



毕业论文(设计)

论文(设计)题目：

基于安全多方计算的隐私集合求交系统设计与实现

姓 名 部文潇
学 号 201922301274
学 院 软件学院
专 业 软件工程
年 级 2019 级
指导教师 孔凡玉

2023 年 6 月 6 日

摘要

随着互联网和大数据应用的发展，隐私保护问题也越来越被人们重视，安全多方计算成为了学术界和工业界的重要课题。安全多方计算是指在无可信第三方的情况下，多个参与方共同计算一个目标函数，并且保证每一方都无法通过计算过程中的交互数据推测出其他任意一方的输入数据。隐私集合求交计算（Private Set Intersection, PSI）是安全多方计算领域的特定应用问题，是两方以上用户在不泄露自己数据的前提下计算集合的交集。

本文主要设计了基于 Diffie-Hellman 密钥交换算法的 PSI 协议，并以此协议为核心，实现了一个支持多参与方在线计算的隐私集合求交系统，旨在为小商户等群体提供客户信息的隐私集合求交平台，便于他们寻找共同客户，进而实现为共同客户个性化推荐商品等需求。系统在架构上分为浏览器前端和服务端，前端使用 React 和 Ant Design 开发，服务端基于 Flask 框架构建，按照功能分为登录/注册模块、房间管理模块、数据加密模块、结果解析模块。本文详细介绍了各模块的设计与开发流程，结合前端长轮询、Session、Redis 缓存等技术实现了各模块的功能。在实际运行展示与测试中，系统各模块稳定运行，实现了所需的功能性需求和非功能性需求，能够及时响应各用户的加密和查询请求，满足用户的使用需求。

最后，总结了论文中完成的工作，并在此基础上，对系统未来能够改进和拓展的方面进行了展望。本文实现了带有数据返回和统计的多方隐私集合求交系统，研究成果具有一定的理论意义与实际应用价值。

关键字：安全多方计算；隐私集合求交；隐私计算；Diffie-Hellman 密钥交换；密码学

ABSTRACT

With the development of Internet and the application of big data, privacy protection has become increasingly important. Secure multi-party computation has become an important topic for both academia and industry. Secure multi-party computation refers to the joint computation of a target function by multiple participants without a trusted third party and ensuring that no participant can infer any other participant's input data through the interactive data during the calculation. Private Set Intersection (PSI) calculation is a specific application problem in the field of secure multi-party computation, which calculates the intersection of sets for two or more users without revealing their data.

This paper designs a PSI protocol based on the Diffie-Hellman key exchange algorithm and implements a privacy set intersection system that supports multi-party online calculation based on this protocol. The purpose of the system is to provide small merchants and other groups a privacy set intersection platform for customer information, which facilitates their search for common customers and provides personalized recommendations for goods and other needs. The system is structured with a browser frontend and a server backend. The frontend is developed using React and Ant Design, while the backend is built on the Flask framework and is divided into login/registration modules, room management modules, data encryption modules, and result parsing modules. This paper details the design and development process of each module, and combines technologies such as frontend long polling, session, and Redis cache to implement the functionality of each module. During actual operation, demonstration, and testing, the system's modules run stably, fulfilling the functional and non-functional requirements, and timely respond to each user's encryption and query requests, meeting their demands.

Finally, this paper sums up the work completed for the system, and based on this, presents prospects for improving and expanding the system in the future. The research result implements a multi-party privacy set intersection system with data return and statistics. It has certain theoretical significance and practical application value.

Key Words: Secure Multi-Party Computation, Private Set Intersection, Privacy calculation, Diffie-Hellman Key Exchange, Cryptography

目 录

第 1 章 前言	1
1.1 隐私集合求交系统开发背景.....	1
1.2 国内外研究现状.....	1
1.3 解决的主要问题.....	2
1.4 论文的组织结构.....	3
第 2 章 隐私集合求交系统需求分析	4
2.1 隐私集合求交系统概述.....	4
2.1.1 项目说明	4
2.1.2 用户操作流程	4
2.2 隐私集合求交系统需求问题描述	5
2.2.1 系统功能性需求	6
2.2.2 系统非功能性需求	6
第 3 章 系统架构设计	8
3.1 系统设计目标和原则	8
3.2 系统技术架构设计	8
3.2.1 系统开发环境	8
3.2.2 系统逻辑架构	9
3.2.3 系统角色设计	9
3.2.4 系统功能组成	10
第 4 章 隐私集合求交算法设计	12
4.1 加密算法.....	12
4.2 前端加密流程.....	15
4.3 服务端响应及求交算法.....	16
第 5 章 隐私集合求交系统前后端设计	18
5.1 系统工作流程.....	18
5.2 系统模块设计	18
5.2.1 系统前端组件设计	18

5.2.2 登录/注册模块详细设计	23
5.2.3 加密房间模块详细设计	25
5.2.4 上传数据模块详细设计	26
5.2.5 结果展示模块详细设计	32
5.3 系统存储设计	32
5.3.1 Redis 数据库设计	32
5.3.2 MySQL 数据库设计	34
5.3.3 Session	35
第 6 章 系统展示与测试	38
6.1 隐私集合求交系统	38
6.1.1 登录与注册	38
6.1.2 房间列表	39
6.1.3 数据上传	43
6.1.4 结果查询	44
6.2 加密算法工具站展示及测试	46
6.2.1 工具站概述	46
6.2.2 常用加密部分	48
6.2.3 AES 算法部分	48
第 7 章 总结与展望	50
7.1 工作总结	50
7.2 展望	50
参考文献	52
致 谢	54

第1章 前言

1.1 隐私集合求交系统开发背景

自 1982 年姚期智博士提出百万富翁问题^[1]以来，隐私计算经过几十年的发展，技术日趋成熟，应用越来越广泛，遍及医学、金融、社交网络等各个领域。尤其是大众创业、万众创新的浪潮下，小型 APP 开发者、电子商务从业者等个人用户对隐私计算的需求越来越多，但目前的隐私集合求交系统仍以大型公司的定制系统（如 Google 用于检验用户密码是否泄露的隐私增强技术^[2]）为主，不便于个人用户的使用。

本系统的开发目标是提供一个方便易用的隐私集合求交网站，通过运行隐私集合求交算法，协助有扩大经营范围的商户安全地求出与其他相关商户的客户集的交集，从而更精准地发现潜在客户群体，以广告投放、个性化推荐等方式为所有参与方创造商业价值。系统支持多方（两个及以上）用户在线进行集合求交任务，具有以下特点：

1. 便捷，系统采用 B/S 架构，前端功能基于 JavaScript 实现，用户无需安装客户端或插件，打开浏览器即可直接使用。
2. 实用：系统提供的功能可以广泛使用于在线广告投放、个性化推荐系统、个人社交媒体等领域。
3. 保密，用户原始数据及密钥均保存在本地，不会上传到服务器，只有密文被上传并参与计算，从而降低了隐私数据泄露的风险。

1.2 国内外研究现状

隐私计算^[3]是在保证数据提供方不泄露原始数据的前提下，对数据进行分析计算的一系列信息技术，保证数据“可用不可见”。隐私集合求交（PSI）是安全多方计算领域的特定问题^[4]，对于两个或多个集合中的元素，在不泄露集合原始信息的前提下，多方协同计算集合的交集。

2004 年，Freedman 等人^[5]构造了第一个安全 PSI 协议，PSI 计算的研究由此开始快速发展，成果不断增加。目前，隐私计算处于快速发展阶段，各国政府的

隐私保护意识逐渐增强、法律不断完善，我国也于 2016 年发布《中华人民共和国网络安全法》，这是是我国网络空间法治建设的重要里程碑，这也让隐私计算日益走近大众的生活。目前，隐私集合求交技术已被应用于医学、金融乃至社会生活等各个领域。例如，对医学数据的集合求交计算可以帮助不同医院的医生更好地诊断疾病，而不泄露患者的隐私信息；金融领域可以在不暴露客户敏感信息的前提下使用隐私集合求交来检查欺诈行为，预测风险，实现智能风控^[6]，这都为人们的生活提供了便利与保障。此外，隐私集合求交技术也为一些对数据隐私要求较高的场景提供了有效的解决方案，如个性化推荐、用户画像等。

根据算法中是否有第三方参与，隐私集合求交技术可以分为传统 PSI 和云辅助的 PSI 两大类^[7]。其中传统 PSI 技术是使用底层密码技术对数据进行加密、各参与方之间执行计算的技术，根据其底层技术的不同又可分为基于公钥加密的 PSI、基于茫然传输的 PSI^{[8][9]}、基于混乱电路的 PSI^[10]等。云辅助的 PSI^[11]是各参与方使用加密函数对数据进行加密后，上传到第三方服务器 P，由 P 对密文进行存储、交集计算，且在此过程中，P 不会得到有关隐私数据的任何信息。

现如今，隐私计算正处于稳定发展阶段，应用规模稳定增长，逐渐与大众的生活紧密相连。随着国家相关法律政策的颁布和实施，

1.3 解决的主要问题

在隐私计算技术发展的同时，大型组织对相关技术的应用越来越广泛，但小商户、个人用户受限于服务的昂贵和复杂，难以真正使用隐私计算技术来创造经济效益。本文的隐私集合求交系统是针对多用户、小数据的隐私数据做集合求交计算这一需求设计并开发的系统，在借助服务端的存储、计算能力，快速求出用户数据交集的同时，不将用户的原始数据、密钥数据上传到服务端，且不对其他用户透露该用户的具体数据信息（只返回交集），从而实现数据的可用不可见。

除隐私集合求交功能外，系统还提供了加密算法工具站页面，以满足用户对隐私数据的其他加密需求。

需求分析方面，充分考虑了用户的需要，以便捷、易用性作为突破点，总结得出开发思路，即：参与计算的用户通过对彼此的验证，进入同一个集合求交过程，上传数据后系统自动执行加密操作；用户查询结果时，系统返回求交结果、

交集大小等信息。基于这一思路，对系统结构、功能进行了合理的设计，能够满足用户对系统的功能需求和非功能需求。

在实现方面，系统开发使用前后端分离的开发方案，前端采用 React + Ant Design 技术栈^[12]，服务端采用 Flask Web 框架开发。根据系统架构要求，关键算法采用基于 Diffie-Hellman 密钥交换算法^[13]的云辅助 PSI 技术，实现了快速、安全的集合求交及数据返回功能在传输和计算阶段都对数据进行加密处理，保证隐私数据安全，最终将交集结果呈现给用户。

1.4 论文的组织结构

本文依次对隐私集合求交系统的需求分析、总体设计、各模块的详细设计进行了说明，并对系统最终实现的功能进行了测试和展示。

第一章为绪论部分，介绍了系统开发的背景，说明了隐私集合求交技术的发展现状和本系统的设计思路，便于读者了解相关技术，理解论文内容。

第二章为隐私集合求交系统的需求分析和总体设计，说明了系统的用户群体、主要功能等信息，展示了系统的整体架构，从功能性需求和非功能性需求两个方面对系统的需求进行了分析。

第三章为系统架构设计，从整体的角度提出了系统设计的目标和原则，并说明了系统开发环境，对系统的结构、角色、功能设计进行了简要的阐述，为系统的具体开发做好了规划和布局。

第四章为算法设计，重点介绍了系统对隐私集合求交算法的实现，包括算法的总流程、前端对数据的处理、服务端对密文的保存和求交，以及算法设计过程中遇到的问题及解决方案。

第五章为前后端设计部分，介绍了系统中设计的长轮询、导航栏、漫游式引导等组件，全面地介绍了系统开发过程中使用的技术。

第六章为系统功能展示与测试，通过系统运行的实际截图，全面展示了系统实现的功能及使用方式，证明系统能够正常运行，实现集合求交和加密工具两大部分的相关功能，并对加密工具站进行测试。

第七章为总结与展望部分，对系统开发过程中学习到的经验与系统当前的不足之处进行总结，并对系统可以增强、优化、扩展的内容作了展望和规划。

第2章 隐私集合求交系统需求分析

2.1 隐私集合求交系统概述

2.1.1 项目说明

在互联网高度发达的信息时代，隐私保护尤为重要。但在使用网络平台（如电子商务平台等）时，用户不可避免地面临着分享个人数据（包括个人基本信息、购物偏好等信息）以换取个性化服务与隐私保护之间的矛盾，但这些个性化服务能够为用户和商户都带来收益，不应当无理由地禁止，由此，用户和互联网服务提供者都产生了对隐私集合求交的需求。

例如：在某电子商务平台上，一家服装店想要拓展童装业务，但服装店很难直接从其原有的客户群体中推断出哪些客户可能有孩子、有购买童装的需求，因此无法精准地投放广告；如果服务店能够与儿童玩具店合作，获得儿童玩具店的客户集合，再与自己已有的客户集合求交集，就能够认为交集客户是潜在的童装客户群体，可以对他们进行个性化推荐。然而，直接使用客户联系方式等个人信息的明文求交会泄露客户隐私，甚至引发法律纠纷；借助隐私集合求交系统，服装店可以安全地与儿童玩具店进行合作，在双方都不知道对方明文数据的情况下，获得与对方的客户交集，再对交集进行分析、针对交集中的客户投放广告，以较低的投入、较高的效率吸引潜在客户，获取经济效益。

2.1.2 用户操作流程

在 2.1.1 节的例子中，假设商户持有每个客户的电子邮箱（在实名制要求下，可以认为每个客户的电子邮箱都是唯一的，可以作为该客户的身份凭证。根据商户的要求，也可以使用手机号等数据加密，与参与本次求交的其他商户协商一致即可）。为了避免有其他人恶意参与集合求交、干扰甚至破坏求交结果，每次隐私集合求交都需要在单独的房间内进行。同时，房主可以设置求交结果的获取权限，让房间成员无法获取求交结果。求交算法运行完毕后，会向有权限获得结果的商户返回。

首先，服装店在系统中创建一个求交房间，等待儿童玩具店的加入；在儿童玩具店申请加入后，必须由房主（服装店）同意申请，儿童玩具店才能正式加入

房间。随后，两商户各自上传其客户的电子邮箱数据（多条数据以空格、逗号、分号或换行符分割），上传或生成密钥用于加密数据，点击提交后，系统自动在前端使用密钥对电子邮箱信息进行加密，并将密文传输到服务端，自动执行一系列加密流程（具体的加密算法在第四章介绍）。计算出求交结果（实际是结果的索引，由于加密过程中没有打乱商户数据的顺序，因此前端可以根据索引，解析出具体的交集电子邮箱），将交集邮箱的数据完整输出到网页上，供商户查看、下载。

在此过程中，服装店、儿童玩具店两个商户各自需要进行的操作如图 2-1 所示。由于需求是服装店中产生的，因此服装店负责创建求交房间，作为房主发起本次隐私集合求交计算。

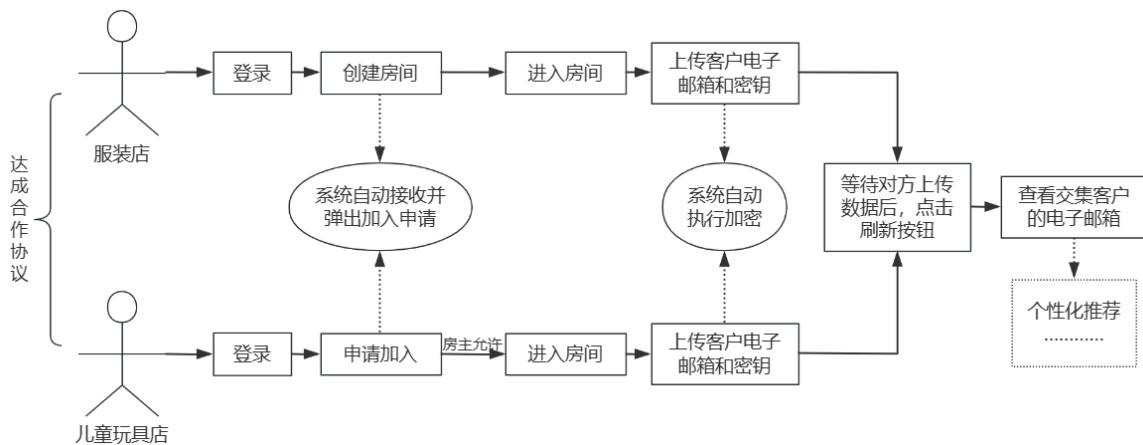


图 2-1 用户操作流程

2.2 隐私集合求交系统需求问题描述

为实现 2.1 节中所述功能和目标，在参与计算前，商户应该提供如下凭证：商户的唯一标识；参与同一计算的商户应能够不受他人打扰地求交；商户的数据和加密用的密钥。在隐私集合求交计算完成后，商户中的一人或所有人能够查看交集具体内容。

反映到系统设计中，即系统应当具备以下功能：登录与注册、创建供隐私集合求交的单独房间、上传数据和密钥、结果返回和处理。

根据上述分析，下文将详细介绍本系统的需求及由此进行的系统设计内容。

2.2.1 系统功能性需求

为了按照流程向商户（下文称为用户）提供服务，系统必须具备如下能力：

1. 用户身份检验

作为在线系统，用户必须注册并登录后才能使用相关功能。同时，在用户登录后进行各项操作前，系统都必须进行身份验证，保证用户只能执行合法操作。

2. 数据上传

要进行求交运算的用户输入原始数据及密钥后，系统自动运行基于 DH 算法的 PSI 协议对数据进行加密，并将原始数据安全地保存在用户本地，仅将密文发送到服务端，随后启动长轮询，向其他用户提供加密服务。

具体来说，如果该用户不是第一个上传数据的用户，那么他需要为比他早上传数据的所有用户加密数据；如果之后还有新用户加入，那么他也需要为新来的用户加密。最终，所有用户的数据都被所有密钥同态加密过，从而能够在密文上求交集。

3. 结果查看

用户上传数据后，可以点击按钮查询求交结果。由于加密是系统通过轮询自动完成，因此用户不需要手动进行其他操作，可以直接查询与当前已经上传数据的所有用户的数据的交集大小、交集内容等信息。

2.2.2 系统非功能性需求

除了满足用户需求外，系统本身应当具备以下特性，以保持系统安全、稳定运行，完成隐私集合求交的任务。

1. 安全性

系统应保护数据不被除用户外的任何人了解、篡改、破坏，将用户隐私数据和密钥保存在前端，浏览器关闭后自动删除；密文保存在服务端，房间关闭后自动删除。

2. 易用性

系统页面简洁清晰，对于每项功能及其使用方法都有完善的提示信息（包括文字提示和漫游式引导）及错误处理机制，便于用户使用。

3. 可扩展性

系统前端使用 React 组件式开发，增加新功能、新页面只需编写为组件即可集成到系统中；服务端使用 Flask 框架，利用其装饰器特性可以方便地增加新服务，可扩展性强，有利于系统将来的完善和优化。

