

模型学习与符号执行结合的安全协议代码分析技术

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)

摘要: 符号执行技术从理论上可以全面分析程序执行空间, 但对安全协议这样的大型程序, 路径空间爆炸和约束求解困难的局限性导致其在实践上不可行。结合安全协议程序自身特点, 提出用模型学习得到的协议状态机信息指导安全协议代码符号执行思路; 同时, 通过将协议代码中的密码学逻辑与协议交互逻辑相分离, 避免了因密码逻辑的复杂性导致路径约束无法求解的问题。在 SSH 协议开源项目 Dropbear 上的成功实践表明了所提方法的可行性; 通过与 Dropbear 自带的模糊测试套件对比, 验证了所提方法在代码覆盖率与错误点发现上均具有一定优势。

关键词: 模型学习; 符号执行; 安全协议代码; 状态驱动

中图分类号: TP311

文献标识码: A

DOI: 10.11959/j.issn.2096-109x.2021067

Security protocol code analysis method combining model learning and symbolic execution

ZHANG Xieli^{1,2}, ZHU Yuefei^{1,2}, GU Chunxiang^{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: Symbolic execution can comprehensively analyze program execution space in theory, but it is not feasible in practice for large programs like security protocols, due to the explosion of path space and the limitation of difficulty in solving path constraints. According to the characteristics of security protocol program, a method to guide the symbolic execution of security protocol code by using protocol state machine information obtained from model learning was proposed. At the same time, by separating cryptographic logic from protocol interaction logic, the problem that path constraints cannot be solved caused by the complexity of cryptographic logic is avoided. The feasibility of the method is demonstrated by the practice on the SSH open source project Dropbear. Compared with Dropbear's test suite, the proposed method has advantages in code coverage and error point discovery.

Keywords: model learning, symbolic execution, security protocol code, state-driver

收稿日期: 2021-02-19; 修回日期: 2021-03-29

通信作者: 顾纯祥, gcxiang520@aliyun.com

基金项目: 国家重点研发计划(2019QY1302)

Foundation Item: The National Key R&D Program of China (2019QY1302)

论文引用格式: 张协力, 祝跃飞, 顾纯祥, 等. 模型学习与符号执行结合的安全协议代码分析技术[J]. 网络与信息安全学报, 2021, 7(5): 93-104.

ZHANG X L, ZHU Y F, GU C X, et al. Security protocol code analysis method combining model learning and symbolic execution[J]. Chinese Journal of Network and Information Security, 2021, 7(5): 93-104.

1 引言

安全协议在理论设计和代码实现中都极易出错。针对协议规范的形式化验证方法由于忽略了协议实现细节,无法保证协议实现的安全。而安全协议较大的代码规模和复杂的交互逻辑让协议代码安全审计变得更加困难。研究新的技术和方法来辅助安全协议代码检测是一项挑战性工作。

模型学习和模糊测试两种技术在安全协议程序分析方面取得了许多有价值的研究成果。Paul等^[1]通过MAT^[2](minimally adequate teacher)模型学习框架推断得到协议程序状态机信息,然后用模型检测器检测协议程序状态机模型中是否存在与协议规范要求不一致的路径。文献[3]用类似的工作思路对安全协议IpSec的开源实现进行了分析,发现了一个违反状态机时序逻辑的bug。文献[4]进一步将模型学习与模糊测试技术相结合,对DTLS进行了分析。直接对模型学习得到的协议程序的状态机进行模型检测,其局限性在于仅能发现与状态机的时序问题,不能用于发掘更细节的错误。符号执行技术可以用较少的测试用例达到较大覆盖率,在安全协议分析方面,文献[5]用符号执行技术分析了TLS的X.509证书解析代码;文献[6]提出在对安全协议符号执行时将密码原语采用重写技术进行处理;文献[7]用密码原语重写的方式分析了WPA2的4次握手协议并发现了其中的脆弱性。这些工作采用符号执行技术分析安全协议中小部分重要代码逻辑,规避了代码路径空间爆炸和约束求解困难问题,但未解决将符号执行技术用于整个安全协议代码的分析时所面临的这两个主要问题。例如,文献[8]仅分析PKCS#1 v1.5的签名验证逻辑,文献[5]只对X.509的证书验证代码进行了符号执行等。文献[9]中对安全协议实施的安全性验证问题进行了全面综述。常规的软件测试方法(如模糊测试)通常依赖人工专家经验对协议规范分析后定制模糊测试策略^[10]。文献[11]将TCP的状态机信息用于对TCP实现的模糊测试中,用协议状态机信息生成Fuzz策略。文献[12-13]也都使用协议规范信息来减少协议程序的搜索空间。文献[14]基于协议规范利用符号执行技术来检测协议实现的脆弱性。

文献[15-16]将模糊处理与符号执行相结合,使用模糊测试来初步探索网络协议,同时执行符号执行探索程序路径和协议状态,比单独使用两项技术的代码覆盖率高。文献[17]提出了用模型推断指导协议的concluc执行分析,方法的主要不足在于模型推断和符号执行交互迭代的过程十分耗时,且未对安全协议的密码学模块进行处理,仍可能面临约束求解困难问题导致符号执行无法继续。

本文方法在文献[17]思路基础上,结合了文献[5-8]方法上的优势,既处理对安全协议程序符号执行时路径空间爆炸问题,又解决因密码学逻辑导致的求解困难问题。该技术将模型学习与符号执行技术相结合来探索协议代码的执行空间,同时将协议中的密码原语代码空间与协议交互逻辑的代码执行空间相分离,降低安全协议代码规模,更具体地:①采用模型学习算法获取协议程序的Mealy machine信息;②以协议程序的状态机信息为基础,引导符号执行来探索感兴趣或重要状态附近的代码执行空间;③在符号执行探索代码执行空间时,绕过密码原语相关函数逻辑,避免符号执行引擎陷入复杂的密码学相关代码逻辑中。

符号执行被视为可以检测深层次语义逻辑问题的分析技术,其可以检测深层代码逻辑同时生成较小测试用例达到较大的代码覆盖率。但在实际分析复杂代码时,通常受限于路径空间爆炸和SMT约束求解困难问题。这些技术局限性在安全协议代码的逻辑分析上更为显著。一方面,由于安全协议设计中并行交互执行逻辑原本就复杂,交互逻辑中多伴随协议状态转换,某些潜在的bug仅在特定的协议执行状态上下文中才能被触发,而符号执行难以或更快速抵达这些状态。另一方面,安全协议中的密码学逻辑,增加了协议代码执行空间规模,尤其密码学中的加解密及哈希算法中的混淆置换等逻辑,使协议程序执行空间急剧膨胀。

