Improved Error Reporting for the Cryptyc Protocol Language

Brian Stoler and Eric Allen

Draft: $Date: 2001/12/08 06:36:56 $ $Revision: 1.6 $

Abstract

The following is an update on the status of the Cryptyc Protocol Language implementation project at Rice University. The purpose of this project is to produce a Cryptyc type checker that provides improved error reporting. In doing so, we hope to increase the applicability of Cryptyc, and similar protocol-checking tools, to real-world domains.

1 Introduction

The Cryptyc Protocol language is an implementation of the typed spi-calculus that allows for protocol verification through type checking. By specifying a typable protocol in this language, a programmer can be assured that the security of the protocol cannot be violated by any other process that is specifiable in (an untyped version of) the same language. Such an approach provides several advantages over protocol verification via an inference engine of a recursively enumerable modal logic, such as BAN logic:

1. Although such logics can prove relations among the various messages sent in a protocol, it is impossible to express claims to the effect that the protocol is impervious to a given set of attacks. This is because such logics do not attempt to model the space of possible attacks. In comparison, the spi-calculus provides an explicit model of an attack space. Although this provides no protection against attack that cannot be modelled in the space, the expressiveness of the calculus is such that it provides a good deal of assurance in the robustness of a protocol.

2. Since the type system presented for Cryptyc is proven to assure "robust safety" against security attacks, a protocol can be verified simply through the process of type checking. Since the Cryptyc type system is not only decidable, but linear in the size of the program, type checking a protocol is much preferable to attempting to prove the protocol correct via an inference engine. Searching for proofs in this way takes time exponential in the size of the proof, if it exists. If it doesn't exist, the process will not halt.

For these reasons, we believe that the use of systems such as Cryptyc for protocol verification holds much promise for improving the robustness of security protocols. Nevertheless, the adoption of such systems has been held back by the lack of development tools and good error-checking and handling facilities for such languages.

As members of the Rice Programming Languages Team, we have had extensive experience with the design and implementation of programming tools for development and error checking. We have found that usefulness of reported errors is drastically improved through the following two practices:

1. Separation of the error checking process from other phases of program processing. Error checking should take the form of a walker over an internal and complete representation of syntactically valid source code (the abstract syntax tree).

2. Retainment of as much structure from the entered text as possible. Canonicalization tends to hide program structure, as does the lack of strong typing of sub-components in source code (e.g., each form of expression in the language should have its own type). Additionally, no structure should be added to the code, as errors propogated into this structure can also obfuscate the true cause of the errors.

Unfortunately, the existing Cryptyc implementation (`http://cryptyc.cs.depaul.edu`) does not follow these practices:

1. Type checking is done during construction of the various components of the AST. Not only does this obfuscate the structure of the type checker, it also divides it across many class boundaries. Furthermore, since the entire AST is not available during the type checking process, there are inherent limitations as to how description error messages from this process can be.

2. The syntax and AST used by the Cryptyc implementation are not what is provided in the paper, and they are not documented.

3. Finally, the AST structure used by Cryptyc does not retain as much structure as it could for the use of error reporting. One important omission is source location that the AST component corresponds to; this leads to incorrect source coordinates being reported in error messages!

We decided that the best course of action to improve the error reporting of Cryptyc type checker (and to make the system more extensible) was to reimplement the system largely from scratch. Our implementation is explained in Section 2 below. We also realized that the usefulness of Cryptyc was substantially hampered by the lack of documentation of the surface syntax used in the implemented tool. We rectified this by documenting the syntax, after substantially simplifying the grammar. The syntax is presented in Section A.

2 Implementation

The reimplementation of Cryptyc was done in these steps:

1. Convert grammar from Metamata Parse to JavaCC.

2. Devise a new abstract syntax tree representation of Cryptyc.

3. Modify the parser to target this structure.

4. Implement a type checker over these new ASTs, taking advantage of the added structure they provide to improve error-checking.

2.1 Converting the grammar to JavaCC

The original Cryptyc parser was written using the Metamata Parse (`http://www.metamata.com`) parser generator. Unfortunately, Metamata Parse is a) a commercial product and b) no longer available at all since Metamata was acquired by Webgain (`http://www.webgain.com`). Luckily, MParse is a modified version of a freeware Java parser generator, the Java Compiler Compiler (JavaCC), which originates from Sun Labs. (JavaCC is in fact now maintained by WebGain and is available at `http://www.webgain.com/products/java_cc`.)