第3章 系统架构设计

本章从技术和功能角度介绍了系统的架构思路及用户角色设计，对系统开发工作进行了总体规划。

3.1 系统设计目标和原则

作为隐私计算系统，用户身份和数据的安全性非常重要。系统主要应当遵循以下3个原则：

1. 数据保密性

系统必须保证用户密码、用户上传数据的保密性，确保敏感信息不会被除用户本人外的任何一方获取。

2. 数据完整性

系统必须确保数据在传输的过程中不会因为外界干扰而损失或被篡改，存储在服务器内的数据不会被其他用户篡改^[14]。

3. 访问控制

系统必须具备良好的访问控制机制，只有经过授权（如通过系统注册、登录）的人员才能访问系统、创建房间、参与隐私集合求交计算，从而保障系统的安全性。

除此之外，系统设计和开发还应该遵循软件工程原则，包括可维护性、可扩展性、可靠性等。

3.2 系统技术架构设计

系统技术架构包括系统开发环境和逻辑架构两部分。在该开发环境下，系统能够稳定运行，实现逻辑架构中设计的各项功能。

3.2.1 系统开发环境

1. 硬件环境

11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz

2. 软件环境

Windows 10 家庭中文版, Visual Studio Code, Chrome 浏览器

3.2.2 系统逻辑架构

系统采用 B/S 架构，用户启动浏览器即可使用系统服务。

从技术角度来看，前端使用了 React 框架、Ant Design 组件库、axios 等库开发，服务端主要使用了 Flask Web 框架、Session、Redis 缓存等技术。前端与服务端使用 POST 方法通信，协同完成求交任务。

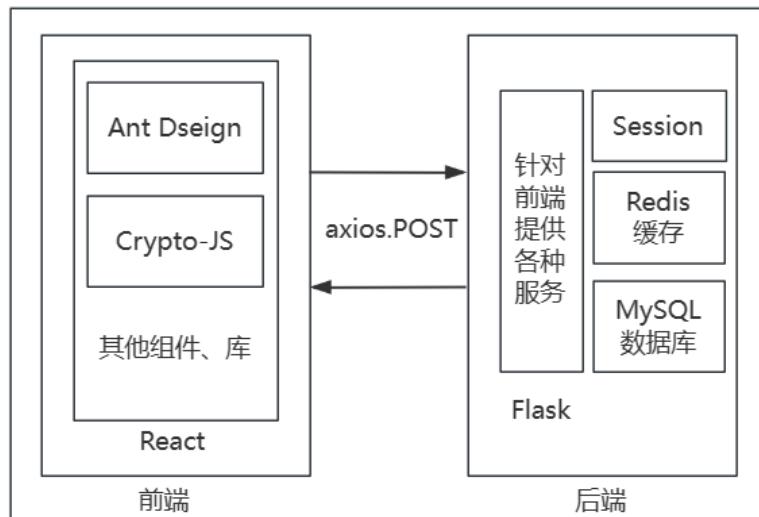


图 3-1 逻辑架构

3.2.3 系统角色设计

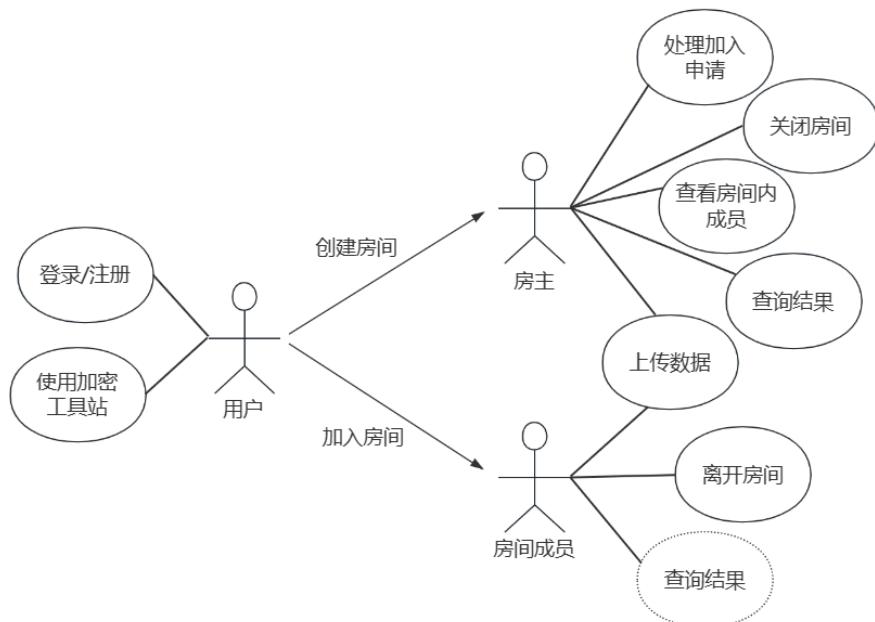


图 3-2 系统用例图

为了让用户能够选择合作对象、安全地共同执行隐私集合求交计算，系统引入房间的概念。参与同一计算的用户需要先进入同一房间，才能进行计算。先由一位用户创建房间、成为房主，其他人申请加入该房间，成为房间成员。

房主是用户创建房间之后成为的新角色。房主可以查看房间内成员列表，会自动接收其他用户的加入申请，并能够选择同意或拒绝；房主可以随时关闭房间，销毁房间内已上传的数据；房主上传数据后，可以查询与其他已上传数据的用户的求交结果，并生成 txt 文件，保存到用户本地。

房间成员是用户加入他人的房间之后成为的新角色。房间成员进入房间后，唯一能够执行的操作就是上传数据和密钥。上传数据前，用户可自由退出房间，不会影响房间的交集计算；在用户上传数据、完成密文传输后，可以查看房间成员和求交结果（如果房主允许），无法退出房间。

房主和房间成员的角色不是固定的，当房间过期、房主关闭房间、成员退出房间时，用户会被自动跳转到房间列表页面，可以成为新的角色。

此外，用户无需登录即可使用加密工具站功能，在工具站中输入的数据也不会被系统保存、上传。

3.2.4 系统功能组成

系统功能模块主要包括隐私集合求交系统、加密算法工具站两大部分。

隐私集合求交系统是本文研究的重点。为了在线进行隐私集合求交计算，用户需要注册、登录，以确定身份；登录后，用户在与约定好的其他用户进行求交计算前，需要先创建或加入房间，以避免大量数据混杂在一起，或被恶意用户破坏求交结果；进入房间后，用户上传数据和密钥，系统会安全保存数据，并将密文发送到服务端，运行求交函数，将交集的结果索引返回到用户前端，再由前端对应解析，得到具体的交集内容并展示在页面中。

加密算法工具站是附加功能。该页面提供了 SHA256 等常见密码散列函数和 AES 算法的自助计算，用户可根据需要，使用工具站进行加密。

根据上文的分析，系统包含的各模块及其关系如图 3-3 所示。其中，数据处理模块和结果展示模块由本系统设计的隐私集合求交算法实现，将在第 4 章中详细介绍。

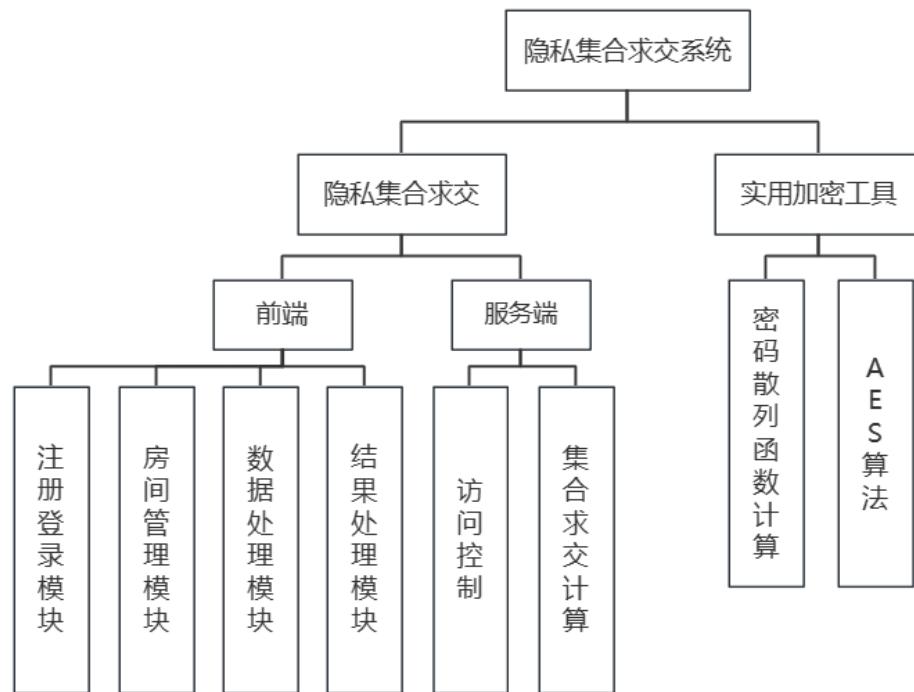


图 3-3 系统功能组成

第 4 章 隐私集合求交算法设计

隐私集合求交算法是本系统的重要内容。该算法是为了实现对多个用户的数
据进行求交，同时不泄露用户数据和密钥给其他任何一方而设计的。

算法的流程为：用户在前端上传数据和密钥；服务端只获取并保存密文，并根据密文求交，将求交结果的索引返回给各参与方用户；前端解析并展示求交结果，实现数据返回功能。这一系列操作保证了用户的隐私数据的安全性，同时满足了隐私集合求交的需求。本章分别从总览、前端流程、服务端流程三个角度，介绍了算法的具体内容。

4.1 加密算法

在学习了基础密码学算法^[15]、尝试了 github 中多个开源隐私集合求交算法后，并没有找到快速、安全且适合 B/S 架构的开源库，因此根据系统的需求和架构，设计了一个通过前后端通信实现的隐私集合求交算法。本系统的算法结合了 Diffie-Hellman 密钥交换算法和分组密码思想^[16]，提出了使用前后端通信来实现的 PSI 协议，借助前端的长轮询和服务端的加锁、响应，实现用户间的加密。

Diffie-Hellman 密钥交换算法：简称 DH 算法，该算法是用于共享秘密密钥的协议，允许通信双方安全地选择出一个共享密钥。算法执行过程如下：

- i. 选择质数 g 和 p ，其中 g 是 p 的原根， g 和 p 的值是公开的。
- ii. 参与者 A 选择自己的密钥 A ，计算 $g^A \bmod p$ 并发送给 B
- iii. 参与者 B 选择自己的密钥 B ，计算 $g^B \bmod p$ 并发送给 A
- iv. 双方各自用自己的密钥加密，得到 $g^{AB} \bmod p$ 和 $g^{BA} \bmod p$ 并返回给对方
- v. 上述两个值是相同的，由此双方得到了共享密钥 $g^{AB} \bmod p$

该算法经过拓展，可以用于实现隐私集合求交算法中的密文计算，具体协议将在下文中详细介绍。

分组密码：将明文数字序列，划分成长度为 n 的组，每组分别在密钥的控制下转换成等长的密文数字序列的方法。分组密码的主要目的是通过扩散、扰乱明文的统计结构，增强加密的安全性。

在本系统中，采用分组方案的主要原因是 JavaScript 的整数大小限制，JavaScript 不支持 double 类型，支持的最大整数是 32 位 int，但 int 大小的数据集

合容易发生碰撞，远远不能满足加密的需要。虽然新版本的 JavaScript 提供了 BigInt 数据类型，但由于前端依赖间的版本兼容等问题，实际使用后发现 BigInt 仍然存在精度不足问题，超出 32 位的部分会被填充为 0，仍无法满足系统的加密需求。因此，决定使用分组加密的方案，将哈希后的字符串转换为十进制串后，分为 5 个一组，实现了加密算法。

算法流程：以两个用户的情况为例，隐私集合求交计算的整体流程见图 4-1.

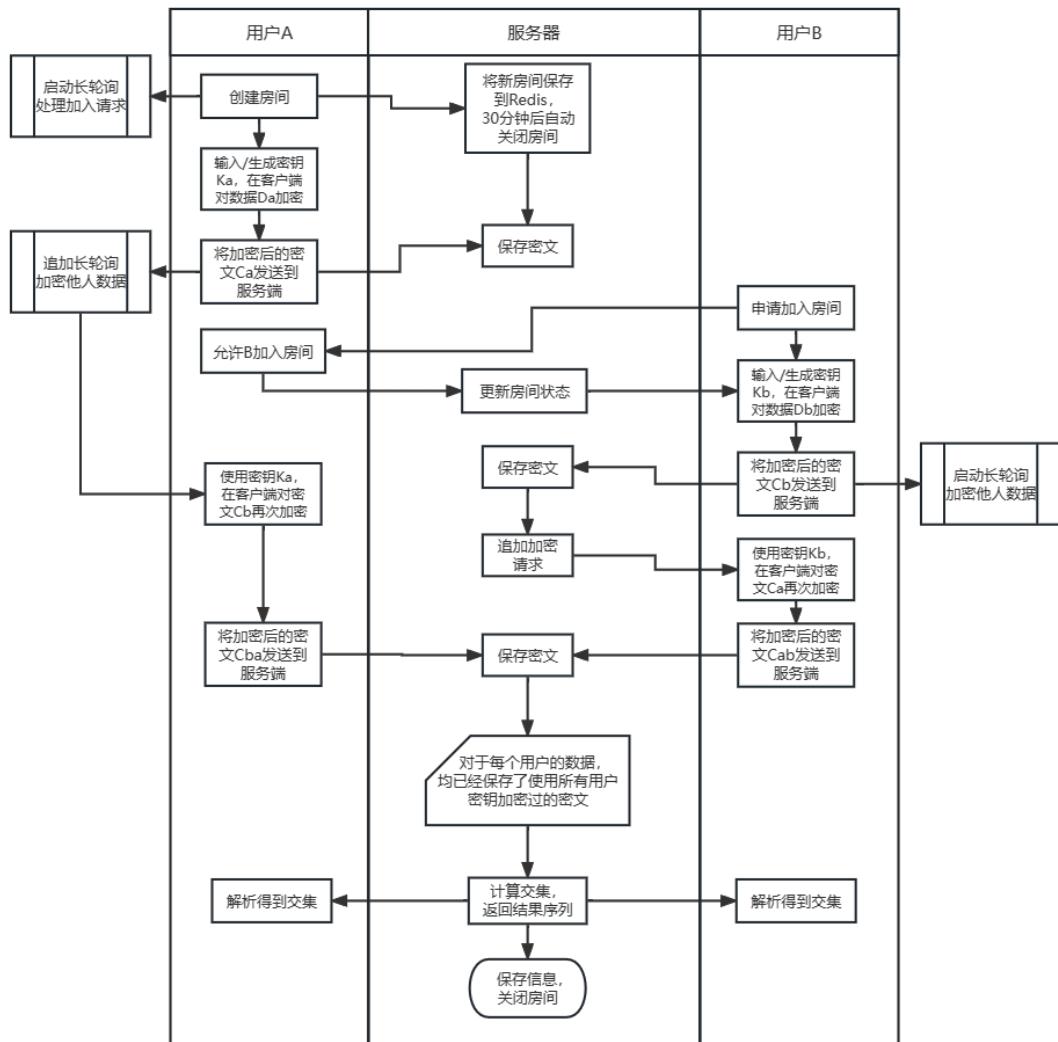


图 4-1 隐私集合求交流程

算法流程比较长，现对其进行具体说明。为了便于叙述，假设 A, B 两名用户参与计算，其明文分别为 P_A 、 P_B 密钥分别为 K_A 、 K_B 。其中，A 是房主，房间内只允许房主查看求交结果，且 A 先于 B 上传数据。

1. 用户 A 上传 P_A (List) 和 K_A (String)，点击 Submit 按钮提交

2. A 前端使用 SHA256 密码散列函数，对 P_A 哈希得到 H_A
3. 对 H_A 中的每个元素，截取其第 9-24 位，并转化为十进制串，得到 S_A 。随后，对于 S_A 中的每个元素 $Item_{Ai}$ ，执行以下操作：
 - i. 将 $Item_{Ai}$ 分割为多个（实际上是 4 个）长度为 5 的串，不足 5 位的填充前导 0。此时 $Item_{Ai}$ 是一个列表，包含 4 个元素，即 4 个十进制串。
 - ii. 对于 $Item_{Ai}$ 中的每个元素 O_i ，使用快速幂算法计算 $O'_i = O_i^{K_A} \bmod 65537$ 。如果 O'_i 不足 5 位，用前导 0 补全。
 - iii. 计算完毕，得到加密后的 $Item'_{Ai}$ ，并将其重新合并为字符串。
4. 重复以上操作，在 A 前端将 S_A 加密为 E_A ， E_A 传输给服务端。A 前端启动长轮询，检查是否有新用户（晚于 A 上传数据的用户）加入。
5. 服务端，检查是否有老用户（比 A 更早上传数据的用户），发现结果为空，上传数据操作执行完毕。
6. 用户 B 上传 P_B 和 K_B ，对于 P_B 和 K_B 重复第 2 至 4 步，将 E_B 传输给后端。B 前端启动长轮询，检查是否有新用户加入，若有则为其加密。
7. 后端检查是否有老用户（比 B 更早上传数据的用户），发现用户 A，获取互斥锁，将 E_A 发送给 B，同时向 A 添加任务：加密用户 B 的密文。
8. B 前端收到加密任务，使用 K_B 对 E_A 加密（重复第 2 至 4 步），将加密后得到的 E_{AB} 传输给服务端，覆盖原本的密文 E_A ，释放互斥锁。
9. A 前端在轮询时发现加密任务，获取互斥锁后，使用 K_A 对 E_B 加密，将 E_{BA} 传输给服务端，覆盖原本的密文 E_B ，释放互斥锁。
10. A 点击 Refresh result 按钮，服务端检查发现 E_{AB} 和 E_{BA} 满足求交条件，可以求交，执行集合求交算法，求出 A 与其他用户的密文交集并将结果（交集内容索引组成的列表）返回。
11. A 前端收到交集索引，根据 A 的明文数据解析后得到交集具体数据，在网页上以表格形式展示结果。
12. B 上传数据后，跳转页面没有刷新按钮，无法获取求交结果，需要保持浏览器运行直到房间关闭。

4.2 前端加密流程

在加密过程中，前端完成了对用户数据的哈希、分组、加密^[17]，并将密文传输至服务端，其流程如图 4-2 所示：

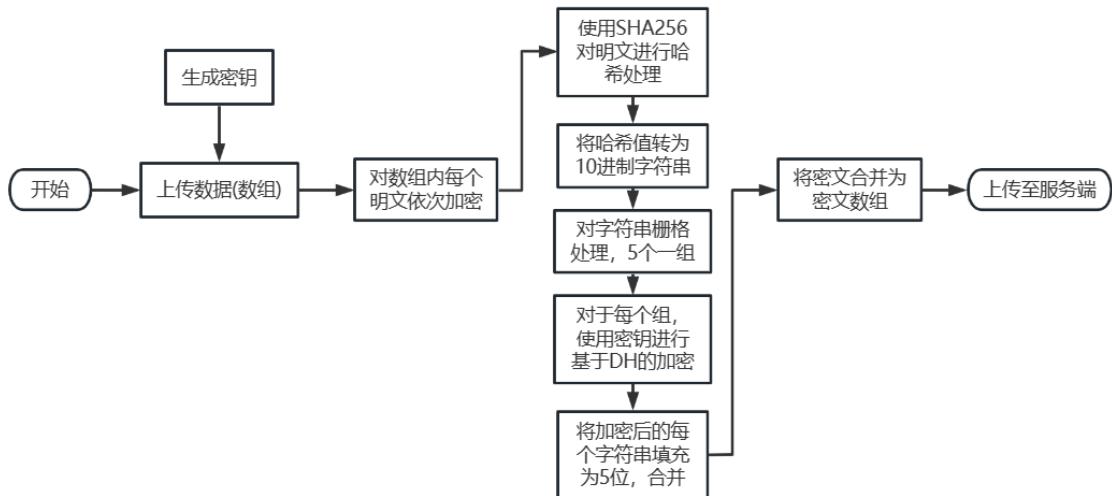


图 4-2 前端加密

将密文分组后，加密的工作量较大。为了提高算法效率，减少用户的等待时间，在系统执行基于 DH 的加密时，使用了快速幂算法加速求幂运算。

快速幂：快速幂利用了幂次的二进制转换，将求一个数的 n 次幂的时间复杂度从 $O(n)$ 降低为 $O_{log(n)}$ 。快速幂的函数实现如下。

```

function QuickPow(a, b, mod) {
    let ans = 1;
    while (b) {
        if (b & 1)
            ans = ans * a % mod;
        a = (a * a)% mod;
        b >>= 1;
    }
    return ans;
}
  