对于第一个问题,在协议代码上采用符号执行探索时,缺乏目标协议实现的协议状态机信息,并未区分不同协议状态。采用模型学习技术获取目标协议程序的Mealy machine,以协议程序的

Mealy machine 信息引导符号执行引擎进行协议执行空间探索,从而可以灵活地切换感兴趣或重要的协议状态空间。对于第二个问题,基于完美密码假设理论将密码学函数的代码逻辑与协议交互逻辑代码空间相分离,符号执行时不进入密码学代码空间。这样既有效提高符号执行的效率,又避免符号执行陷入密码学代码逻辑或因混淆变换逻辑导致 SMT 无法求解得到符号公式。

本文在 SSH 协议的开源项目 Dropbear 上实践了该方法,并通过与 Dropbear 源码中自带的模糊测试套件进行了对比,表明本文方法在协议代码的测试覆盖率和发现错误点数量上均有一定提高。

本文主要贡献如下。

(1) 提出了协议程序的状态驱动来指导协议源码符号执行的技术思路,可以有效解决对协议源码做符号执行分析时面临的路径空间爆炸问题。

(2) 基于完美密码假设理论采用具体执行和封装策略处理安全协议代码中的密码学部分,可以避免在对安全协议代码符号执行时因密码学逻辑造成的 SMT 求解困难问题。

(3) 在 SSH 开源实现 Dropbear 上成功实践了本文方法的技术思路,并通过与 Dropbear 自带的模糊测试套件的对比实验表明了本文方法的有效性。

2 基础知识

2.1 符号执行

符号执行是一种用于为软件生成高覆盖率测试用例和发现程序中深层错误的技术。经典符号执行的基本思想是用符号变量替代具体值作为程序输入,以模拟执行方式为程序执行空间中的每条执行路径生成输入符号变量表达式作为路径约束条件,通过 SMT 求解器对每条路径(或感兴趣路径)的路径约束进行求解来判断其可达性。程序执行时为程序维护一个当前执行状态 σ 和路径约束 ρ , 状态 σ 内容为变量到符号表达式的映射,路径约束 ρ 为符号表达式的无量词一阶逻辑公式。在程序执行开始时,状态 σ 为空集,路径约束 ρ 为 true。在执行过程中根据代码逻辑不断更新执行状态 σ 和路径约束 ρ ; 最后通过 SMT 约束求解器求解路径约束 ρ , 若路径对应的约束条件

有解,则该条路径可达,即存在具体输入值可以使程序执行该条路径;反之,则该条路径不可达。其基本思想是:如果程序在这些具体的输入值上执行,则将采用与符号执行完全相同的路径,并以相同的方式结束。

路径爆炸及约束求解瓶颈是符号执行技术当前面临的关键挑战。在理论上程序执行路径的数量与代码中的条件分支数量成指数级关系,难以在给定时间内探索所有执行空间;此外,因为 SMT 问题本身是被证明的 NPC 问题,尽管现代 SMT 求解器采用了技巧使 SMT 求解可行,但仍可能存在为每条执行路径生成的路径约束无法求解的情况,导致符号执行引擎无法终止和继续探索其他路径。在代码量较多的大型程序中这些限制尤为突出,特别是安全协议实现这样逻辑复杂的程序,在利用符号执行分析时,由于协议自身复杂的执行交互设计和密码学相关函数的混淆变换逻辑,加剧了符号执行引擎的负担。

2.2 模型学习

为符号执行提供协议的抽象信息来引导其对协议执行空间的探索。本文借鉴 Paul 等^[1]工作思路,通过模型学习算法抽取协议的 Mealy machine 作为协议执行的高级状态信息。本节介绍 Mealy machine 和经典的状态机学习框架。

一个 Mealy machine 由一个六元组表示 $\mathcal{A} = (S, I, O, \delta, X, s_0)$, 其中 S 是一个状态的有限集, I 是输入的有限集, O 是输出的有限集, δ 是状态的转换关系集合 $S \times I \rightarrow S$, X 是输出的转换关系集合 $S \times I \rightarrow O$, s_0 是初始状态, 且 $s_0 \in S$; δ 中的一个转换关系 $s_k \times i_k \rightarrow s_{k+1}$ 表示从当前状态 s_k 出发, 接收输入 i_k 后, 状态转换为 s_{k+1} ; X 中的一个转换关系 $s_k \times i_k \rightarrow o_k$ 表示从当前状态 s_k 出发, 接收输入 i_k 后, 输出为 o_k ; 其中, $s_k \in S$, $s_{k+1} \in S$, $i_k \in I$, $o_k \in O$ 。Mealy machine 的输出取决于当前状态和当前输入。更近一步, 从初始状态出发, 给定一个输入序列, 状态转换序列和输出序列也是确定的。因此, 当初始状态给定的情况下, 可以用 Mealy machine 中从初始状态 s_0 出发到某一目标状态 s_i 的所有输入序列集合来表示 s_i 状态的信息, 这是将协议实现的 Mealy machine 抽象信息用于指导协议代码空间上的符号

执行的理论基础之一。

本节介绍经典的模型学习框架 MAT^[2] (如图1所示)。MAT 框架适用于软件系统的黑盒学习模型, 将目标系统的模型学习看作 Learner 和 Teacher 之间的 query-response 游戏。Learner 通过向 Teacher query 来推断状态机模型。Teacher 为被测软件系统, 其拥有状态机信息, 故又称其为 SUL (system under learning)。Learner 在学习初始阶段, 掌握输入集合 I 和输出集合 O 。

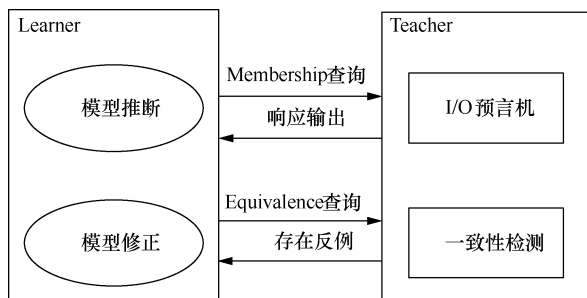


图1 MAT 学习框架
Figure 1 MAT framework

Learner 通过以下两种类型的查询来获取状态机信息。

Membership query: Learner 向 Teacher 查询一个输入序列对应的输出序列, Teacher 给出相应的输出序列。如采用 L^* 算法, 从 query 得到的输入序列/输出序列信息, 推断出 Mealy machine。

