left-hand side denotes the string in the trace to be matched. The right-hand side denotes how to rewrite the match strings. The notations `$1c`, `$2c`, and so on match constants. The first six rules rewrite memory references to `%%EA`. The last four rules add an additional `%` to registers.

For each x86_64 instruction in `ecp_nistz256_add.gas`, a translation rule is needed for convertion. Instructions starting with `#` will be skipped. Let us look at the rule for the x86_64 `xor` instruction:

```
#! xor $1v, $1v -> mov $1v 0@uint64
```

The left-hand side denotes the pattern in the trace to be matched. The right-hand side denotes how to rewrite the matched pattern. The strings prefixed with `%%` are matched by `$1v`, `$2v`, and so on. The x86_64 instructions from the trace separate arguments by `,`. They moreover write sources before destinations. The CryptoLine commands however write destinations before sources. In this rule, the same register appears in both arguments to the `xor` instruction. The register is set to zero effectively. The rule thus sets the CryptoLine variable to `0@uint64`. Note that all constants in CryptoLine must be given a type. The notation `0@uint64` denotes the constant zero of the unsigned 64-bit integer type.

Our next rules are for the x86_64 `mov` instruction:

```
#! mov $1v, $2v -> mov $2v $1v
#! mov $1ea, $2v -> mov $2v $1ea
#! mov $1v, $2ea -> mov $2ea $1v
```

The string `%%EA` is matched by `$1ea`, `$2ea`, and so on. Recall that `itrace.py` reports the effective address appeared in each instruction. The script `to_zdsl.py` will also rewrite each matched `%%EA` to the corresponding effective address. In CryptoLine, there is no memory model. All memory stores are modeled by CryptoLine variables. By convention, the memory store with the address ***addr*** are modeled by the variable `L`***addr***. Using effective addresses as variable names allows us to avoid tedious address computation. It greatly simplifies our model construction. Our rules simply swap the order of source and destination.

The rules for arithmetic instructions are also straightforward:

```
#! add $1ea, $2v -> adds carry $2v $2v $1ea
#! adc $1ea, $2v -> adcs carry $2v $2v $1ea carry
#! adc \$0x0, $1v -> adc $1v $1v 0@uint64 carry
#! sub $1ea, $2v -> subb carry $2v $2v $1ea
#! sbb $1ea, $2v -> sbbs carry $2v $2v $1ea carry
#! sbb \$0x0, $1v -> sbbs carry $1v $1v 0@uint64 carry
```

In CryptoLine commands, all arguments are explicit. Consider, for instance, the x86_64 `add` instruction. It puts the sum of the two arguments in the destination *and* sets the carry flag implicitly. In CryptoLine, two commands are provided for addition: `add` updates the destination with the sum of sources; `adds` updates the destination with the sume of sources *and* the destination flag with carry. In our rules, the CryptoLine variable `carry` denotes the carry flag. We therefore use the `adds` and `adcs` for the x86_64 `add` and `adc` instructions respectively. Finally, CryptoLine provides subtraction commands for carry or borrow flags. The `subb` commands updates the destination with the difference of sources and the destination flag with *borrow*; the `subc` commands updates the destination with the difference of sources and the destination flag with *carry*. In x86_64, the `sub` instruction updates the carry flag with borrow. Our rule hence uses the CryptoLine `subb` command.

Our last rule is for the x86_64 `cmovb` instruction:

```
#! cmovb $1v, $2v -> cmov $2v carry $1v $2v
```

The instruction sets the destination to the value of source if the carry flag is true. The CryptoLine command `cmov` sets the destination to the value of the first source if the source flag is true; it sets the destination to the value of the second source otherwise.