```

完成加密、上传数据后，数据和密钥保存在用户前端。同时前端启动长轮询，定期询问房间内是否有新用户。如果有新用户上传了数据，当前用户需要为其加密，系统会自动运行加密算法，使用 sessionStorage 中的密钥为其加密。

最终，后端保存的密文全部被更新为由每个用户各加密一次的密文，从而可以直接进行遍历，执行集合求交算法、获取交集索引。

4.3 服务端响应及求交算法

响应：

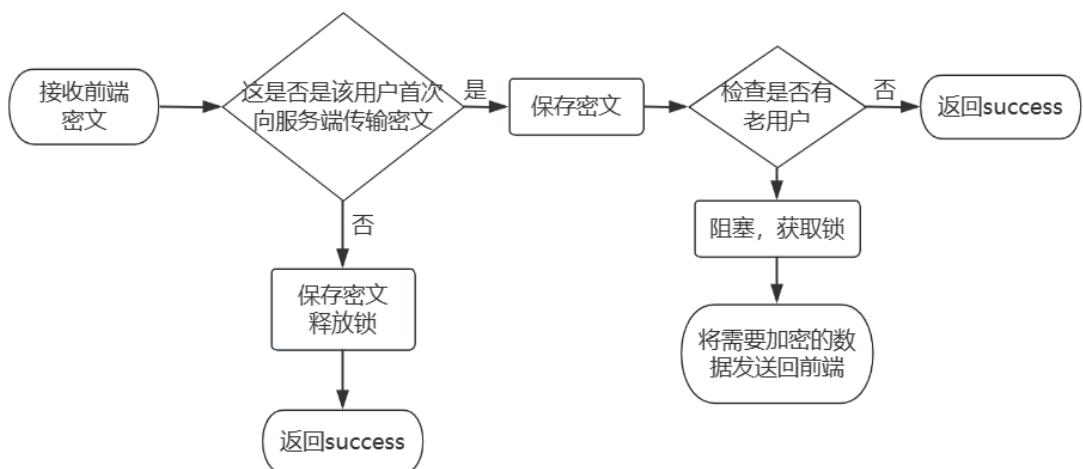


图 4-3 服务端响应

通过用户向服务端发送密文时，有两种可能的情况：

1. 这是用户加入房间后首次发送密文（且发送的是对自己的数据加密后得到的密文）。此时，服务端还需要检查是否有其他用户已经先于该用户上传了密文，如果有，那么该用户需要再使用自己的密钥对这些密文进行加密。
2. 这是用户加入房间后第二次（或更多次）发送密文（发送的是对他人密文再次加密后的密文）。这代表用户已经完成了 1 中的步骤，正在响应服务端对于其他用户的加密请求，只需简单地将本次上传的密文保存即可。

求交：

从 Redis 中所保存的密文的角度来看，算法流程如下：

1. 用户 A 上传使用密钥 K_A 加密后的密文 E_A

2. 用户 B 上传使用密钥 K_B 加密后的密文 E_B
3. 用户 B 使用密钥 K_B 将 E_A 更新为 E_{AB}
4. 用户 A 使用密钥 K_A 将 E_B 更新为 E_{BA}
5. 此时已经可以查询 A 和 B 的交集结果。由于 DH 算法满足乘法同态，对于相同的明文， $E_{AB}=E_{BA}$. 当 A 查询交集时，计算流程如下：
 - i. 获取其他用户列表，列表中只有一个用户 B.
 - ii. 对于 B 密文列表中的每一个元素，遍历 A 的密文，比较是否有完全相同的密文；如果有，则保存 A 中对应的索引。
 - iii. 如果有其他用户，对其他用户重复 ii 中的操作
 - iv. 将保存索引的列表返回给前端，前端根据索引和用户明文数据获取具体的交集内容，展现在结果表格中。
6. 用户 C 上传使用密钥 K_C 加密后的密文 E_C
7. 用户 C 使用密钥 K_C 将 E_{AB} 更新为 E_{ABC} , E_{BA} 更新为 E_{BAC}
8. 用户 B 使用密钥 K_B 将 E_C 更新为 E_{CB}
9. 用户 A 使用密钥 K_A 将 E_{CB} 更新为 E_{CBA}
10. 此时可以查询 A, B, C 的交集结果，方法同上

其中，第 8、第 9 步的顺序可能会颠倒，具体取决于在 C 上传数据后，A 和 B 谁先向服务端发送轮询请求。此处的顺序不影响算法执行过程，也不影响求交结果。

在用户 A 点击按钮、尝试获取结果时，服务端会对 A 和其他所有用户两两组合，执行时间复杂度为 $O(n^2)$ 的循环算法，找出重合密文的索引（这里发出请求的是 A，因此取 A 的索引）。计算完毕后，将索引返回给用户 A 的前端。由于加密算法不会打乱数据原有的顺序，因此，若 A 的第 i 个密文与 B 中某个密文相同，那么 A 的第 i 个明文也必定与 B 中某个明文相同，即 i 存在于 A 与 B 的交集中，由此可以解析出交集的真实数据。

经测试，系统加密 10000 条数据用时约为 4s，对两个 10000 规模的数据集求交用时约为 6s，速度能够满足用户需求。

第 5 章 隐私集合求交系统前后端设计

第 3 章概括说明了系统的业务需求和系统总体架构，第 4 章说明了系统核心算法，本章在上述设计的基础上，结合系统代码和界面，详细说明系统内各模块的设计思路和实现方式，以及服务端使用的数据存储结构。

5.1 系统工作流程

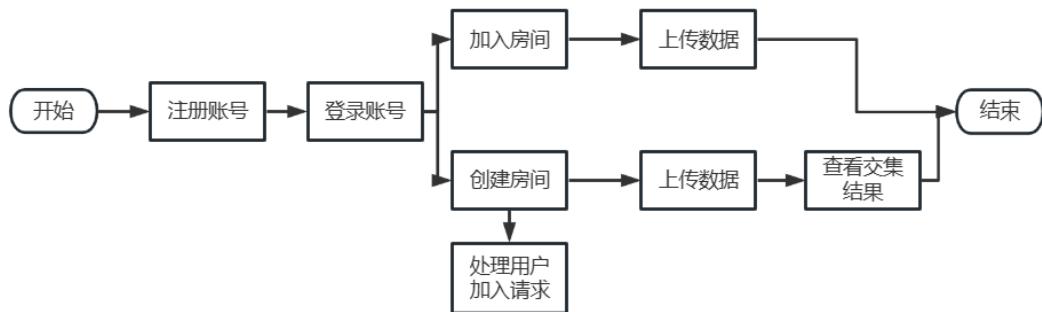


图 5-1 总体工作流程

本系统作为一个在线系统，用户必须注册、登录账号后才能通过前端和服务端的身份认证，使用相关功能。

登录后，用户进入房间即可启动隐私集合求交流程。在用户上传数据和密钥后，系统自动运行第 4 章中介绍的隐私集合求交算法，进行加密和求交操作。在加入房间的所有用户均上传数据后，房主及有权限的房间成员可以刷新结果列表，查看求交结果。

5.2 系统模块设计

根据图 5-1 的总体工作流程，可以将隐私集合求交系统的前端页面划分为四大部分：登录/注册、房间列表、上传数据、结果展示等。后端针对每个页面执行的功能编写服务，响应请求。每个部分都由若干小模块组成。

5.2.1 系统前端组件设计

路由：系统有多个页面，不同页面间的跳转使用 react-router-dom6 实现。该路由系统的结构示意图如下：

```
<BrowserRouter>
  <Routes>
    <Route path='/login' element={<Login />} />
    <Route path='/register' element={<Register />} />
    .....
  </Routes>
</BrowserRouter>
```

在 BrowserRouter 组件中，路由系统通过 history 模式切换路径。与 5.0 版本的 HashRouter 相比，6.0 版本刷新后不会丢失数据，url 也更美观；但与此同时，BrowserRouter 组件中无法再使用 window 自带的 hashChange 监听器。为了实现系统前端的访问控制，监听当前路径，系统使用 React 提供的 useEffect 钩子监听 window.location.pathname 的值，来获取当前路径、多态地处理用户操作。

栅格化布局：为了让页面简洁、美观，易于开发和增加新内容，前端页面使用 Ant Design 的栅格化布局系统。

这种布局方式以“行（Row）”和“列（Col）”划分页面，每个页面都有若干行，每行有 24 列。布局的代码可按如下方式编写：

```
<Row>
  <Col>
    { /* working components */ }
  </Col>
  { /* can add more col here */ }
</Row>
{ /* can add more row */ }
```

对于登录、注册这样有明显的行列划分，且只需简单地将组件居中处理的页面，如上图所示即可实现；对于其他有特殊要求的页面，可以通过 Col 组件的 offset、margin 等属性对组件的位置进行调整，实现整个页面的美观布局。

步骤条：用户执行隐私集合求交计算总共分为三步：加入房间、上传数据、等待他人上传数据并查看求交结果。Ant Design 的 Step 组件可以清晰地展示用户当前所在的步骤，提示用户下一步操作。

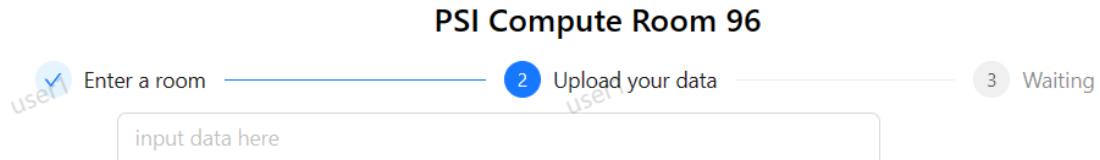


图 5-2 步骤条示例

垂直导航栏：用户登录后，可能会需要手动跳转到各个页面，因此在系统的功能页面中会展示一个简单的导航栏。

由于 Ant Design 的不同组件嵌套使用时，经常出现错位等问题，因此系统没有在 Layout 组件中嵌套 Row 和 Col 组件，而是使用 Menu 组件。用户登录后会显示导航栏，导航栏固定在所有功能页面的右下角。

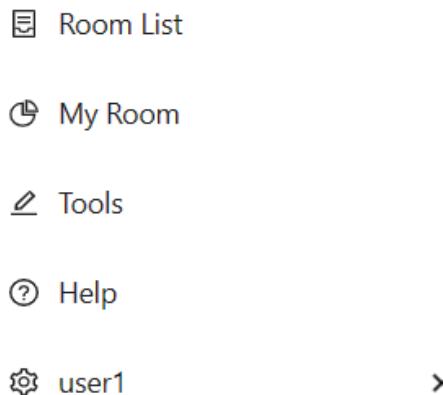


图 5-3 导航栏

每个选项（Item）的功能如下。

Room List：点击后跳转到房间列表页面。

My Room：点击后跳转到用户所在房间，根据用户当前的具体状态，可以分为如图 5-4 所示的四种情况：

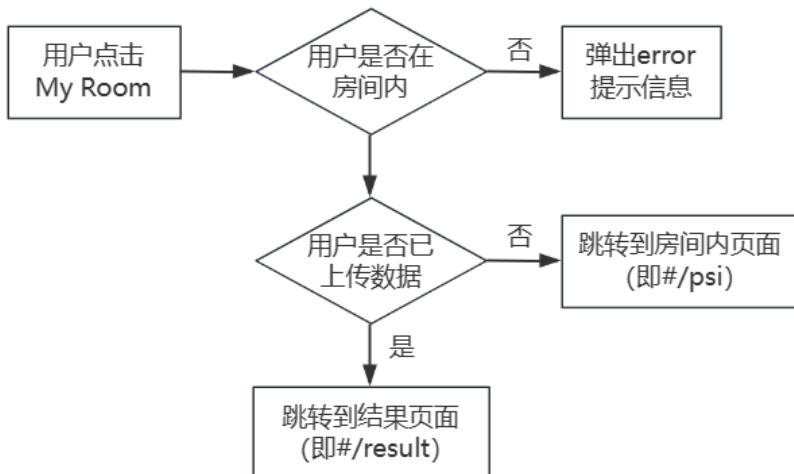


图 5-4 My Room 跳转逻辑

Tools: 点击后跳转到加密工具站。

Help: 点击后，如果用户当前在房间或房间列表页面，由于这些页面的功能相对复杂，系统启动漫游式引导组件，分步骤说明当前页面的操作方式；如果用户当前在其他页面，系统弹出对话框，简要说明当前页面的使用方法，如图 5-5 展示了结果展示页面的对话框帮助。

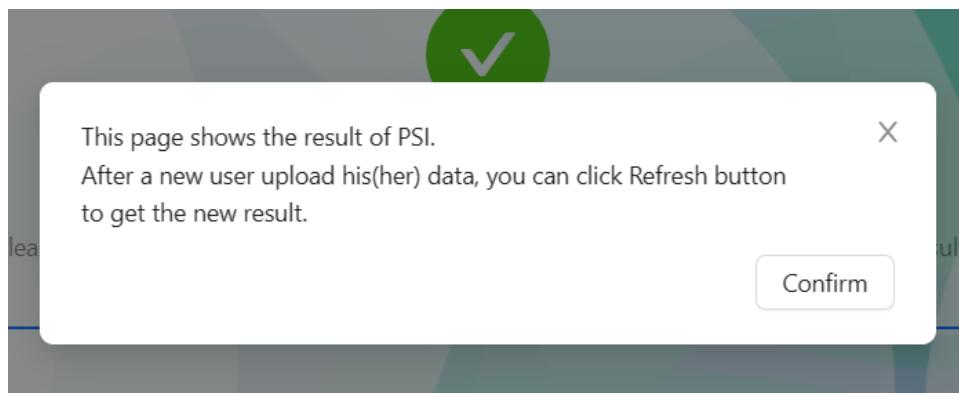


图 5-5 对话框帮助

Logout: 用户将鼠标悬停在自己的用户名上时，会弹出 Logout 子选项。点击 Logout 按钮，系统会自动清除与在线状态相关的 Session（包括前端的 name 属性和服务端的 Session），将用户标记为离线，然后跳转到登录页面。长轮询也会自动停止。

漫游式引导：由于页面内按钮较多，初次使用的用户可能不熟悉使用方法，而大段的文字说明为了清晰、直观地帮助用户上手使用，系统设计了漫游式引导功能。

漫游式引导基于 Ant Design 的 Tour 组件实现，通过 React 的 useRef 钩子，绑定到每个组件真实 DOM 节点的位置，用户点击 Next 和 Previous 按钮可以按步骤学习使用方法，效果展示如图 5-6 所示。

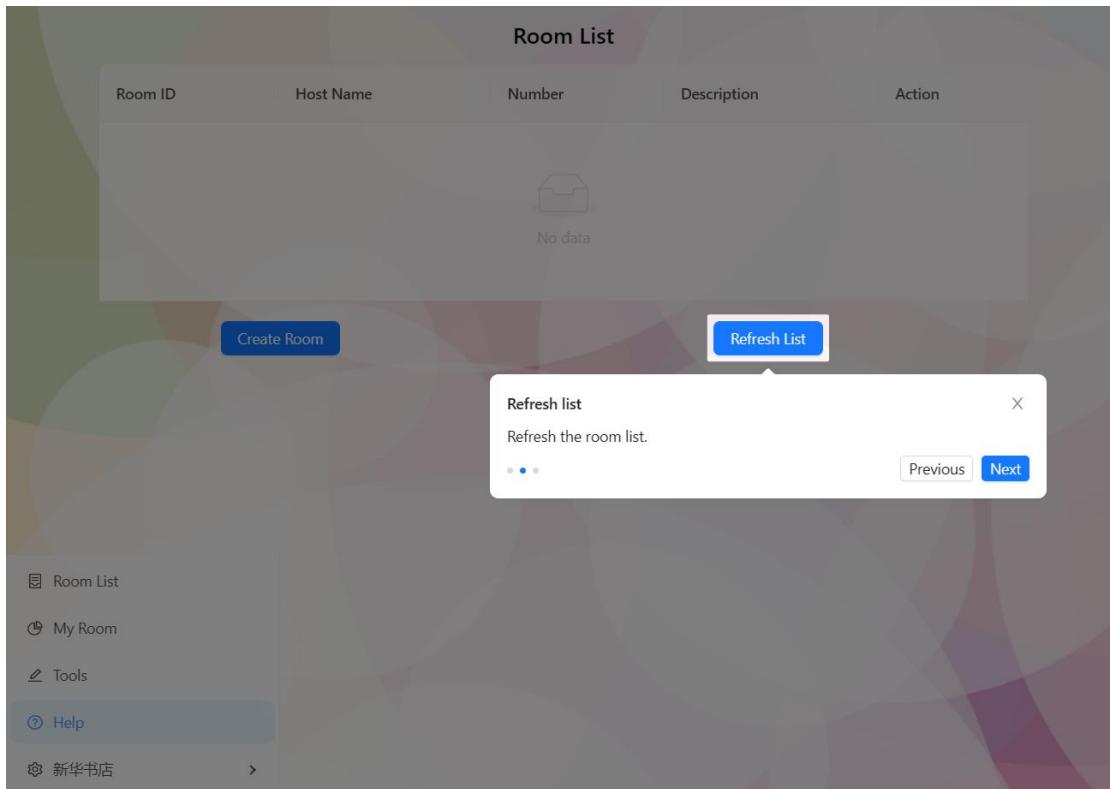


图 5-6 漫游式引导

由于系统开发时，没有使用 Redux 技术，而触发漫游式引导的 Help 按钮位于与路由系统并列的一个全局组件中（该全局组件负责前端访问控制、长轮询、导航栏的实现），而每个页面的引导方法又必须写在页面内，否则无法绑定 ref，因此需要解决跨组件通信问题。

为此，系统中专门设计了 generateContext 方法，将具体的上下文实现和上下文功能的调用解耦。该方法包括 call 和 setImpl 两个函数。call 函数用于调用 setImpl 中设定的实现函数，而 setImpl 用于绑定具体的实现函数，调用方法向上传递，绑定方法向下绑定，从而在系统中实现了对于不同层次、不同组件中函数的调用。

通过参数传递的方式，setImpl 传递给路由系统的子组件，绑定到具体页面的具体实现方法；call 传递给全局组件中的导航栏组件，在用户点击 Help 选项时自动调用 call 方法，进而调用子组件中的 impl 方法，从而实现跨组件通信（实际上

是表兄弟通信，具体的功能组件与导航栏中漫游式引导组件的最近公共祖先是 Router 组件)。

```

function generateContext(){

    let implemented = false;

    let impl = undefined;

    return [

        function call(){

            if(!implemented){

                throw new Error("Context not implemented");

            }

            impl()

        },

        function setImpl(newImpl){

            impl = newImpl;

            implemented = true;

        }

    ]

}

