软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(6):1581−1596 [doi: 10.13328/j.cnki.jos.006238]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

C2P:基于 Pi 演算的协议 C 代码形式化抽象方法和工具*

张协力[1,2], 祝跃飞[1,2], 顾纯祥[1,2], 陈 熹[1,2]

[1](数学工程与先进计算国家重点实验室,河南 郑州 450001)

[2](网络密码技术河南省重点实验室,河南 郑州 450002)

通讯作者: 顾纯祥, E-mail: gcxiang5209@163.com

摘 要: 形式化方法为安全协议分析提供了理论工具,但经过形式化验证过的协议标准在转换为具体程序实现时,可能无法满足相应的安全属性.为此,提出了一种检测安全协议代码语义逻辑错误的形式化验证方法.通过将协议 C 源码自动化抽象为 Pi 演算模型,基于 Pi 演算模型对协议安全属性形式化验证.最后给出了方案转换的正确性证明,并通过对 Kerberos 协议实例代码验证表明方法的有效性.根据该方案实现了自动化模型抽象工具 C2P 与成熟的协议验证工具 ProVerif 结合,能够为协议开发者或测试人员检测代码中的语义逻辑错误提供帮助.

关键词: 协议实现;形式化验证;Pi 演算;模型抽取;ProVerif

中图法分类号: TP311

中文引用格式: 张协力,祝跃飞,顾纯祥,陈熹.C2P:基于 Pi 演算的协议 C 代码形式化抽象方法和工具.软件学报,2021,32(6):1581−1596. http://www.jos.org.cn/1000-9825/6238.htm

英文引用格式: Zhang XL, Zhu YF, Gu CX, Chen X. C2P: Formal abstraction method and tool for c protocol code based on Pi caculus. Ruan Jian Xue Bao/Journal of Software, 2021,32(6):1581−1596 (in Chinese). http://www.jos.org.cn/1000-9825/6238.htm

C2P: Formal Abstraction Method and Tool for C Protocol Code Based on Pi Caculus

ZHANG Xie-Li[1,2], ZHU Yue-Fei[1,2], GU Chun-Xiang[1,2], CHEN Xi[1,2]

[1](State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

[2](Henan Key Laboratory of Network Cryptography Technology, Zhengzhou 450002, China)

Abstract: Formal method provides a theoretical tool for security protocol analysis, but the theoretical security is not equivalent to the actual security. A verified protocol standard may not meet the required security properties when converted into a concrete program. Hence, a formal verification method for detecting semantic logic errors in security protocol code is proposed. By automatically abstracting the C source code of the protocol into Pi calculus model, protocol security properties are verified based on the Pi calculus. Finally, the correctness of the scheme transformation is proved and the validity of the method is verified by a Kerberos protocol instance code. C2P tools implemented can help protocol developers to detect semantic logic errors in code.

Key words: protocol implementation; formal verification; Pi calculus; model extraction; ProVerif

形式化方法在对安全协议的分析中取得了显著成果,并促进了很多协议草案的设计完善.但理论上的安全并不能保证实际部署中协议的安全属性满足,经过形式化验证安全的协议,在转换成具体实现时可能依然存在安全问题.协议实现中的安全问题既有一般性的低级别错误如缓冲区溢出,也包含了语义实现中的逻辑错误:第 1 类安全问题可以借助通用的代码审计技术等来弥补;第 2 类语义逻辑错误问题更为隐蔽,难以借助现有的软件测试技术如 Fuzzing 来发掘.

* 基金项目: 国家重点研发计划(2019QY1302)
Foundation item: National Key Research and Development Program of China (2019QY1302)
本文由“形式化方法与应用”专题特约编辑田聪教授推荐.
收稿时间: 2020-07-29; 修改时间: 2020-10-26, 2020-12-19; 采用时间: 2021-01-18; jos 在线出版时间: 2021-02-07

为了发现协议实现中的语义逻辑错误,研究人员提出了对协议实现代码同样进行形式化验证的思路[1].程序设计语言本身在语法语义上复杂表达能力丰富,难以直接在其上进行形式化的验证推演,因而对协议代码的形式化分析都是遵循先抽象再验证的工作思路开展.通常做法是:将协议源码抽象为某种形式化语言模型,再基于该抽象模型进行协议安全属性的形式化验证.不同语言实现的协议在分析时存在较大差异性,多数文献都是针对协议的某种具体的编程语言实现提出相应的形式化方法[2−4].近些年,协议代码形式化验证方面的研究工作目前以对 JavaScript[5,6],Python[6]和 Java[7]这类高级语言编写的协议代码分析为主,而针对较偏底层的 C 语言协议实现形式化工作相对较少,且在实用性上存在一定限制,例如不支持 C 的指针语法[1,8]、对条件分支和函数调用无法自动化抽象[9,10]等.而安全协议开发中C语言的所占比重,对协议C代码的形式化验证工作更具现实安全需求.

本文的研究对象是 C 语言实现的协议代码的逻辑语义检测,忽略一般的代码级问题如内存类 bug,旨在设计并实现对协议 C 源码自动地形式化验证方法.根据 C 语言面向过程的编程特点,按照函数调用-函数体语句-表达式的顺序将协议 C 程序逐步抽象为 Pi 演算模型,利用成熟的协议自动验证工具 ProVerif[11]对代码的抽象模型进行验证.ProVerif 是一个在 Dolev-Yao 敌手模型下自动化分析协议安全属性的工具,能够自动化证明可达性属性和观察等价性.这些能力允许对机密性和认证属性进行分析,此外,还可以用于刻画诸如匿名性、可追溯性等,并且可以在无限数量的会话和无限的消息空间下进行推理.该工具还能够进行攻击路径重构.这些针对协议的成熟的解决方案是一般的代码验证工作所不具备的.根据抽象方案实现了自动化抽象工具 C2P(https://github.com/85lx/C2P),并将其开源.在对协议实现代码中的密码与通信原语及相关标准库函数处理时,采用当前这一方向的通用处理方式,将其视为黑盒,不探索这些库函数的细节;提出了获取静态函数调用关系序列的算法,并在静态执行下函数调用关系序列基础上,给出了函数调用关系在 Pi 演算逻辑语言中的表达方法;提出了基于函数上下文的表达式归约算法,对 C 语言中涉及到的基本代数和逻辑运算进行规约,降低了抽象模型的逻辑复杂度.

本文的主要贡献如下:

(1) 提出一种对协议实现的C源码形式化验证的方法.该方法可以将协议C代码转换为Pi演算抽象模型,进一步在 Dolve-Yao 敌手模型下验证相关的安全属性是否满足;

(2) 提出了函数调用时序图(function call sequence graph,简称 FCSG)结构和静态执行下函数调用序列获取算法,并基于静态执行下的函数调用序列,给出了 C 程序中函数调用在 Pi 演算模型中的抽象方法,给出了其合理性证明;

(3) 依据方案实现了协议C代码的自动化抽象工具C2P并将其开源,通过 Kerberos 协议代码,验证分析说明了方法和工具的有效性.

本文第 1 节介绍相关工作.第 2 节简介协议源码验证及 Pi 演算基础知识.第 3 节阐述协议 C 源码到 Pi 演算模型的抽象方法.第 4 节介绍 C2P 工具实现和方法合理性证明以及代码实例上的实验.最后,第 5 节总结全文.

