# 内生安全系统原型系统方案

一、模块功能设计

建模模块

验证模块

代码生成模块

二、信息处理设计（输入、输出）（何）

三、关键数据结构设计

（一）、模型数据结构

**1．Model**类

核心的底层数据结构，用于存储由xml文件生成的模型组件（包括类图、状态机、序列图等）。

属性：

list<Process\*> processes;

list<Property\*> properties;

list<InitialKnowledge\*> initialKnowledges;

list<SequenceDiagram\*> sequenceDiagrams;

list<AttackTree\*> attackTrees;

processes属性用于存储Model模型中的进程；

properties用于存储需要验证的安全属性；

initialKnowledges用于存储各个process中的初始知识；

sequenceDiagrams用于存储描述协议交互信息的序列图。

**2．Process**类

Model类中核心的数据结构，用于存储协议中进程的相关信息，包括属性、方法、信道、有限状态机等。

属性：

list<Attribute\*> attributes;

list<Method\*> methods;

list<Channel\*> channels;

list<StateMachine\*> stateMachines;

attributes是用于存储Process中定义的属性，支持原生的int、bool数据类型，用户还可以根据需求自己定义数据类型；

methods是用于存储Process中定义的所有方法；

channels是用于存储Process中定义的所有信道，信道是各进程之间用于通信的通道；

stateMachines是Process对应的状态系统，用来定义Process的行为。

**3．StateMachine**类

该数据结构用于存储进程行为的状态机，状态机中的每条转移表示从其源状态到目标状态所要进行的条件判断以及行为操作。

属性：

list<Vertex\*> vertices;

list<Edge\*> edges;

Vertex\* startVertex;

vertices是用于存储StateMachine中的状态集合；

edges用于存储StateMachine中所有的边集合，每条边上允许定义一个Guard语句以及若干个Action语句，其中Guard语句用于根据条件判断是否执行后面的语句，Action语句用于对属性的操作；

startVertex是用于存储StateMachine中的初始状态。

**4．SequenceDIagram**类

该数据结构用于存储进程交互的序列图，状态机中的每条转移表示从其源状态到目标状态所要进行的条件判断以及行为操作。

属性：

list<Vertex\*> vertices;

list<Edge\*> edges;

Vertex\* startVertex;

vertices是用于存储SequenceDIagram中的状态集合；

edges用于存储StateMachine中所有的边集合，每条边上我们运行定义一个Guard语句以及若干个Action语句，其中Guard语句用于根据条件判断是否执行后面的语句，Action语句用于对属性的操作；

startVertex是用于存储SequenceDIagram中的初始状态。

**5．Property**类

该数据结构用于存储用于验证的性质，其中包括系统安全性质以及信息安全性质。

5.1 SafetyProperty类

属性：

LTLFomular\* fomular ;

5.2 ConfidentialProperty类

5.3 AuthenticityProperty类

**6．AttackTree**类

（二）、Parse类

**1. Xml相关类**

Xml类定义xml在内存中的数据结构。Dtd类定义xml的文档头，其中定义了xml文件的元信息。XmlNode类定义了xml树结构的结点（也即子树）。XmlParser工具类用于实现Xml数据结构和系统中某些直接数据结构的相互转换。

Xml类的属性：

XmlNode\* root;

Dtd\* dtd;

其中root记录xml结构树的树根，即整棵xml树，dtd记录了xml的文档头。

XmlNode类的属性：

string name;

map<string, string> properties;

list<XmlTree\*> children;

bool isLeaf;

string content;

其中name记录了当前结点的名称，properties记录了当前结点的属性-值的映射，children记录了当前结点的直接孩子列表，isLeaf指示当前结点是否是叶子结点，如果是叶子结点，在content字段中记录结点的字符串信息。

XmlParser类的方法：

static Model\* forward(const Xml& xml);

static Xml\* backward(const Model& model);

static Process\* parseProcess(const XmlNode& subxml);

static FiniteStateMachine\* parseFST(const XmlNode& subxml);

static Property\* parseProperty(const XmlNode& subxml);

static InitialKnowledge\* parseIK(const XmlNode& subxml);

static SequenceDiagram\* parseSequenceDiagram(const XmlNode& subxml);

static AttackTree\* parseAttackTree(const XmlNode& subxml);

其中forward用于将Xml数据结构转换为Model数据结构，backward用于将Model转换为Xml数据结构，parseProcess用于从xml子树中解析出一个进程，parseFST用于从xml子树中解析出一个有限状态机，parseProperty用于从xml子树中解析出一个用于验证的性质，parseIK用于从xml子树中解析出一个先验知识，parseSequenceDiagram用于从xml子树中解析出一个序列图，parseAttackTree用于从xml子树中解析出一个攻击树模型。

（三）、Verifier类（王）

**1、Verifier类**

验证过程中的核心数据结构，用于记录验证过程中的Translator转化类，Model相对应的Beagle模型和Proverif模型的验证结果以及验证结果为false时的CounterExample类。

**属性：**

Translator\* translator;

CounterExample\* beagleCE;