Knowing this, we simply converted the MParse grammar syntax into JavaCC format. This was feasible since the structure remained basically the same and only the surface syntax of the grammar file changed. Most of the work was done by automated search-and-replaces.

2.2 Designing new AST and modifying the parser to target it

After cleaning up the grammar some, we followed the structure of the grammar to model the abstract syntax. We used a tool called ASTGen to generate the many source files needed to define the AST. ASTGen takes a compact representation of the class hierarchy and expands it out into the complete source code. The input file for the Cryptyc AST (at present) is presented in Section B.

It is interesting to note that this AST structure, as well as the specific parser rules, was refined frequently as we worked on the project. We discovered many cases where the grammar (and AST) were allowing a wider range of values than actually made sense in the language. In these cases we modified the parser and AST to make component types more restrictive, which provides better static information about what values are possible at each node in the tree.

2.3 The Type Checker

2.3.1 Analysis

Type checking the Cryptyc language involves the accumulation of multisets of effects over each process of a protocol, considering the sequence of statements of a process in reverse order. A protocol is well-formed iff the accumulated multiset of each process is empty.

In addition to this high-level check on a protocol, it is also necessary to check that the following conditions hold:

1. All variables are bound in the scope in which they are used.
2. The defined parametric types are applied only to arguments of the appropriate type.
3. The types of the expressions contained in the various statements of the protocol match the conditions stipulated by the language. For example, check statements may only check to variables of Nonce type.

These seemingly straightforward checks on a protocol are complicated by the rules for typing and variable binding of the language. New variables can be bound in the middle of patterns, such as those occuring in input statements, check statements, etc. Union types can be matched by any of their constituent types. Additionally, the types of the language can be defined in terms of other types, binding the parameters of a type to expressions. Therefore, the determination of whether an expression matches a particular type is dependent not just on its explicit type, but also on how this type is defined in terms of the other types in the protocol (and, recursively, on how those types are defined, etc.). Finally, the Cryptyc type system defines structural equivalence on types, meaning that two types with different names are still equivalent so long as their underlying structure is the same.

2.3.2 Design

In order to deal with these issues, we broke the process of type checking a parsed protocol description into two phases:

1. Accumulation of the global environments.
2. Traversal of the syntax tree, extending the environments at each new lexical scope.

The first phase of the process, accumulation of the environments, actually involves the construction of three distinct environments: the declared types, the type bindings of public fields, and the type bindings of private fields. Since type equivalence in Cryptyc is determined through structural equivalence, the types stored in these environments were reduced to their *ground structure*, i.e., all references to other defined types were removed. The removal of these references was accomplished by inductively defining a notion of *evaluation* of an applied type T in a substitution environment E, and type environment Δ, over the set of types as follows:

- To evaluate a type application, extend E, bind the formal parameters of the applied type to the actual parameters of the application, and recursively evaluate the body of the applied type.
- To evaluate a variable, lookup the expression it is bound to in E, and return the type of this expression in Δ.
- To evaluate all other types, recursively evaluate their subcomponents and re-assemble the results.

The ground structure of a type is simply its evaluation, as determined by these rules. By reducing types to their ground structure, it is much easier to determine during type checking whether two types match: each type is evaluated, and the resultant ground structures are compared directly. The only caveat to this rule is that union types must be matched against types of any of their constituent forms.

The second phase of the type checking process, traversal of the tree, is done through the use of another visitor over syntax trees that keeps the needed type environments in fields as necessary, as well as the queue of discovered type errors. A recursive descent is performed over each process, and multisets of effects are accumulated.

Since the type checker is working only over syntactically well-defined trees, it is possible at each type error to continue processing the rest of the tree in such a way as to continue catching new type errors. In order to do this, a queue of error messages is kept by the visitor. Messages are added to this queue as they are discovered, but the control flow of type checking is never disturbed. To deal with cases where errors prevent the assignment of any reasonable type to an expression (or effect to an effect expression) the special AST classes ErrorType and ErrorEffect are used. These AST components are never generated by the parser, but they serve to hold places for erroneous code in the type environments.