1 相关工作

对于协议代码的形式化验证,与之前的协议标准的形式化分析工作有着很大区别.程序设计语言本身在语法语义上复杂表达能力丰富,难以直接在其上进行形式化的验证推演,因而对现有协议代码的形式化分析都是遵循的先抽象再验证的逻辑思路开展.文献[11]是对协议实现安全性分析最早的工作之一,通过程序开发人员在代码中的注释建立信任断言模型,将协议的 C 语言程序抽象成霍恩子句,基于霍恩子句的逻辑理论,在信任断言模型中进行协议机密性验证.文献[8]是第 1 个将软件模型检测与标准协议安全模型相结合、自动分析 C 语言中实现协议中的认证性和机密性的框架.文章中用谓词抽象技术将具体的协议程序翻译成更为抽象的 ASPIER 模型,并采用 ASPIER 分析并发现了 SSL 握手协议的 OpenSSL0.9.6c 源代码中存在的“版本回滚”漏洞.不足之处在于:该项工作没有很好地处理 C 语言的主要特性是指针和内存操作这两大特性,因而需要大量人工参与,实用性受限;另一方面的不足是,文中采用自定义的 ASPIER 抽象模型相对较复杂,研究人员得熟练掌握

ASPIER 抽象模型才可以在此基础上开展分析工作,因而通用性也有一定损失.类似的工作有文献[13–15],都是将现有的协议代码抽象成自定义的模型再进行逻辑推演.

文献[9]的工作弥补了这一不足,将安全协议的 C 语言代码转换成为 Pi 演算模型.Pi 演算是一种通用而语法简洁的协议模型描述语言,且可以直接作为协议自动验证工具 ProVerif 的输入语言模型,从而在符号模型下对协议 C 代码进行自动化的形式化验证分析.相关团队在文献[10]中将该项工作拓展到计算模型,利用 CryptoVerif 工具进行计算模型下的协议形式化验证.这一工作的不足之处在于:仅支持单一执行路径,不支持函数调用和条件分支、循环迭代等,在一定程度上削弱了验证框架的适用范畴.但该工作所体现的将协议的 C 语言代码抽象成已经存在且较成熟的 Pi 演算逻辑模型的研究思路,既方便后续工作拓展,也有利于基于已有形式化自动化验证工具进行协议代码的安全属性分析.与之类似的工作还有文献[16],其采用类似的思路将程序设计语言 F#所编写的协议代码转换为 Pi 演算和 Blanchet 演算,然后借助 ProVerif 工具进行自动化的形式化验证.还有一部分工作将这种思路应用到 Java 语言编写的协议代码形式化验证中,如文献[7],通过将安全协议的 Java 代码抽象成 Blanchet 演算,从而利用 CryptoVerif 工具对抽象后的模型进行自动化的形式化验证分析.文献[17,18]也是采用先抽象再验证的思路,与其他工作的不同之处在于,这些工作应用了 Petri 网理论模型:首先将安全协议程序转换成对应的 Petri 网模型,然后借助 Petri 网理论来分析协议的通信行为是否满足相关的安全属性.此项工作的意义在于为协议实现的形式化分析研究引进了新的工具 Petri 网理论,但同时仍然有很多待研究解决的问题.

本文沿用了先模型抽象再验证的研究思路.与现有工作相比,直接在源码层面做抽象,支持更多的 C 语法抽象如函数调用,适用于代码规模较大的协议实现分析.另一方面,现有工作的共性不足之处在于工作缺乏延续性,或仅限于团队内部延续,这很大程度上源于该领域在复现他人工作方面并不平凡.为此,本文将 C2P 工具的代码开源,希望对这一现状有所改善.

2 相关基础

2.1 协议源码验证

为理解协议协议源码验证和传统的源码安全测试工作的不同,在此给出协议源码的验证的定义.

定义 2.1(协议源码验证). 给定一个协议 *Pro* 的规范 $S(Pro)$和由程序设计语言 L 编码的协议实现 $P(Pro)[L]$,验证 $S(Pro)$中可满足的安全属性φ在 $P(Pro)[L]$中也是可满足的过程:

$$S(Pro)\models\varphi\Rightarrow P(Pro)[L]\models\varphi$$

称为对协议 *Pro* 的 L 源码的安全属性验证,简称为协议源码验证.

根据定义 2.1,协议源码验证工作的对象是某种具体编程语言实现的安全协议源码,如本文的研究对象即 C 语言实现的协议源码,即 $P(Pro)[C\text{-}Language]$.研究的目标是安全属性是否满足,而非一般的代码级问题.

2.2 Pi演算语法

Pi 演算(Pi calculus)[12]是一个用于并行系统之间通信行为建模的理论,其模型简洁且具有强大的表达能力.著名的安全协议验证工具 ProVerif 的输入语言便是扩展的 Pi 演算.本节对这种扩展的 Pi 演算语言进行简单介绍,为便于描述,后文提到的 Pi 演算特指 ProVerif 的输入模型.

Pi 演算语法中的项(term)是由名(names)、变量(variables)和构造/析构函数(constructor/destructor)组成的.形式化定义如下:

$$M,N::=a,b,c,\ldots|x,y,z,\ldots|h(M_1,\ldots,M_k)|M=N|M<>N|M\&\&N|not(M),$$

其中,$a,b,c\in C$;$x,y,z\in V$;$h\in\Sigma$为 names 的集合,V 为变量集合,Σ为函数符号集合;=和<>表示等式和不等式关系,&&,||和 not 对应与或非逻辑关系;项可以用于描述安全协议通信中的消息,函数可以用于对加密原语的描述.

在 Pi 演算中,进程(process)是用于描述一系列协议行为.Pi 演算中,进程的定义见表 1.其中:空进程表示无实义操作;并发进程为两个进程同时运行;创建新名是为进程 P 创建私有名;条件语句为比较项 M 和 N,相等则执行

进程 *P*,否则执行进程 *Q*;赋值语句为变量 *x* 赋值为项 *M*,赋值成功执行进程 *P*,失败执行进程 *Q*;接收消息和发送消息操作模拟协议通信中的发送和接收消息行为.

Table 1 Process grammar in Pi calculus

表 1 Pi 演算进程语法

P|*Q*|*R*::=	进程(process)
0	空进程
P|*Q*	并发进程
!*P*	重复进程
new *n*:*t*; *P*	创建新名
if *M*=*N* **then** *P* **else** *Q*	条件
let *x*=*M* in *P* **else** *Q*	赋值
in(*M*,*x*:*t*); *P*	接收消息
out(*M*,*N*); *P*	发送消息

一个协议实体的通信行为可以用这些语法进行描述.例如对称加解密操作的表示中,加密过程看作不解义的构造函数(constructor):*enc*(*x*,*y*)解密操作视为析构函数(destructor),加解密之间的联系用等式理论代替:

$$dec(enc(x,y),y)=x.$$

类似地,公钥加密算法的表示分别对应如下:

$$enc(x,pk(y));$$

$$dec(enc(x,pk(y)),y)=x,$$

其中,*pk*(*y*)表示私钥 y 对应的公钥,同样为不解译函数.采用类似的等式理论可以表示私钥签名等操作.

ProVerif 所支持的 Pi 演算的主要语法描述如上,更多细节可参考相关文档[11].

3 从协议 C 代码提取 Pi 演算模型

对安全协议代码进行形式化分析的基本思路是:将协议 C 源代码转换成为 Pi 演算模型,之后借助协议自动化分析工具 ProVerif 对抽象后的 Pi 模型进行形式化验证分析.我们依据 C 语言面向过程的这种以函数调用组织起来的结构特点,依次从 C 程序中的函数调用、函数体语句、语句中的表达式这 3 个不同的层次来对安全协议 C 程序代码来实施形式化抽象.图 1 给出了协议 C 源码验证的整体框架.