```

5.2.2 登录/注册模块详细设计

为了实现数据上传、存储，以及使用不同密钥加密密文进而求交，系统需要对每个用户加以区分，因此注册和登录是必需的。

1. 登录

用户需要在输入框中分别输入用户名和密码（密码显示为*号），点击提交后，前端自动检查输入是否为空，若有输入为空则弹出信息提示用户；检查不为空后，系统将用户名和 SHA256 加密后的密码发送给服务端，查找 MySQL 数据库中是否有符合要求的记录。

登录页面的前端结构如下：

```
<Row>
    <Col>
        { /* website title */
    </Col>
</Row>
<Row>
    <Col>
        <Input /> { /* input username */ }
    </Col>
</Row>
<Row>
    <Col>
        <Input.Password /> { /* input password */ }
    </Col>
</Row>
<Row>
    <Col>
        <Button /> { /* submit */ }
    </Col>
    <Col>
        <Link /> { /* click to register page */ }
    </Col>
</Row>
```

服务端接收用户名、密码和服务请求后，首先在 Session 中查看该用户名对应的用户是否已经登录，不可重复登录系统；如果未登录，且查找到相符的记录，用户可以成功登录，前端和服务端中的 Session 更新该用户的登录状态，并跳转到房间列表界面；如果没有查找到相符的记录，根据查找失败的原因，弹出错误提示，以指导用户成功登录。

2. 注册

如果用户是首次使用该系统，需要先注册再登录。

注册页面的前端结构与登录类似，由 4 行组成，分别为：用户名、邮箱、密码、再次输入密码，这四个输入框。

除了对用户名和密码的长度、格式，以及对邮箱格式的检查外，注册页面还使用 React 提供的 `useEffect` 钩子实现了对两次输入密码是否一致的实时检测，且会对用户弹出警告信息。

用户输入所有信息并点击 `register` 按钮，前端检查合法后发送注册请求到服务端。服务端首先检查用户名是否已经存在。如果存在，无法以此用户名注册；如果不存在，可以成功注册，服务端将数据存入 MySQL 数据库中，前端返回注册成功，并跳转到登录界面。

3. 访问控制

为了保证系统的安全性，在用户登录后，前端和服务端的 Session 会保存用户的部分数据，以进行访问控制。对 Session 和访问控制的详细介绍见 5.3.3 节。

5.2.3 加密房间模块详细设计

为了保证求交运算的安全和私密性，方便用户理解和使用，系统的隐私集合求交计算以房间为单位，用户需要先进入一个房间，才能正式使用系统的隐私集合求交功能。

在房间列表中，用户可以浏览已创建的房间及其基本信息，包括房间 ID、房主用户名、当前房间内人数、房间描述等。

若用户要创建新房间，点击 `Create Room` 按钮，会弹出创建房间对话框，用户需填写房间描述、选择房间成员能否查看求交结果，完成后点击

若用户有想要加入的房间，点击 `join` 即可发送加入申请，房主处理后，会即时弹框提示申请结果。如果加入成功，会自动跳转到房间内。

成员列表抽屉：在房间内的上传数据页面，为了减少按钮数量，以免用户误操作，房间成员列表以抽屉+自动弹出的形式展示。抽屉使用了 Ant Design 的 `Drawer` 组件，通过单独编写的 `usePopup` 钩子监听用户鼠标位置，

当用户鼠标移动到页面右侧（右 18%范围内）时，系统自动弹出抽屉，鼠标移出后自动收起。

```

<Row>
    <Col>
        { /* room list title */
    </Col>
</Row>
<Row>
    <Col>
        <Table /> { /* represent room list */
    </Col>
</Row>
<Row>
    <Col>
        { /* click to create a new room, which will show a modal. */
        <CreateRoom />
    </Col>
    <Col>
        { /* click to refresh the room page */
        <RefreshList />
    </Col>
</Row>

```

5.2.4 上传数据模块详细设计

上传数据是系统的关键功能。在房间内页面中，用户需要上传数据、填写或选择密钥，然后点击 Submit 按钮提交数据。随后，数据和密钥保存在浏览器 sessionStorage 中，用户关闭浏览器即可删除；前端自动使用密钥对数据加密，将密文发送到服务端，随后启动长轮询，为新用户提供加密服务。长轮询是系统实现求交功能的重要组件，下面详细介绍该组件。

页面组件结构如下：

```
/* Steps component*/

<Row>
  <Col>
    <TextArea /> { /* input data */ }

  </Col>
</Row>
<Row>
  <Col>
    { /* user can input or click random to generate a key. */ }

    <Space.compact>
      <Input /> <Button />

    </Space.compact>
  </Col>
  <Col>
    { /* click to submit data and key */ }

    <Button />
  </Col>
</Row>
```

长轮询：除求交算法设计外，上传数据部分的难点主要在于：对传统的前端与服务端而言，请求是单向的（只允许前端向服务端发送请求，服务端不能向前端发送请求）。而本系统使用的隐私集合求交算法要求每位用户都为其他所有用户加密一次，为了实现这一操作，服务端必须能够告知前端新用户的加入，并要求其为新用户提供加密服务。

为解决这个问题，通过对 `async` 和 `await` 关键字的学习，系统前端额外设计了 `HeartBeat` 组件来实现长轮询，通过长轮询完成前端加密任务^[18]。

长轮询组件的核心代码如下。当服务端想要结束长轮询时，只需返回一个‘`terminate`’字符串，前端会自动识别并停止轮询：

```
export default function HeartBeat({ url, postData, callback, delay }) {  
    const [term, setTerm] = useState(false)  
  
    const sendPost = useCallback(async () => {  
        await new Promise((resolve) => setTimeout(resolve, delay))  
  
        return axios.post(url, postData)  
            .then(resp => resp.data)  
            .then(data => {  
                callback(data)  
  
                if(data.status === 'terminate') {  
                    console.log("terminate query")  
  
                    throw new Error("terminate query")  
                }  
            })  
            .then(() => {  
                return true  
            })  
            .catch((err) => {  
                return false  
            })  
    }, [url, postData, callback, delay])  
  
    const func = async () => {  
        while(!term && await sendPost()) {}  
    }  
  
    useEffect(() => {  
        func()  
  
        return () => { setTerm(true) }  
    }, [])  
  
    return (<></>)  
}
```

在设计长轮询前，对这一需求的原始解决方案是：在用户上传数据后，手动点击 refresh 按钮获取加密结果；此时，服务端检查是否有新用户，若有则提醒老用户，然后使用老用户的密钥为其加密。

这一方案的主要弊端在于：对于第 n 个进入房间的用户，要想查看求交结果，则必须等待先于他进入房间的所有人都点击一次 refresh 按钮为他加密；而在过程中，如果又有新用户加入了房间，第 n 名用户就需要再进行一轮等待……这一连锁反应大大降低了用户获取交集结果的效率。实际上，加入房间申请也面临着类似的问题（必须有一个获取新用户按钮，房主点击按钮才能看到申请加入的信息，但房主不可能频繁地点击按钮查询，用户加入房间会变得困难），这一现象违背了系统设计的初衷。

然而，前端长轮询变相实现了“服务端向客户端发送请求”，在系统中有多个应用，如：上传数据模块的功能以长轮询为核心，实现了用户无感知的自动处理申请、自动加密，用户只需保持浏览器打开，即可实现加密功能；浏览器关闭后会自动删除数据和密钥，保障隐私数据安全。

通过引入<HeartBeat />组件，系统可以随时启动长轮询，处理其他用户发来的各类申请。房间成员的长轮询如图 5-7 所示。

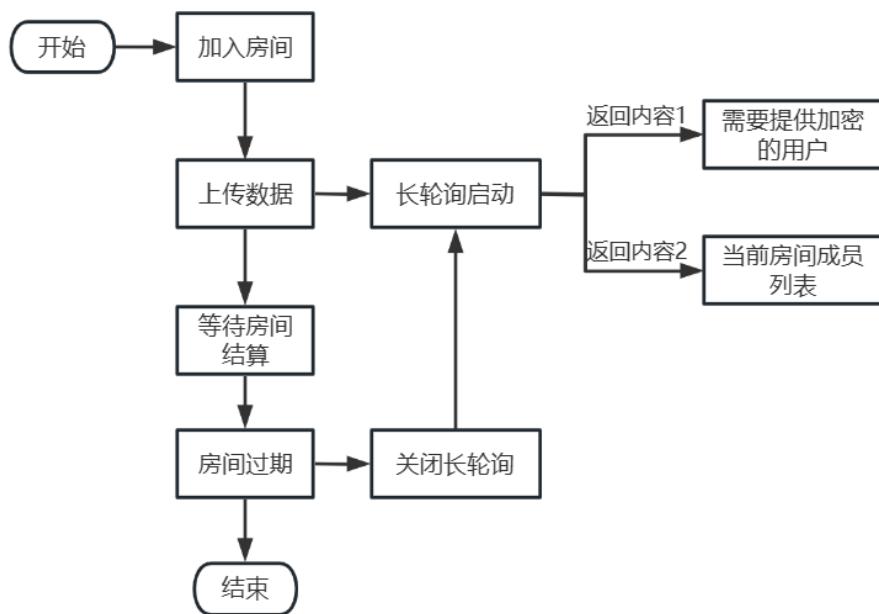


图 5-7 房间成员长轮询

由于房主还需要处理其他成员的加入申请，因此房主与成员的轮询内容不完全相同。房主的长轮询流程如图 5-8 所示。

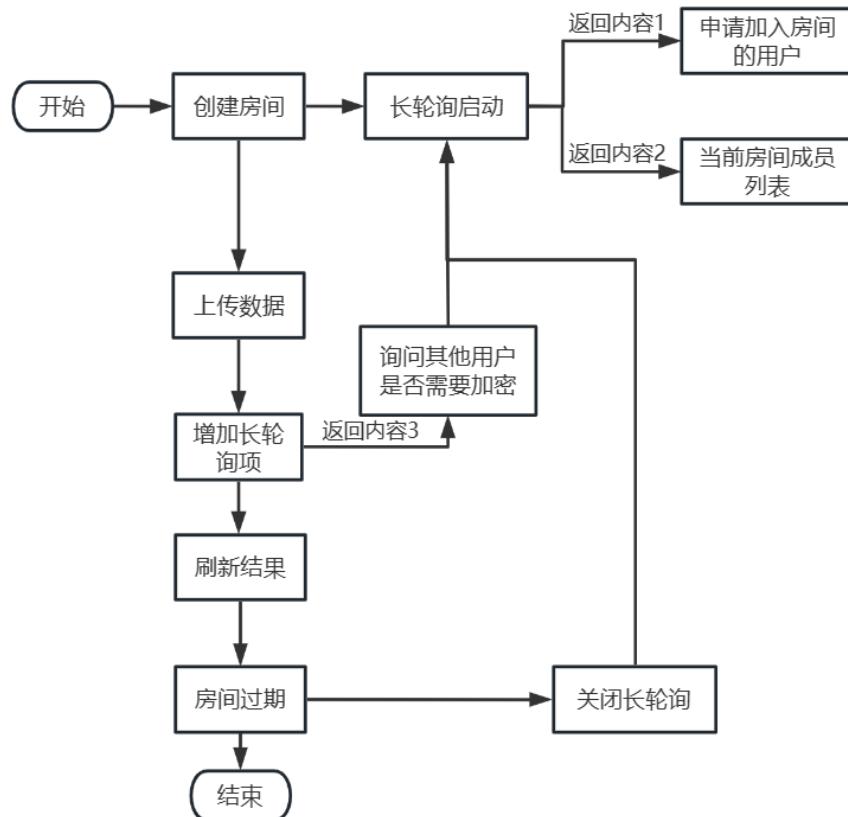


图 5-8 房主长轮询

以房主接收加入房间申请为例，前端在长轮询期间定期向服务端发送请求、接收数据，服务端返回的 json 内容如图 5-9 所示。

图 5-9 长轮询数据

互斥锁：由长轮询中所述的加密流程，容易发现在加密数据时可能出现数据不一致的问题，以如下场景为例：用户 A, B, C 先后进入房间，用户 A 上传数据 E_A 后，用户 B 和 C 几乎同时上传了数据。根据加密要求，B 和 C 都需要对 A 的数据进行加密，于是他们从服务端的 Redis 中取出 E_A ，分别加密为 E_{AB} 、 E_{AC} ，再保存到服务端。此时，B 和 C 都认为自己已经完成了对老用户 A 的加密任务，但实际上，服务端 Redis 中保存的是只经过了 B 和 C 中一个用户加密的密文，无法正确执行求交算法。

为了避免类似的多个用户同时更新导致冲突的情况，系统使用互斥锁保证数据一致性。互斥锁利用了 Redis 中的 `setnx()` 方法实现，其逻辑为：只有在设置的键不存在时，才能成功设置并返回 1，否则返回 0。释放锁时，只需将该键删除即可。

```
# success returns 1, failure returns 0

def lock(mutex):
    logger.debug('lock')
    return conn.setnx(mutex, 1)

def unlock(mutex):
    logger.debug('unlock')
    return conn.delete(mutex)
```

服务端每次对 Redis 中保存的密文进行更新时，都需要先获取互斥锁，更新结束后再释放互斥锁，以保证数据的一致性。

密文更新时，如果获取锁失败，有两种处理方式：一是放弃本次获取锁，等下次轮询时再尝试获取锁。这种方式在老用户为新加入用户加密时采用，因为老用户还会不断进行长轮询，在较短的时间内总可以完成加密任务；二是先睡眠很短的一段随机时间，之后再尝试获取锁。这种方式在用户首次上传数据同时为老用户加密时采用，因为这类加密的相关方法只会在首次加密时被调用，一旦执行失败，整个房间的集合求交都将无法进行。

5.2.5 结果展示模块详细设计

结果展示模块使用了 Ant Design 的 Result 组件和 Steps 组件，求交结果以 Table 的形式展示，用户可点击表格中对应的 download 按钮下载交集内容。由于结果展示页面中用到的组件本身都带有布局，且与其他页面所用的行列式布局接近，因此没有再使用栅格化布局，页面结构如下。

```
<Heart /> { /* Start long polling */ }

<Result /> { /* a series of icons from Ant Design */ }

<tinyStep /> { /* show steps guide */ }

<Table /> { /* show interaction results */ }
```

为新用户提供加密的长轮询组件在该页面启动，原因是只有上传了数据的用户才能跳转到该页面，而跳转到该页面也就意味着用户必须提供为新用户加密的服务。理论上，在所有用户上传数据后的数秒内，用户密文就能完成更新（也就是说，只要当前用户不是第一个上传数据的用户，直接点击 Refresh List 按钮就有很大的概率能够直接获取结果），等待系统刷新后可以获取求交结果。

结果展示模块不进行计算，只负责将服务端传来的交集索引解析为具体交集内容，并展示给用户。

5.3 系统存储设计

对于服务端，用户信息、房间信息等需要长期保存且很少更改的数据存储在 MySQL 中；房间状态、密文等频繁更新的信息保存在 Redis 中，利用 Redis 设置数据存活时间，时间到后自动清除。

两个数据库共同实现服务端的数据存储功能，既提高效率，也保证了系统的安全性和稳定性。

5.3.1 Redis 数据库设计

为了便于存取和使用，本文规定，系统中对 Redis 键的设置均为“关键字+编号”的形式。如房间 1 的房主 ID 为 1，则 Redis 中保存的键为“ROOM1”，值为“1”。

当需要保存用户标识符作为键/值时，统一使用用户 ID 作为标识（可通过 MySQL 数据库查询用户名和用户 ID 的对应关系）。

Redis 数据库中存储的键值如表 5-1 所示。

表 5-1 Redis 数据库

关键字	编号	值	说明
ROOM	房间 ID	房主 ID	保存房间和房主的映射关系
REQUEST	房间 ID	列表[用户 ID]	所有请求加入该房间的用户
APPLY	用户 ID	房间 ID	用户申请加入房间的记录
JOIN	房间 ID	‘exist’	查询到键值存在，就说明该房间可以申请加入
RESOFJOIN	用户 ID	‘accept’或‘reject’	用户申请加入房间的结果
MEMBER	房间 ID	列表[用户 ID]	房间内所有用户
READY	房间 ID	列表[用户 ID]	房间内上传了数据的所有用户
DATA	用户 ID	数据	用户的密文。存入 Redis 时需要使用 listToString 函数转化为字符串，读取时使用 StringToList 函数转化为列表
TASK	用户 ID	列表[用户 ID]	编号中的用户需要为其列表中的所有用户加密密文
BACK	用户 ID	房间 ID	根据用户 ID，反向查找用户所在的房间
LOCK	用户 ID	1	互斥锁

根据系统要求，每个用户上传的数据都是由多条数据（通常是电子邮箱的密文）组成的列表，因此加密后的密文也是列表。

由于 Redis 数据库中无法存储嵌套列表，系统在保存 DATA 时，先调用 ListToString 方法将密文列表转换为字符串，取出 DATA 时再调用 StringToList 方法将字符串转换为列表，便于进行计算。

5.3.2 MySQL 数据库设计

与 Redis 数据库不同，MySQL 数据库中保存的是需要长期储存、较少改动的数据，包括用户信息、房间信息、用户与房间的映射情况 3 张数据表。该数据库中的内容主要有 3 个用途：

1. 用于在用户登录时验证用户身份、辅助进行访问控制。
 2. 用户登录后通过 MySQL 数据库查看房间列表、检查用户是否能够创建或加入房间。
 3. 用户上传数据、刷新求交结果时，服务端通过 MySQL 数据库查询该用户是否有权限查看结果，以此决定返回内容。

图 5-10 是数据库的 E-R 图。

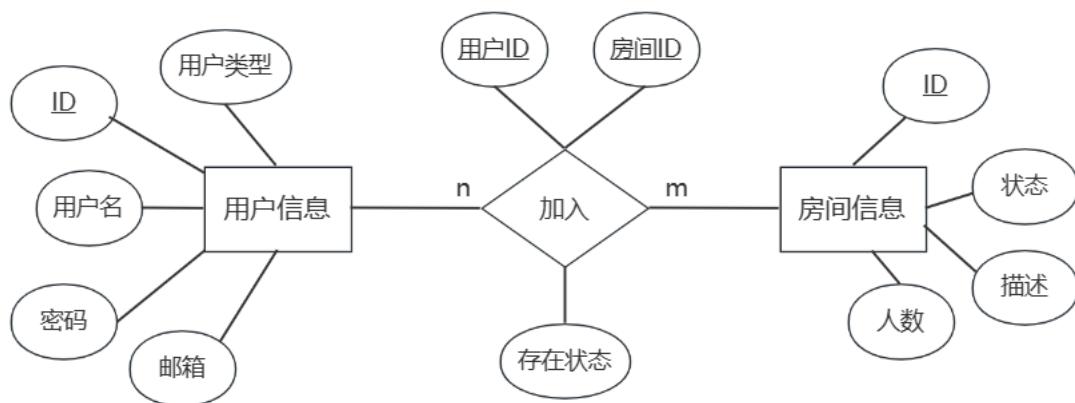


图 5-10 数据库 E-R 图

用户信息数据表 (UserInfo):

表 5-2 用户信息

字段	数据类型	长度	能否为空	说明
ID	INT	4	N	用户编号
Username	VARCHAR	20	N	用户名
Password	VARCHAR	80	N	密码
Email	VARCHAR	40	N	电子邮箱
Type	INT	4	Y	用户类型

其中，密码存储的是使用 SHA256 加密后的密文；用户类型分为用户（0）和管理员（1）两类。未来会对后台管理界面等内容进行完善。

房间信息数据表 (RoomInfo):

表 5-3 房间信息

字段	数据类型	长度	能否为空	说明
<u>RoomID</u>	INT	4	N	房间编号
<u>HostID</u>	INT	4	N	房主用户编号
<u>Number</u>	INT	4	N	当前人数
<u>Description</u>	VARCHAR	100	Y	房间描述
<u>CreateTime</u>	DATETIME	8	N	房间创建时间
<u>Type</u>	INT	4	Y	是否允许成员查看结果
<u>State</u>	INT	4	N	房间当前状态

其中，房间类型 Type 为 0 表示普通房间成员不能查看结果，Type 为 1 表示房间内所有人均可查看结果；房间当前状态 State 为 0 表示房间存在，为 2 表示房间已过期，相关数据已销毁。

用户与房间的映射情况数据表 (User_Room):

表 5-4 用户-房间映射情况

字段	数据类型	长度	能否为空	说明
<u>UserID</u>	INT	4	N	用户编号
<u>RoomID</u>	INT	4	N	所在房间编号
<u>Alive</u>	INT	4	N	房间是否存在

其中，Alive 为 0 表示房间已过期（表中存储的关系也已经过期），Alive 为 1 表示房间存在（但未必允许加入）。

当 Redis 中的数据提示房间过期时，服务端执行更新操作，将 RoomInfo 表中对应的 State 修改为 2，User_Room 表中对应的 Alive 全部修改为 0.