Equivalence query: Learner 向 Teacher 询问一个 Mealy machine A' 是否正确, 即 A' 是否与实际的状态机 A 相等。例如, 采用 W-method 算法测试查询, 检测学习到的状态机是否与实际状态机一致。若存在反例, 则修正 Mealy machine。直到在限定查询中无反例, 则认为当前学习到的 Mealy machine 与 Teacher 的 Mealy machine 是近似等价的。

3 方法

3.1 方法概述

本文方法主要探索如何在安全协议代码上利用符号执行方法进行高效的代码空间探索, 检查协议代码实现中存在的问题。由于安全协议代码逻辑复杂, 直接对安全协议代码采用符号执行分析是不可行的。为此采用了两种技巧来提高符号执行引擎在安全协议代码探索时的效率: ① 为符

号执行引擎提供协议目标代码的高级抽象信息 Mealy machine 作为指导, 便于符号执行引擎探索感兴趣的协议状态附近的执行空间; ② 将安全协议代码中与密码学相关的代码从符号执行分析的代码空间中去除, 以减少密码学中混淆变换引入的不必要的逻辑, 从而降低协议代码执行空间的分支路径。

方案的整体思路如下。按照 MAT 框架, 以协议二进制为学习目标 (即 Teacher/SUL), 采用模型学习算法 L^* , 学习出目标协议实现的 Mealy machine; 从 Mealy machine 中抽取从初始状态到任意状态的消息序列, 以得到的消息序列集合近似代替 Mealy machine 的局部信息。对消息序列进行转换, 通过 mapper 程序将其映射为实际的数据包序列; 符号执行引擎加载执行协议代码, 以转换后的数据包序列作为 socket 通信输入, 探索该消息序列下的安全协议代码执行空间。测试者可以自由选择数据包序列并将感兴趣的消息字段符号化进行定制化探索。在符号执行引擎对协议代码空间探索时, 将密码学相关函数套件进行屏蔽, 可极大降低协议代码复杂度, 也可让符号执行引擎探索工作集中在协议代码与环境交互时的协议执行逻辑上。

3.2 模型学习抽取 Mealy machine

本文根据 MAT 学习框架, 以黑盒的方式学习目标协议程序的高级抽象信息 Mealy machine。

为了避免模型学习陷入无法终止或者变为不确定性状态, 需要保证学习的目标协议程序是一个确定性的程序, 即在同一状态下, 对相同的报文数据输入, 输出报文数据是相同的。换言之, 在协议程序的初始状态下, 对于确定的报文序列, 其输出报文序列是确定的。这一点, 通常需要将待学习的目标协议程序的初始化配置文件进行固定设置即可保证。本文的目标不包括检测目标协议程序的配置是否存在问题, 而关注于协议程序在其通用的初始化配置下与对等端的交互处理逻辑是否存在 bug。可以通过重设配置, 重复整个方案工作来探索某一具体配置问题。

一个 Mealy machine 最根本的蕴含信息是输入输出序列, 因为状态是由输入输出序列确定的。在学习协议程序的 Mealy machine 前, 需要先确

定什么是抽象模型的输入输出集合。参考协议规范中状态机描述方式，以消息类型来抽象表示一个具体的协议数据报文。自然地，一个输入序列被抽象为输入消息类型序列，一个输出序列被抽象为输出消息类型序列。根据协议规范，确定一个具体协议实现的输入消息类型和输出消息类型的有限集。

对于每一个待分析的协议程序，需要一个转换程序将抽象的消息类型转换为具体的、程序可接收的消息报文。这个转换程序可以通过协议程序交互的对等端代码进行改造实现，如在学习 SSH 协议服务端程序 SSHD 的 Mealy machine 信息时，通过改造 SSH 客户端程序得到消息转换程序。本文采用 L* 学习算法学习目标协议程序的 Mealy machine 信息，具体的学习步骤如下。

- ① 学习程序通过决策生成输入消息类型。
- ② 转换程序将输入消息类型映射为具体的输入报文数据，发送给目标协议程序。
- ③ 目标协议程序在接收输入报文数据后生成输出报文数据。
- ④ 转换程序将输出报文数据映射为抽象的输出消息类型。
- ⑤ 学习程序根据输出消息类型更新状态信息并生成新的输入消息类型。
- ⑥ 迭代步骤②~步骤⑤，直到完成学习目标，即模型收敛或达到设定学习时间。
- ⑦ 对学习到的模型采用随机抽样检查一致性，若存在不一致，则修正(refine)模型。

3.3 状态驱动：Mealy machine 的抽象

由 3.1 节描述的模型学习方法，本文可以采用黑盒交互的方式学习到待分析协议程序的 Mealy machine 信息。这里基于假设学习到的 Mealy machine 是一个包含（或称为实践上近似包含）被分析的协议程序完整状态集合和消息集合的高级抽象结构，其中蕴含的协议状态转换关系是正确的。在此假设前提下，用其指导后续对协议程序代码执行空间的搜索分析。

按照上文学习到的 Mealy machine 是一个六元组集合，直接用其指导符号执行是难以实践的。学习到的 Mealy machine 可以看作节点之间有并行边存在的有向图结构。节点表示协议程序的抽象状态，边代表状态的变迁，而边的属性即

input/output 信息。为了将学习到的协议程序高级抽象信息 Mealy machine 作为安全协议代码符号执行时的指导信息，本文以状态驱动集 (state-driver set) 来指导某一个具体协议状态下的符号执行。状态驱动集定义如下。

定义 1 对于一个 Mealy machine $\mathcal{A} = (S, I, O, \delta, X, s_0)$ ，对任意状态 $s \in S$ ，若消息序列 $[i_0, i_1, \dots, i_l]$ 满足以下性质，则将序列 $[i_0, i_1, \dots, i_l]$ 称为状态 s 的一条状态驱动，其中 $i \in I$ 。