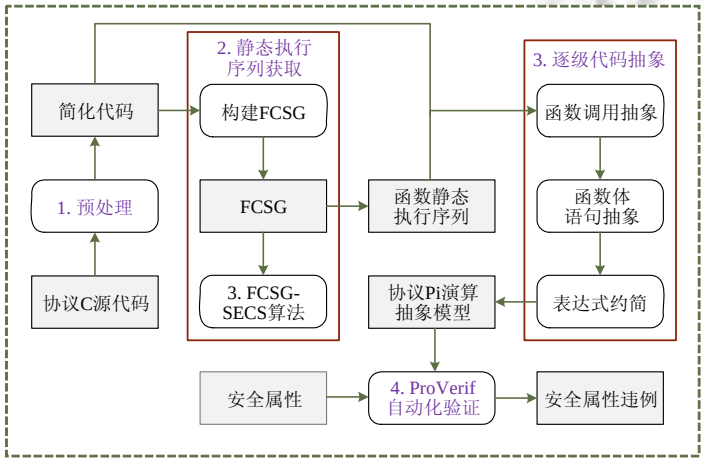

Fig.1 Overall security verification framework of C protocol code

图 1 协议 C 源码的安全验证架构

整个验证框架分为 4 个步骤.

(1) 对待分析的目标协议源码进行预处理,预处理任务主要包括基本的预编译处理,如消除宏相关指令、

头文件处理以及对密码原语操作相关函数、通信模块相关函数以及标准库相关函数的重写;

(2) 对简化后的代码,通过构建函数调用时序图和运行 FCSG-SECS 算法,得到源码对应的函数静态执行序列;

(3) 在函数调用静态执行序列基础上,对函数调用关系抽象得到协议源码的 Pi 演算模型框架,再逐步函数体语句抽象、表达式约简得到协议源码的 Pi 演算抽象模型;

(4) 对得到的抽象模型,借助 ProVerif 自动化验证协议规范中规定的安全属性.

第(4)步验证工作借助了现成的支持 Pi 演算模型验证的 ProVerif 工具,只需添加相应的待验证安全属性,此工作不再赘述.

下面对前 3 步内容涉及的细节进一步阐述.

3.1 对协议代码的预处理

直接在程序设计语言尤其是 C 这样偏底层的语言编写的协议代码上进行形式化分析工作比较困难,需要对协议源码进行预处理,转换为适合模型抽取的简化代码.

本方案对源码的预处理包含两方面工作.第 1 部分工作是常规的编译预处理,消除宏定义、条件编译宏以及头文件包含等.这部分处理借助通用的 C 编译器前端预处理选项即可.有所区分的是:“头文件包含”仅处理协议源码中自定义头文件,对于标准库头文件以及源码项目第三方头文件采用直接注释.这样是为了配合预处理的第 2 部分工作:函数黑盒化处理.具体需要识别并处理的预处理指令见表 2.

Table 2 Common compilation preprocessing instructions

表 2 常见编译预处理指令

编译预处理指令	说明
#define	宏定义
#undef	撤销宏
#if	条件宏
#ifdef	宏已定义,则编译
#ifndef	宏未定义,则编译
#elif	和#if 配合使用
#endif	结束条件编译
#include	头文件包含

对一个函数的黑盒化处理是指不进入该函数的函数体中去解析该函数的执行逻辑是否正确,而是默认该函数的实现逻辑和它所声明的执行功能一样.密码学是安全协议实现安全属性的重要基石,在安全协议对应的代码中也存在很多密码原语相关的操作函数.密码学上一般通过多轮混淆(confusion)、扩散(diffusion)和移位(shift)等操作来抵抗敌手的分析.密码原语函数的相关代码中同样存在这些逻辑,对这些逻辑的分析会增加协议源码的分析难度.在基于 Dolve-Yao 模型的协议规范形式化分析通常采用完美密码假设来处理这一问题,在对代码的形式化分析中同样可以采用这一思路.将协议代码中密码原语操作函数按照 Pi 演算逻辑中的项重写和等式规则直接替换.同样的,对于协议代码中的通信模块相关函数如 socket 编程接口,也不去探索这些函数实现细节,用逻辑语言中的消息发送和接收逻辑替代.这一处理可以扩大到标准库或者第三方库中的函数,甚至是协议源码中任意可信任的函数实现.协议代码的预处理交由用户手工标识配合程序半自动化完成,因为 C2P 工具的使用对象定位为协议代码开发者或者对代码熟悉的专业人员,而非一般用户.

3.2 获取函数静态执行序列

函数是对程序代码逻辑模块化的编程方式,采用函数组织程序有助于将复杂的程序代码结构分割成代码块集合.一个 C 语言程序的结构可以表示为一个函数执行序列.

考虑到 C 程序的面向过程的函数结构,我们从程序的函数静态执行序列入手,对程序从结构上进行解析.为了得到函数静态执行序列,我们设计了能全面反映协议 C 程序代码的函数调用关系的数据结构函数调用时序图,用缩写 FCSG 表示.

FCSG 是一个四元组结构 $G=(F,R,S,f_0)$,其中,

- F 是一个符号集合;
- R 是定义在 F 上的二元关系:$R=\{\langle f_i,f_j\rangle_k|f_i\in F,f_j\in F,k\in Z^+\}$,下标 k 表示对集合中相同的 f_i 和 f_j 构成的关系的区分.且$\forall k\geqslant 2$,若$\langle f_i,f_j\rangle_k\in R$,则$\langle f_i,f_j\rangle_{k-1}\in R$.当 $k=1$ 时,简记为 $r=\langle f_i,f_j\rangle$;
- S 是集合 R 到正整数集 Z+的映射,满足:
  - $\forall\langle f_i,f_j\rangle_k\in R$,有 $S(\langle f_i,f_j\rangle_k)=s,s\geqslant k$;
  - $\forall\langle f_i,f_j\rangle_k\in R,\forall\langle f_i,f_{j'}\rangle_{k'}\in R$,若$\langle f_i,f_{j'}\rangle_{k'}\neq\langle f_i,f_j\rangle_k$,则 $S(\langle f_i,f_{j'}\rangle_{k'})\neq S(\langle f_i,f_j\rangle_k)$;
  - $\forall S(\langle f_i,f_j\rangle_k)=s$,若 $s>1$,则$\exists\langle f_i,f_{j'}\rangle_{k'}\in R,S(\langle f_i,f_{j'}\rangle_{k'})=s-1$;
- $f_0\in F$,且$\nexists f\in F,\langle f,f_0\rangle\in R$.

符号集合 F 对应一个 C 程序 P 中的所有函数集合.考虑 C 语言中不支持函数重载,故可用函数名表示程序中的函数.集合 R 用来刻画程序 P 中的函数调用关系,$\forall f_i\in F,f_j\in F$,若程序 P 中的函数 f_i 的定义中存在函数 f_j 的调用语句,且在函数中的调用是第 k 次出现,则有$\langle f_i,f_j\rangle_k\in R$;若该次调用是 f_i 函数体中出现的第 s 个函数调用语句,则有 $S(\langle f_i,f_j\rangle_k)=s$.程序 P 的入口函数记为 f_0,显然,一个良好程序的入口函数不会在其他函数中被调用.在关系集合 R 中,k 是对相同的函数调用关系的编号,用以区分这些相同函数调用.

在协议代码 FCSG 结构的基础上,定义获取函数调用静态执行序列(static execution of call sequence,简称 SECS)算法,将其命名为 FCSG-SECS 算法,算法描述如下:

算法 1. 函数调用静态执行序列获取算法 FCSG-SECS(G).

输入:$G=(F,R,S,f_0)$;

输出:函数调用静态执行序列 *SECS*.

1. *SECS*=[·];
2. **If** $G.F=\varnothing$ **then**
3. **Return** [·];
4. **EndIf**
5. $R_0=\{\langle G.f_0,f'\rangle_k|\langle G.f_0,f'\rangle_k\in G.R\}$;
6. 将 R_0 按照 $G.S$ 的映射值升序排序;
7. **Foreach** $\langle f_0,f'\rangle_k\in R_0$ **do**
8. *SECS.append*($\langle f_0,f'\rangle_k$);
9. $G'=(F/f_0,R/\langle f_0,f'\rangle_k,S/(\langle f_0,f'\rangle_k\rightarrow s'),f')$;
10. *SECS.append*(*FCSG-SECS*(G'));
11. **EndFor**
12. **Return** *SECS*

函数调用时序图中蕴含了静态模拟执行代码的函数调用顺序.从入口函数出发,对协议代码对应的 FCSG 按出边的标号属性顺序(升序),遍历所有函数调用关系,便可以得到程序代码的函数调用执行序列.我们用图 2 中的示例代码来具体说明.图 2 是一段示例代码和其对应的 FCSG 图.