5.3.3 Session

Session 机制是对 cookie 的改进，借助 cookie 实现，用于在前端和服务端之间共享和存储少量数据，经常被用来实现用户身份验证和授权。用户访问网站时，服务端会自动为该用户生成一个唯一的 Session ID，用于标识该用户的身份，并通过 cookie 将 Session ID 发送到前端，由前端保存。此后，该用户每次向

服务端发送请求时，服务端都会读取 Session ID，找到该用户的 Session 数据，实现特定的服务。

在本系统中，Session 主要用于进行访问控制^[19]。这里的访问控制包括两个部分，一是用户未登录不能访问与房间有关的页面；二是用户进入房间后，上传数据前只能进入/psi 页面，上传数据后只能进入/result 页面。

用户登录后，前端将用户名保存在 Session Storage 中作为身份凭证，同时也保存到服务端的 Session 中，其有效期设为 3 天。只有在 3 天期限已到，或用户主动退出登录时，后端才会将其标记为失效。Session 有效期间，用户不能在其他地方重复登录。

在用户登录后、向服务端发送其他操作请求时（如加入房间、上传数据等），都必须传递自己的身份凭证，系统将凭证一起发送给服务端，检查用户是否已经登录，确认用户身份是否合法，再向用户提供服务。Session 内容以服务端为准，即每次涉及访问控制时，都需要先询问服务端，确认用户身份后再提供相应的服务，这一功能通过 Flask 框架提供的@app.before_request 函数实现。

当用户试图跳转到/psi 页面或/result 页面时，系统需要检查该用户是否上传过数据，并为用户重定向到正确的路径，检查流程如下所述。

首先，系统在前端获取 key = sessionStorage.getItem("key")，向服务端发送请求获取 roomID = checkUser(username)。根据这两个值是否存在，可以分为以下 4 种情况来处理：

1. Key 存在，roomID 存在。这说明用户已经加入了房间（且该房间此时仍然存在），且已经上传了数据，应当跳转到/psi 页面。
2. Key 存在，roomID 不存在。这说明用户曾经加入房间并上传过数据，但该房间现在已经过期或被关闭。应当进行过期处理，清除前端保存的除用户名外的所有数据，不进行页面跳转操作。
3. Key 不存在，roomID 存在。这说明用户加入了房间（且该房间此时仍然存在），但还没有上传数据。应当跳转到/psi 页面。
4. Key 不存在，roomID 不存在。这说明用户尚未加入房间，或者加入的房间已经过期或被关闭，不进行页面跳转操作。

注意必须将用户的跳转和系统经过判断后重新确定的跳转区分开，否则系统会不断地刷新（因为每次刷新后都跳转到/psi 或/result，于是上述函数又会被调用、重定向路径），无法正常工作。

用户点击 Logout 按钮时，系统通知服务端删除 Session 数据，同时在前端删除 Session Storage 中的内容，跳转回登录界面。

用户上传数据后，密钥和原始数据被临时保存在 Session Storage 中，前者用于为其他用户加密，后者用于获取具体的求交结果并展示给用户；若用户没有权限查询求交结果，原始数据将被自动清除。隐私数据不会被泄露给服务端，为了保证系统的正常运行，用户需妥善保存密钥和原始数据。

第 6 章 系统展示与测试

6.1 隐私集合求交系统

6.1.1 登录与注册

用户进入系统后，首先进入登录页面。未登录时就可以使用工具站（准确来说，是在用户的登录过期后仍能使用工具站，也可以通过访问路径 <http://localhost:3000/tools>，直接跳转到工具站），但隐私集合求交这一功能必须登录后才可使用。

在用户填写输入框，并点击登录/注册按钮后，前端对数据进行检查，确认没有空数据、不合法数据，检查通过后再传输给服务端进行处理。

登录页面如图 6-1 所示：

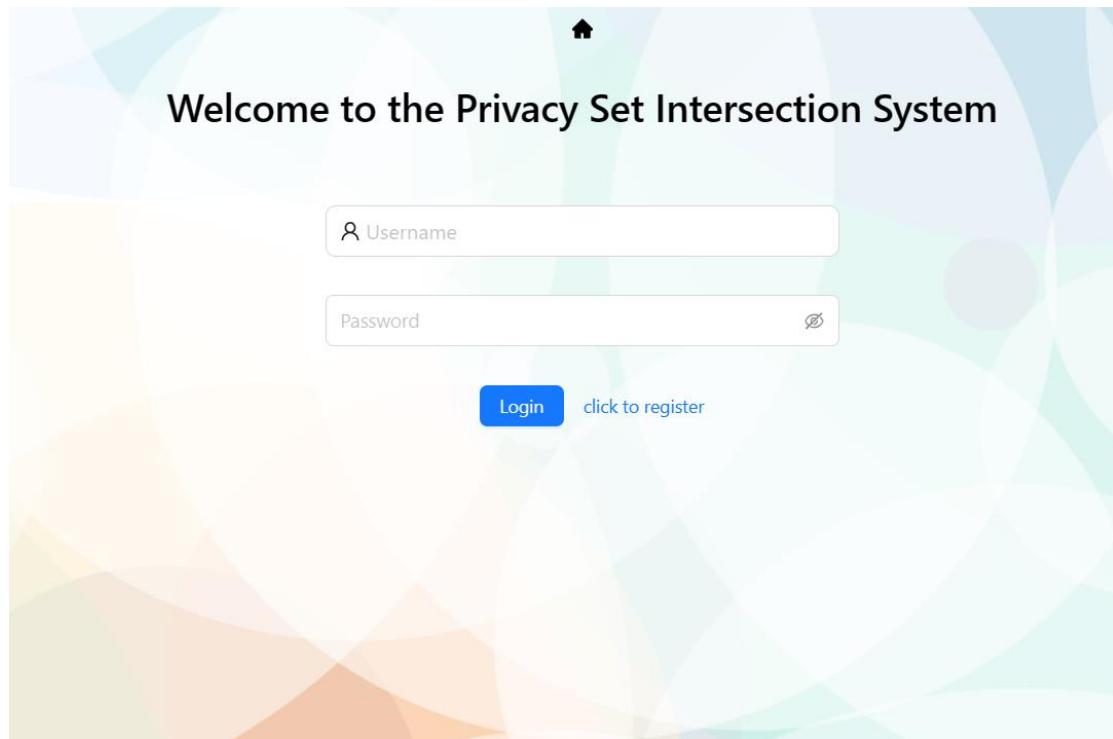


图 6-1 登录界面

为了保证数据安全和一致性，用户不能重复登录系统。在用户登录的用户名和密码都正确时，服务端会检查 Session 中是否已经存在该用户。重复登录会弹出错误提示，不进行其他操作，用户无法登录。

如果用户没有账号，点击“click to register”即可进入注册页面，填写相关信息注册账号。注册页面如图 6-2 所示：

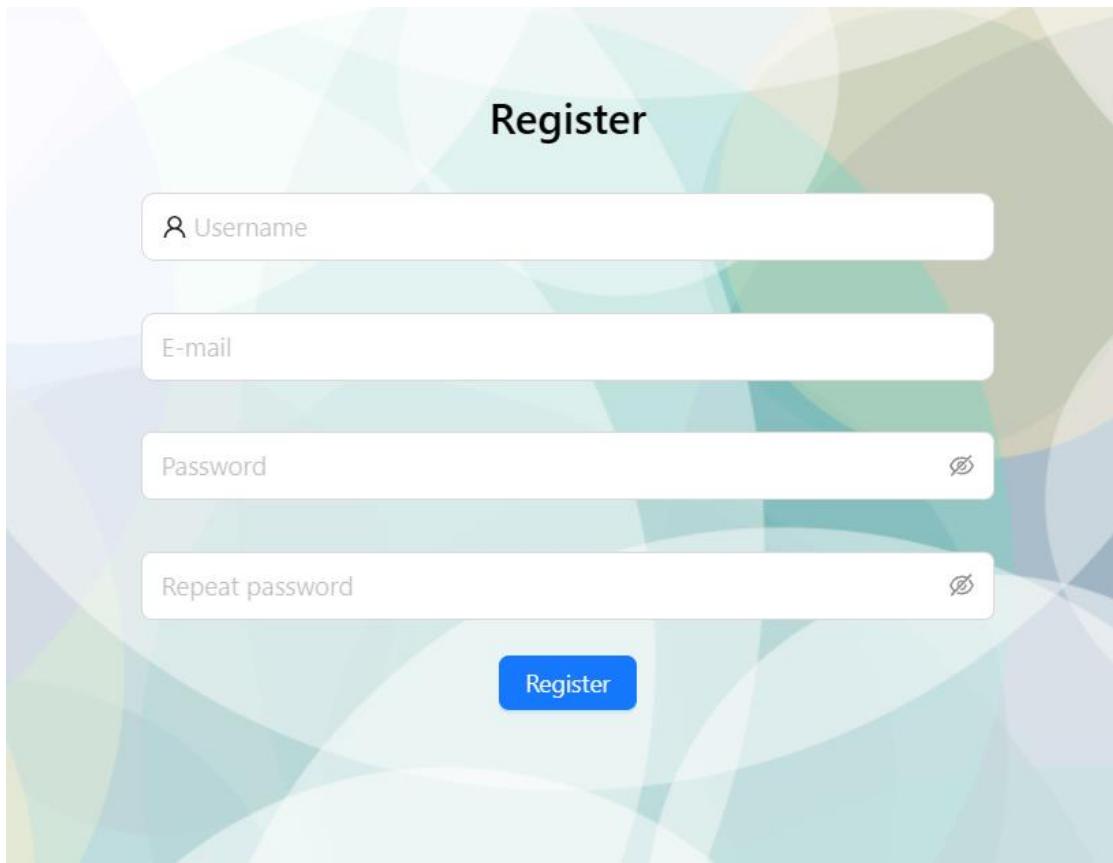


图 6-2 注册页面

如果用户输入的邮箱不符合一般邮箱格式，会弹出错误提示。以@为分隔标志，将邮箱分为①和②两部分，本系统认可的邮箱格式应为：

- ①（以字母开头的数字、字母、下划线组合）
- ②（数字、字母组合）.（至少 2 个字母）

如果两次输入密码不一致，前端会即时出现警告文字，提示用户。

6.1.2 房间列表

用户登录后跳转到房间列表界面。此时由于已经登录，左下角显示导航栏，用户可以点击导航栏中的选项，跳转到对应的页面。

房间列表是用户登录后进入的第一个功能页面。从此页面开始，访问控制已经启动，用户要创建、加入房间都必须保证自己已经登录，否则无法操作，系统会要求用户重新登录。

若用户没有房间，可以选择加入其他房间或点击 Create Room 按钮创建房间；若用户已在房间内，点击任一按钮或导航栏中的 My Room 选项后会自动跳转到房间内。

房间的有效期由 Redis 设置，房间在创建 20 分钟后不允许新用户加入，创建 30 分钟后自动过期。如果用户停留在房间列表页面，长时间没有主动刷新列表，列表中展示的房间可能会过期，此时再点击 join 会弹出错误提示，用户需要刷新列表重新寻找房间。

如果要创建房间，用户可点击 Create Room 按钮，输入房间描述（可以为空）、选择成员权限，点击 OK 按钮创建新房间。

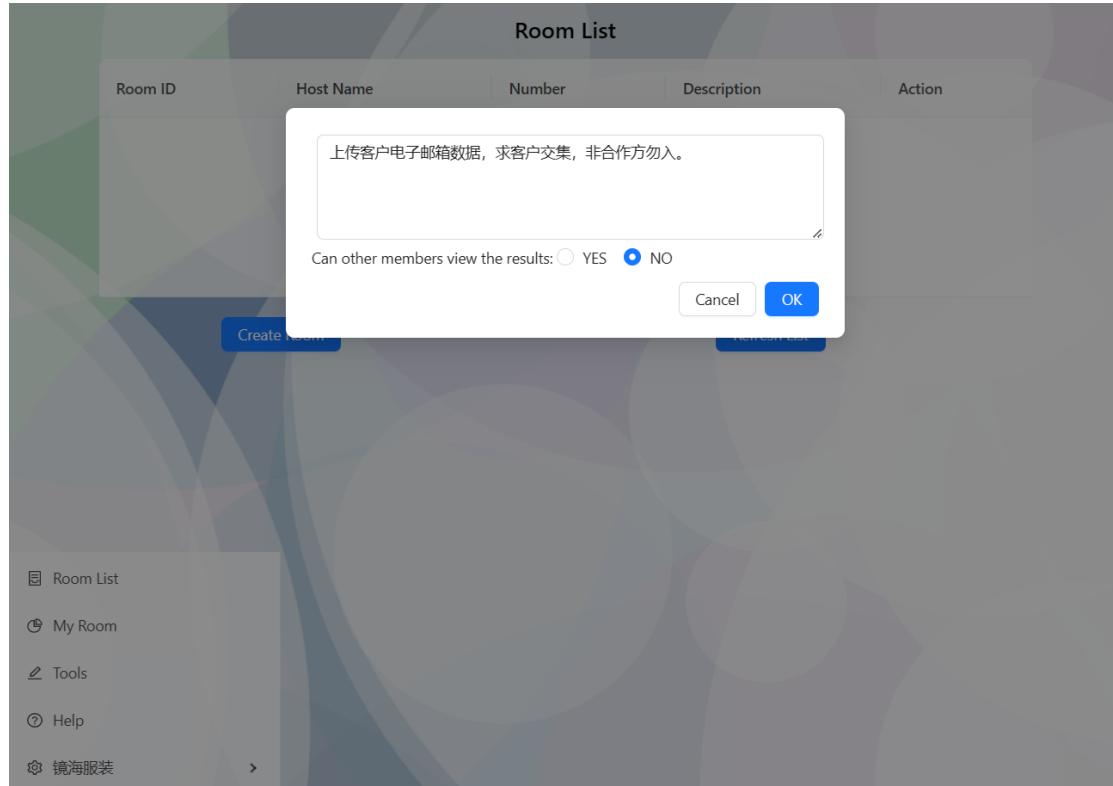


图 6-3 创建房间

房间成功创建后，自动跳转至房间内的上传数据界面，同时提示房间 ID。此时，其他用户登录或点击 Refresh List 按钮刷新房间列表，即可发现该房间。其他用户可以点击 Action 中的 join 申请加入房间，申请加入后，用户前端启动轮询等待加入结果。此时不可申请加入其他房间，也不可再创建其他房间，可以访问工具站。

由于长轮询是在房间列表页面运行的，切换页面会关闭长轮询。如果用户在点击导航栏的对应选项、跳转到其他页面后想查看是否成功加入房间，必须手动点击导航栏中的 My Room 选项。如果成功加入会跳转到房间界面，申请未处理或被拒绝则会提示用户当前不在房间内。

为了用户能够及时接收、处理加入结果，不建议用户在提交加入申请后访问工具站或切换至其他页面。

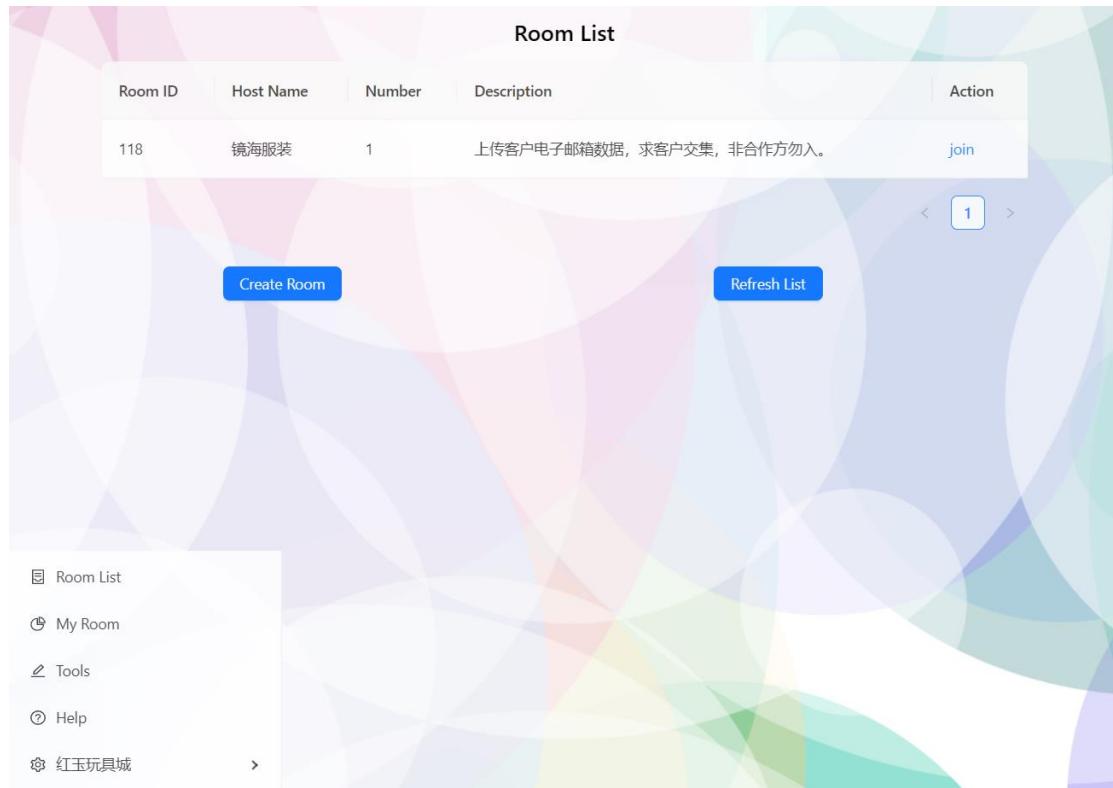


图 6-4 有房间的房间列表

申请发送后，房主通过长轮询即时地收到弹框提示的申请信息。可选择拒绝或通过加入申请。

用户在自己的申请被处理后，会收到通知。若申请被通过，可以点击 Confirm 按钮跳转到房间内；申请被拒绝时可以重新选择创建或加入房间（仍然可以申请加入该房间）。

房主的长轮询组件以及接收申请、弹出对话框的组件与路由组件是平行的，在前端启动后持续运行。因此只要房间存在，无论房主处于哪个页面，都能即时接收并处理他人加入房间的申请。

