CryptoLine

July 26, 2022

1 Introduction

CryptoLine is a tool and a language for the verification of low-level implementations of mathematical constructs. In CryptoLine, users can specify two kinds of properties, namely algebraic properties and range properties. Algebraic properties involve equalities and modular equalities in the integer domain while range properties involve bit-accurate variable ranges. CryptoLine verifies algebraic properties and range properties separately. Verification of algebraic properties is reduced to ideal membership queries which are solved by external computer algebra systems. Verification of range properties is reduced to Satisfiability Modulo Theories (SMT) queries which are solved by external SMT solvers.

2 CryptoLine Language

An *identifier* is a regular string started by a letter or an underscore, followed by letters, digits, or underscores.

$$\mathit{id} ::= (\mathit{letter} \mid \mathit{underscore})[\mathit{letter} \mid \mathit{digit} \mid \mathit{underscore}]$$

All constants and variables in CryptoLine are typed. Let w be a positive integer. $\mathsf{uint}w$ and $\mathsf{sint}w$ in CryptoLine denote the types of bit-vectors with width w in the unsigned and two's complement signed representations respectively. The type $\mathsf{uint1}$ is also written as bit.

$$\mathit{typ} \quad ::= \quad \mathsf{uint1} \mid \mathsf{sint2} \mid \mathsf{uint2} \mid \mathsf{sint3} \mid \cdots \mid \mathsf{uint}w \mid \mathsf{sint}(w+1)$$

A *constant* is an integer, a binary number, a hexadecimal number, a named constant, or arithmetic expressions over constants.

$$\begin{array}{rcl}
\mathit{const} & ::= & \mathit{simple\_const} \\
 & | & \mathbf{(}\ \mathit{complexy\_const}\ \mathbf{)} \\
\mathit{simple\_const} & ::= & \mathbb{Z} \\
 & | & 0b[0-1]^{+} \\
 & | & 0x[0-9a-fA-F]^{+} \\
 & | & \mathbf{\$}\mathit{id} \\
\mathit{complexy\_const} & ::= & \mathit{const} \\
 & | & \mathbf{-}\ \mathit{complexy\_const} \\
 & | & \mathit{complexy\_const}\ \mathbf{+}\ \mathit{complexy\_const} \\
 & | & \mathit{complexy\_const}\ \mathbf{-}\ \mathit{complexy\_const} \\
 & | & \mathit{complexy\_const}\ \mathbf{*}\ \mathit{complexy\_const} \\
 & | & \mathit{complexy\_const}\ \mathbf{**}\ \mathit{complexy\_const} \\
\mathit{typed\_const} & ::= & \mathit{const@typ} \\
 & | & \mathit{typ}\ \mathit{const}
\end{array}$$

The value of a named integer c is read by $\$c$. CryptoLine supports the following arithmetic operators over constants: unary minus (-), addition (+), subtraction (-), multiplication (*), and exponent (**). A *typed constant* is a constant with its type explicitly specified.

A *variable* is an identity. A *typed variable* is a variable with its type explicitly specified. An *lval* is either a variable or a typed variable.

$$\begin{array}{rcl}
\mathit{var} & ::= & \mathit{id} \\
\mathit{typed\_var} & ::= & \mathit{var@typ} \mid \mathit{typ}\ \mathit{var} \\
\mathit{lval} & ::= & \mathit{var} \mid \mathit{typed\_var}
\end{array}$$

The notation $t^{*}_{\circ}$ and $t^{+}_{\circ}$ respectlvey represents a possibly empty and a non-empty sequence of $\circ$-separated t.

An *atom* is either a typed constant, a variable, or a typed variable. It is not necessary to specify the variable type explicitly in an atom because CryptoLine can infer the type automatically.

$$\mathit{atom} ::= \mathit{typed\_const} \mid \mathit{var} \mid \mathit{typed\_var}$$

An *algebraic expression* is evaluated over $\mathbb{Z}$.