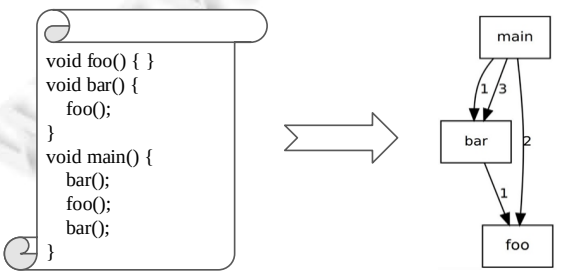

Fig.2 Example for function calls and FCSG

图 2 函数调用及 FCSG 示例

示例代码对应的 FCSG 表示为 $G=(F,R,S,f_0)$,其中,

- $F=\{main,bar,foo\}$;
- $R=\{\langle main,bar\rangle,\langle bar,foo\rangle\langle main,foo\rangle,\langle main,bar\rangle_2\}$;
- $S=\{\langle main,bar\rangle\rightarrow 1,\langle main,foo\rangle\rightarrow 2,\langle main,bar\rangle_2\rightarrow 3,\langle bar,foo\rangle\rightarrow 1\}$;
- $f_0=main$.

通过 FCSG-SECS 算法可得到程序代码在静态执行下的函数访问序列:

$$[\langle main,bar\rangle,\langle bar,foo\rangle,\langle main,foo\rangle,\langle main,bar\rangle,\langle bar,foo\rangle].$$

这与代码在静态执行过程中的函数调用顺序是一致的(实际执行中的函数调用序列可能会因为条件判断语句等因素产生偏差).为了简化分析,还可以根据人工经验,将一些无用的 FCSG 分支修剪.

从程序 P 的 C 语言程序源码构建程序对应的 FCSG,根据程序源码的抽象语法树识别定位所有函数声明、函数定义、函数调用点位置,进一步构建协议代码实现的函数调用时序图.有很多现成的开源工具可以从程序源码中抽取函数调用关系图,FCSG 的构建只需要在这些逻辑上添加函数调用的时序信息即可.

3.3 逐级代码抽象

函数调用执行序列构成了协议 C 源码的程序框架结构.通过对源码中函数调用关系的抽象,构建协议 C 源码对应的 Pi 演算抽象模型框架,本文对函数调用关系的抽象依赖于函数调用静态执行序列;再通过函数体语句抽象过程将一般的代码语句转换为形式化逻辑;最后,基于函数上下文关系将表达式约简得到协议源码的 Pi 演算抽象模型.

3.3.1 函数调用关系抽象

将 C 语言程序中的函数调用关系转换为 Pi 演算的形式化逻辑描述这一工作并不是平凡的.函数调用关系在 Pi 演算逻辑中表示的难点如下.

- Pi 演算逻辑中的进程宏仅支持并行进程语法,不能在逻辑上直接表达函数调用者与被调用者之间的关系;
- Pi 演算中的构造和析构函数对整个函数过程进行了黑盒抽象,不适用于一般函数的逻辑抽象.

Pi 演算的形式化逻辑中虽然有过程宏、函数等概念,但这二者与程序设计语言中的函数并不等同.如前文所述:Pi 演算中的函数可以用于代码中抽象密码原语操作、基础通信操作以及 C 标准库中的函数;Pi 演算中的进程宏(process macros)与 C 语言程序中的函数相近,可以用于定义子进程(sub-processes);缺乏处理结果返回机制,且在语法上仅支持子进程的并行,不能直接表达多个子进程之间的顺序执行结构的语义.

为了在 Pi 演算逻辑中模拟 C 程序中的普通函数调用语义,本文在函数静态执行序列的基础上,综合运用了 Pi 演算逻辑中的进程宏、并行进程、私有通信这 3 种机制,实现了对 C 程序中的函数调用关系进行等价逻辑转换(证明见第 4.2 节).

对函数调用抽象的步骤如下.

1) 基于 FCSG-SECS 算法获取函数调用静态执行序列;
2) 根据函数调用静态执行序列,为每一次函数调用关系都分配了全局唯一的正整数调用编号,简记为 $CALL\_ID$.如:对于在哈数调用静态执行序列中出现的第 n 个函数调用关系$\langle A,B\rangle$,其对应的编号 $CALL_ID\langle A,B\rangle$为 n;
3) 以 $CALL\_ID$ 为编号,为每次调用关系创建两个临时信道 $C_\langle CALL_ID\rangle$和 $S\_\langle CALL\_ID\rangle$,对 $CALL\_ID=n$,对应的临时信道为 C_n,S_n;
4) 对所有的函数调用关系执行转换,如图 3 所示.
   (1) 第 $CALL\_ID=n$ 次函数调用关系$\langle A,B\rangle$看作相应的 A 进程和 B 进程并行执行;
   (2) 调用者 A 在信道 C_n 上发送空消息,并在相应的 S_n 信道上等待接收返回值信息,然后继续执行剩余代码逻辑;

(3) 被调用者 *B* 在信道 *C_n* 上监听,有空消息到来时执行 *B* 函数体逻辑,最后将返回值在信道 *S_n* 上发出,然后执行结束.

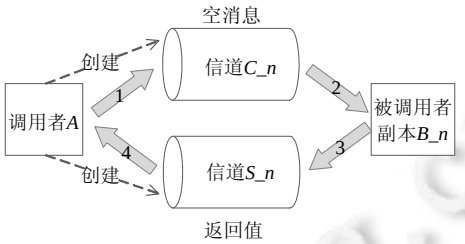

Fig.3 Function call simulation

图 3 函数调用模拟

第 4)步骤中,将每次函数调用都看作调用者和被调用者在专有信道上的并行通信行为.双方采用在临时信道来模拟调用者与被调用者之间的执行同步.两个进程在形式上是并行的,通过通信同步执行顺序,同时也解决了函数返回值传递问题.图 4 给出了函数抽象更具体的转换细节:左边为示例的 C 代码,描述了 *A* 中调用了函数 *B*;中间为转换后 *A* 函数的 Pi 演算抽象;右边为 *B* 函数的 Pi 演算抽象.在 Pi 演算中,*A* 和 *B_n* 进程并行,*A* 中新建了两个信道 *C_n* 和 *S_n*,并通过 *out*(*C_n*,(·))发送空消息触发 *B_n* 中的 *in*(*C_n*,(·));同样,*B* 进程中的 *out*(*S_n*,*value*)将 *B* 函数返回值发送回 *A*,并触发其函数调用后的执行逻辑.*C_n*,*S_n* 与 *B* 函数原有参数一起传送给 *B_n* 进程.

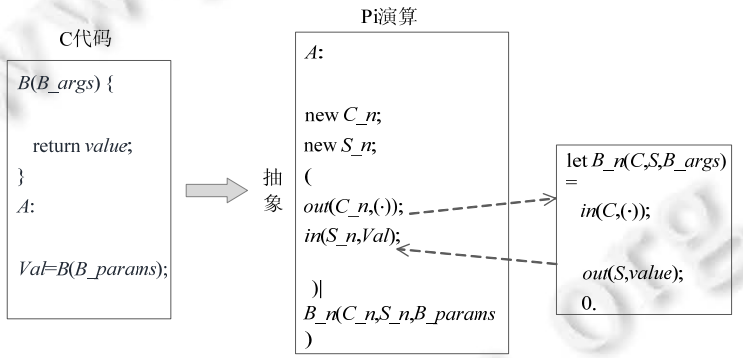

Fig.4 Function call abstraction

图 4 函数调用关系抽象

需要强调的是,为了符合 Pi 演算的语法规范和满足正确的逻辑,在实际转换时,

- 两个临时信道由调用者以创建新名的方式创建,以过程宏参数的形式传给被调用函数,其作用范围仅局限于调用者的函数调用点与整个被调用函数体中;
- 每个函数的每次调用都会被翻译,被调用函数的函数体部分的翻译会被复制一份副本,函数名也以添加后缀_⟨*CALL_ID*⟩的形式加以区分;
- 同一函数再次被调用时,仅需对其中涉及调用的临时信道名进行替换.