CounterExample\* proverifCE;

bool verifyBeagleResult;

bool verifyProverifResult;

translator是用来进行模型转换的类。

BeagleCE包含了验证结果为false时验证工具Beagle返回的CounterExample。

ProverifCE包含了验证结果为false时验证工具Proverif返回的CounterExample。

verifyBeagleResult是模型Model转换成Beagle模型后的验证结果。

verifyProverifResult是模型Model转换成Proverif模型后的验证结果。

**重要方法：**

void setTranslator(Translator\* \_translator);

bool verifyBeagleModel();

bool verifyProverifModel();

CounterExample\* getCounterExample(string verifyToolName);

setTranslator方法主要是设置Verifier类中的转换器Translator类。

verifyBeagleModel方法是将Model通过调用Translator中的方法转换成对应的Beagle模型之后，再调用Beagle工具进行验证并返回验证结果。

verifyProverifModel方法是将模型Model转换为Proverif模型后在调用Proverif进行验证并返回验证结果。

getCounterExample方法根据给定的参数verifyToolName选择对应的验证工具得到的验证结果的CounterExample并返回。

**2、Translator类**

这个类是负责将之前得到的模型Model进行转换的类。主要分成两个部分BeagleTranslator和ProverifTranslator分别是对应的验证工具的转换类。

**属性：**

Model\* model;

BeagleModel\* beagleModel;

ProverifModel\* proverifModel;

model中存放的是在之前生成的模型数据结构。

beagleModel中的是由model生成的Beagle模型的数据结构。

proverifModel中的是生成的Proverif模型的数据结构。

**重要方法：**

Translator类中的方法主要分成两个部分，一个是用来进行Model到BeagleModel的转换的方法，另一类是从Model到ProverifModel的转换的方法，下面将分别介绍这两部分中会用到的一些重要方法。

**Model到BeagleModel：**

BeagleModel\* generateBeagle();

makeHeader();

makeModules();

getProperties();

generateBeagle方法主要是从Transltor类中的Model生成对应的Beagle模型数据结构beagelModel，其中又包含了三个方法：makeHeader(), makeModules()和getProperties()。

makeHeader方法是负责生成beagleModel中的文件头部分的方法，主要包含了Beagle模型中模块的位置，变量以及标签的定义部分。

makeModules方法是根据Model中的Process的定义生成对应的Beagle模型的中的模块module的内容。主要是根据每一个Process中的StateMachine类中的edges生成对应的迁移，以及根据Model中的channel通信行为建立对应的带有label标签的迁移。

getProperties方法是从Model文件中的SafetyProperty生成对应的Beagle模型所需要验证的Property语句。

**Model到ProverifModel：**

ProverifModel\* generateProverif();

void makingStartingProcess();

void makeHeader();

void makeBlocks();

generateProverif方法主要做的是将Translator类中的Model转换成对应的Proverif模型数据结构proverifModel，其中主要用到了三个方法：makingStartingProcess(), makeHeader()和makeBlocks()。

makingStartingProcess方法主要做的是生成Proverif模型中的主进程的内容以及从Model的initialKnowledges中获得每一个进程所使用到的initalKnowledge。

makeHeader方法根据Model中methods生成对应的Proverif模型的函数定义，以及生成Model中ConfidentialProperty和AuthenticityProperty所对应的query查询语句。

makeBlocks方法主要是将Model中的每一个Process都根据各自的StateMachine生成对应的Proverif模型。

**3、BeagleModel类**

BeagleModel类是Verifier类中用来表示Model转换成Beagle模型之后的数据结构。BeagleModel中包含了BeagleModule,BeagleProperty和BeagleDeclaration类，分别代表了Beagle模型中的模块，验证属性以及声明。

**属性：**

list<BeagleModule\*> modules;

list<BeagleProperty\*> properties;

list<BeagleDeclaration\*> declarations;

modules中的是BeagleModel中由Model中的Process获得的对应的Module类数据结构。

properties中的是BeagleModel中由SafetyProperty转换而来的Beagle模型中对应的验证属性的数据结构。

declarations中的是BeagleModel中Beagle模型的位置，变量以及标签的声明的数据结构。

**重要方法：**

void addModule(BeagleModule\* \_module);

void addProperty(BeagleProperty\* \_property);

void addDeclaration(BeagleDeclaration\* \_declaration);

addModule方法可以向BeagleModel类中添加新的BeagleModule类，其存放的是对应的Process类转换得到的BeagleModule。

addProperty方法可以向BeagleModel类中添加由SafetyProperty类转换的来的BeagleProperty内容。

addDeclaration方法是添加在generateBeagle方法中生成的Beagle模型需要新的声明语句。

**4、ProverifModel类**

ProverifModel类是Verifier类中用来表示Model转换成Proverif模型之后的数据结构。ProverifModel中包含了ProverifProcess和ProverifDeclaration类，分别表示了Proverif模型中的进程以及声明。

**属性：**

ProverifProcess\* mainProcess；

list<ProverifDeclaration\*> declarations;

mainProcess是Proverif模型中的主进程。

declarations包含了Proverif模型中用到的所有声明，包括了Method对应的函数的声明，ConfidentialProperty对应的Query的声明以及AuthenticityProperty对应的Query的声明。