$$\begin{array}{rclcl}
\mathit{eexp} & ::= & \mathit{simple\_const} & | & \mathit{var} \\
 & | & \mathbf{-}\ \mathit{eexp} & | & \mathit{eexp}\ \mathbf{+}\ \mathit{eexp} \\
 & | & \mathit{eexp}\ \mathbf{-}\ \mathit{eexp} & | & \mathit{eexp}\ \mathbf{*}\ \mathit{eexp} \\
 & | & \mathit{eexp}\ \mathbf{**}\ \mathit{eexp} & | & \boldsymbol{limbs}\ \mathit{const}\ \mathbf{[}\ \mathit{eexp}^{+}_{,}\ \mathbf{]} \\
 & | & \mathbf{(}\ \mathit{eexp}\ \mathbf{)} & &
\end{array}$$

limbs n $[e_1, \ldots, e_m]$ represents $e_1 + e_2 2^n + e_3 2^{2n} + \cdots + e_m 2^{(m-1)n}$. A *range expression* is evaluated over bit vectors. *const* w n is a bit-vector of width w and value n. $\sim$ (*neg*) is logical negation. ! (*not*), & (*and*), | (*or*), ˆ (*xor*) are respectively bit-wise negation, bit-wise AND, bit-wise OR, and bit-wise XOR.

umod is unsigned remainder. *srem* is 2's complement signed remainder (sign follows dividend). *smod* is 2's complement signed remainder (sign follows divisor). *uext* and *sext* are respectively unsigned and signed extension operations.

$$
\begin{array}{rcll}
\mathit{rexp} & ::= & \mathbf{(}\ \mathit{rexp}\ \mathbf{)} & \mid\ \textbf{const}\ \mathit{const}\ \mathit{const} \\
 & \mid & \mathbf{-}\ \mathit{rexp} & \mid\ \mathit{rexp}\ \mathbf{+}\ \mathit{rexp} \\
 & \mid & \mathit{rexp}\ \mathbf{-}\ \mathit{rexp} & \mid\ \mathit{rexp}\ \mathbf{*}\ \mathit{rexp} \\
 & \mid & \mathbf{\sim}\ \mathit{rexp} & \mid\ \textbf{neg}\ \mathit{rexp} \\
 & \mid & \mathbf{!}\ \mathit{rexp} & \mid\ \textbf{not}\ \mathit{rexp} \\
 & \mid & \mathit{rexp}\ \mathbf{\&}\ \mathit{rexp} & \mid\ \textbf{and}\ \mathit{rexp}\ \mathit{rexp} \\
 & \mid & \mathit{rexp}\ \mathbf{|}\ \mathit{rexp} & \mid\ \textbf{or}\ \mathit{rexp}\ \mathit{rexp} \\
 & \mid & \mathit{rexp}\ \hat{}\ \mathit{rexp} & \mid\ \textbf{xor}\ \mathit{rexp}\ \mathit{rexp} \\
 & \mid & \textbf{umod}\ \mathit{rexp}\ \mathit{rexp} & \mid\ \textbf{srem}\ \mathit{rexp}\ \mathit{rexp} \\
 & \mid & \textbf{smod}\ \mathit{rexp}\ \mathit{rexp} & \mid\ \textbf{limbs}\ \mathit{const}\ \mathbf{[}\ \mathit{rexp}^{+}_{,}\ \mathbf{]} \\
 & \mid & \textbf{uext}\ \mathit{rexp}\ \mathit{const} & \mid\ \textbf{sext}\ \mathit{rexp}\ \mathit{const}
\end{array}
$$

A *predicate* is represented by an algebraic predicate and a range predicate.

$$
\mathit{pred} \quad ::= \quad \textbf{true} \quad \mid \quad \mathit{epred}\ \mathbf{\&\&}\ \mathit{rpred}
$$

An *algebraic predicate* is evaluated over the integer domain. $e_1 = e_2$ ($eq\ e_1\ e_2$) is an equality over algebraic expressions. $e_1 = e_2$ ($mod\ [m_1, \ldots, m_n]$) ($eqmod\ e_1\ e_2\ [m_1, \ldots, m_n]$) is a modular equality. p_1 /\ p_2 ($and\ p_1\ p_2$) is a logical conjunction of p_1 and p_2. The conjunction of a sequence of algebraic predicates $e_1, \ldots, e_n$ is written as /\ $[e_1, \ldots, e_n]$ ($and\ [e_1, \ldots, e_n]$).

$$
\begin{array}{rcll}
\mathit{epred} & ::= & \mathbf{(}\ \mathit{epred}\ \mathbf{)} & \mid\ \textbf{true} \\
 & \mid & \mathit{eexp}\ \mathbf{=}\ \mathit{eexp} & \mid\ \textbf{eq}\ \mathit{eexp}\ \mathit{eexp} \\
 & \mid & \mathit{eexp}\ \mathbf{=}\ \mathit{eexp}\ \mathbf{(}\ \textbf{mod}\ [\mathit{eexp}^{+}_{,}]\ \mathbf{)} & \mid\ \textbf{eqmod}\ \mathit{eexp}\ \mathit{eexp}\ [\mathit{eexp}^{+}_{,}] \\
 & \mid & \mathit{epred}\ \mathbf{/\backslash}\ \mathit{epred} & \mid\ \textbf{and}\ \mathit{epred}\ \mathit{epred} \\
 & \mid & \mathbf{/\backslash}\ \mathbf{[}\ \mathit{epred}^{+}_{,}\ \mathbf{]} & \mid\ \textbf{and}\ \mathbf{[}\ \mathit{epred}^{+}_{,}\ \mathbf{]}
\end{array}
$$