转换时,对于函数调用逻辑之外的其他函数体语句保持不变.通过对程序静态执行下的函数调用序列中的所有函数调用关系进行转换之后,便得到了一个协议 C 程序对应的 Pi 演算的形式化描述框架,主要剩余函数体语句部分未进行抽象.在实验章节,我们通过 ProVerif 自动化证明了此函数抽象方案的正确性.

3.3.2 函数体语句抽象

对函数调用关系的抽象,实现了协议代码到 Pi 演算模型框架的转换,还需对模型中未转换的函数体语句部分进行抽象后才能作为 ProVerif 工具的输入.这里分析的 C 语言函数体中的代码块,暂不考虑匿名函数等语法

现象,因为它们的处理方式和正常函数的处理类似.在对语句进行抽象处理时,将表达式 *Expr* 视为原子项,对于表达式的处理将在第 3.3.3 节阐述.首先介绍 C 中的变量类型处理,再介绍各种类型语句的抽象方法.

A. 变量类型转换

方案将 C 语言中的变量基本类型简化为两种基本类型:数字型 *num* 和字符串类型 *bitstring*.其中,*num* 类型包含了 C 语言中常见的 *int*,*float*,*long*,*double* 等.对于字符及数组等抽象为 *bitstring* 类型.为了支持条件分支语句逻辑,还应该引入对 *bool* 类型的抽象.Pi 演算模型中,内建类型中支持 *bool* 类型以及与之相伴随的 true 和 false 两个 bool 型常量,这里直接将 C 语言中的 bool 类型及 true,false 与之对应抽象即可.

在此基础上增加指针类型 *ptr*,并通过引入 *PTR* 构建操作和对应的析构操作 *VAR* 来实现对 C 语言中基本类型指针的支持.

- *PTR_num*(*a*:*num*):*ptr* 获取 *num* 类型变量 *a* 的内存地址;
- *PTR_bitstring*(*s*:*bitstring*):*ptr* 获取 *bitstring* 类型变量 *s* 的内存地址;
- *VAR_num*(*p*:*ptr*):*num* 获取地址 *p* 处的值,返回类型为 *num*;
- *VAR_bitstring*(*p*:*ptr*):*bitstring* 获取地址 *p* 处的值,返回类型为 *bitstring*.

用 Pi 演算中的重写规则来将这两种操作进行关联.

type num.

type bitstring.

type ptr.

fun *PTR_num*(*num*):*ptr*.

reduc forall *a*:*num*,*p*:*ptr*;*VAR_num*(*PTR_*(*ptr*))=*a*.

fun *PTR_bitstring*(*bitstring*):*ptr*.

reduc forall *s*:*bitstring*,*p*:*ptr*;*VAR_bitstring*(*PTR_bitstring*(*ptr*))=*p*.

本方案暂不支持函数指针以及指针的运算操作,当协议代码中涉及到这些语法时,自动化抽象程序仅将这些语法部分标注出来,留作人工处理.

B. 语句抽象

对代码块的处理,我们以 C 语言中的语句(statement)作为基本的翻译单元.考虑协议的 C 代码中常见的语句类型,主要有变量声明/定义语句、表达式语句、条件分支语句、函数调用语句、返回语句、循环结构语句等.其中,函数调用语句和返回语句在上一小节中已经进行了抽象处理,本方案暂不支持 C 程序中的循环结构语句,涉及到循环结构的部分需要依赖人工经验进行手工转换(见表 3).

Table 3 Common statements type in C language

表 3 C 中常见语句类型

语句类型	说明
VarDeclStatement	变量声明和定义
ExprStatement	表达式
IfStatement/*SwitchStatement*	条件分支
NullStatement	空
CallStatement	函数调用
ReturnStatement	函数返回
ForStatement/*WhileStatement*	循环

下面逐一给出这些语句到 Pi 演算逻辑的转换方法.

- *VarDeclStatement*

变量声明和定义语句在抽象转换时并无特殊区别,因为 C2P 并不关注协议 C 代码的语法正确性.这是编译工具所做的事情.

全局变量与局部变量定义语句在具体的抽象细节上稍有不同,具体抽象规则描述如下:

VarDeclstatement::=⟨*Ctype*⟩⟨*varname*⟩[*Initial*],

其中,

$$Ctype::=int|float|char|\ldots.$$

全局变量的抽象方式:

$$free\langle varname\rangle:\langle type\rangle.$$

局部变量的抽象方式:

$$new\langle varname\rangle:\langle type\rangle,$$

其中,*type* 是依照前文所述的变量类型转换方法进行替换:

$$\langle type\rangle=abs(\langle Ctype\rangle).$$

对于含有初始化部分⟨*Initial*⟩的变量定义语句,采用和 *ExprStatement* 中的赋值表达式相同抽象方法处理.

- *ExprStatement*

一个表达式作为独立的 C 语言语句,表达式语句是 C 语言中最基础的语句类型.严格意义上,函数调用语句也属于表达式语句的一种,但这里的 *ExprStatement* 不包含函数调用的表达式语句,因为函数调用表达式语句的抽象过程中已被处理过.在 C 语言中的表达式语句的作用是对表达式进行求值,然后丢弃求值结果,像 *a*+3.这种表达式作为单独语句是没有意义的,有用的是执行表达式之后会对参与的变量(一个或多个)的值进行修改的表达式语句,最常见是赋值语句和自增减语句.自增减其本质也是一种赋值操作,若作为单独语句,是可以和赋值语句等价转换.

赋值语句的语法如下:

$$\langle Lvarname\rangle=\langle subExpr\rangle.$$

对应的 Pi 演算抽象:

$$\text{let } \langle Lvarname\rangle=abs(\langle subExpr\rangle) \text{ in}$$

对于⟨*subExpr*⟩抽象 *abs*(⟨*subExpr*⟩),我们会进行基于表达式上下文化简,具体参考第 3.3.3 节.

- *IfStatement*/*SwitchStatement*

C 语言语法中存在两类条件分支语句:If-Else 和 Switch-Case.Switch case 结构可以转化为 If-Else 结构,所以我们重点讨论 *IfStatement*.*SwitchStatement* 的抽象可以先将其转换为 *IfStatement*,再对其进行抽象转换为 Pi 演算模型.Pi 演算中支持 if-else 条件判断,因此仅需要将相应的部分条件表达式 condition,if 分支代码块 If_block,else 分支代码块 Else_block 等,利用本方案中的对应类型进行抽象即可.对于没有 else 的分支,自动补充 else 部分抽象:else 0.具体的抽象规则描述如图 5 所示,*abs*(⟨*c_code*⟩)表示对⟨*c_code*⟩进行抽象后的结果.

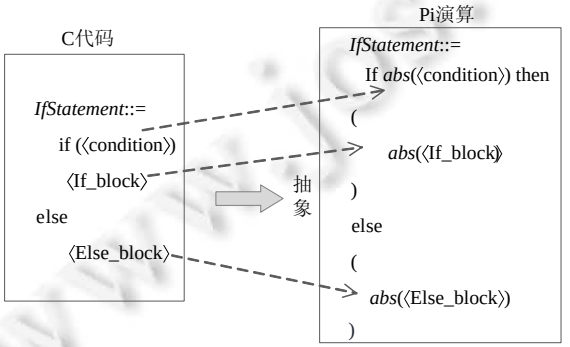

Fig.5 *IfStatement* abstraction

图 5 条件语句抽象

- *ForStatement*/*WhileStatement*

考虑到循环控制结构在 Pi 演算抽象模型中无合适的模型对应,因此和现有的一些方案类似,本文方案对循环结构无确定的自动化抽象转换规则.在自动化分析阶段采用保留循环,仅对其中代码块中的子表达式和子语

句进行抽象,最后采用人工分析进行抽象转换,这其中或引入执行语义的损失.