3 Cryptyc Error Messages

Here we compare an example of the error diagnostics produced by the old implementation of Cryptyc to the the what we'd expect from an ideal implementation.

3.1 Example 1: Incorrect version of the wide-mouthed frog protocol

The source code for this example is presented in C.2.

3.1.1 Original results

Type error! At line 88: WMFNonce (a, Bob, sKey) = Nonce (end (a sending Bob key sKey)) does not match WMFNonce (Bob, a, sKey) = Nonce (end (Bob sending a key sKey)).

Note that line 88 is rather useless; it points to the “server Responder ...” line!

3.1.2 New results

Hopefully this should, at least:

1. Explain what line/column contained each of the two things that don’t match.
2. Identify what the two things that did not match are and why they were supposed to match.
3. Include all more than one error per run, when appropriate.

4 Future work

Given the current implementation of Cryptyc, there are several ways in which the system could be extended to better meet the needs of security protocol designers in the field:

1. The inclusion of asymmetric keys. The set of protocols expressible solely through the use of symmetric keys is rather limited. Since many real protocols involve the use of asymmetric keys, the extention of the language to include such keys would be extremely useful. Such a process would involve devising the new syntax and typing rules, and the construction of a new type soundness proof for the extended language.

2. Attack traces duing error reporting. Although type checking has advantages over general theorem proving in terms of performance, the proofs it constructs are not always helpful to a human programmer trying to understand how an error would correspond to undesirable behavior at runtime. Concievably, it would be possible to bridge this gap by including, with a type error message, an example execution trace of the protocol in which an attacker takes advantage of the type error discovered.

3. Evaluation/compilation of protocols. Cryptyc allows for the checking of a protocol specification, but this specification must then be implemented in a real-world setting. Human implementation of a specification is an error prone process, and therefore has the potential to introduce security holes. It would be preferable if we could instead allow for the direct evaluation of a Cryptyc protocol. By providing evaluators at both the client and server sites, parties sharing a Cryptyc protocol could communicate securely through the evaluators. Alternatively, a Cryptyc compiler could allow stand-alone source code that implements a protocol to be installed on the various machines. An offline version of such a tool may help a programmer to interact dynamically with the processes of a protocol as he designs it.

4. A protocol development environment. The various extensions above could be incorporated into an integrated development environment that would provide support for language-specific source-editing tasks, as well as a user-friendly interface to functionality such as type-checking and evaluation. Furthermore, following the paradigms set by DrScheme and DrJava, a read-eval-print loop could be built into such an environment. Combined with an offline evaluator, the repl would allow a programmer to play the role of attacker against his protocol, as he designed it, subjecting it to various attacks. Such a tool could have strong pedagogic value: a student could learn precisely what security he is provided by the Cryptyc type checker, and how various attacks are thwarted.

A Cryptyc language syntax

We have extracted and refined the (undocumented) grammar from the original Cryptyc parser, and have constructed a corresponding JavaCC input file for our implementation. We include this grammar below for the benefit of would-be Cryptyc programmers attempting to implement new protocols in the language.

A.1 Syntax overview

compilationUnit ::= (privateFieldDeclaration | publicFieldDeclaration | typeDeclaration | serverProcessDeclaration | clientProcessDeclaration | importDeclaration)*

importDeclaration ::= `"import"` identifier `";"`

publicFieldDeclaration ::= `"public"` variableDeclaration `";"`

privateFieldDeclaration ::= `"private"` variableDeclaration `";"`

typeDeclaration ::= `"type"` identifier (`"("` variableDeclarations `")"`)? `"="` type `";"`

serverProcessDeclaration ::= "server" variableMessage "at" variableMessage "is" "(" variableDeclaration ")" processBody

clientProcessDeclaration ::= "client" variableMessage "at" variableMessage "is" processBody

processBody ::= "{" statement "}"

variableDeclaration ::= identifier ":" type

variableDeclarations ::= (variableDeclaration ("," variableDeclaration)*)?

effect ::= ("end" "(" variableMessageSequence ")" | "check" "(" variableMessage ")")

effects ::= (effect ("," effect)*)?