After putting all translation rules in the beginning of `ecp_nistz256_add.gas`, the x86_64 trace can be converted to a CryptoLine model with following command:

```
$ $CL_HOME/script/to_zdsl.py ecp_nistz256_add.gas > ecp_nistz256_add.cl
```

The content of `ecp_nistz256_add.cl` looks like:

```
proc main (L0x55555557c000, ...) =
{
  true
  &&
  true
}

(* #! -> SP = 0x7fffffffd9b8 *)
#! 0x7fffffffd9b8 = 0x7fffffffd9b8;
```

```
(* #push   %r12                   #! ...                    *)
#push   %%r12                     #! ...
(* #push   %r13                   #! ...                    *)
#push   %%r13                     #! ...
(* mov    (%rsi),%r8              #! EA = L0x7fffffffd9c0; ... *)
mov r8 L0x7fffffffd9c0;
(* xor    %r13,%r13               #! PC = 0x55555557c327 *)
mov r13 0@uint64;
...
(* #repz retq                     #! PC = 0x55555557c39c *)
#repz retq                        #! ...

{
  true
  &&
  true
}
```

The notation `proc main (...) =` denotes the main subroutine in CryptoLine. The arguments to the main subroutine are the uninitialized variables reported by `to_zdsl.py`. In this case, they are memory stores for input arguments and constants used in `ecp_nistz256_add`. The expression in the brackets `true && true` denote the pre-condition. It is followed by CryptoLine commands for x86_64 instructions. Each x86_64 instruction is put in CryptoLine comments prefixed by `#` or enclosed by `(*` and `*)`. It is then followed by the CryptoLine command generated with the translation rules. Finally, the expression in the ending brackets `true && true` denotes the post-condition.

Let us first make arguments more readable by replacing the main subroutine declaration with:

```
proc main (uint64 a0, uint64 a1, uint64 a2, uint64 a3,
           uint64 b0, uint64 b1, uint64 b2, uint64 b3,
           uint64 m0, uint64 m1, uint64 m2, uint64 m3) =
```

We will use `a`'s and `b`'s for the two input elements and `m`'s for the modulo $p256$. The pre-condition for the main subroutine is

```
{
  true
  &&
  and [ m0 = 0xffffffffffffffff@64, m1 = 0x00000000ffffffff@64,
        m2 = 0x0000000000000000@64, m3 = 0xffffffff00000001@64,
        limbs 64 [a0, a1, a2, a3] <u limbs 64 [m0, m1, m2, m3],
        limbs 64 [b0, b1, b2, b3] <u limbs 64 [m0, m1, m2, m3]
  ]
}
```

Recall that two different engines are employed in CryptoLine. In order to differentiate properties passed to different engines, all CryptoLine properties are of the form `P && Q`: P is passed to CAS's and Q is to SMT solvers. For `ecp_nistz256_add`, the SMT-based engine suffices because it does not involve non-linear computation. Our pre-condition simply passes `true` to the CAS-based engine. For the SMT-based engine, the pre-condition assumes `m`'s to be the modulo $p256$. The field elements `a`'s and `b`'s moreover are less than $p256$ in the unsigned representation. The expression `limbs` n $[a_0, a_1, \ldots, a_m]$ is short for

$$a_0 \times 2^{0\times n} + a_1 \times 2^{1\times n} + \cdots + a_m \times 2^{m\times n}.$$

Our next step is to put input field elements and constants to correspond memory stores. By consulting `ecp_nistz256_add.gas`, we add the following CryptoLine commands after the precondition:

```
mov L0x7fffffffd9c0 a0; mov L0x7fffffffd9c8 a1;
mov L0x7fffffffd9d0 a2; mov L0x7fffffffd9d8 a3;

mov L0x7fffffffd9e0 b0; mov L0x7fffffffd9e8 b1;
mov L0x7fffffffd9f0 b2; mov L0x7fffffffd9f8 b3;

mov L0x55555557c000 0xffffffffffffffff@uint64;
mov L0x55555557c008 0x00000000ffffffff@uint64;
mov L0x55555557c010 0x0000000000000000@uint64;
mov L0x55555557c018 0xffffffff00000001@uint64;
```

At the end of `ecp_nistz256_add.cl`, we copy the result from memory stores by the following command:

```
mov c0 L0x7fffffffda00; mov c1 L0x7fffffffda08;
mov c2 L0x7fffffffda10; mov c3 L0x7fffffffda18;
```

Finally, we specify the post-condition for the SMT-based engine:

```
{
  true &&
  and [ eqmod limbs 64 [c0, c1, c2, c3, 0@64]
              limbs 64 [a0, a1, a2, a3, 0@64] +
              limbs 64 [b0, b1, b2, b3, 0@64]
              limbs 64 [m0, m1, m2, m3, 0@64],
        limbs 64 [c0, c1, c2, c3] <u limbs 64 [m0, m1, m2, m3] ]
}
```

The post-condition states that the output field element `c`'s is congruent to the sum of input field elements modulo $p256$, and the output field element is less than the modulo in the unsigned representation. Note that the congruence is computed with $5 \times 64 = 320$ bits instead of 256 bits.