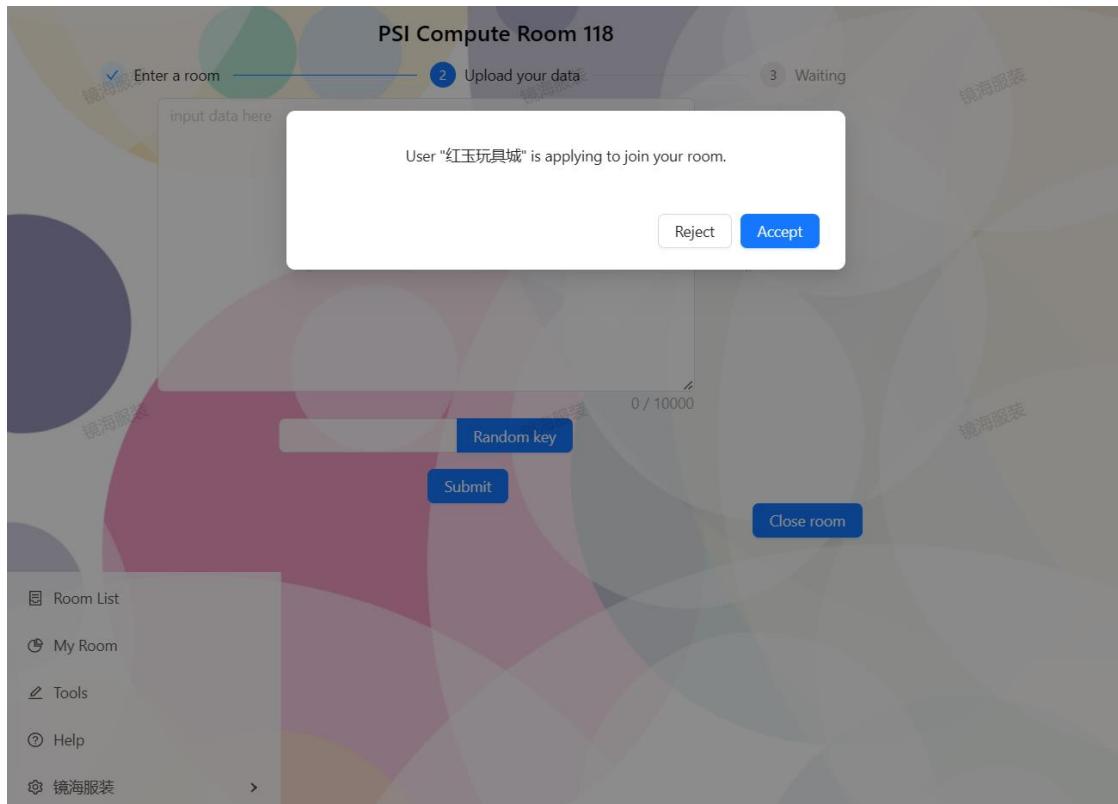


图 6-5 房主接收加入申请

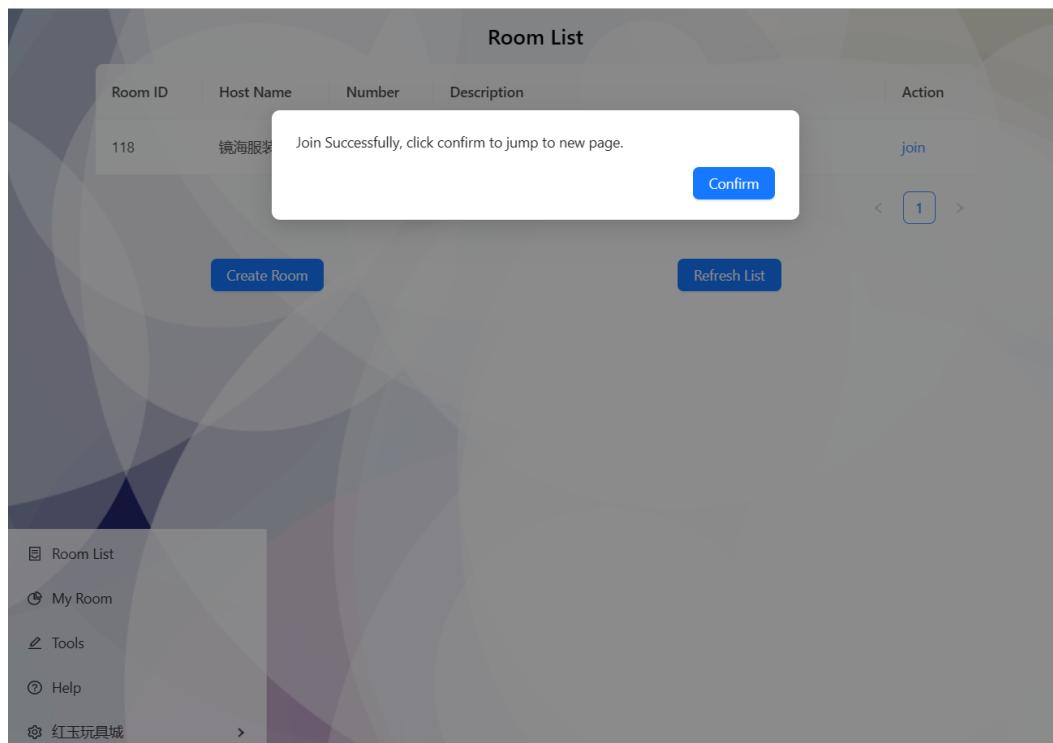


图 6-6 加入申请通过

房主的长轮询启动后，会同时获取申请加入的用户列表和房间成员列表。成员列表以抽屉的形式展示，用户将鼠标移动到页面的右侧时会自动弹出成员列表抽屉，如图 6-7 所示。用户自己的名字会加粗、浮起展示。

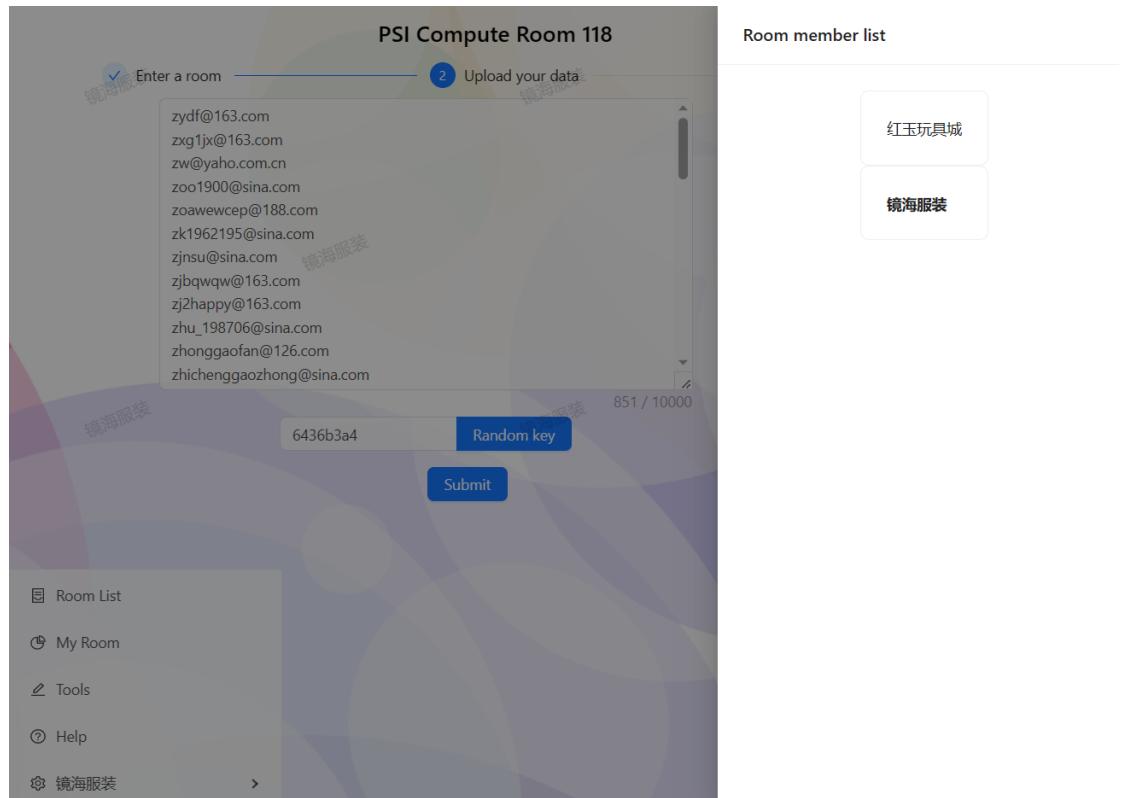


图 6-7 房间成员列表

而申请加入房间、尚未上传数据的用户相当于非正式成员，虽然存在于成员列表中，但没有自己的数据和密钥，尚未启动轮询，此时可以随意退出房间；上传数据就会成为正式成员，无法退出房间（否则加密算法无法正常运行），可以查看成员列表。

6.1.3 数据上传

由于房间的有效期设置，房间创建 30 分钟后服务端会清除 Redis 中存储的相关数据，并在前端发送请求时通知前端删除 sessionStorage 中保存的数据和密钥，以保障用户数据安全。为了避免房间过期导致求交计算失败，用户应在加入房间后尽快上传数据。

上传数据后，用户还需要输入或生成密钥。考虑到安全性，建议用户使用随机密钥，自动生成 64 位密钥用于加密。

如果用户不熟悉操作方式，可以点击导航栏中的 Help 选项，开启对当前页面的漫游式引导，按照步骤完成数据上传。

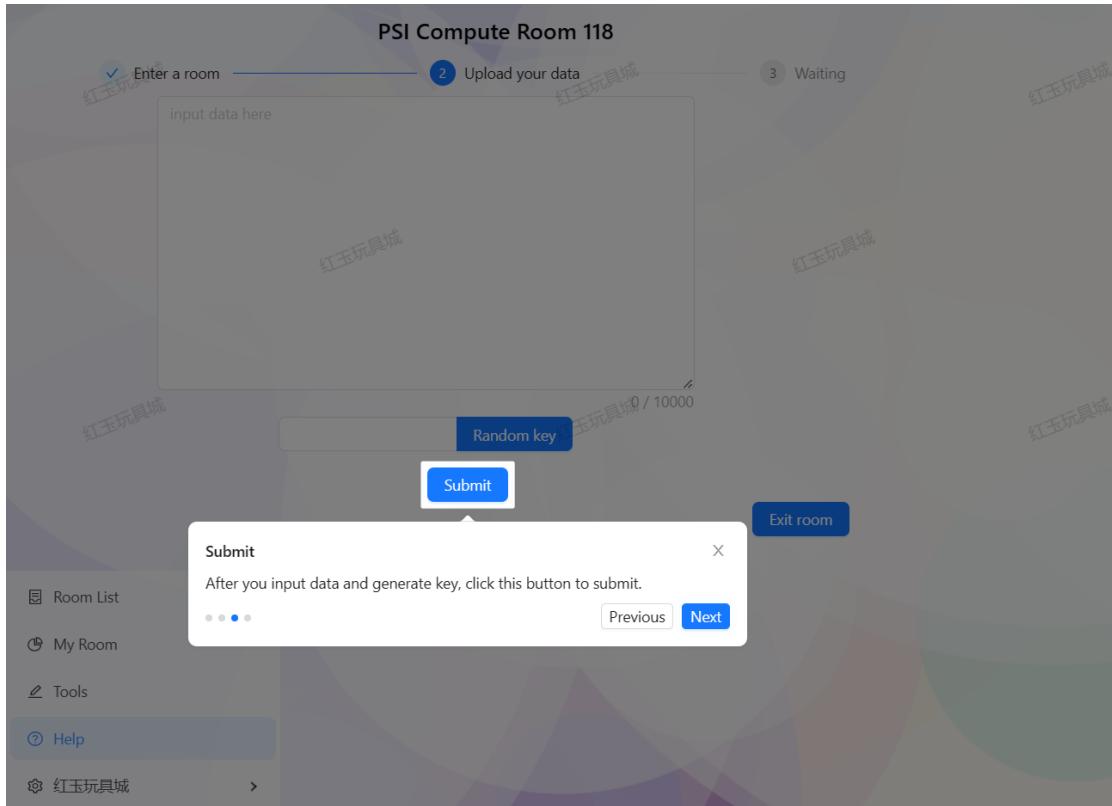


图 6-8 房间内漫游式引导

用户填充数据后，点击 Submit 按钮，系统自动在前端加密数据，将密文传输至后端。成功发送后，跳转至结果页面，用户可刷新查看求交结果。

此时用户密钥保存在 sessionStorage 中，一旦浏览器关闭就会被清除。在用户试图关闭页面时，系统会提示用户停留在该页面，但由于浏览器对安全的要求，提示词无法编辑，仍可能出现误关闭的情况。为避免用户意外关闭浏览器，系统会自动将数据和密钥以 txt 文件的形式保存在用户的浏览器下载文件夹中。但目前系统还无法读取生成的 txt 文件中的内容，载入 txt 文件、重启长轮询的功能将在未来逐步完善。

6.1.4 结果查询

上传数据后，房主或能够查看结果的成员可以点击刷新按钮，立即获取求交结果。点击 download 可以将交集内容保存为 txt 文件。经检查其内容无误，未来会对其格式进行改进，或者保存为 csv 文件，便于用户查看。

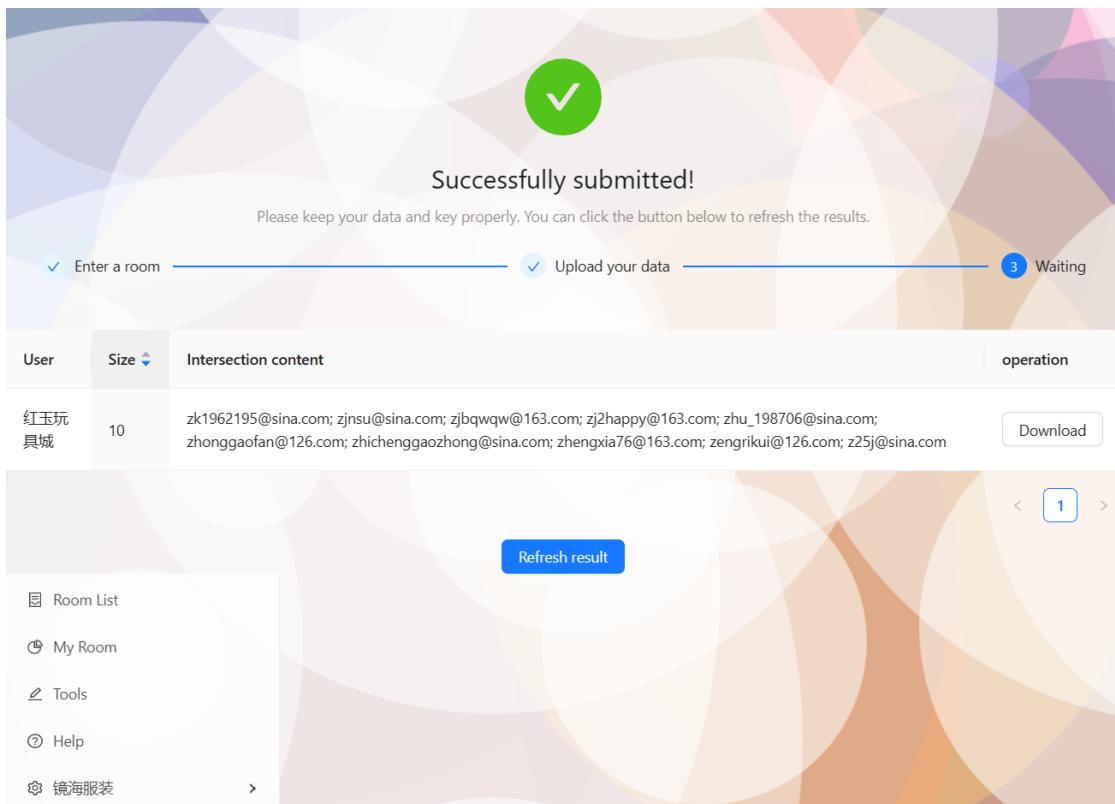


图 6-9 查看求交结果

若房主设置了成员不能查看求交结果，则房间成员无法获取结果列表，只需停留在此页面，在房间过期前保持浏览器持续运行即可。

为了防止用户误操作关闭页面，用户在结果展示界面执行关闭操作时，浏览器会自动弹框阻止关闭。

此时，如果用户仍然关闭了页面，只要没有关闭浏览器，在房间有效期内再次进入系统时仍然能重启长轮询，继续计算流程；若关闭了浏览器，`sessionStorage`会被清空，目前无法重启长轮询，也无法执行求交操作，只能重新创建房间、重新进行求交操作。

图 6-10 展示了无法获取求交结果的用户页面，以及当用户尝试关闭页面时弹出的提示（该弹框是由浏览器控制的，无法修改其中的内容）。

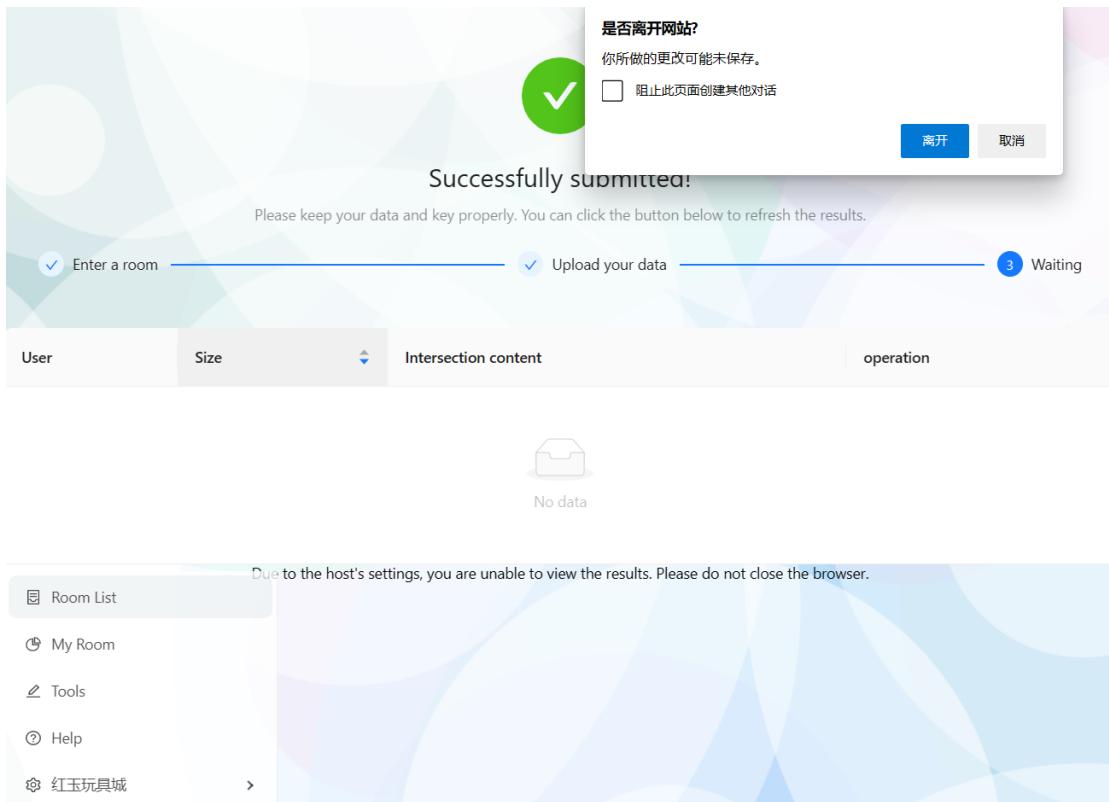


图 6-10 无法查看求交结果

6.2 加密算法工具站展示及测试

6.2.1 工具站概述

在需求分析和系统的初步设计过程中，我发现许多用户都有使用密码散列函数或密码算法对数据进行加密，用于学习、传输的需求，系统附加了一个小型加密算法工具站。

加密工具站无需登录即可使用，但用户未登录时不会显示导航栏，无法跳转到其他页面。未登录时的页面总体效果如图 6-11 所示。

该加密站分为上、下两部分，第一部分提供常见的 SHA1、SHA256 算法加密；第二部分提供 AES 加密算法的加密和解密（系统内置初始向量，要求用户输入或生成密钥）。页面中的算法均使用 Crypto-JS 库实现。