A *range predicate* specifies the ranges of variables. CryptoLine offers comparisons such as equality (=), modular equalities (*equmod*, *eqsmod*, *eqsrem*), unsigned less than (<), unsigned less than or equal to (<=), unsigned greater than (>), unsigned greater than or equal to (>=), signed less than ($<s$), signed less than or equal to ($<= s$), signed greater than ($> s$), and signed greater than or equal to (>= s).

$$
\begin{array}{lllll}
\mathit{rpred} & ::= & (\ \mathit{rpred}\ ) & | & \textbf{\textit{true}} \\
 & | & \mathit{rexp}\ =\ \mathit{rexp} & | & \textbf{\textit{eq}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp}\ =\ \mathit{rexp}\ (\ \mathit{umod}\ \mathit{rexp}\ ) & | & \textbf{\textit{equmod}}\ \mathit{rexp}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp}\ =\ \mathit{rexp}\ (\ \mathit{smod}\ \mathit{rexp}\ ) & | & \textbf{\textit{eqsmod}}\ \mathit{rexp}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp}\ =\ \mathit{rexp}\ (\ \mathit{srem}\ \mathit{rexp}\ ) & | & \textbf{\textit{eqsrem}}\ \mathit{rexp}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} < \mathit{rexp} & | & \textbf{\textit{ult}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} <= \mathit{rexp} & | & \textbf{\textit{ule}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} > \mathit{rexp} & | & \textbf{\textit{ugt}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} >= \mathit{rexp} & | & \textbf{\textit{uge}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} < \textbf{\textit{s}}\ \mathit{rexp} & | & \textbf{\textit{slt}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} <= \textbf{\textit{s}}\ \mathit{rexp} & | & \textbf{\textit{sle}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} > \textbf{\textit{s}}\ \mathit{rexp} & | & \textbf{\textit{sgt}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \mathit{rexp} >= \textbf{\textit{s}}\ \mathit{rexp} & | & \textbf{\textit{sge}}\ \mathit{rexp}\ \mathit{rexp} \\
 & | & \sim \mathit{rpred} & | & \textbf{\textit{neg}}\mathit{rpred} \\
 & | & \mathit{rpred}\ /\backslash\ \mathit{rpred} & | & \textbf{\textit{and}}\ \mathit{rpred}\ \mathit{rpred} \\
 & | & \mathit{rpred}\ \backslash/\ \mathit{rpred} & | & \textbf{\textit{or}}\ \mathit{rpred}\ \mathit{rpred} \\
 & | & /\backslash\ [\ \mathit{rpred}^{+}_{,}\ ] & | & \textbf{\textit{and}}\ [\ \mathit{rpred}^{+}_{,}\ ] \\
 & | & \backslash/\ [\ \mathit{rpred}^{+}_{,}\ ] & | & \textbf{\textit{or}}\ [\ \mathit{rpred}^{+}_{,}\ ]
\end{array}
$$