4.1.2. *Verification.* We are ready to verify our CryptoLine model for `ecp_nistz256_add`. Try

```
$ $CL_HOME/cv.exe -v -isafety ecp_nistz256_add.cl
```

The transcript is shown below:

```
Parsing Cryptoline file:                [OK]            0.001247 seconds
Checking well-formedness:               [OK]            0.000218 seconds
Transforming to SSA form:               [OK]            0.000105 seconds
Normalizing specification:              [OK]            0.000097 seconds
Rewriting assignments:                  [OK]            0.000122 seconds
Verifying program safety:
         Cut 0
             Round 1 (1 safety conditions, timeout = 300 seconds)
                 Safety condition #0    [OK]
         Overall                        [OK]            0.044187 seconds
Verifying range specification:          [OK]            2.203067 seconds
Rewriting value-preserved casting:      [OK]            0.000023 seconds
Verifying algebraic specification:      [OK]            0.000412 seconds
Verification result:                    [OK]            2.249944 seconds
```

Congratulations! You have verified the x86_64 `ecp_nistz256_add` subroutine in OpenSSL successfully. As we have seen, CryptoLine provides useful scripts for model construction. They are not perfect and still require human intervention. Some practices will help verifiers get familiar with the verification flow.

Exercise: Construct a model for `ecp_nistz256_sub` in `ecp_nistz256-x86_64.pl` and verify it.

4.2. `ecp_nistz256_mul_mont`. Our next example is to verify the assembly subroutine `ecp_nistz256_mul_mont` from OpenSSL.[1] The assembly subroutine takes two field elements a and b in $\mathbb{Z}_{p256}$ as inputs, computes their Montgomery product c, and store c in memory. Mathematically, the inputs and output satisfy the following modular equation:

$$c \equiv a \times b \times 2^{-256} \mod p256, \text{ equivalently}, c \times 2^{256} \equiv a \times b \mod p256$$

4.2.1. *Model Construction.* The executable binary `top` built in the first example also calls the assembly subroutine. The trace for `ecp_nistz256_mul_mont` can be extracted by `itrace.py` with the same binary:

```
$ $CL_HOME/scripts/itrace.py top ecp_nistz256_mul_mont ecp_nistz256_mul_mont.gas
```

The trace `ecp_nistz256_mul_mont.gas` looks like the following:

```
ecp_nistz256_mul_mont:
# %rdi = 0x7fffffffd9f0
# %rsi = 0x7fffffffd9b0
# %rdx = 0x7fffffffd9d0
# %rcx = 0x7fffffffd9b0
# %r8  = 0x-9
# %r9  = 0xfffffffe
#! -> SP = 0x7fffffffd9a8
mov    $0x80100,%ecx           #! PC = 0x55555557d1e0
and    0x5e35(%rip),%ecx       # ...
push   %rbp                    #! EA = L0x7fffffffd9a0; ...
push   %rbx                    #! EA = L0x7fffffffd998; ...
       ...
```

By calling convention, we know the input field elements are stored at `0x7fffffffd9b0` and `0x7fffffffd9d0`; the output is stored at `0x7fffffffd9f0`. The subroutine uses more registers. Unsurprisingly, we need additional translation rules for memory addresses and registers.

```
#! $1c(%rdi) = %%EA
#! (%rdi) = %%EA
#! $1c(%rsi) = %%EA
#! (%rsi) = %%EA
#! $1c(%rdx) = %%EA
#! (%rdx) = %%EA
#! $1c(%rbx) = %%EA
#! (%rbx) = %%EA
#! -$1c(%rip) = %%EA
#! %r$1c = %%r$1c
#! %rax = %%rax
#! %rbx = %%rbx
#! %rcx = %%rcx
#! %rdx = %%rdx
#! %rbp = %%rbp
#! %eax = %%eax
```