**重要方法：**

void setMainProcess(ProverifProcess\* \_mainProcess);

void addDeclaration(ProverifDeclaration\* \_declaration);

setMainProcess方法负责生成Proverif模型中的主进程对应的数据结构以及每个进程需要用到的initialKnowledge。

addDeclaration方法会将generateProverif中用到的转换Proverif模型所需要的新的声明语句添加至declarations中。

**5、CounterExample类**

CounterExample类是负责存储验证结果为false时，验证工具返回的反例的数据结构。CounterExample类中包含了两个子类BeagleCounterExample和ProverifCounter Example分别表示了Beagle和Proverif所生成的CounterExample数据结构。

**属性:**

string resultFile;

string type;

list<Vertex\*> vertices;

list<Edge\*> edges;

vertex\* startVertex;

list<string> attackerTraces;

int step;

resultFile是用字符串记录的验证工具返回的结果文件。

type记录了当前CounterExample类是Beagle工具生成的还是Proverif工具生成的。

vertices中的是BeagleCounterExample的反例路径中所包含的所有节点。

edges中的是BeagleCounterExample的反例路径中的所有迁移边。

startVertex记录的是BeagleCounterExample中的反例路径的其实节点。

attackerTraces中记录的是ProverifCounterExample中所包含的攻击者攻击的路径。

step中存放的是attackerTraces中的路径包含了多少步。

**重要方法：**

string getCounterExampleType();

string getResultFile();

void createFromFile();

list<string> getAttackerTraces();

getCounterExampleType方法会得到当前的CounterExample类是由那个验证工具所生成的。

getResultFile方法会获得验证工具所返回的验证结果文件。

createFromFile方法会从验证结果文件生成对应的CounterExample。若是Beagle工具的结果文件则会生成对应的反例路径，若是Proverif工具的结果文件则会生成对应的攻击者路径。

getAttackerTraces方法则会返回ProverifCounterExample中的攻击者路径。

（四）、CodeGenerator类（王）

该类是用于代码生成模块的抽象类。

包含的属性有：Model\* model;

包含的方法有：void generateCode(string path)

该函数定义了生成代码的方法接口，被SimulationGenerator和ImpleCodeGenerator实现。

**1. SimulationGenerator类**

该类用于生成可以模拟系统运行过程的C代码，生成的代码中的方法未被实现。

1.1 重要方法

string generateHeaderFile(const Process\* proc):

用于生成.h文件，文件中包括需要使用的库函数引用，以及需要用到的生成的.h文件和通讯需要用到的部件。

string generatePragmas(const FiniteStateMachine\* fsm):

生成Pragma，其中包括对自动机的状态的define。

string generateMain(const FiniteStateMachine\* fsm):

生成Main函数的部分，其中重要的是用generateFSMCode方法和Process中的FSM生成自动机的迁移，自动机的运行过程是一个循环，记录当前所在的状态，并根据是否满足迁移的guard进行自动机的迁移，并执行迁移上的动作。

string generateProcMethods(const Process\* proc):

生成proc中的方法的代码，此处生成时并不对方法进行实现

void generateSimCCode(string path, list<Process\*> procs):

生成所有的代码文件到文件目录path中，该方法对procs中的每一个进程都生成.h和.c文件到目录path。

**2. ImpleCodeGenerator类**

该抽象类为有具体实现代码的类的设计提供接口。

2.1 重要方法

string generateDecryptAlgorithm()=0;

生成解密算法的函数。

string generateEncryptAlgorithm()=0;

生成加密算法的函数。

void generateCode(string path, Process\* proc):

对一个Process最终生成其源代码的方法，并将文件输出到目录path。

2.2 说明

ImpleCodeGenerator还应有对应的子类实现，例如CPPCodeGenerator，在这些类中需要维持一个插件的list，生成加密解密算法的具体部分需要调用相应的插件。

**3. CPPCodeGenerator类**

该类是ImpleCodeGenerator的子类，用于生成单个Process的实现，该类的每个实例都有一个插件的列表，在生成代码的过程中如果要生成算法，会在插件列表中寻找是否存在符合条件的相关插件，并用于代码生成。

3.1 属性

list<Plugin\*> plugins：

保存了一个插件的数组，插件在代码生成的过程中用于生成实际的算法代码

3.2 方法

void addPlugin(Plugin\* plugin)

Plugin\* getFirstPlugin()

Plugin\* rmPlugin()

void generateHeaderFile(string path, Process\* proc)

void generateSrcFile(string path, Process\* proc)

string generateMethods(Process\* proc)

string generateMain(Process\* proc)

string generateDecryptAlgorithm(string algorithmId): 根据需要使用的算法在插件列表中寻找是否有对应的插件，用其生成相应代码。

String generateEncryptAlgorithm(string algorithmId)

void generateCode(string path, Process\* proc)

五、接口设计

内部接口设计

1.1 建模模块与模型接口 （刘）

1.2 验证模块与模型接口 （王）

1.3 代码生成模块与模型接口 （李）

外部接口设计（何）

六、执行序列设计（何）