- *NullStatement*

空语句不执行任何操作但也符合 C 语言的语法规范,我们将其等价转换为 Pi 演算中的空过程 0.

3.3.3 表达式约简

表达式是 C 语言语句中的核心组件,对表达式的抽象逻辑处于整个抽象方案的最底层环节.变量和常量是最基本的表达式:对于变量,我们直接在抽象逻辑中将其作为相应抽象类型的变量;常量将其视为不解义符号.对于由变量和代数运算符等构成的表达式,先按照代数运算规则对其进行相应的计算约简,再将约简后的表达式中的代数运算符抽象为 Pi 演算中的构建操作(constructor).

所谓约简,即在当前函数中,对表达式根据表达式的求值运算规则进行求值运算.

基于函数上下文关系的表达式约简算法见算法 2.

算法 2. 基于函数上下文关系的表达式约简算法.

输入:单个函数定义中的表达式序列 *E*;

输出:约简后的表达式序列 *E′*.

1. *Context*=[·];
2. *E′*=[·];
3. **Foreach** *e* in *E* **then**
4. *Context.append*{*e*};
5. *e′=Simplify*(*Context*,*e*); //据表达式上下文 *Context* 对表达式 *e* 进行化简
6. *E′.append*(*e′*);
7. **EndFor**
8. **Return** *E′*

在一个函数的上下文中,维护一个表达式的上下文内容,根据表达式的上下文内容逐行对当前的表达式进行约简.最后,函数中出现的局部变量都会被函数的参数变量、全局变量以及常量的代数表达式所替代.经过约简后的表达式,基本的代数运算和逻辑运算已经完成,无法根据当前信息进一步对表达式进行运算.无法进一步运算的表达式中的代数运算符视为透明操作,用 Pi 演算中的 Constructor 来表示.以最基本的二元运算符+为例进行说明.对于二元操作符+,抽象为 constructor add: fun *add*(*num*,*num*):*num*.其他操作符类似,*n* 元操作符抽象为 *n* 元函数即可.

基于函数表达式上下文关系代码约简示例如图 6.

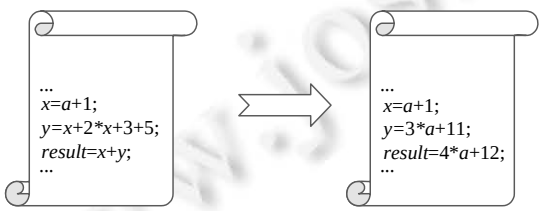

Fig.6 Sample code for expression reduction

图 6 表达式约简示例

4 C2P 实现与评估

4.1 实 现

本节介绍关于依照上文方案开发的 C2P 工具的实现.

C2P 工具以人工预处理后的源码为输入,将其依照上文所述的抽象方法转换为对应的 Pi 演算模型.在实现时,我们利用 C 编译器前端开源项目 clang 及 llvm 库,采用 C++编程语言开发了 C2P 工具代码,约 1.5KLoC.

从协议 C 代码到 Pi 演算的转换工作本质是实现协议 C 代码的翻译,需要对目标代码的 C 语言进行语法分析、词法分析、语义分析等传统编译前端工作,这些可以借助 lex 和 yacc 接口实现.C2P 工具采用了 clang 编译器的 AST 访问接口,因而其对协议 C 代码的解析,直接建立在目标协议 C 代码的抽象语法树基础上.Clang AST 为开发者提供了强大的抽象语法树解析访问接口.简而言之,C2P 借助 Clang AST 接口对 C 代码的抽象语法树进行前序遍历,并采用 Clang AST 的 Rewriter 类对源码进行抽象重写.

在处理协议代码之前,除采用编译器前端处理预编译宏指令外,需要手工标记协议代码中的密码学函数和通信函数,C2P 工具在遍历 AST 的时候会自动跳过被标记的函数.当前,我们采用为这些函数的函数名添加固定前缀来标记.这一步处理还可以改进为配置文件的形式,从预置文件中读取相应的函数名,并在遍历时跳过这些函数.C2P 需要对预处理后的代码进行两次遍历:第 1 次遍历识别出目标协议代码中的所有的函数定义和函数调用关系(除标记函数外),将其以链表结构存储,在第 1 次遍历之后,便得到了代码的 FCSG 图;第 2 次遍时在 FCSG 图上运行算法 1,获取函数静态执行序列,同时根据函数调用静态执行序列的顺序,完成函数调用关系抽象、函数体语句抽象.具体地,对所有 FCSG 图中的函数体语句进行抽象重写,函数调用关系抽象需要处理函数体定义、函数调用语句、函数返回语句等的抽象翻译;非函数调用的一般语句按照第 3.3.2 节中的抽象方案进行抽象转换即可;在处理每个函数定义时,维护一个表达式上下文结构,按照算法 2 进行约简.

实现中的要点如下.

- 基于 Clang AST,在抽象语法树层面对 C 代码语法解析,而非从词法和语法分析做起.Clang AST 提供了方便的操作接口,采用 *Clang*::*Rewriter* 类对源码进行转换,对该类作了少许修改,以满足需求;
- 在 Clang 的 *CallGraph* 抽取代码基础上,添加了函数调用关系图的时序属性,构成了函数调用时序图 FCSG 抽取逻辑.在此基础上添加了 FCSG-SECS 算法,获取函数调用静态执行序列,在执行 FCSG-SECS 算法的同时,完成函数调用关系的抽象;
- 函数体语句抽象是继承了 *Clang*::*VisitStmt* 类,为不同类型语句转换重写方法;表达式约简中,在基于函数上下文关系的表达式约简算法中借助了著名数学工具库 pari,对基础的代数运算和逻辑表达式进行约简.

更具体的实现细节详见 C2P 项目源码.

4.2 合理性证明

在本节给出本文方法中关于函数调用关系抽象的合理性证明.

抽象正确性指在抽象后的执行逻辑与原 C 代码执行逻辑上是一致的,包含两个方面.

(1) 抽象后的模型执行顺序与原协议 C 代码执行顺序一致;

(2) 抽象后的模型不改变攻击者(attacker)的知识集.

第 1 点,为了说明抽象方案不改变原有的代码执行顺序;第 2 点旨在证明抽象方案不改变任何消息原有的机密性,即抽象后的模型不改变攻击者(attacker)的知识集.即证明:对于任意消息 *m*,其隐私性不因抽象方案而改变(任意消息 *m* 若是私有的,则转换后的 Pi 演算模型下无法推出攻击者的知识集中包含 *m*).在 Pi 演算逻辑中,信息机密性的改变是由直接或间接的信道传递而导致的,抽象方案引入了新的信道 *s_id* 和 *c_id*.进一步将问题简化为验证在 *s_id* 信道直接传递 *m* 的情况下,消息 *m* 的机密性不因在新添信道 *s_id* 中的传递而发生变化.即,转换后的 Pi 演算模型中 query *attacker*(*m*)不成立.query *attacker*(*m*)是 Pi 演算中关于消息 *m* 机密性的求解表达式,表示在 Dolev-Yao 敌手模型假设下,求解攻击者的知识集中是否能包含消息 *m*.

不失一般性,将问题简化为对图 7 中转换模型求证以下两个命题成立.

命题 4.1. 事件 *before_call*,*in_call*,*after_call* 严格依序发生:

query *inj-event*(*in_call*)==>*inj-event*(*before_call*);

query *inj-event*(*after_call*)==>*inj-event*(*in_call*).