性质 1 $s = \delta(\delta(s_0, i_0), i_1, \dots, i_l)$ 。

性质 2 在性质 1 中，转换关系均不重复出现。

$\mathcal{A}$ 中， s 的所有状态驱动构成其状态驱动集 $\mathcal{D}_s$ 。

性质 1 表明在 Mealy machine $\mathcal{A} = (S, I, O, \delta, X, s_0)$ 中，某个状态 s 的一条状态驱动 d 是从初始节点 s_0 出发，经过若干次 δ 转换到达 s 状态所需要的转换条件（输入消息）构成的序列。若协议程序从初始状态依序接收驱动 d 中的消息，其状态会转换到状态 s 。性质 2 表明在一条驱动中不包含冗余的状态转换条件，用于避免重复的分析。为便于表达，记为 $s_0 \xrightarrow{d} s$ 。同时为每个状态定义状态探索集 (state-explorer set)，作为符号执行阶段符号化时的语料库。

定义 2 对于一个 Mealy machine $\mathcal{A} = (S, I, O, \delta, X, s_0)$ ，对任意状态 $s \in S$ ， $i \in I$ ，若 $\delta(s, i) \in S$ ，则称 i 为状态 s 的一条状态探索 state-explorer，状态 s 的所有状态探索构成其状态探索集。

本文给出在 Mealy machine 中提取状态驱动集的算法。为便于算法描述，对 δ 中状态转换关系 $r: s_k \times i_k \rightarrow s_{k+1}$ 记作 $r.\text{src} = s_k$ ， $r.\text{dst} = s_{k+1}$ ， $r.\text{cond} = i_k$ 。

算法 1 Mealy machine 状态驱动集提取算法

输入 Mealy machine $r.\text{cond} = i_k$ ，目标状态 s

输出 s 的状态驱动集 $\mathcal{D}_s$

- 1) $\mathcal{D}_s \leftarrow \emptyset$, Visited $\leftarrow []$, Stack RS
- 2) $R_{s_0} \leftarrow \{(r'.\text{src}, r'.\text{dst}, r'.\text{cond}) \mid r'.\text{src} = s_0 \ \& \ r' \in \delta\}$
- 3) RS.push(R_{s_0});
- 4) **while** RS.isEmpty() = false **do**
- 5) $R \leftarrow \text{RS.top}()$;

```

6)      foreach  $r$  in  $R$  do
7)      if  $r.dst = s$  then
8)       $d \leftarrow []$ ;
9)      foreach  $v$  in Visited do
10)  $d.append(v.cond)$ ;
11)      endfor
12)       $\mathcal{D}_s \leftarrow \mathcal{D}_s \cup \{d\}$ ;
13)      elif  $r.dst \notin \{v.src \mid \text{for all } v \text{ in Visited}\}$  then
14)       $R_{r.dst} \leftarrow \{(r'.src, r'.dst, r'.cond) \mid r'.src=r.dst \ \& \ r' \in \delta\}$ ;
15)       $RS.push(R_{r.dst})$ ;
16)      Visited.append( $r$ );
17)      endif
18)      endfor
19)       $RS.pop()$ ;
20)      Visited.pop();
21) endwhile
22 Return  $\mathcal{D}_s$ 

```

给定一个 Mealy machine $\mathcal{A} = (S, I, O, \delta, X, s_0)$ 和一个目标状态 s , 可以根据算法 1 抽取 $\mathcal{A}$ 中从初始状态转换为目标状态 s 的消息序列集合, 即 s 的状态驱动(state-driver)集。状态 s 的 state-driver 集替代了 Mealy machine 高级抽象信息, 为协议程序的符号执行提供辅助信息来指导其更高效地执行。

3.4 用状态驱动指导符号执行

一个协议状态的状态驱动实质是从协议程序的初始状态到达该状态的抽象消息序列。要将协议程序从初始的 reset 状态引导到具体的协议状态需要具体的数据包序列来触发。这里可以重复利用模型学习时抽象消息到具体数据包的转换器(Mapper)。转换器可以根据协议规范, 将一个状态驱动中不同抽象消息拓展成具体的、协议程序可接收的数据包流。且因为 Mealy machine 是从协议程序中学习推断得到的, 这些具体的数据包按照状态驱动中的顺序可以依次被协议程序接收, 并将协议程序的状态调整到该状态驱动对应的状态上。

采用 L*算法获取到目标协议程序的状态机信息后, 可以根据分析需要选择一个或多个协议

状态对其采用符号执行技术做分析。首先依据算法 1 获取感兴趣状态的状态驱动集, 通过将状态驱动集中的状态驱动映射到具体数据流后输入协议程序中, 调整协议程序达到相应的协议状态。尽管一个状态的不同状态驱动都能够将协议程序引导到该协议状态, 但不同状态驱动引导的同一协议状态的程序上下文会有差别。同一抽象状态的不同上下文, 也会造成符号执行在分析过程中所覆盖的路径有所差异。仅一条状态驱动不能完全覆盖该协议状态的执行路径。为此, 在分析某一状态时, 多个协议程序在初始状态下并行执行, 通过不同状态驱动对应的 packets flow 来调整其符号执行时的起始状态, 如图 2 所示。

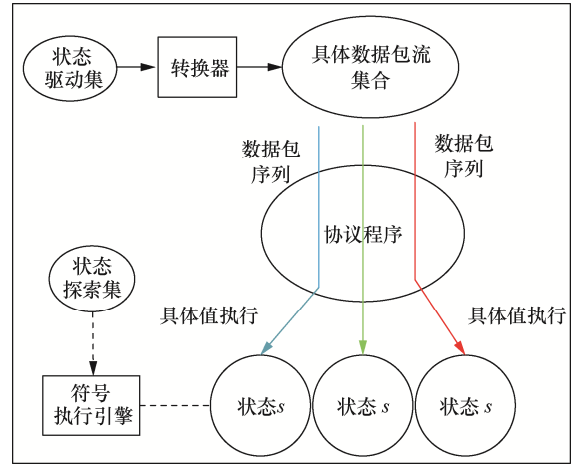


图 2 状态驱动指导协议符号执行

Figure 2 Protocol symbol execution under the guidance of state-driver set

因为有目标协议程序的 Mealy machine 的辅助信息, 通过如图 2 所示的操作过程, 可以灵活选取某个协议状态, 自由控制协议程序执行到该状态并以此为起点对协议程序进行符号执行分析。让符号执行引擎的注意力集中在某个指定状态上, 避免了因协议程序路径复杂, 符号执行分析时无法及时有效探索重要程序空间。在由 state-driver 调整好协议程序到目标状态后, 利用该状态的 state-explorer 集来制定具体的符号执行策略, 即确定一个 state-explorer 对应的数据包中哪些域是 symbolic 域, 哪些是 concolic 域。一个状态的 state-explorer 对应协议程序在该状态下可接收的一个消息类型, 所以该方法的符号执行策略是一次符号执行且仅利用一个数据包为基础语料