There are numerous *instructions* supported by CRYPTOLINE. *mov x a* assigns destination variable x by the value of the source atom a. *cmov x c* a_1 a_2 assigns destination variable x by the value of the source atom a_1 if the condition bit c is 1, and otherwise by the value of the source atom a_2. *add x* a_1 a_2 assigns x by the addition of the source atoms a_1 and a_2. Note that *add* may overflow. *adds c x* a_1 a_2 assigns x by the addition of the source atoms a_1 and a_2 with carry bit c set. *adc x* a_1 a_2 *y* assigns x by the addition of the carry bit y and the source atoms a_1 and a_2. *adcs* is the same as *adc* except the carry bit is set. There are also instructions *sub* for subtraction; *subc*, *sbc* and *sbcs* for subtraction with carry; *subb*, *sbb*, and *sbbs* for subtraction with borrow. *mul* and *muls* are half multiplication operations. The differenace is that *muls* sets the carry bit if the multiplication under- or over-flow. *mull* is full multiplication with results split into high part and low part. *mulj* is also full multiplication without splitting the results. *nondet* assigns a variable by a nondeterministic value. *set x* assigns the bit variable x by 1 while *clear x* assigns the bit variable x by 0. *shlxan* shifts the source atom a left by n and stores the results in x. *shlsoxan* is the same as *shlsxan* except that the bits shifted out are stored in o. *shrxan* shifts the source atom a right logically by n and stores the results in x. *shrsxoan* is the same as *shrxan* except that the bits shifted out are stored in o. *sar* and *sars* are the same as *shr* and *shrs* respectively except that the right shift is arithmetic. $cshlx_hx_la_1a_2n$ concatenates the source atoms a_1 (high bits) and a_2 (low bits), shifts the concatenation left by n, stores the high bits of the shifted concatenation in x_h, and stores the low bits shifted right by n in x_l. $cshrx_hx_la_1a_2n$ concatenates the source atoms a_1 (high bits) and a_2 (low bits), shifts the concatenation right logically by n, stores the high bits of the shifted concatenation in x_h, and stores the low bits in x_l. $cshrx_hx_loa_1a_2n$ is the same as $cshrx_hx_la_1a_2n$ except that the bits shifted out are stored in o. $splx_hx_lan$ splits the source atom a at position n, stores the high bits in x_h, and stores the low bits in x_l. *split* is the same as *spl* except that the high bits and the low bits are extended to the size of a. While the low bits are always zero extended, the high bits are sign extended if a is signed and otherwise zero extended. $joinxa_1a_2$ assigns x by the concatenation of the source atoms a_1 (high bits) and a_2 (low bits). *and*, *or*, *not*, and *xor* are bit-wise operations. *casttxa* assigns x by the source atom a casted to the type t. *vpctxa* is the same as *casttxa* except that the integer interpretation of x must be the same as the integer interpretation of a. *assert* tells CryptoLine to verify the specified predicate. *assume* tells CryptoLine to assume the specified predicate. *cut e* && *r* is an alias of one *ecut e* followed by a *rcut r*. For *ecut*, CryptoLine verifies the specified algebraic predicate and starts afresh with the predicate assumed when verifying algebraic properties. Similarly for *rcut*, CryptoLine verifies the specified range predicate and starts afresh with the predicate assumed when verifying range properties. *ghost* can introduce logical variables that must only be used in specifications such as *assert*, *assume*, *cut*, *ecut*, *rcut*, and postconditions. The predicate in a *ghost* instruction is always assumed. *call p* $(a_1, a_2, \ldots, a_n)$ executes a defined procedure p with arguments $a_1, a_2, \ldots, a_n$.

instr	::=	***mov*** *lval atom*	|	***cmov*** *lval lval atom atom*
	|	***add*** *lval atom atom*	|	***adds*** *lval lval atom atom*
	|	***adc*** *lval atom atom var*	|	***adcs*** *lval lval atom atom var*
	|	***sub*** *lval atom atom*		
	|	***subc*** *lval lval atom atom*	|	***subb*** *lval lval atom atom*
	|	***sbc*** *lval atom atom var*	|	***sbcs*** *lval lval atom atom var*
	|	***sbb*** *lval atom atom var*	|	***sbbs*** *lval lval atom atom var*
	|	***mul*** *lval atom atom*	|	***muls*** *lval lval atom atom*
	|	***mull*** *lval lval atom atom*	|	***mulj*** *lval atom atom*
	|	***nondet*** *lval*		
	|	***set*** *lval*	|	***clear*** *lval*
	|	***shl*** *lval atom const*	|	***shls*** *lval lval atom const*
	|	***shr*** *lval atom const*	|	***shrs*** *lval lval atom const*
	|	***sar*** *lval atom const*	|	***sars*** *lval lval atom const*
	|	***cshl*** *lval lval atom atom const*		
	|	***cshr*** *lval lval atom atom const*	|	***cshrs*** *lval lval lval atom atom const*
	|	***spl*** *lval lval atom const*	|	***split*** *lval lval atom const*
	|	***join*** *lval lval atom const*		
	|	***and*** *lval atom atom*	|	***or*** *lval atom atom*
	|	***xor*** *lval atomatom*	|	***not*** *lval atom*
	|	***cast*** *typ lval atom*	|	***vpc*** *typ lval atom*
	|	***assert*** *pred*	|	***assume*** *pred*
	|	***cut*** *pred_clause*	|	***ecut*** *epred_clause*
	|	***rcut*** *rpred_clause*	|	***ghost*** *typed_var*$^+_,$: *pred*
	|	***call*** *id* (*atom*$^*_,$)	|	***nop***

Instructions *add*, *adds*, *adc*, *adcs*, *sub*, *subc*, *subb*, *sbc*, *sbcs*, *sbb*, *sbbs*, *mul*, *muls*, *mull*, *mulj*, *split*, and *spl* also have specific unsigned and signed versions with prefix "u" or "s". For example, *uadd* and *sadd* are respectively unsigned and signed versions of *add*.