type ::= ("(" type ")" | publicType | privateType | keyType | nonceType | structType | unionType | typeReference)

privateType ::= "Private"

publicType ::= "Public"

keyType ::= "Key" type

nonceType ::= "Nonce" "(" effects ")"

structType ::= "Struct" "(" variableDeclarations ")"

unionType ::= "Union" "(" variableDeclarations ")"

typeReference ::= identifier ("(" messages ")")?

statement ::= (establishStatement | inputStatement | outputStatement | decryptStatement | newStatement | checkStatement | castStatement | beginStatement | endStatement)?

establishStatement ::= "establish" variableMessage "at" variableMessage "is" "(" variableDeclaration ")" ";" statement

outputStatement ::= "output" variableMessage "is" "(" messages ")" ";" statement

inputStatement ::= "input" variableMessage "is" "(" patterns ")" ";" statement

decryptStatement ::= "decrypt" variableMessage "is" "(" ciphertextPattern ")" ";" statement

newStatement ::= "new" "(" variableDeclaration ")" ";" statement

beginStatement ::= "begin" "(" variableMessageSequence ")" ";" statement

endStatement ::= "end" "(" variableMessageSequence ")" ";" statement

checkStatement ::= "check" variableMessage "is" variableMessage ";" statement

castStatement ::= "cast" variableMessage "is" "(" variableDeclaration ")" ";" statement

message ::= (unionConstructorMessage | variableMessage | tupleMessage | ciphertextMessage)

variableMessage ::= identifier

unionConstructorMessage ::= identifier tupleMessage

tupleMessage ::= "(" messages ")"

ciphertextMessage ::= "{" messages "}" variableMessage

variableMessageSequence ::= (variableMessage)*

messages ::= (message ("," message)*)?

pattern ::= (variablePattern | unionConstructorPattern | constantPattern | tuplePattern | ciphertextPattern)

constantPattern ::= identifier

variablePattern ::= variableDeclaration

unionConstructorPattern ::= identifier tuplePattern

tuplePattern ::= "(" patterns ")"

ciphertextPattern ::= "{" patterns "}" variableMessage

patterns ::= (pattern ("," pattern)*)?

identifier ::= <IDENTIFIER>

A.2 Syntax description

compilationUnit ::= (privateFieldDeclaration | publicFieldDeclaration | typeDeclaration | serverProcessDeclaration | clientProcessDeclaration | importDeclaration)*

A CompilationUnit represents one Cryptyc source file.

importDeclaration ::= "import" identifier ";"

An ImportDeclaration instructs the parser to include the contents of the specified file when parsing this file.

identifier: The name of the file to import, without the suffix ".cry". This name is resolved relative to the directory that the present source file is in.

publicFieldDeclaration ::= "public" variableDeclaration ";"

A PublicFieldDeclaration defines a global variable that is visible to all processes, including opponent processes. Note that the type of the variable declared here must be untrusted, reflecting the fact that only untrusted-typed values are leaked to opponents.

privateFieldDeclaration ::= "private" variableDeclaration ";"

A PrivateFieldDeclaration defines a global variable that is visibile to all processes EXCEPT opponent processes. Thus, the variable declared here can have any type.

typeDeclaration ::= "type" identifier ("(" variableDeclarations ")")? "=" type ";"

A TypeDeclaration defines a new data type.

identifier: The name of the new type

variableDeclarations: The user-defined type can have type parameters. The parameters are simply variable bindings. References to a type with different actual type parameters refer to different types. So, if there is a type: `type HostKey (h : Host) = Key(h);` Then HostKey(Alice) and HostKey(Bob) are distinct types. Note that variable bindings are in scope for future bindings in the parameter list, allowing usage like this: `type Nonce (h : Host, key : HostKey(h)) = Nonce ( end (h getting key) );`

type: The value of the type this declaration is creating. Type parameters are in scope inside this binding.

serverProcessDeclaration ::= `"server"` variableMessage `"at"` variableMessage `"is"` `"("` variableDeclaration `")"` processBody

A ServerProcessDeclaration defines a new server process.

variableMessage #1: The name of the service that this process defines. This name must resolve to a public field, which must be of an untrusted type.

variableMessage #2: The name of the host where this process runs. This name must resolve to a public field, which must be of untrusted type.

variableDeclaration: This is the variable declaration for the socket that this server process uses to communicate with its client. The type of socket variable must be untrusted.

processBody: The body of the process is simply a sequence of statements, inside braces.