命题 4.2. 消息 *m* 的隐私性不因在临时信道 *s_id* 上传递而发生改变.若消息 *m* 是私有的,则攻击者知识集中无法推导出消息 *m*,即 query *attacker*(*m*)不成立.

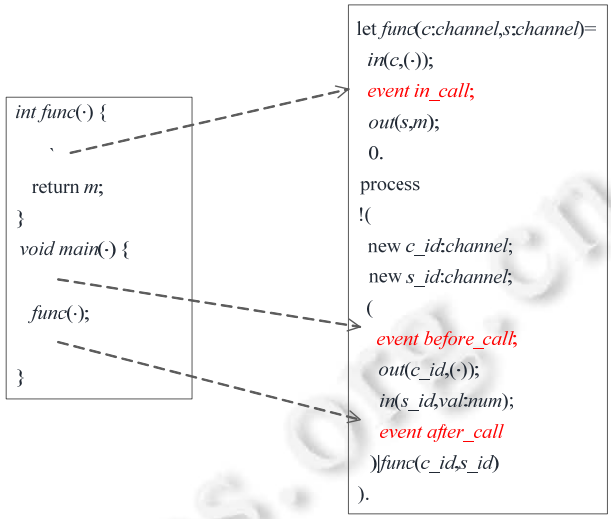

Fig.7 Case of call abstract

图 7 函数调用抽象逻辑正确性案例

这里采用 ProVerif 工具自动化证明命题 4.1 和命题 4.2,如图 8 所示.

```
-- Query inj-event(in_call) ==> inj-event(before_call)
Completing...
Starting query inj-event(in_call) ==> inj-event(before_call)
goal reachable: begin(before_call, @sid = endsid_53, @occ4 = @occ_cst) -> end(endsid_53,in_call)
RESULT inj-event(in_call) ==> inj-event(before_call) is true.
-- Query inj-event(after_call) ==> inj-event(in_call)
Completing...
Starting query inj-event(after_call) ==> inj-event(in_call)
goal reachable: begin(in_call, @sid = endsid_108, @occ9 = @occ_cst) -> end(endsid_108,after_call)
RESULT inj-event(after_call) ==> inj-event(in_call) is true.
-- Query not attacker(m[])
Completing...
Starting query not attacker(m[])
RESULT not attacker(m[]) is true.
```

Fig.8 Proof of correctness

图 8 正确性证明结果

由 ProVerif 对命题 4.1 和命题 4.2 的证明结果可知,两个命题均成立.事实上,借助 ProVerif 工具可以进一步证明:在多个函数调用存在情况下,命题依然成立.据此,本文关于函数调用关系抽象的正确性得到证明,即根据此方法得到的 Pi 演算抽象模型在执行逻辑上与协议 C 代码时一致的.

4.3 Kerberos协议代码分析

本节以一份 Kerberos 协议的示例代码作为分析对象,进一步说明本文方案对协议代码验证的有效性.该分析对象为 github 上下载的一份 Kerberos 协议的 Demo 代码.该份代码实现的协议流程如下.

(1) $A\rightarrow B$:$A,B,N1$;

(2) $S\rightarrow A$:$IV2,E(N1,B,Kab,IV\_ticket,ticket)_{Ka,IV2}$;

(3) $A\rightarrow B$:$A,B,IV\_ticket,ticket,IV3,E(N2)_{Kab,IV3}$;

(4) $B\rightarrow A$:$IV4,E(N2,N3)_{Kab,IV4}$;

(5) $A\rightarrow B$:$IV5,E(N2,N3)_{Kab,IV5}$;

(6) $B\rightarrow A$:$IV6,E(data)_{Kab,IV6}$;

(7) $A\rightarrow B$:$IV7,E(data^*)_{Kab,IV7}$.

其中,$ticket=E(timestamp,A,Kab)_{Kb}$,$Ka$ 和 Kb 分别为实体 A,B 与可信第三方 S 共享的密钥,E 为对称加密算法, IV 为加密算法中用的初始向量.协议主要通信目标为实体 A 通过可信第三方 S 授权访问 B 上的资源 $data$.

代码抽象的部分结果如图 9 所示.

```
fun pi_concat_num(num,num):num[data].
fun pi_concat_bs(bitsting,bitstring):bitstring.
reduc forall x:bitstring,y:bitstring); pi_first(pi_concat_bs(x,y))=x.
reduc forall x:bitstring,y:bitstring); pi_second(pi_concat_bs(x,y))=y.
fun pi_BS2N(bitstring):num [data].
fun pi_N2BS(num):bitstring [data].
fun pi_makeKeyFromPassword(bitstring):bitstring .
type:key.
fun pi_key(bitstring,bitsting):key [data] .
fun pi_encrypt(bitstring,key):bitstring.
reduc forall x:bitsing; pi_decrypt(pi_encrypt(x,key),key)=x.
type ticket_ .
fun ticket (bitstring,num) : ticket_ [data] .
type tkt_req_ .
fun tkt_req (num,bitstring,bitstring) : tkt_req_ [data] .
type tkt_res_ .
fun tkt_res (num,bitstring,bitstring,ticket_) : tkt_res_ [data] .
type svc_req_ .
fun svc_req (bitstring,bitstring,ticket_,num) : svc_req_ [data] .
type svc_res_ .
fun svc_res (num,num) : svc_res_ [data] .
type svc_ack_ .
fun svc_ack (num) : svc_ack_ [data] .
```

Fig.9 Partial results of code abstraction

图 9 代码转换部分结果

抽象完成后,通过人工添加协议安全属性及部分语法修正,利用 ProVerif 工具自动化验证结果,如图 10 所示.

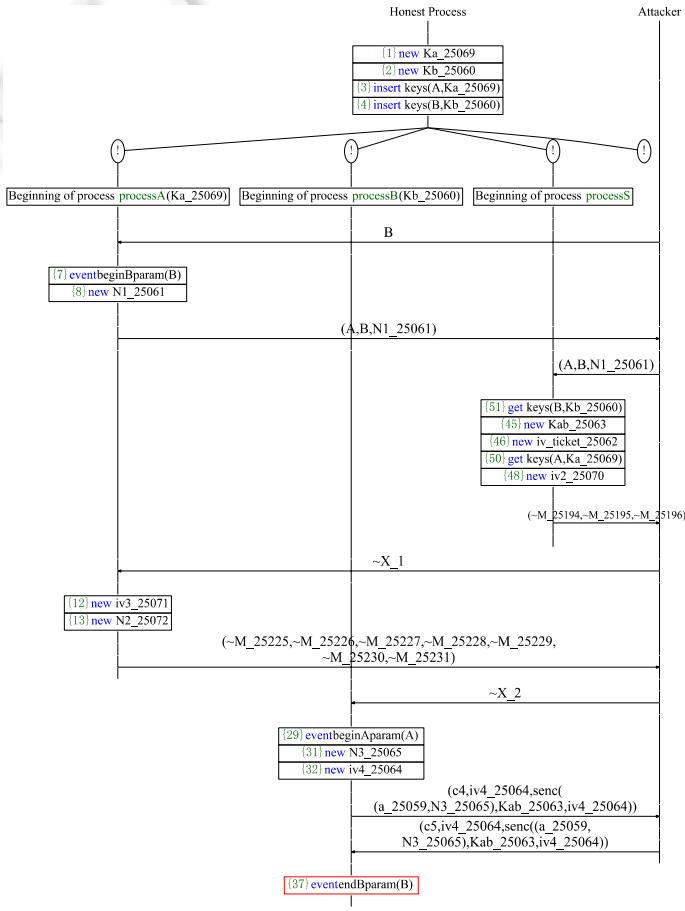

Fig.10 An attack path of Kerberos code

图 10 Kerberos 代码认证性攻击路径

通过转换后的分析表明:代码中存在一条攻击路径,在认证性证明时,单射一致性被打破,参见图 10.代码实现逻辑中,在第 5 步消息接收时,并未对 *IV*5 与 *IV*4 进行比较,即验证 *IV*5 值的新鲜性,导致攻击者可以在不拥有任何密钥的情况下,仅利用截获消息来构造攻击路径,从而导致 *B* 对 *A* 认证并不满足单射一致性,协议的认证性被破坏.若可信第三方 *S* 在局部时间内为 *A* 和 *B* 生成的密钥 *Kab* 不变,攻击者便可以成功构造攻击路径.