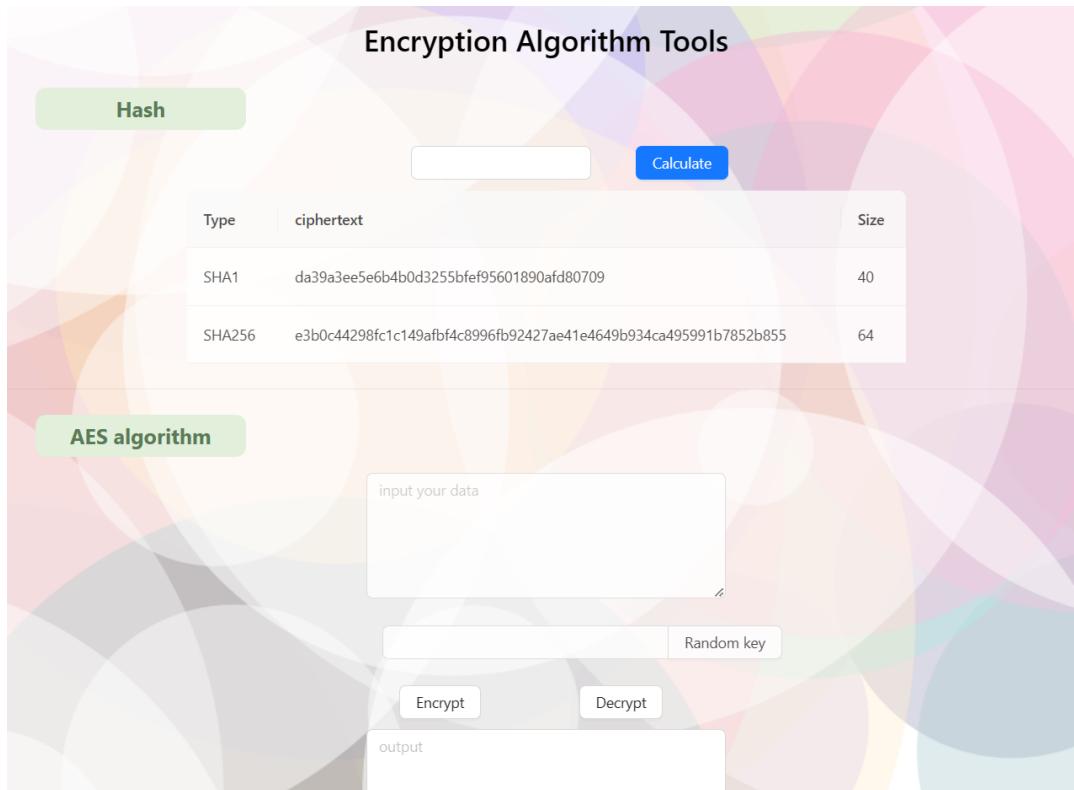


图 6-11 工具站总览

如果用户对使用方法有疑问，可在登录后点击导航栏中的 Help 选项，打开帮助对话框。帮助对话框的内容如图 6-12 所示。

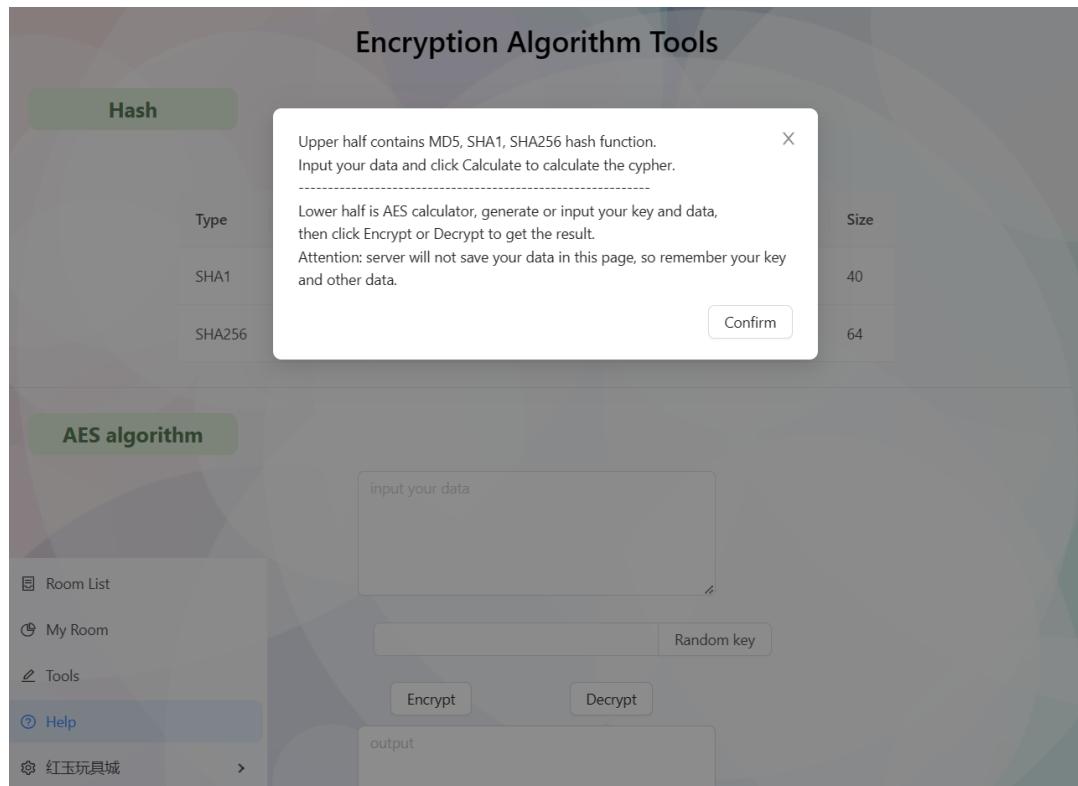


图 6-12 工具站帮助

6.2.1 常用加密部分

用户输入数据并点击 Calculate 按钮后，自动计算并展示出三种密码算法的哈希结果、哈希值长度。

The screenshot shows a user interface for calculating hash values. At the top left is a green button labeled "Hash". In the center is a white input field containing the text "红玉玩具城". To the right of the input field is a blue "Calculate" button. Below the input field, there is a table with two rows. The first row has columns for "Type" (SHA1), "ciphertext" (dc210ad182746729b9740ffe69c38ad9700cdfe2), and "Size" (40). The second row has columns for "Type" (SHA256), "ciphertext" (26fd010336a16e3f7e1fe0a1d09996c5e334dd74e4bad8269c1ef232f00c6bd1), and "Size" (64). The background of the interface features abstract geometric shapes in shades of purple, blue, and teal.

Type	ciphertext	Size
SHA1	dc210ad182746729b9740ffe69c38ad9700cdfe2	40
SHA256	26fd010336a16e3f7e1fe0a1d09996c5e334dd74e4bad8269c1ef232f00c6bd1	64

图 6-13 加密结果

6.2.3 AES 算法部分

AES 算法是一种对称加密算法，被广泛认为是最安全的加密算法之一。该算法需要用户提供密钥，系统随机生成初始向量，对数据进行加密或解密。为了保证数据的安全，密钥长度选择 128 位，即 32 位十六进制数，推荐用户点击 Random Key 按钮自动生成合法密钥，并将密钥以可信的方式保存下来。

下面展示了使用 AES 算法加密和解密数据的示例。输入数据和密钥后，点击 Encrypt 按钮，系统会使用密钥对输入的内容进行加密；点击 Decrypt 按钮则会对输入的内容进行解密。

The screenshot shows a user interface for the AES algorithm. At the top left is a green button labeled "AES algorithm". In the center is a white input field containing the text "红玉玩具城". Below the input field, there is a row of buttons: "Random key" (white background), "Encrypt" (light blue background), and "Decrypt" (light blue background). Further down, there is another input field containing the ciphertext "U2FsdGVkX1/0IEy6PUX67z6Mlz6GOfiQXSVYeVcu2K A=". The background of the interface features abstract geometric shapes in shades of grey, blue, and green.

图 6-14 AES 加密



图 6-15 AES 解密

此处密钥的长度必须为 128 位，即 32 个字符。用户输入不符合长度的密钥时，会弹出错误提示。为了避免麻烦，建议用户点击 Random Key 按钮生成一段密钥，然后将密钥以可靠的方式保存下来。

第7章 总结与展望

7.1 工作总结

本文的主要工作如下：

1. 通过学习密码学知识，研究隐私集合求交算法的常见方法，阅读相关论文，理解了隐私集合求交的思想，并据此构思了适用于小用户的隐私集合求交系统的设计思路。
2. 对系统进行需求分析和总体设计，明确系统架构，分析系统各模块工作流程，对服务端的 Redis 缓存和 MySQL 数据库进行初步设计，为系统的开发做准备。
3. 实现了用户访问控制，完成了注册登录模块、房间管理模块、数据处理模块、结果处理模块四大模块的开发。
4. 本文的重点在于设计并实现了适用于 B/S 架构的多方隐私集合求交算法，保护了用户隐私数据的安全，跳出了 JavaScript 数据类型的限制，计算完毕后能返回具体的交集内容，操作简单，结果可视化展示，计算速度快，符合用户的需要。

系统开发工作完成之后，对系统功能进行了验证与测试，计算结果符合预期，且能够阻止用户的非法操作，可以上线运行。

7.2 展望

目前，系统满足了需求分析与设计阶段对隐私集合求交系统提出的功能性与非功能性需求，但由于开发时间和水平有限，系统在细节方面还有许多有待改进的地方，功能也可以更加完善。主要有以下几个方面：

1. 现在系统的加密算法在前端执行，速度受限于前端，无法充分利用服务端的计算资源。如果借助相关传输协议，将带有密钥的加密函数进行序列化处理，以可执行逻辑流的形式传输给服务端，让服务端在无法了解函数和密钥的具体数值的情况下进行加密计算和求交，能够进一步提高隐私集合求交的速度和安全性^[20]。

2. 可以增加发送邮件的功能。在用户刷新获取求交结果后，点击下载按钮，系统自动将结果生成为一封电子邮件，发送到用户注册时使用的电子邮箱上。
3. 可以增加对用户关闭浏览器后又重连的处理。现在的系统会在用户上传数据后自动将其保存为 txt，未来可以增加从 txt 文件中再次读取密钥明文、重新启动长轮询等中断处理功能。
4. 可以对交集结果以图表的形式进行可视化展示，如以柱状图形式展示当前用户与其他用户的交集数量。在此基础上可以增加个人中心页面，用户可以在此处查看过去的求交计算，以及与不同用户的交集大小，系统对此进行数据分析，为用户提出建议。
5. 可以增加管理员后台界面，允许管理员对非法房间、恶意用户等违规现象进行查处和封禁处理。

在未来的工作中，我会从实用的角度出发，对以上内容进行重点研究和开发，继续完善系统。

参考文献

- [1] Yao A. Protocols for secure computations[C]. Proceedings of the IEEE Symposium on Foundations of Computer Science, 1982: 160-164.
- [2] Chen, H., Laine, K., & Rindal, P. (2017). Fast Private Set Intersection from Homomorphic Encryption. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS'17. doi:10.1145/3133956.3134061
- [3] Cao, F., & Zhang, C. Privacy-Preserving Data Mining: Models and Algorithms. Springer US, 2014.
- [4] 符芳诚,侯忱,程勇等.隐私计算关键技术与创新[J].信息通信技术与政策,2021,47(06):27-37.
- [5] Freedman M, Nissim K, Pinkas B. Efficient private matching and set intersection[C/OL]: Proc of the 23rd Int Conf on the Theory and Applications of Cryptographic Techniques. Berlin: Springer,2004
- [6] 朱之伟,张锡安.浅析金融领域的隐私计算应用[J].金融发展研究,2023,No.495(03):90-92.DOI:10.19647/j.cnki.37-1462/f.2023.03.013.
- [7] 魏立斐,刘纪海,张蕾,王勤,贺崇德.面向隐私保护的集合交集计算综述[J].计算机研究与发展,2022,59(08):1782-1799.
- [8] 徐秋亮,蒋瀚,赵圣楠.安全多方计算关键技术：茫然传输协议[J].山东大学学报(理学版),2021,56(10):61-71.
- [9] Freedman MJ, Ishai Y, Pinkas B, et al. Keyword search and oblivious pseudorandom functions. In: Theory of Cryptography Conference. Springer Berlin Heidelberg, 2005: 303-324.
- [10] 宋祥福. 云环境下实用安全计算与隐私保护关键技术研究[D]. 山东大学, 2021. DOI:10.27272/d.cnki.gshdu.2021.000547.
- [11] 张静,田贺,熊坤,汤永利,杨丽.基于云服务器的公平多方隐私集合交集协议[J/OL].计算机应用:1-6[2023-05-08]
- [12] Vladimir Novick, React Native - Building Mobile Apps with JavaScript, Packt Publishing, 2017
- [13] Diffie W, Hellman M E. New directions in cryptography[J]. IEEE transactions on Information Theory, 1976, 22(6): 644-654.
- [14] J. Yan, R. K. Thomas, E. Stefanov, and A. Uzun, "Protecting accounts from credential stuffing with password breach alerting," in Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, Aug. 2018, pp. 1009–1025.
- [15] Douglas R. Stinson 著, 冯登国 等译, 密码学原理与实践 (第 3 版), 电子工业出版社, 2016.1
- [16] Wu Q, Yu M, Song X. An Efficient and Secure Private Set Intersection Algorithm Based on Homomorphic Encryption and Diffie-Hellman Key Exchange. In: Proceedings of the 9th International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage. Lanzhou, China: IEEE, 2019:278-288.
- [17] 陈恭亮 著, 信息安全数学基础 (第 2 版), 清华大学出版社, 2014.11
- [18] Avcu S, Kaynar O. Design and implementation of a web-based online examination system[J]. Journal of Information Technology and Software Engineering, 2018, 8(1):1-1.
- [19] Fu X, Wang Q, Sun Y, Qin Y. Research of Session-Based Access Control Model in Cloud Computing[J]. Journal of Engineering and Applied Sciences, 2017, 12(20): 5211-5216.

- [20] S. Subashini, V.Kavitha. A survey on security issues in service delivery models of cloud computing. Journal of Network and Computer Applications 34(2011)1-11.

致 谢

为了开发隐私集合求交系统、完成这篇毕业论文，我学习了有关 React 和 Flask 的许多知识，研究了多种密码算法，对前端开发和密码学都有了更深入的认识，也丰富了我的项目经验，令我受益匪浅。在我的知识学习和毕业设计工作中，有许多人都为我提供了指导和帮助，让我少走了弯路、提高了工作效率，在此，我想要对他们表达诚挚的感谢。

首先，我要感谢我的指导教师孔凡玉老师，孔凡玉老师对我的密码学学习和论文的撰写都提供了细致而专业的指导，为我的开发工作指明了方向，并对我的需求分析、系统设计等方方面面提供了重要的指导和意见，耐心解答各类问题，帮助我顺利地完成了毕业设计。

其次，我要感谢与我同组的同学们，在密码学方面与他们的交流和对相似问题的思考，促使我们共同进步，也为我的算法设计提供了思路，对我成功设计并完成隐私集合求交系统的开发起到了重要作用。

最后，我要感谢我的家人和朋友们。他们在生活和学习中一直鼓励着我，在生活上和毕业设计的工作中都为我提供了许多帮助，无论何时都坚定地支持我，让我能够充满动力和信心地完成这篇毕业设计。

用于存储系统的可更新不经意密钥管理

摘要

本文介绍不经意密钥管理系统（KMS），作为对能够支撑大规模数据存储与部署中密钥管理的传统基于包装的KMS算法的更安全的替代。新系统基于不经意伪随机函数（OPRF），隐藏了系统中的密钥和对象标识符，提供了无条件安全的密钥传输、密钥可验证性、减少；存储空间，等等。此外，我们展示了如何在分布式阈值实现中提供这些功能，以增强对服务器泄露攻击的保护。

我们为该系统扩展了支持密钥更新（密钥轮换）的可更新加密功能，在 KMS 服务器定期更改OPRF密钥时，一个有效的更新过程允许 KMS 服务的客户端通过非交互方式更新所有加密数据，以使其只能由新密钥解密。这增强了前向和后向安全性，即分对 KMS 持有的客户端 OPRF 密钥的未来和历史的安全性。并且，与传统KMS相比，我们的解决方案支持不与KMS进行数据加密交互的公钥加密（只有客户端解密需要此类通信）。

我们的解决方案以最近关于可更新加密的相关工作为基础，但对远程 KMS 环境的功能显著增强功。除了关键的安全改进之外，我们的设计非常高效，已经准备投入实践。我们将报告实验性的实现和性能。

CCS目录

Security and privacy → Key management

安全和隐私-密钥管理

关键词

key management, updatable encryption, Oblivious PRF, OPRF

密钥管理，可更新加密，不经意密钥传输OPRF

引言

不断扩展的云存储基础设施是现代计算的一大支柱。然而，提供用于保护存储数据的密钥的密钥管理系统（KMS）几十年来并没有根本性变化。这个设置包括了三个独立的部分：客户端C，以加密形式保存用户数据的远程存储服务器StS（例如云服务器），和用于存储加密密钥的密钥管理服务器KmS。客户端每次需要加密或解密数据时都使用 KmS 的服务。这个想法是基于 KmS 可以更好地保密密钥，而 StS 可以更好、更可靠地存储大量数据。因此，KmS 负责保护机密，而 StS 负责保护可用性。

该系统在实践中的典型部署（包括大型基于云的操作，如AWS[2]，Microsoft[39]，IBM[27]，Google[24]）使用了传统的包装-解封方案来管理数据加密密钥（*dek*），如图1所示。当客户端C需要加密数据对象时，它选择一个对称密钥*dek*对其进行加密，然后将*dek*发送给密钥管理系统KmS，后者根据存储在KmS中的客户端特定主密钥*k_c*对*dek*进行包装（即加密），并将结果（包装包）返回给C。最后，C存储包装包，数据会在存储服务器StS中使用*dek*进行加密。当C需要检索一个对象时，它从StS中得到相应的密文，将包装包发送给KmS，KmS使用*k_c*对其进行解封（解密），并将解密得到的*dek*发回给C，C用其解密密文。

Parties: key management server KmS, storage server StS, client C (= data owner).

Functions: Symmetric authenticated encryption scheme Enc; wrapping functions Wrap, Unwrap (used to encrypt/decrypt data encryption keys).

Keys: KmS stores a client-specific wrapping key k_c for each client.