Many translation rules for x86_64 instructions can be re-used. They are listed below:

[1]Depending on the x86_64 microarchitecture, the assembly subroutine `ecp_nistz256_mul_mont` has two implementations: `__ecp_nistz256_mul_montx` and `__ecp_nistz256_mul_montq`. We will verify `__ecp_nistz256_mul_montx` here.

```
#! add $1v, $2v -> adds carry $2v $2v $1v
#! adc $1v, $2v -> adcs carry $2v $2v $1v carry
#! cmovb $1v, $2v -> cmov $2v carry $1v $2v
#! mov $1c, $2v -> mov $2v $1c@uint64
#! mov $1v, $2v -> mov $2v $1v
#! mov $1ea, $2v -> mov $2v $1ea
#! mov $1v, $2ea -> mov $2ea $1v
#! sbb $1v, $2v -> sbbs carry $2v $2v $1v carry
```

Three rules are modified slightly. They are:

```
#! xor $1v, $1v -> mov $1v 0@uint64;\nclear carry;\nclear overflow
#! adc $1c, $2v -> adc $2v $2v $1c@uint64 carry
#! sbb $1c, $2v -> sbbs carry $2v $2v $1c@uint64 carry
```

The x86_64 `xor` instruction actually clears carry and overflow flags. This is not modeled previously but needed in `ecp_nistz256_mul_mont`, so the rule is modified accordingly. The string
`n` represents a line break. In `ecp_nistz256_mul_mont`, more constant literals are used. We therefore use `$1c` to match constants in the rules for `adc` and `sbb`.

Two additional addition instructions are used in `ecp_nistz256_mul_mont`. The `adcx` and `adox` instructions compute the sum with the carry and overflow flags as carry respectively. Their translation rules are similar to those for `adc`:

```
#! adcx $1v, $2v -> adcs carry $2v $2v $1v carry
#! adox $1v, $2v -> adcs overflow $2v $2v $1v overflow
```

The x86_64 `mulx` computes the product of the `rdx` register and the source. The 128-bit product is then stored in the destinations. The CryptoLine `mull` command computes the product of the last two arguments, stores the more significant half in the first argument and the less significant half in the second. We thus use the following rule for `mulx`:

```
#! mulx $1v, $2v, $3v -> mull $3v $2v rdx $1v
```

Finally, the x86_64 instruction `shlx` *`r s`* `d` and `shrx` *`r s d`* shifts the value of *`s`* to the left and right respectively by the value in *`r`*. The shifted result is stored in *`d`*. In CryptoLine, the `shl` *`d s c`* shifts the value of *`s`* by the constant *`c`* bits to the left. The `split` *`h l s c`* command splits *`s`* by the constant *`c`* into two parts: the lowest *`c`* bits are stored in *`l`* and other bits are stored in *`h`*. It is tempting to use the following rules:

```
#! shlx $1v, $2v, $3v -> shl $3v $2v $1v
#! shrx $1v, $2v, $3v -> split $3v dc $2v $1v
```

There is a problem in these rules. The `shl` and `split` commands only allow constant shifting and splitting. We need to change the variable `$1v` to a constant. After examining `ecp_nistz256_mul_mont`, we see the first argument of all `shlx` and `shrx` instructions is always `%r14`. Moreover, `%r14` is set to `$0x20` and never changed. We can ask CryptoLine to check the value of `$1v` is always 32 and then use 32 as the shifting and splitting constant. The CryptoLine `assert` *`P`* `&&` *`Q`* command checks both *`P`* and *`Q`* must be true. The verification fails if any of *`P`* or *`Q`* can be false. Consider the following rules:

```
#! shlx $1v, $2v, $3v -> assert $1v=32 && true;\nshl $3v $2v 32
#! shrx $1v, $2v, $3v -> assert $1v=32 && true;\nsplit $3v dc $2v 32
```

The `assert $1v=32 && true` command ensures `$1v` must be 32 at this location. If so, we use the constant 32 instead of the variable `$1v`. Note that we ask an external CAS to check if `$1v` is equal 32. If you would like to use the SMT-based engine, use `assert true && $1v=32@64` instead.