库，即符号执行所探索的协议状态机的深度仅为 1，如图 3 所示。一方面，深度为 2 的探索空间可以在对该条 state-driver 对应的下一个状态探索时覆盖，更大的深度依次类推；另一方面，将探索空间限定在该协议状态的局部可以进一步为符号执行引擎减轻工作负荷。

3.5 密码学代码的处理

上文介绍了在对安全协议代码采用符号执行时路径空间爆炸问题的解决方案，即用状态驱动有针对性地引导符号执行的路径空间探索，而非非常规地扁平化地执行分析。本节介绍在安全协议代码符号执行时面临的另一个典型问题，即符号表达式约束求解难的解决方法。

安全协议代码中密码学操作函数占据一定的比例，且属于频繁被调用的处理逻辑模块。典型的安全协议（如 TLS、SSH 等）在会话协商完成之后，所有的消息都会有加解密和认证码操作，用于确保通信数据的机密性和不可篡改性等。经典的密码算法会采取如扩散、混淆等技术对输入进行处理，一般还要求具备雪崩效应，即对输入的微小改变会导致输出完全不同。但这些复杂的处理逻辑严重阻碍了协议代码的符号执行分析过程，主要原因有两方面：第一，进入密码算法中的迭代、混淆等处理逻辑内部进行分析是复杂且收效甚微的，因为多数情况下安全协议中的密码算法会采用 NIST 标准算法和成熟的密码库实现，这些算法由草案成为标准的过程中经过了密码学家和学者的专业经验分析；第二，当符号化变量经过这些密码算法的处理代码后，路径约束表达

式会变得很复杂，当前最先进的 SMT 求解器（如 Z3、STP 等）需消耗大量运算时间来求解这些表达式。本文尝试对常用的加密算法（AES、chacha20 等），以及哈希算法（SHA-1、MD5 等）的 C 代码实现采用符号执行分析，实践结果表明，产生的路径约束表达式会让 SMT 求解器消耗相当长时间而致使符号执行引擎难以有效分析。以 AES-128-CBC 的加密函数为例，仅将输入的 1 字节符号化后产生的符号表达式，STP 求解器耗费数十分钟才求解完毕。即便采用具体执行的穷举策略，数秒内就可穷尽出该问题的求解，这与符号执行的技术原理有关。

为解决安全协议代码中密码学相关逻辑造成的路径约束表达式复杂而难求解的问题，本文基于经典协议形式化验证理论中的 Dolev-Yao 模型来解决，即把 Dolev-Yao 敌手模型中的完美密码假设理论拓展到密码算法实现上。在分析时，本文将协议交互逻辑和密码学模块逻辑分离，并假设密码学相关模块的代码实现是完美和值得信任的。尽管其中仍可能存在脆弱点，但这些问题的发掘需依赖较高的密码学专业知识和经验，对密码学相关代码的安全性分析超出本文的研究范围。当符号执行分析到密码学函数模块时，根据协议状态所处的不同阶段分情况采取具体执行和“绕过”两种策略来分析密码函数的代码逻辑，以避免增加符号约束公式的求解复杂度。

典型的安全协议通常可以分为会话协商和传输通信两个阶段，如图 4(a)所示。会话协商阶段通信双方除确定协议软件和协议版本外，还会通

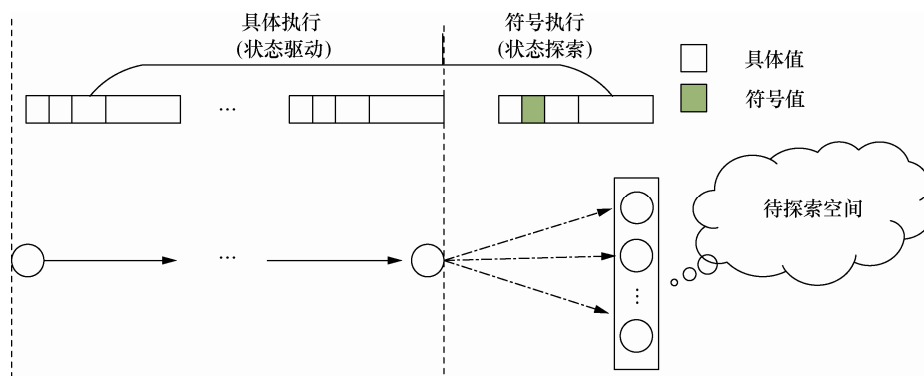


图 3 将 state-driver 和 state-explorer 组合对某一协议状态符号执行
Figure 3 Analyze a protocol state by combining state-driver and state-explorer

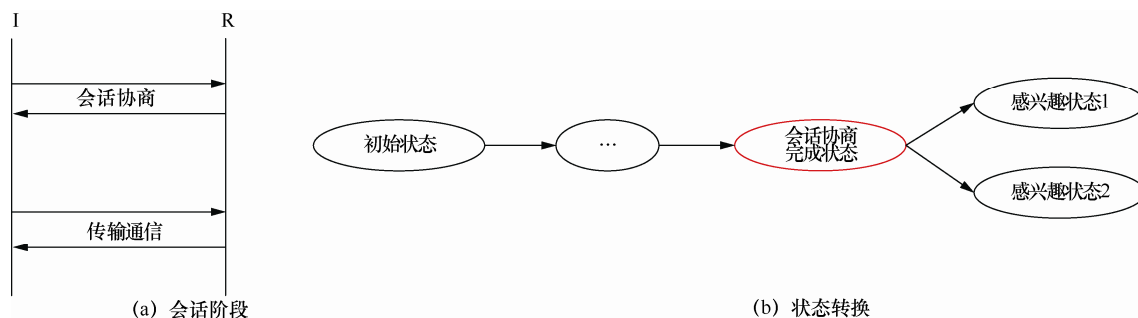


图 4 安全协议会话阶段和状态转换
Figure 4 Session phases and state transitions of a security protocol