表 4 列出了 C2P 方法与相关文献工作的对比情况.文献[9,10]在函数调用语法处理上均需要人工将所有函数调用改写为 inline 模式,再利用预编译程序处理 inline.本文 C2P 方法可以直接支持函数调用语法的自动化处理.此外,文献[9,10]仅支持单一执行路径,不支持条件分支语法;而 C2P 方法给出了条件分支的抽象方法.在支持安全属性验证方面,C2P 与文献[9]都采用了 Pi 演算模型,可以支持符号模型的安全属性验证;而文献[10]采用 Blanchet 演算,支持计算模型的安全属性验证.符号模型基于完美密码假设理论,即认为密码学相关设计完美,攻击者不具备特殊密码学能力;计算模型则在概率条件下考虑攻击者的破密能力.一般认为:计算模型复杂度高,但更符合实际安全.因此,把本文方案迁移到计算模型,也是待拓展工作之一.

Table 4 Comparison of related work

表 4 相关工作对比

文献	抽象模型	支持的 C 语法			安全模型
		函数调用	条件分支	指针	
文献[9]	Pi 演算	人工改写 inline	不支持	部分支持	符号模型
文献[10]	Blanchet 演算	人工改写 inline	不支持	部分支持	计算模型
C2P	Pi 演算	支持	支持	部分支持	符号模型

5 结 语

本文基于 Pi 演算模型给出了一种对 C 语言开发的安全协议源码进行形式化验证的方法,继承了以往工作先抽象再验证的工作思路以及密码学原语相关函数黑盒化的技巧,并将之拓展到通信模块和可信标准库或第三方库;通过定义 FCSG 及函数调用静态执行序列获取算法得到函数静态执行序列,并在此基础上,依照函数调用、函数体语句、表达式约简的顺序逐级对代码进行抽象转换成 Pi 演算模型,借助 ProVerif 自动化验证协议安全属性;最后,利用 ProVerif 给出了相关抽象的逻辑正确性证明,并通过 Kerberos 协议代码案例分析说明了方法的有效性;依据方案实现了 C2P 工具并将之开源,以弥补这一领域工作延续性不足的问题.

References:

[1] Goubault-Larrecq J, Parrennes F. Cryptographic protocol analysis on real C code. In: Cousot R, ed. Proc. of the Int'l Workshop on Verification, Model Checking, and Abstract Interpretation. Berlin: Springer-Verlag, 2005. 363−379. [doi: 10.1007/978-3-540-30579-8_24]

[2] Avalle M, Pironti A, Sisto R. Formal verification of security protocol implementations: A survey. Formal Aspects of Computing, 2014,26(1):99−123. [doi: 10.1007/s00165-012-0269-9]

[3] Meng B, Lu JT, Wang DJ, He XD. Survey of security analysis of security protocol implementations. Journal of Shandong University (Natural Science), 2018,53(1):1−18 (in Chinese with English abstract). [doi: 10.6040/j.issn.1671-9352.2.2017.067]

[4] Zhang HG, Wu FS, Wang HZ, Wang ZY. A survey: Security verification analysis of cryptographic protocols implementations on real code. Chinese Journal of Computers, 2018,41(2):288−308 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.00288]

[5] Kobeissi N. Formal verification for real-world cryptographic protocols and implementations [Ph.D. Thesis]. Paris: Ecole Normale Supérieure de Paris, 2018.

[6] He X, Liu Q, Chen S, Huang C, Wang DJ, Meng B. Analyzing security protocol Web implementations based on model extraction with applied PI calculus. IEEE Access, 2020,8:26623−26636. [doi: 10.1109/ACCESS.2020.2971615]

[7] Li ZM, Meng B, Wang DJ, *et al*. Mechanized verification of cryptographic security of cryptographic security protocol implementation in JAVA through model extraction in the computational model. Journal of Software Engineering, 2015,9(1):1−32. [doi: 10.3923/jse.2015.1.32]

[8] Chaki S, Datta A. ASPIER: An automated framework for verifying security protocol implementations. In: Proc. of the 22nd IEEE Computer Security Foundations Symp. New York: IEEE, 2009. 172−185. [doi: 10.1109/CSF.2009.20]

[9] Aizatulin M, Gordon AD, Jürjens J. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: Chen Y, ed. Proc. of the 18th ACM Conf. on Computer and Communications Security. New York: ACM, 2011. 331−340. [doi: 10.1145/2046707.2046745]

[10] Aizatulin M, Gordon AD, Jürjens J. Computational verification of C protocol implementations by symbolic execution. In: Yu T, ed. Proc. of the 2012 ACM Conf. on Computer and Communications Security. New York: ACM, 2012. 712−723. [doi: 10.1145/2382196.2382271]

[11] Blanchet B. Modeling and verifying security protocols with the applied Pi calculus and ProVerif. Foundations and Trends® in Privacy and Security, 2016,1(1-2):1−135.

[12] Milner R. Communicating and Mobile Systems: The Pi Calculus. London: Cambridge University Press, 1999.

[13] Kiyomoto S, Ota H, Tanaka T. A security protocol compiler generating C source codes. In: Proc. of the 2008 Int'l Conf. on Information Security and Assurance (ISA 2008). Piscataway: IEEE, 2008. 20−25.

[14] Backes M, Maffei M, Unruh D. Computationally sound verification of source code. In: Al-Shaer E, ed. Proc. of the 17th ACM Conf. on Computer and Communications Security. New York: ACM, 2010. 387−398. [doi: 10.1145/1866307.1866351]

[15] Jürjens J. Automated security verification for crypto protocol implementations: Verifying the Jessie project. Electronic Notes in Theoretical Computer Science, 2009,250(1):123−136. [doi: 10.1016/j.entcs.2009.08.009]

[16] Bhargavan K, Fournet C, Gordon AD, Tse S. Verified interoperable implementations of security protocols. ACM Trans. on Programming Languages and Systems (TOPLAS), 2008,31(1):1−61. [doi: 10.1145/1452044.1452049]

[17] Tang WS, Gou ZL, Ahmadon MAB, Yamaguchi S. On verification of implementation of security specification with Petri nets' protocol inheritance. In: Proc. of the IEEE 5th Global Conf. on Consumer Electronics. Piscataway: IEEE, 2016. 1−4. [doi: 10.1109/GCCE.2016.7800491]

[18] Ahmadon MAB, Yamaguchi S, Gupta BB. Petri net-based verification of security protocol implementation in software evolution. Int'l Journal of Embedded Systems, 2018,10(6):503−517. [doi: 10.1504/IJES.2016.10011276]

附中文参考文献:

[3] 孟博,鲁金钿,王德军,何旭东.安全协议实施安全性分析综述.山东大学学报(理学版),2018,53(1):1−18. [doi: 10.6040/j.issn.1671-9352.2.2017.067]

[4] 张焕国,吴福生,王后珍,王张宜.密码协议代码执行的安全验证分析综述.计算机学报,2018,41(2):288−308. [doi: 10.11897/SP.J.1016.2018.00288]

张协力(1992－),男,硕士,主要研究领域为网络安全协议.

顾纯祥(1976－),男,博士,教授,博士生导师,主要研究领域为网络安全,密码学.

祝跃飞(1962－),男,博士,教授,博士生导师,主要研究领域为网络安全,密码学.

陈熹(1988－),男,硕士,讲师,主要研究领域为网络空间安全,密码学,计算机网络.