The translation rule for `shlx` nevertheless would not work. Safety conditions would fail during verification if they were used. To explain what safety conditions are, recall that CryptoLine employs two different engines. Every CryptoLine command therefore has two different interpretations: one for the SMT-based, the other for the CAS-base engine. The `shl` *`d s c`* command is interpreted as the logical left shift in bit-vector theory in the SMT-based engine. It is interpreted by the equation $d = s \times 2^c$ in the CAS-based engine. Two different interpretations need to be related, otherwise their results may differ unexpectedly. To relate both interpretations of `shl`, CryptoLine checks safety conditions to see if information might be lost in the command. For `shl`, the safety condition is that only zeros are shifted out. Thus, both interpretations coincide. In `ecp_nistz256_mul_mont`, this is not the case. We need to translate the x86_64 `shlx` instruction differently to avoid the safety condition failure.

Let us go back to `ecp_nistz256_mul_mont.gas`. Consider the following rule for `shlx`:

```
#! shlx $1v, $2v, $3v -> assert $1v=32 && true;\nsplit ddc $3v $2v 32;\nshl $3v $3v 32
```

After check `$1v` is 32, it splits `$2v` into two. The high 32-bit value is stored in `ddc`. The low 32-bit value in `$2v` is then shifted to the left by 32 bits.

To further improve our translation rules, let us see how `shlx` and `shrx` are used in `ecp_nistz256_mul_mont.gas`:

```
shlx   %r14,%r8,%rbp               #! PC = 0x55555557d72e
adc    %rcx,%r11                   #! PC = 0x55555557d733
shrx   %r14,%r8,%rcx               #! PC = 0x55555557d736
```

The `shlx` instruction puts the low 32-bit of `r8` in `rbp`. Then `shrx` puts the high 32-bit of `r8` in `rcx`. In the CryptoLine fragment, the variable `ddc` is in fact equal to `rcx`. Let us change the rule for `shrx` to check it. Consider the following rule:

```
#! shrx $1v, $2v, $3v -> assert $1v=32 && true;\nsplit $3v dc $2v 32;\nassert true
&& $3v=ddc;\nassume $3v=ddc && true
```

After obtaining `$3v`, the new rule asks the SMT-based engine to check if `$3v` is equal to `ddc`. The equation is then passed to the CAS-based engine by the CryptoLine `assume` command. This is a common technique to pass information between engines. We ask one engine to verify a property with `assert`, and then pass the property to the other engine with `assume`.

We are ready to apply the translation rules. After commenting out irrelevant instructions in trace, use the following command:

```
$ $CL_HOME/scripts/to_zdsl.py ecp_nistz256_mul_mont.gas > ecp_nistz256_mul_mont.cl
```

It remains to declare input parameters and specify properties about `ecp_nistz256_mul_mont`. The declaration and pre-condition are similar to `ecp_nistz256_add`:

```
proc main (uint64 a0, uint64 a1, uint64 a2, uint64 a3,
           uint64 b0, uint64 b1, uint64 b2, uint64 b3,
           uint64 m0, uint64 m1, uint64 m2, uint64 m3) =
{
  and [ m0 = 0xffffffffffffffff, m1 = 0x00000000ffffffff,
        m2 = 0x0000000000000000, m3 = 0xffffffff00000001 ]
  &&
  and [ m0 = 0xffffffffffffffff@64, m1 = 0x00000000ffffffff@64,
        m2 = 0x0000000000000000@64, m3 = 0xffffffff00000001@64,
        limbs 64 [a0, a1, a2, a3] <u limbs 64 [m0, m1, m2, m3],
        limbs 64 [b0, b1, b2, b3] <u limbs 64 [m0, m1, m2, m3]
  ]
}
```