过密钥交换协商出后续传输通信阶段所要使用的各种密码套件和相应的密钥。会话协商阶段完成后，通信双方或多方后续通信所使用的加解密、签名、完整性校验等各类密码算法及密钥信息已经确定。在这之后的会话传输通信将在由会话协商阶段产生的密钥及密码套件构建的安全信道中进行。图 4(b)是相应的状态转换图，此处省略了转移条件信息。从初始状态经历会话协商后到达会话协商完成状态，会话通信阶段的各个状态均在此状态基础上根据相应的转移条件触发到达。

根据安全协议密码算法应用的不同阶段，对其处理有所不同。在会话协商阶段的各个状态探索时，输入数据包序列中的密钥参数字段，即密码算法的输入采用 **concrete execution** 策略；同时对非密钥参数字段符号化来探索其他的路径分支。采用具体值可以快速通过密码算法代码处理过程，而不至造成分析引擎停留在密码函数处理逻辑的时间过长，更不会形成复杂的约束条件，**concrete** 输入即使经过复杂处理过程的输出仍是 **concrete** 值。对于会话通信阶段，所有的数据都会被加密，若全都采用具体值分析，则仅能分析单一的执行路径，而采用符号执行，则仍面临约束表达式求解困难的问题。本文把一个密码函数 f 的符号执行分析，改为对其替代封装 (**wrapper**) 函数 wrap_f 的执行。**wrapper** 函数 wrap_f 与原密码函数 f 的输入输出具有相同的形式，但仅从代码形式上的模拟并非执行逻辑。图 5 是加密函数 **cipher** 的 **wrapper** 函数 wrap_cipher 示例，经过模拟函数 wrap_cipher 后得到的密文和明文是相同的，仅会对传入参数做一些基本的合规性检测。对密码函数的输入长度与输出长度不等的情况，

采用简单的截取或填充的方式处理。符号变量 sym_var 经过密码函数 f 后变为 $f(\text{sym_var})$ ，在对应的 **path condition** 符号公式求解时，会被替换成 $\text{wrap_f}(\text{sym_var})$ 。由于 f 函数是一个复杂函数，而 wrap_f 是一个简单函数，这样的替换会极大降低 SMT 求解器的工作负担。

```
void wrap_cipher(void* p, size_t pl, void* k, size_t kl,
                 void* c, size_t cl) {
    assert(pl == cl);
    assert(kl == K.SIZE);
    memcpy(c, p, pl);
    return;
}
```

图 5 密码模块模拟函数示例
Figure 5 An example of a cipher simulation function

4 实验评估：在 Dropbear 上的实践

为了评估方法在安全协议实现分析方面的有效性，本文选取了被广泛使用的安全协议 SSH 及其开源项目实现 Dropbear 源码作为分析测试对象。SSH 协议被设计用于在不受信任的网络中实现安全远程登录和其他安全网络服务，具体协议细节可参考 RFC4251^[18]。该协议由传输协议、用户身份验证协议和连接协议 3 个组件构成。传输协议主要任务是在不安全网络上提供安全通道，包括通信的机密性和完整性、可选的压缩方案以及客户端对服务端的身份认证方式等；用户身份验证协议规定了服务端对客户端登录用户身份认证的机制；连接协议用于在安全的（机密性和完整性）和认证授权过的通道上以多路复用的方式传输应用数据。SSH 协议被视为当前服务器远程配置和运维的首选方案。Dropbear 是轻量级的 SSH 协议开源项目，支持 POSIX 平台且内存占用

量小，可以作为 **Openssh** 替代方案，当前被广泛部署在嵌入式设备上。本文选取 **SSH** 协议和 **Dropbear** 作为评估测试对象的主要原因有两点：第一，SSH 协议是应用广泛的典型安全协议，协议的复杂度适中且具有众多开源实现，便于研究分析；第二，Dropbear 是 C 代码开源项目，代码量适中，适合后续采用基于源码的符号执行 **KLEE** 来作分析，且 Dropbear 项目中自带基于 **libFuzzer** 库的模糊测试用例，便于测试方案的评估对比。

4.1 对 Dropbear 进行模型学习

方案实施的第一阶段是模型学习，即利用 MAT 模型学习框架对 DropbearSSH 进行模型学习，推导其 Mealy machine 模型。这部分工作主要基于 Paul 等^[1]SSH 模型学习的开源工作并有少许细节上的改动。实验设置如下：被测对象为在 vmware 虚拟机 ubuntu16.04 系统中编译运行 Dropbear v2020.80 版本服务端程序；Learner 运行在 4 GB 内存、处理器为 Intel core i5-8500。

Ubuntu16.04 系统的虚拟机中。

在采用 `modellearning` 技术对 Dropbear 的模型之前,需要先确定输入集合。本文沿用文献[1]的策略把 SSH 消息类型作为输入集合。同时,为了简化状态机规模便于后续符号执行工作,本文对输入集合进行了筛选,如去除了 IGNORE、UNIMPL、DEBUG 等。表 1 列出了实验时采用的输入消息集合、对应的消息类型,有关这些消息类型的详细说明可参考 SSH 的 RFC 文档^[18]和文献[1]。

最终确定的状态数量为 25 个。如图 6 所示, 红色轨迹标注了常规测试套件的协议测试路径, 将其称为主干路径。主干路径的状态迁移如下。

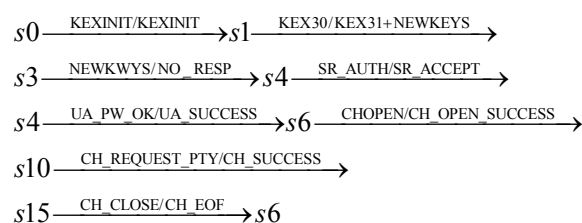


表 1 Dropbear 的模型学习输入/输出字母表
Table 1 The input/output alphabet for learning Dropbear

SSH 子协议	输入表 (I)	输出表 (O)
SSH-TRANS	KEXINIT, KEXINI_PORCEED, KEX30, NEWKEYS, SR_AUTH, SR_CONN	KEXINIT, KEX31, NEWKEYS, UA_SUCCESS, UA_FAILURE,
SSH-USERAUTH	UA_PW_OK, UA_PW_NOK	CH_SUCCESS, CH_EOF,
SSH-CONNECT	CH_OPEN, CH_CLOSE, CH_REQUEST_PTY, CH_EOF	SR_ACCEPT, NO_RESP