clientProcessDeclaration ::= `"client"` variableMessage `"at"` variableMessage `"is"` processBody

A ClientProcessDeclaration defines a new client process.

variableMessage #1: The name of the service that this process defines. This name must resolve to a public field, which must be of an untrusted type.

variableMessage #2: The name of the host where this process runs. This name must resolve to a public field, which must be of untrusted type.

processBody: The body of the process is simply a sequence of statements, inside braces.

processBody ::= `"{"` statement `"}"`

A process body defines the body of a client or server process. It is simply a brace-surrounded sequence of statements. Note that this production only specifies the body to be one statement. This is because each statement contains its successor statement, forming a list.

variableDeclaration ::= identifier `":"` type

A VariableDeclaration declares a new variable.

identifier: The name of the variable to be created

type: The type of the variable to be created

variableDeclarations ::= (variableDeclaration (`","` variableDeclaration)*)?

VariableDeclarations is a comma-delimited list of VariableDeclaration items.

effect ::= (`"end"` `"("` variableMessageSequence `")"` | `"check"` `"("` variableMessage `")"`)

An Effect represents an "effect". It can either be an end effect or a check effect. An end effect is produced by an end statement, and the messages in the end statement are the same as the messages in the corresponding end effect. A check effect is produced by checking a variable. This ensures that each variable is checked only once.

effects ::= (effect ("," effect)*)?

effects is a comma-separated list of effects.

type ::= ("(" type ")" | publicType | privateType | keyType | nonceType | structType | unionType | typeReference)

All data in Cryptyc have types. Here we declare all the possible forms of types.

privateType ::= "Private"

PrivateType is a built-in Cryptyc type. It (or its aliases) is used as the type of data that is secret. This then instructs the type checker to not allow the secret data to be directly sent over a socket. It must be encrypted to be transmitted safely.

publicType ::= "Public"

PublicType (or the untrusted type) is a built-in Cryptyc type. It (or its aliases) is used as the type of data that can be leaked to opponents without harm. Thus, it is used as the type for data that is OK to leak, and it is also the type of all encrypted data. This is because, unless the encryption is broken (which Cryptyc does not model), encrypted data can be safely passed on a public channel.

keyType ::= "Key" type

KeyType is the type that all symmetric crypto keys must have. Each instance of a KeyType is defined to encrypt/decrypt messages of a given type, which is specified when creating a KeyType.

nonceType ::= "Nonce" "(" effects ")"

A NonceType is the type of a nonce that is used to verify freshness for some sequence of effects. A nonce is casted to a NonceType(end x) after there has been a begin statement x. Then, the casted nonce is sent back to its caller, who reads it from the socket as a NonceType. Then, this new value can be compared to the original nonce challenge that was sent.

effects: This is the set of effects that this nonce is used to verify.

structType ::= "Struct" "(" variableDeclarations ")"

A StructType is a type representing a structure (a tuple).

variableDeclarations: These are the declarations of the components of the structure.

unionType ::= "Union" "(" variableDeclarations ")"

A UnionType is a type representing a tagged union. Unions are used because encryption keys can only encrypt one type of message. So, in order to use one key to encrypt two different types of message, a union of the two message types is created, and then the key can be a Key(uniontype).

variableDeclarations: Each possible case of the union is declared here as a variable. The tag is the name of the "variable" and the type of the "variable" is the type that the union holds when using that tag.

typeReference ::= identifier ("(" messages ")")?

A TypeReference is a reference to a user-defined type, with any applicable type parameters specified.

identifier: The name of the type being referenced

parameters: Some user-defined types take in messages as parameters. Here the actual parameter values are specified.

statement ::= (establishStatement | inputStatement | outputStatement | decryptStatement | newStatement | checkStatement | castStatement | beginStatement | endStatement)?