Sometimes a predicate has to be proved with facts that have been cut off. CryptoLine offers the specification of hints required to prove a predicate.

$$
\begin{array}{rcll}
\mathit{pred\_clause} & ::= & \mathit{true} & \mid\ \mathit{epred\_clause}\ \textbf{\&\&}\ \mathit{rpred\_clause} \\
\mathit{epred\_clause} & ::= & \mathit{epred} & \mid\ \mathit{epred}\ \textbf{\textit{prove with}}\ [\mathit{prove\_with}^{+}_{,}] \\
 & \mid & \mathit{epred\_clause}^{+}_{,} & \\
\mathit{rpred\_clause} & ::= & \mathit{rpred} & \mid\ \mathit{rpred}\ \textbf{\textit{prove with}}\ [\mathit{prove\_with}^{+}_{,}] \\
 & \mid & \mathit{rpred\_clause}^{+}_{,} & \\
\mathit{cas} & ::= & \textbf{\textit{singular}} & \mid\ \textbf{\textit{sage}} \\
 & \mid & \textbf{\textit{magma}} & \mid\ \textbf{\textit{mathematica}} \\
 & \mid & \textbf{\textit{macaulay2}} & \mid\ \textbf{\textit{maple}} \\
\mathit{prove\_with} & ::= & \textbf{\textit{precondition}} & \mid\ \textbf{\textit{all cuts}} \\
 & \mid & \textbf{\textit{all assumes}} & \mid\ \textbf{\textit{all ghosts}} \\
 & \mid & \textbf{\textit{cuts}}\ [\mathbb{N}^{+}_{,}] & \mid\ \textbf{\textit{algebra solver}}\ \mathit{cas} \\
 & \mid & \textbf{\textit{algebra solver}}\ \mathit{cas}\ \mathit{path} & \mid\ \textbf{\textit{algebra solver smt}}\ \textbf{:}\ \mathit{path} \\
 & \mid & \textbf{\textit{range solver}}\ \mathit{path} &
\end{array}
$$

Note that the indices of *ecut* and *rcut* are numbered separately (starting from 0). When verifying algebraic properties, *rcut* instructions are ignored. When verifying range properties, *ecut* instructions are ignored. For example, consider the following program.

```
mov x 15@uint16;
ecut x = 15;
mov y 3@uint16;
cut y = 3 && and [x = 15@16, y = 3@16];
add z x y;
rcut z = 18@16;
```

If we want to prove e *prove_with* [*cuts*[1]] && r *prove_with* [*cuts*[1]], then $y = 3$ will be assumed when proving the algebraic property e while $z = 18@16$ will be assumed when proving the range property r.

A *procedure* is a parameterized program together with its specification (precondition and postcondition).

$$\mathit{proc} ::= \textbf{\textit{proc}}\ \mathit{id}\ \textbf{(}\ \mathit{formals}\ \textbf{)} = \textbf{\{}\ \mathit{pre}\ \textbf{\}}\ \mathit{prog}\ \textbf{\{}\ \mathit{post}\ \textbf{\}}$$

The *formal parameters* of a procedure may be separated by a semicolon into *inout* and *out* variables.

$$\mathit{formals} ::= \mathit{typed\_var}^{*}_{,} \mid \mathit{typed\_var}^{*}_{,}\ \textbf{;}\ \mathit{typed\_var}^{*}_{,}$$

Variables before the semicolon are inout variables while variables after the semicolon are out variables. Formal parameters without a semicolon are all inout variables. The difference between inout and out variables is that when calling a procedure, actual parameters of the inout formal variables must be defined but this is not required for the actual parameters of the out formal variables. However, this does not mean that an out variable can be read before initialized. Every variable must be initialized before reading its value. A *precondition* is a predicate.

$$pre ::= pred$$

A *postcondition* is a predicate clause.

$$post ::= pred\_clause$$

A *statement* is a declaration of a procedure or a named integer.

$$stmt \quad ::= \quad proc \quad | \quad \textbf{\textit{const}}\ id \ = \ const$$

A *program* is a sequence of semicolon separated statements. The entry point of the program is the *main* procedure. Other procedures called in main are inlined.

$$prog ::= stmt^{+}_{;}$$