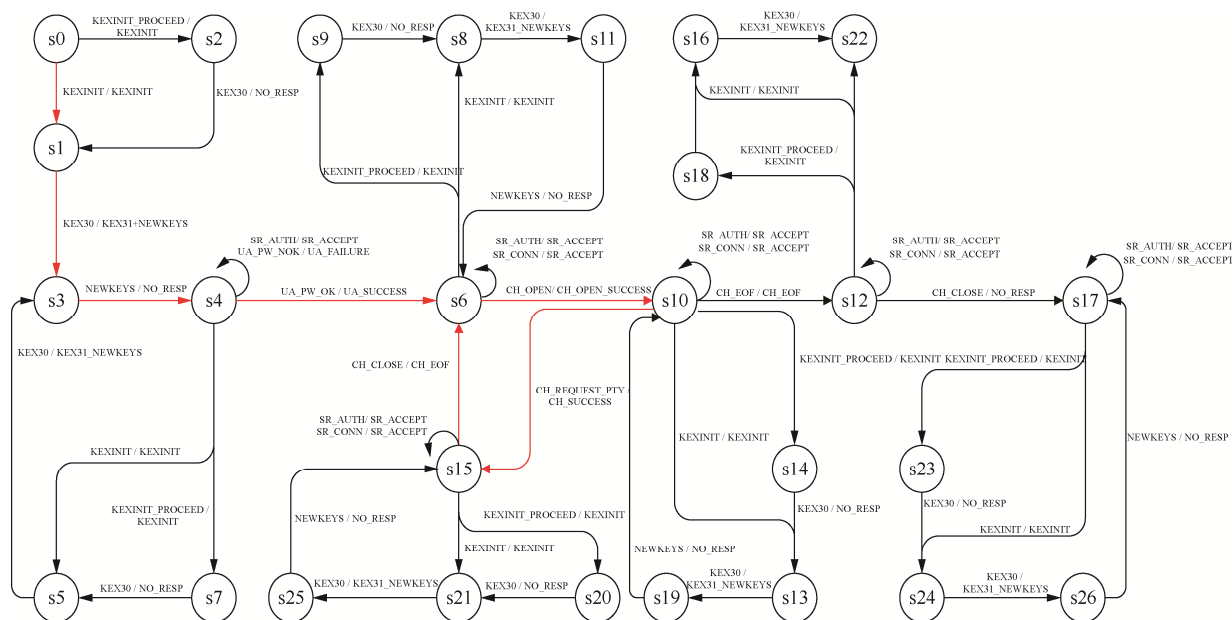


图 6 模型学习推断的 Dropbear 状态机 (Mealy machine) 模型
Figure 6 Learned state machine (Mealy machine) model for Dropbear

经过简单的分析,可以辨别出状态编号对应的实际状态含义。如 s0 为初始化状态, s4 为密钥协商完成状态, s6 为用户通过认证后的状态等。学习得出的 Mealy machine 中包含的主干路径为常规测试套件的测试路径,因此可以得出初步的定性结论:学习得到的 Mealy machine 信息来指导符号执行,在理论上不低于常规测试套件的代码覆盖率。

4.2 Dropbear 源码的符号执行

在得到 Dropbear 程序的 Mealy machine 信息后,可以选取感兴趣的协议状态,用算法 1 获取该状态的状态驱动集。表 2 列出了 Dropbear 的各抽象状态对应的状态驱动集数量,即驱动集的基数。

表 2 Dropbear 各状态对应的状态驱动数
(状态驱动集基数)

Table 2 Number of state drivers for each state of Dropbear (cardinality of state driver set)

状态	基数	状态	基数	状态	基数	状态	基数
s1	2	s8	36	s15	54	s22	324
s2	1	s9	18	s16	324	s23	486
s3	2	s10	18	s17	162	s24	972
s4	2	s11	36	s18	162	s25	324
s5	12	s12	54	s19	108	s26	972
s6	6	s13	108	s20	162		
s7	6	s14	54	s21	324		

以 s15 状态为例,根据算法 1 得到的状态驱动集共有 54 条状态驱动,如图 7 所示。从表 1

```
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'SR_AUTH', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_NOK', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_AUTH', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_AUTH', 'CH_REQUEST_PTY' ]
[ 'KEXINIT_PROCEED', 'KEX30', 'KEX30', 'NEWKEYS', 'UA_PW_OK', 'SR_CONN', 'CH_OPEN', 'SR_CONN', 'CH_REQUEST_PTY' ]
```

图 7 状态 s15 状态驱动集

Figure 7 The state driver set of state s15

中可以看出，一部分状态的状态驱动数量达到几百条甚至近千条的规模，如状态 `s24` 的状态驱动数量达 972 条。这意味着有 972 种不同的输入可以让 Dropbear 服务端程序从初始状态到达状态 `s24`。Mealy machine 仅是协议程序的高级抽象信息，到达一个状态的路径数量达几百条，当对应到协议程序的执行空间时，执行路径的分支数量会急剧增加。在没有状态信息指导的情况下，符号执行引擎很难快速分析这些路径分支。

状态驱动到具体数据包序列的转换器 mapper 重用了模型学习阶段的转换模块，调用了手工改写的 Paramiko 库。符号执行工具采用了基于 llvm 的开源分析引擎 KLEE，它可以对源码编译后的 llvm 字节码进行符号执行。由于 KLEE 不支持 socket 函数建模，但可以支持文件的静态读写。为此，在实验时 socket 相关的函数都采用文件读写操作进行了替代。

为了说明本文方法的有效性，对比实验对象为 Dropbear 自带的 fuzz 测试模块 `fuzzer-preauth`。主要对比 SSH 传输协议阶段代码的分析效率，主要的评价指标为代码覆盖率和检出错误率。其中，代码覆盖率采用 Clang 基于源码的覆盖率功能 (`-fprofile-instr-generate -fcoverage-mapping` 选项) 测算结果，以执行的源代码行数占比作为代码覆盖率的计算方式，并去除为测试添加的代码及覆盖率 0% 的代码文件。检测的目标代码错误为人工设计并添加到分析对象代码中的代码问题。添加错误的位置平均分布在传输协议阶段相关的 4 个消息处理例程 (`recv_msg_kexinit`, `recv_msg_kexdh_init`, `recv_msg_newkeys`, `recv_msg_service_request`) 以及公共代码中，各部分各 4 个，即共有 20 个代码错误，具体实验结果如表 3 所示。