Statement is a statement that goes inside a Cryptyc process body. Each statement has a pointer to its succeeding statement.

establishStatement ::= `"establish"` variableMessage `"at"` variableMessage `"is"` `"("` variableDeclaration `")"` `";"` statement

An EstablishStatement forms a connection to a server process.

variableMessage #1: The name of the service to connect to

variableMessage #2: The host where the service to connect to runs

variableDeclaration: Declaration for the socket variable, which is used to communicate with the server.

outputStatement ::= `"output"` variableMessage `"is"` `"("` messages `")"` `";"` statement

An OutputStatement writes data to a socket.

variableMessage: Variable containing the socket to communicate over

messages: The messages containing the data to send

inputStatement ::= `"input"` variableMessage `"is"` `"("` patterns `")"` `";"` statement

An InputStatement reads data from a socket.

variableMessage: Variable containing the socket to communicate over

patterns: Patterns showing the form of data to be input. See pattern documentation for more details. Note that embedded VariablePatterns in patterns will result in new variables being declared!

decryptStatement ::= `"decrypt"` variableMessage `"is"` `"("` ciphertextPattern `")"` `";"` statement

A DecryptStatement decrypts data, allowing access to its unencrypted components.

variableMessage: The variable containing the data to be decrypted.

pattern: Pattern showing the form of data to be decrypted. This must be a ciphertext pattern, of course. See ciphertext pattern documentation for more details. Note that embedded VariablePatterns in patterns will result in new variables being declared!

newStatement ::= `"new"` `"("` variableDeclaration `")"` `";"` statement

A NewStatement declares a new variable, giving it fresh (but unknown and irrelevant) contents. This extinguishes any record of a previous check to that variable, since this is a new, fresh instance.

variableDeclaration: The declaration of the new variable and its type

beginStatement ::= `"begin"` `"("` variableMessageSequence `")"` `";"` statement

A BeginStatement marks the beginning of some event, which must correspond to an EndStatement of the same event in order for the protocol to be safe.

variableMessageSequence: The event that is being begun is made up of a series of messages.

endStatement ::= `"end" "(" variableMessageSequence ")" ";"` statement

An EndStatement marks the end of some event, which must correspond to a BeginStatement of the same event in order for the protocol to be safe.

variableMessageSequence: The event that is being ended is made up of a series of messages.

checkStatement ::= `"check"` variableMessage `"is"` variableMessage `";"` statement

A CheckStatement tests that a nonce read in is has the same value as a previously created nonce.

variableMessage #1: The original nonce, which must have been created by a NewStatement in the same process as this check.

variableMessage #2: The returned nonce, which must be of some NonceType. If the check succeeds, the effect of this nonce type is added to the set of current effects.

castStatement ::= `"cast"` variableMessage `"is" "(" variableDeclaration ")" ";"` statement

A CastStatement converts a nonce challenge that was input into a nonce of some NonceType to send it back.

variableMessage: The variable containing the nonce challenge, which must be of some untrusted type.

variableDeclaration: This declaration is where a casted version of the challenge is put. The type of this new variable must be some NonceType. Also, for this cast to succeed, if the type is NonceType(end x), a begin x statement.

message ::= (unionConstructorMessage | variableMessage | tupleMessage | ciphertextMessage)

A Message is a value or a reference to a value.

variableMessage ::= identifier

A VariableMessage is a reference to a previously defined variable.

unionConstructorMessage ::= identifier tupleMessage

A UnionConstructorMessage creates an instance of a union type. For example, if these types has been declared: `type S = Struct ( a : Public, b : Public ); type UT = Union( sTag : S, pubTag : Public );` Then these are both valid ways to output an instance of type UT: `output socket is (sTag(a, b)); output socket is (pubTag(a));` presuming both a and b are bound to public variables.

identifier: The tag specifying which case of which union is being referenced.

tupleMessage: A tuple containing the contents to be used to to instantiate the union.

tupleMessage ::= `"("` messages `")"`

A TupleMessage is a value that is a structure or tuple.

messages: The components (comma separated) of the tuple

ciphertextMessage ::= `"{"` messages `"}"` variableMessage

A CiphertextMessage is a value that represents a sequence of messages encrypted by a key.

messages: The components of the plaintext to be signed. If there is more than one message here, they form an implicit tuple.

variableMessage: A reference to the variable containing the key used for encryption. This variable must have type Key(x), where x is the type of the messages.

variableMessageSequence ::= (variableMessage)*

variableMessageSequence is a sequence of variable messages, with no delimiters.

messages ::= (message ("," message)*)?

messages is a comma-separated list of messages.

pattern ::= (variablePattern | unionConstructorPattern | constantPattern | tuplePattern | ciphertextPattern)