Note that the modulo `m`'s appear in both parts of pre-condition. Since we will use the CAS-based engine, we need to tell the engine about `m`'s. Similarly, we initialize memory stores with input parameters and constants.

```
mov L0x7fffffffd9b0 a0; mov L0x7fffffffd9b8 a1;
mov L0x7fffffffd9c0 a2; mov L0x7fffffffd9c8 a3;

mov L0x7fffffffd9d0 b0; mov L0x7fffffffd9d8 b1;
```

```
mov L0x7fffffffd9e0 b2; mov L0x7fffffffd9e8 b3;

mov L0x55555557c000 0xffffffffffffffff@uint64;
mov L0x55555557c008 0x00000000ffffffff@uint64;
mov L0x55555557c010 0x0000000000000000@uint64;
mov L0x55555557c018 0xffffffff00000001@uint64;
```

At the end of `ecp_nistz256_mul_mont.cl`, the results `c`'s are obtained from memory stores.

```
mov c0 L0x7fffffffd9f0; mov c1 L0x7fffffffd9f8;
mov c2 L0x7fffffffda00; mov c3 L0x7fffffffda08;
```

And we use the following post-condition:

```
{
  eqmod limbs 64 [0, 0, 0, 0, c0, c1, c2, c3]
        limbs 64 [a0, a1, a2, a3] * limbs 64 [b0, b1, b2, b3]
        limbs 64 [m0, m1, m2, m3]
&&
  limbs 64 [c0, c1, c2, c3] <u limbs 64 [m0, m1, m2, m3]
}
```

In the post-condition, we ask the CAS-based engine to verify $c \times 2^{256} \equiv a \times b \mod p256$. For the range check $c < p256$, we employs the SMT-based engine.

4.2.2. *Verification.* We are ready to verify our model. Type

```
$ $CL_HOME/cv.exe -v -isafety ecp_nistz256_mul_mont.cl
```

CryptoLine reports the algebraic specification fails. We will add more annotations to our model. We have seen how information can be passed between engines in the translation rules for `shlx` and `shrx`. Another useful information to pass from the SMT-based to the CAS-based engines is addition carries. When carries propagate along long additions, the last carry is almost always zero. Such information is easily inferred with the SMT-based engine. In `ecp_nistz256_mul_mont`, two threads of long additions are computed interleaved. One uses the x86_64 `adcx` instruction and the other uses `adox`. There are three pairs of interleaving long additions. At the end of each pair, we annotate `ecp_nistz256_mul_mont.cl` with the following commands:

```
(* NOTE: can't carry *)
assert true && and [carry=0@1,overflow=0@1];
assume and [carry=0,overflow=0] && true;
```

Here, we ask the SMT-based engine to verify both carry and overflow are zeros, and then pass the information to the CAS-baesd engine.

The last annotation we need to add is for the conditional moves at the end of `ecp_nistz256_mul_mont`. Similar to `ecp_nistz256_add`, the conditional moves check if the Montgomery product is less than $p256$ by subtraction. If not, the difference is returned. The SMT-based engine suffices to verify this in `ecp_nistz256_add`. We will verify the conditional moves in the SMT-based engine and pass the information to the CAS-based engine. Let us save the Montgomery product before subtraction with the following:

```
ghost r12o@uint64, r13o@uint64, r8o@uint64, r9o@uint64, r10o@uint64 :
      and [r12o=r12, r13o=r13, r8o=r8, r9o=r9, r10o=r10]
   && and [r12o=r12, r13o=r13, r8o=r8, r9o=r9, r10o=r10];
```

The keyword `ghost` declares five reference variables `r12o`, `r13o`, `r8o`, `r9o`, and `r10o`. These reference variables can only appear in `assert` and `assume` commands and hence cannot the computation of `ecp_nistz256_mul_mont`. After the conditional moves, we add two CryptoLine commands:

```
(* NOTE: final reduction *)
assert true &&
       eqmod limbs 64 [r12, r13, r8, r9, 0@64]
```

```
proc main (...) =
{ P0 && Q0 }
(* Phase I *)
cut P1 && Q1;
(* Phase II *)
cut P2 && Q2:
(* Phase III *)
{ P3 && Q3 }
```