由于本文方法在 NEWKEYS 消息之前采用具体合法值输入密码学函数代码中，NEWKEYS 之后采用绕过策略，而 Libfuzzer 测试套件未对此有特殊区分，因此将 Dropbear 源码中的 libtomcrypt

库源码不计入源码覆盖率的计算中时，LibFuzzer 测试套件的代码覆盖率上下降了约 4%；而 state-driver 方式在不含 libtomcrypt 库时代码覆盖率优于代码覆盖率。在检测出错误方面，Libfuzzer 测试套件可以检测出 13 个设计的错误，而 State-driver 方式可以覆盖到这些错误的执行点。这与 20 个错误的构造位置有关，在设计这些错误时特意选取了常规方式不容易到达的分支执行点上，而这正是符号执行技术的优势所在。对比实验未采用 Mealy machine 信息指引的常规执行策略，经实践发现，即使 KLEE 这样先进的符号执行引擎在处理类似 Dropbear 的安全协议源码时也不能直接适用，最终会因系统资源耗尽而终止。

5 结束语

符号执行在对安全协议程序分析时，存在路径 SMT 求解困难等两个显著问题，这也是符号执行技术对大型软件程序分析所面临的局限性。本文方法不改进通用符号执行技术，而从安全协议程序的自身特点出发，提出了一种对安全协议代码空间进行符号执行的技术思路。通过将模型学习得到的状态机信息转换为状态驱动集，以状态驱动集指导符号执行引擎对协议程序的某一状态进行符号执行分析，可以灵活高效地控制协议程序空间的路径探索；基于 Dolev-Yao 模型的完美密码假设理论，将密码学函数逻辑空间与协议交互逻辑空间分离，可以避免因密码学逻辑的复杂性造成路径约束求解困难问题。

参考文献：

- [1] FITERĂU-BROȘTEAN P. Model learning and model checking of SSH implementations[C]//Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. 2017.
- [2] ANGLUIN D. Learning regular sets from queries and counterexamples[J]. Information and Computation, 1987, 75(2): 87-106.
- [3] GUO J, GU C, CHEN X, et al. Model learning and model checking of IPSec implementations for internet of things[J]. IEEE Access,

表 3 方法与自带测试套件对比
Figure 3 Comparison of the method presented with the built-in test suite

测试方法	代码覆盖率	代码覆盖率 (不含 libtomcrypt 库)	发现错误数
Libfuzzer 测试套件	38.49%	34.64%	13
State-driver 方式	36.25%	42.37%	20

- 2019, 7: 171322-171332.
- [4] FITERAU-BROSTEAN P, JONSSON B, MERGET R, et al. Analysis of DTLS implementations using protocol state fuzzing[C]//29th {USENIX} Security Symposium ({USENIX} Security 20). 2020: 2523-2540.
- [5] CHAU S Y, CHOWDHURY O, HOQUE E, et al. Symcerts: practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations[C]//2017 IEEE Symposium on Security and Privacy (SP). 2017: 503-520.
- [6] CORIN R, MANZANO F A. Efficient symbolic execution for analysing cryptographic protocol implementations[C]//International Symposium on Engineering Secure Software and Systems. 2011: 58-72.
- [7] VANHOEF M, PIESENS F. Symbolic execution of security protocol implementations: handling cryptographic primitives[C]//12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18). 2018.
- [8] CHAU S Y, YAHYAZADEH M, CHOWDHURY O, et al. Analyzing semantic correctness with symbolic execution: a case study on pkcs# 1 v1. 5 signature verification[C]//Network and Distributed Systems Security (NDSS) Symposium 2019. 2019.
- [9] 孟博, 鲁金钊, 王德军, 等. 安全协议实施安全性分析综述[J]. 山东大学学报:理学版, 2018, 53(1): 1-18.
MENG B, LU J T, WANG D J, et al. Survey of security analysis of security protocol implementations[J]. Journal of Shandong University (Natural Science), 2018, 53(1): 1-18.
- [10] AICHERNIG B K, MOSTOWSKI W, MOUSAVI M R, et al. Model learning and model-based testing[M]//Machine Learning for Dynamic Software Analysis: Potentials and Limits. 2018: 74-100.
- [11] JERO S, HOQUE M E, CHOFFNES D R, et al. Automated attack discovery in TCP congestion control using a model-guided approach[C]//NDSS. 2018.
- [12] JERO S, LEE H, NITA-ROTARU C. Leveraging state information for automated attack discovery in transport protocol implementations[C]//2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2015: 1-12.
- [13] WALLS R J, BRUN Y, LIBERATORE M, et al. Discovering specification violations in networked software systems[C]//2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). 2015: 496-506.
- [14] SONG J S, CADAR C, PIETZUCH P. SymbexNet: testing network protocol implementations with symbolic execution and rule-based specifications[J]. IEEE Transactions on Software Engineering, 2014, 40(7): 695-709.
- [15] ALSHMRANY K, CORDEIRO L. Finding security vulnerabilities in network protocol implementations[J]. arXiv preprint arXiv: 2001.09592, 2020.
- [16] 谢肖飞, 李晓红, 陈翔, 等. 基于符号执行与模糊测试的混合测试方法[J]. 软件学报, 2019, 30(10): 3071-3089.
XIE X F, LI X H, CHEN X, et al. Hybrid testing based on symbolic execution and fuzzing[J]. Journal of Software, 2019, 30(10): 3071-3089.
- [17] CHO C Y, BABIC D, POOSANKAM P, et al. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery[C]//USENIX Security Symposium. 2011: 139.
- [18] YLONEN T, LONVICK C. The secure shell (SSH) protocol architecture[S]. RFC4251, Internet Engineering Task Force, 2006.

[作者简介]



张协力 (1992-), 男, 陕西兴平人, 数学工程与先进计算国家重点实验室博士生, 主要研究方向为网络安全协议。



顾纯祥 (1976-), 男, 安徽霍山人, 博士, 数学工程与先进计算国家重点实验室教授, 主要研究方向为网络安全、密码学。



祝跃飞 (1962-), 男, 浙江兰溪人, 博士, 数学工程与先进计算国家重点实验室教授、博士生导师, 主要研究方向为网络安全、密码学。



陈熹 (1988-), 男, 湖北建始人, 硕士, 数学工程与先进计算国家重点实验室讲师, 主要研究方向为网络安全、密码学。