A pattern is used inside output and decrypt statements to show the form of the data that is being processed. The pattern models the overall structure of the data, allowing its constituent parts to be extracted.

constantPattern ::= identifier

A ConstantPattern is a pattern representing a reference to an already bound variable.

variablePattern ::= variableDeclaration

A VariablePattern is a pattern representing an unknown piece of data of a known type. When this pattern is applied, the unknown is required to have the type of the variable declared here, and the unknown value is put into the variable that this pattern declares.

unionConstructorPattern ::= identifier tuplePattern

A UnionConstructorPattern is a pattern representing a case of a tagged union. For example, if these types has been declared: type S = Struct (a : Public, b : Public); type UT = Union(sTag : S, pubTag : Public); Then these are both valid ways to create an instance of type UT: input socket is (sTag(a : Public, b : Public)); input socket is (pubTag(a : Public));

identifier: The tag specifying which union and which case this is

tuplePattern: A pattern representing the value of the union

tuplePattern ::= "(" patterns ")"

A TuplePattern is a pattern representing a tuple (a structure). It is represented as a comma-separated list of subpatterns that are surrounding by parentheses.

ciphertextPattern ::= "{" patterns "}" variableMessage

A CiphertextPattern represents encrypted data.

patterns: The patterns representing the plaintext data

variableMessage: The variable containing the key used to decrypt the data.

patterns ::= (pattern ("," pattern)*)?

patterns is a comma-separated list of patterns.

identifier ::= <IDENTIFIER>

An identifier consists of a letter, followed by a sequence of letters and digits. Note that in Cryptyc, dollar signs, underscores and single quote marks are all considered letters.

B New Cryptyc AST declaration

The abstract syntax description below is written in the language of ASTGen, an open-source tool by co-author Brian Stoler to generate automatically a composite-pattern hierarchy of classes and interfaces, along with the appropriate visitor classes.

```
// Source for AST classes for NewCryptyc to be run through ASTGen
// $Id: AST.ast,v 1.18 2001/12/08 03:40:43 eallen Exp $

visitmethod visit;
visitormethodprefix for;
tabsize 2;
outputdir ast;
allownulls no;

package edu.rice.cs.newcryptyc.ast;
import edu.rice.cs.newcryptyc.*;

// a parse consists of a list of Declarations.
begin ast;
interface CryptycAST(SourceLocation sourceLocation);

  abstract CryptycASTBase(ignoreForEquals SourceLocation sourceLocation);
    CompilationUnit(Declaration[] declarations);

    abstract Declaration();
      ImportDeclaration(CompilationUnit unit);

      abstract ProcessDeclaration(VariableMessage service, VariableMessage location, Statement body);
        ClientProcessDeclaration();
        ServerProcessDeclaration(VariableDeclaration socketVariable);

      TypeDeclaration(String name, VariableDeclaration[] parameters, Type value);
      PublicFieldDeclaration(VariableDeclaration variable);
      PrivateFieldDeclaration(VariableDeclaration variable);

    abstract Type();
      PublicType();
      PrivateType();
      NonceType(Effect[] effects);
      KeyType(Type dataType);
      TypeReference(String name, Message[] parameters);
      abstract CompoundType(VariableDeclaration[] components);
        StructType();
        UnionType();
      ErrorType();

    VariableDeclaration(String name, Type type);

    abstract Statement();
```

```
      LastStatement();
      NotLastStatement(Statement next);
        EstablishStatement(VariableMessage service, VariableMessage host, VariableDeclaration socketVa
        InputStatement(VariableMessage socket, Pattern[] patterns);
        OutputStatement(VariableMessage socket, Message[] messages);
        DecryptStatement(VariableMessage ciphertext, CiphertextPattern pattern);
        NewStatement(VariableDeclaration variable);
        CheckStatement(VariableMessage first, VariableMessage second);
        CastStatement(VariableMessage source, VariableDeclaration destination);
        BeginStatement(VariableMessage[] messages);
        EndStatement(VariableMessage[] messages);

    abstract Message();
      VariableMessage(String name);
      UnionConstructorMessage(String tag, TupleMessage contents);
      TupleMessage(Message[] components);
      CiphertextMessage(Message[] plaintext, VariableMessage key);

    abstract Effect();
      EndEffect(VariableMessage[] messages);
      CheckEffect(VariableMessage message);
      ErrorEffect();

    abstract Pattern();
      ConstantPattern(String name); // a reference to a pre-existing variable
      VariablePattern(VariableDeclaration declaration);
      UnionConstructorPattern(String tag, TuplePattern contents);
      TuplePattern(Pattern[] components);
      CiphertextPattern(Pattern[] plaintext, VariableMessage key);

end;
```