(A) Original

```
proc main0 (...) =
{ P0 && Q0 }
(* Phase I *)
{ P1 && Q1 }
```

(B) Part I

```
proc main1 (...) =
{ P1 && Q1 }
(* Phase II *)
{ P2 && Q2 }
```

(C) Part II

```
proc main2 (...) =
{ P2 && Q2 }
(* Phase III *)
{ P3 && Q3 }
```

(D) Part III

FIGURE 1. The CRYPTOLINE `cut` Command

```
             limbs 64 [r12o, r13o, r8o, r9o, r10o]
             limbs 64 [m0, m1, m2, m3, 0@64];
assume eqmod limbs 64 [r12, r13, r8, r9, 0]
             limbs 64 [r12o, r13o, r8o, r9o, r10o]
             limbs 64 [m0, m1, m2, m3, 0] && true;
```

The `assert` command asks the SMT-based engine to verify the result is congruent to the Montgomery product modulo $p256$. The information is then passed to the CAS-based engine in `assume`.

Using the following command, CRYPTOLINE reports `ecp_nistz256_mul_mont` is verified:

```
$ $CL_HOME/cv.exe -v -isafety ecp_nistz256_mul_mont.cl
```

Exercise: Construct a model for `ecp_nistz256_sqr_mont` in `ecp_nistz256-x86_64.pl` and verify it.

4.3. Compositional Reasoning with `cut`. The `ecp_nistz256_mul_mont` subroutine computes in two phases. The first phase computes the Montgomery product and stores it in five registers `r12`, `r13`, `r8`, `r9`, and `r10`. The second phase reduces the Montgomery product by modulo $p256$ and stores the final result in four registers `r12`, `r13`, `r8`, and `r9`. Since the two phases appear to be independent, they may be verified independently.

The CRYPTOLINE `cut` *P* `&&` *Q* command provides a simple mechanism to divide a verification task by parts. Consider the CRYPTOLINE model in Figure ??. The `cut` command effectively splits the model into three parts shown in Figure ?? to ??. Observe that `P1 && Q1` is the post-condition in Figure ?? but the pre-condition in Figure ??. Similarly, `P2 && Q2` is the post-condition in Figure ?? but the pre-condition in Figure ??. CRYPTOLINE reports successful verification when all three parts are verified successfully. Informally, `P1 && Q1` is established and then assumed to verify `P2 && Q2`. `P2 && Q2` is then assumed to prove `P3 && Q3`. If we know how to divide a large cryptographic program into phases, the `cut` command allows us to verify the program by parts.

Back to `ecp_nistz256_mul_mont`, it is natural to divide the subroutine by its two phases. Let us add the following command just before the `ghost` declaration:

```
cut eqmod limbs 64 [0, 0, 0, 0, r12, r13, r8, r9, r10]
          (limbs 64 [a0, a1, a2, a3] * limbs 64 [b0, b1, b2, b3])
          limbs 64 [m0, m1, m2, m3] &&
    and [limbs 64 [r12, r13, r8, r9, r10] <u
         2@320 * limbs 64 [m0, m1, m2, m3, 0@64],
         m0 = 0xffffffffffffffff@64, m1 = 0x00000000ffffffff@64,
         m2 = 0x0000000000000000@64, m3 = 0xffffffff00000001@64,
         r14=0x00000000ffffffff@64, r15=0xffffffff00000001@64,
         r12=rbx, r13=rdx];
```

The `cut` command states the Montgomery product is stored in the five registers `r12`, `r13`, `r8`, `r9`, and `r10` and the product is less than twice of the modulo. The remaining equations collect necessary assumptions to verify the reduction modulo $p256$.

With the simple modification, we can verify `ecp_nistz256_mul_mont.cl` again:

```
$ $CL_HOME/cv.exe -v -isafety ecp_nistz256_mul_mont.cl
```

On Raspberry Pi 4 (1.8GHz ARM Cortex-A72 with 8GB RAM), the model without `cut` is verified in 153 seconds. In contrast, the model with `cut` is verified in 52 seconds. The `cut` command can significantly reduce the verification time if used properly.

Exercise: Add `cut` to your model for `ecp_nistz256_sqr_mont` and compare verification time.