C Example protocols

C.1 prelude.cry

All of the other protocols include this prelude to define some basic types.

```
/*
 * Some standard definitions for use in Cryptyc programs
 * Alan Jeffrey v0.0.8 2001/03/08
 */

/*
 * Some standard types for use in protocols.
 */

type Payload = Private;
```

```
type Host = Public;
type Server = Public;
type Client = Public;
type Word = Public;
type Socket = Public;
type Challenge = Public;
type CText = Public;

/*
 * Some standard host names for use in protocols.
 */

public Alice : Host;
public Bob : Host;
public Charlie : Host;
public Eve : Host;
public Sam : Host;
```

C.2 fail-wmf.cry

```
/*
 * The Abadi/Gordon Wide Mouthed Frog protocol.
 * This is a varient of the Wide Mouthed Frog protocol
 * with nonce challenges rather than timestamps.
 * In this failed version, we don't use tags, so we get a type error.
 * Alan Jeffrey, v0.0.1 2001/02/08
 */

/*
 * Include the standard prelude
 */

import prelude;

/*
 * This file declares an initiator, a responder and an intermediary:
 */

public Initiator : Client;
public Responder : Server;
public Intermediary : Server;

/*
 * These make use of a Key Distribution Center (not defined here).
 */
```

```
public KDC : Server;
public KDCHost : Host;

/*
 * Words used in the session name.
 */

public sending : Word;
public key : Word;

/*
 * Types used in the protocol.
 */

type SKey =
  Key (Payload);

type WMFNonce (a : Host, b : Host, sKey : SKey) =
  Nonce (end (a sending b key sKey));

type WMFMsg (a : Host) =
  Struct (b : Host, sKey : SKey, nonce : WMFNonce (a, b, sKey));

type WMFKey (h : Host) =
  Key (WMFMsg (h));

/*
 * Types used for the server's dialogue with the KDC.
 */

type WMFKDCMsg = Struct (h : Host, keyH : WMFKey (h));
type WMFKDCKey = Key (WMFKDCMsg);

/*
 * The shared keys.
 */

private keyA : WMFKey (Alice);
private keyB : WMFKey (Bob);
private keyKDC : WMFKDCKey;

/*
 * The initiator.
 */

client Initiator at Alice is {

  new (sKey : SKey);
```

```
  begin (Alice sending Bob key sKey);
  establish Intermediary at Sam is (socket : Socket);
  output socket is (Alice);
  input socket is (nonceA : Challenge);
  cast nonceA is (nonceA' : WMFNonce (Alice, Bob, sKey));
  output socket is (Alice, { Bob, sKey, nonceA' }keyA);

}

/*
 * The responder.
 */

server Responder at Bob is (socket : Socket) {

  new (nonceB : Challenge);
  output socket is (nonceB);
  input socket is ({ a : Host, sKey : SKey, nonceB' : WMFNonce (a, Bob, sKey) }keyB);
  check nonceB is nonceB';
  end (a sending Bob key sKey);

}

/*
 * The intermediary.
 */

server Intermediary at Sam is (socketA : Socket) {

  input socketA is (a : Host);
  establish KDC at KDCHost is (socketKDC : Socket);
  output socketKDC is (a);
  input socketKDC is ({ a, keyA : WMFKey (a) }keyKDC);
  new (nonceA : Challenge);
  output socketA is (nonceA);
  input socketA is (a, { b : Host, sKey : SKey, nonceA' : WMFNonce (a, b, sKey) }keyA);
  check nonceA is nonceA';
  output socketKDC is (b);
  input socketKDC is ({ b, keyB : WMFKey (b) }keyKDC);
  establish Responder at b is (socketB : Socket);
  input socketB is (nonceB : Challenge);
  cast nonceB is (nonceB' : WMFNonce (a, b, sKey));
  output socketB is ({ a, sKey, nonceB' }keyB);

}
```