Encryption of object (ObjId, Obj) by client C :

- (1) C chooses random Enc key dek (data encryption key);
- (2) C sends (ObjId, dek) to KmS;
- (3) KmS returns (ObjId, wrap = $\text{Wrap}_{k_c}(\text{dek})$)
(Note: KmS authenticates C before using k_c);
- (4) C sends (ObjId, wrap, $\text{Enc}_{\text{dek}}(\text{Obj})$) to StS for storage.

Decryption of object ObjId by C :

- (1) C retrieves (ObjId, wrap, $\text{Enc}_{\text{dek}}(\text{Obj})$) from StS;
- (2) C sends (ObjId, wrap) to KmS;
- (3) KmS returns (ObjId, dek = $\text{Unwrap}_{k_c}(\text{wrap})$)
- (4) C decrypts $\text{Enc}_{\text{dek}}(\text{Obj})$ using dek.

Figure 1: Traditional Wrapping-based Key Management

这种密钥包装机制虽然有效，并且得到了广泛的部署，但存在重大的潜在漏洞。首先，加密密钥 dek 是以明文方式暴露给 KmS；第二，因为上一条的原因， dek 的安全性乃至所有加密数据的安全性依赖于客户端和 KmS 之间的通道安全性。这种通常由 TLS 实现的通道容易受到大量攻击，包括实现配置错误、认证攻击、中间人攻击等；第三，即使在正常运行中，密钥 dek 也对解密 TLS 流量的任何中间件和端点可见。此外，KmS 可以通过包装值追踪被加密/解密的对象。另一个缺点是通过 KmS 轮换客户端密钥的成本：更改 k_c 值为一个新的 k'_c 需要客户端（或 StS）将所有的包装都发送到 KmS，在 k_c 下展开，并在 k'_c 下重新包装。这不仅是一个性能问题，也是一个安全问题（需要很长的时间才能更新完所有包，并安全地删除 k_c ）。

不经意 KMS。 我们的第一项成果是对密钥管理系统的一个简单改进，基于不经意伪随机函数 OPRF [22, 29, 41]，解决了上述漏洞并提供了传统系统中不存在的附加功能。OPRF 是持有 PRF 密钥的服务器和持有输入的客户端之间的交互式方案，在交互结束时，客户端在其输入上了解到 PRF 的输出，而服务器没有得到任何信息（无论是输入值还是函数的输出值）。OPRF 已经得到了大量的应用，且实现非常高效，例如基于 DH 问题的规则椭圆曲线组 [16, 21, 26, 40, 47]（见图 3）。

Components: G : group of prime order q ; H, H' : hash functions with ranges $\{0, 1\}^\ell$ and G , respectively, where ℓ is a security parameter.

PRF F_k Definition: For key $k \leftarrow_R \mathbb{Z}_q$ and $x \in \{0, 1\}^*$, define

$$F_k(x) = H(x, (H'(x))^k)$$

Oblivious F_k Evaluation between client C and server S

- (1) On input x , C picks $r \leftarrow_R \mathbb{Z}_q$; sends $a = (H'(x))^r$ to S .
- (2) S checks that the received a is in group G and if so it responds with $b = a^k$.
- (3) C outputs $F_k(x) = H(x, b^{1/r})$.

Figure 3: DH-based OPRF function dh-op [29]

在我们的不经意密钥管理系统 (OKMS) (见图2) 中, 一个客户端C需要数据加密密钥 dek 来加密一个数据对象, 它通过OPRF协议与OKMS服务器进行交互。C的输入是数据对象的标识符, 而服务器的输入是一个OPRF密钥 (通常每个客户端都是唯一的, 记为 k_c), 并且C使用OPRF的输出作为 dek (注1: 或者可以把OPRF的输出用作加密密钥 kek , 来加密本地的密钥 dek) .这样一来, OKMS服务器并不知道 dek (甚至不知道对象标识符), 因而系统不依赖于外部安全通道 (如TLS) 来传递 dek ; 相反, dek 得到了OPRF的安全属性保护 (注2: TLS连接可以传输辅助信息或客户端凭证, 但不能被用于传输数据加密密钥)。

Functions: OPRF F and symmetric authenticated encryption scheme Enc.

OPRF Keys: KmS stores a client-specific OPRF key k_c for each client.

Encryption of object Obj by client C: C runs OPRF protocol with KmS where C inputs object identifier ObjId and KmS inputs key k_c . C sets $dek = F_{k_c}(\text{ObjId})$ and stores the pair $(\text{ObjId}, \text{Enc}_{dek}(\text{Obj}))$ at storage server StS.

Decryption of encrypted object ObjId by client C:

As in the encryption case, C interacts with KmS to compute $dek = F_{k_c}(\text{ObjId})$ and decrypts Obj using dek.

Verification of correct computation of dek: Use a verifiable OPRF [28].

Figure 2: Oblivious KMS (OKMS)

这解决了传统KMS系统的两个主要漏洞：一是 dek 对服务端的可见性，二是密钥在客户端和服务端间传输的潜在暴露风险。此外，通过使用最有效的基于DH的OPRF实现，对这些威胁的保护是无条件的。即使是计算性无界服务器（知道OPRF密钥），或者网络窃听器，都不能了解到关于 dek 或输入到OPRF的对象标识符。注意，在OKMS中，敌手解密密文的唯一方法是冒充合法伙伴客户，或者了解到OPRF密钥 k_c 和响应的对象ID值。与之相对，在传统系统中，即便Kms密钥被保护得很好（例如保存在硬件模块内部），数据加密密钥 dek 也可能在传输至保护区外时遭到攻击。

OPRF方法提供了额外的属性，这进一步增强了安全性，且全方位地超越了传统方法。首先，它提供了可验证性，即Kms能够向客户C证明返回的 dek 确实是根据客户端提供的对象标识符计算出的OPRF函数值。这可以防止在返回 dek 时出现错误导致的数据丢失（由于计算错误或对抗行为）；实际上，使用不正确或不可恢复的密钥加密数据可能会导致无法挽回的数据损失。其次，OKMS也使用基于DH的OPRF，这意味着它可以作为多服务器阈值方案进行分发，一旦损坏导致服务器数量低于定义的阈值，OPRF密钥就会得到保护。最后，我们所描述的系统还具备可更新性，即通过Kms对客户端的主密钥 k_c 进行周期性密钥轮换，使用高效（非交互式）的步骤来更新密文，使其可以由新密钥解密，而不能被以前的密钥解密。此过程不会危害数据的机密性，因此可以由StS执行。该系统的设计是我们工作的主要技术贡献，接下来将对此进行讨论。

可更新的不经意KMS。传统的密钥管理系统在上文中已经讨论过（可参考图1），它需要服务器Kms定期更新客户端的密钥 k_c ，这种更新被称为密钥轮换，需要在公开 k_c 时限制数据的公开。对于传统包装系统，用将 k_c 更改为新的 k'_c 时需要对客户端的所有密文进行解封和重新包装，并在存储服务器和KMS服务器之间传输所有这些包装值。此外，在所有密文都更新为由新密钥 k'_c 加密的密文之前，旧密钥 k_c 不能被删除，从而大大延长了 k_c 的暴露时间。

在存储系统（以及其他应用程序）中更新客户端密钥的需求引出了可更新加密[9]的概念，其目标是为密钥轮换问题提供更有效、更安全的解决方案。多种可更新加密方案[9, 10, 20, 36]已经被提出。在这项工作中，我们以不经意密钥管理系统为背景进行研究，提出了可更新的不经意密钥管理系统(UOKMS)的设计。

在UOKMS中，为了执行客户端密钥 k_c 的轮换，服务器Kms根据新旧密钥 k_c, k'_c 计算出一个短更新令牌 Δ 函数，并将 Δ 传输给客户端C。C的存储服务器StS可以使用 Δ 将所有使用 k_c 的派生密钥加密的密文转换为可由新的 k'_c 解密但不能被旧的 k_c 解密的密文。此操作保护了数据的安全性，它在存储服务器 StS 本地执行，无需与 Kms 进行任何交互，并且它只修改密文的一小部分（与加密数据的长度无关），从而使整个操作非常高效。在安全方面，它可以防止客户端密钥 k_c 的未来和历史的安全性遭到威胁。

与传统KMS和我们自己的OKMS方案相比，上述 UOKMS 方案具备另一个主要性能优势：数据加密不需要与KMS服务器交互，只需要在解密数据时进行交互。更一般地说，我们的 UOKMS 支持公钥加密，因此所有人都可以为客户端 C 加密数据，但只有 C 可以通过与 KMS 服务器的交互来解密。

阈值可更新OKMS。OKMS和UOKMS方案都可以通过分布式服务器实现，只要妥协的服务器数不超过阈值，客户端的OPRF密钥就是安全的。这些系统继承了阈值OPRF结构的高效性[30]（在UOKMS方案中使用OPRF变体的情况下也是如此）。在UOKMS设置中，令牌 Δ 的更新是通过分布式服务器之间高效的多方计算实现的。这些解决方案保留了OPRF的可验证性属性，可以在实现时对客户端透明，即无论是单服务器还是多服务器，客户端的操作和代码都是相同的。详见第5节。

正式模型和分析。我们在可更新的不经意KMS 安全模型中正式分析了我们的解决方案，该模型与最近的可更新加密模型（或密钥轮换加密）[9, 10, 20, 36]有高度相似之处，也有显著的差异。一个关键区别在于密钥管理设置，我们的客户端与两个外包远程服务Kms和StS交互。这会导致客户端和Kms之间的通信通道产生潜在的安全漏洞。过去的可更新加密模型没有这个问题，它们将客户端和Kms本质上视为并置实体。我们的解决方案的另一个独特之处是Kms方采用不经意计算。还有一个区别是，虽然典型的存储设置不需要公钥加密，我们仍然将此设计包含在我们的可更新模型中。

我们的可更新模型能够承受同时对KmS和StS的攻击，包括客户端密钥 k_c 、更新令牌值 Δ 的泄露，以及攻击方查看密文并向StS写入密文的情况。我们使用基于模拟的安全模型提供针对未来和过去攻击的安全性，即前向和后向安全。显然，该模型不允许能够让攻击者得到一定程度胜利的攻击组合（例如，在学习KMS密钥 k_c 的时间段内解密挑战密文）。该模型适用的不经意设置是：与KmS通信（且拥有C的凭证）的攻击者可以解密任何一个与KmS交互过的密文q，但所有其他密文仍然安全。考虑到攻击者对密文更新使用预言机的能力，以及身份验证加密的应用，该模型为不经意和可更新的加密提供了类似于CCA的安全性。我们的UOKMS方案的安全性证明（将在第4节介绍）在Gap One-More Diffie-Hellman假设[5, 32]的强化变体下携带随机预言机模型，我们将证明它在通用组模型中保持不变。

实现与效果。在第6节中，我们展示了OKMS和UOKMS方案具体应用的性能信息，从而展现了该技术的实用性，特别是服务器每秒支持大量操作和客户端请求的能力。在OKMS中，客户端时间大约为0.4毫秒（加密）和0.2毫秒（解密）。UOKMS系统性能更佳：客户端单线程单CPU核，每秒可分别承受41000/6000/14000次以上的加密/解密/更新操作，且只在解密时需要服务器进行操作。我们还通过部署到Amazon EC2实例的(U)OKMS Server的原型实现展示了良好的吞吐量和延迟结果。在单一服务器和多服务器的部署中，该实现每秒均能回答超过3000个请求。最后，我们介绍在硬件安全模块（HSM）中管理KmS密钥的经验。

1.1 与过去工作的比较

我们首个针对基于云（和其他）存储系统的密钥管理核心问题提出了全面可更新解决方案（利用了不经意计算），且率先为此类设置开发了安全模型。我们的目标和模型与最近的可更新加密（UE）模型有相似之处[9、10、20、34、36]，但也有许多显著差异。其中最突出的是使用不经意作为解决远程密钥管理系统产生的潜在漏洞的一种方式，而不是像以往的可更新加密工作一样，假设与客户端并置的漏洞。我们解决方案的其他新功能包括从KmS无条件隐藏数据加密密钥和对象标识符，以及通过阈值实现分布式UOKMS服务。我们的可更新解决方案是与密文无关的（即更新令牌的大小与密文的数量和要更新的数据的大小相独立），与先前的几个UE方案相同[9、20、34、36]。其中，我们的方案是最高效的，只需要一个来自KmS服务器的短更新值 Δ 和更新操作的每个对象的单个求幂，例如在[34, 36]的方案中，每个密文块进行两次取幂。我们的UOKMS方案可以扩展以提供类似于[34, 36]模型中密文的不可区分性和不可链接性，但它继承了这种方案的低效率，导致其在大规模数据存储部署中并不实用（注3：先前的几项工作，例如[34, 36]将密文在更新周期内的不可链接性视为主要设计目标，但实现它要求以O(n)幂次的复杂度更新长度n的密文。我们相信，在大多数实际设置中，可链接性可以通过元数据、对象标识符等实现，因此不值得为此付出高昂的计算成本）。最后，这是第一个支持公钥加密的模型和解决方案，包括在不经意加密下类似CCA的安全性。我们在第3.2节中进一步阐述了我们与先前可更新加密工作的关系。

可更新加密与代理重加密（PRE）密切相关，特别要提到的是，我们研究的核心Diffie-Hellman技术直接与Blaze等人的PRE方案相关[7]。最近，[17, 18]在PRE的上下文中处理前向保密和腐败后安全，与我们的做法相同，他们在上下文中定义了进化密钥。然而，PRE的要求，特别是[17]中提出的要求，比我们方案的要求更严格。其中包括使用受托人的公钥而不是输入其密钥来生成更新值、实现单向性、支持通用DAG委托图、确保密文不可区分等。因此，他们需要使用更多且效率较低的技术；这突出表现为[18]建立在基于配对的结构和HIBE[8, 14]上，而[17]使用来自[11, 46]的基于格的全同态技术。另一方面，尽管它们具有更强的性能，但这些方案都不支持不经意计算。

我们对OPRF函数的使用可以认为是“OPRF即服务”的应用程序，这是一个团队在[19]中提出一个术语。我们从该工作中借用了可更新不经意PRF的概念，但他们的应用针对密码验证协议，而我们的方案是一个通用的加密存储系统（此外，[19]的协议效率明显低下，因为它使用具有双线性映射的组来获得更强的可更新“部分遗忘”PRF的概念，而我们不需要）OPRF也用于“带保护的密码秘密分享”[28]，它可以实现分布式密码安全存储，但不能更新主加密密钥。此外，这两种解决方案都专门用于密码验证的客户端，而UOKMS则适用于任何客户端到KMS或客户端到StS的身份验证机制。

与U-PHE的比较。UOKMS的目标与[35]可更新密码强化加密 (U-PHE) 有一些相似之处。在U-PHE中，服务器S为客户端存储加密数据。数据的加密和解密需要S持有客户端的密码，并与附加服务器R交互，称为速率限制器。了解S状态（但不是存储密码的客户端）的攻击者，如果不猜测客户端密码并与速率限制器R交互，就无法解密客户端数据。该解决方案提供了与我们类似的可验证性和可更新性，我们的UOKMS模型可以将S视为存储服务器St，将R视为密钥管理服务器KmS。然而，与UOKMS不同的是，在U-PHE中，服务器S知道客户端的解密消息和客户端的密码（尤其在依赖TLS传输密码时），而在UOKMS中，客户端加密和解密数据，服务器均不了解。而且，U-PHE解密协议不是不经意的，换言之，服务器R和KmS均可以识别解密后的密文。此外，与上面[19, 28]的情况一样，PHE专门用于密码身份验证情况，而UOKMS独立于客户端使用的身份验证方式，允许任何形式的客户端身份验证凭据。[35]的U-PHE方案比我们的UOKMS效率低，特别由于它们的加密是交互式的，而我们不是，它们的解密和更新大约比我们慢两倍，并且速率限制器的阈值实现[35]比我们的阈值KmS代价高得多。

2 可更新不经意KMS

下面展示我们的主要成果，UOKMS（可更新不经意KMS），它建立在引言中描述的不经意KMS的一般方法之上，接下来将回顾这些知识。

2.1 不经意密钥管理系统

图2描述了不经意KMS(OKMS)协议，作为下一节中可更新方案的基础。OKMS在引言中被描述为一种更安全的替代方法（图1），如今在存储系统中广泛使用，特别是大型云部署中。当使用图3中基于DH的OPRF方案dh-op时，可以实现一个高效的OKMS（见第6节）并适应可验证性和分布式实现的扩展（第5节）。OKMS方案的安全性及其使用dh-op的实现遵循OPRF的属性（特别是[28, 29]中研究的）。我们没有正式地分析OKMS方案，而是在第3节和第4节中对其扩展的可更新OKMS设置进行分析。（可以通过将UOKMS模型特化为单个更新周期来获得OKMS的模型和分析）。

Components: G : group of prime order q ; H, H' : hash functions with ranges $\{0, 1\}^\ell$ and G , respectively, where ℓ is a security parameter.

PRF F_k Definition: For key $k \leftarrow_R \mathbb{Z}_q$ and $x \in \{0, 1\}^*$, define

$$F_k(x) = H(x, (H'(x))^k)$$

Oblivious F_k Evaluation between client C and server S

- (1) On input x , C picks $r \leftarrow_R \mathbb{Z}_q$; sends $a = (H'(x))^r$ to S .
- (2) S checks that the received a is in group G and if so it responds with $b = a^k$.
- (3) C outputs $F_k(x) = H(x, b^{1/r})$.

Figure 3: DH-based OPRF function dh-op [29]

2.2 可更新OKMS

相关规则和最佳实践要求密钥管理系统定期更新客户端密钥 k_c （密钥轮换）。目标是将密钥 k_c 泄露的负面影响限制在尽可能短的时间内和尽可能少的数据内。这对于保护长期存储的数据的密钥尤为重要，因为它在许多云存储应用程序（从用户照片到受监管的财务信息的任何内容）中都很常见。在KMS服务器KmS轮换密钥 k_c 为一个新的密钥 k'_c 时，存储服务器StS保存的所有用 k_c 保护的密文也需要更新。更新后的密文应该可以由 k'_c 解密，但不能由 k_c 解密。目标是得知了 k_c 但只看到更新后密文的攻击者不能了

解任何关于加密数据的信息（在 k'_c 尚未泄露时）。类似地，如果攻击者已经看到使用未公开的 k_c 加密的密文并且后来获得了 k'_c ，它仍然不能从密文中学到任何东西。这提供了前向安全性（针对未来暴露的安全性）和后向安全性（针对过去暴露的安全性）。显然，这还要求更新过程本身不能向StS泄露加密信息（例如，由StS加密和重新加密的数据将被视为不安全的）

Setting: Generator g of group G of prime order q ; symmetric authenticated encryption scheme Enc, Dec with keys of length security parameter ℓ ; hash function $H : G \rightarrow \{0, 1\}^\ell$.

Client keys: KMS server KmS stores a client-specific random key $k_c \in \mathbb{Z}_q$ for each client; storage server StS stores certified public value $y_c = g^{k_c}$ for client C .

Encryption of object Obj: To encrypt Obj under key y_c , pick $r \leftarrow_{\mathbb{R}} \mathbb{Z}_q \setminus \{0\}$, set $w = g^r$ and $dek = H(y_c^r)$, and output ciphertext triple $c = (\text{ObjId}, w, \text{Enc}_{\text{dek}}(\text{Obj}))$.

Decryption of ciphertext $c = (\text{ObjId}, w, e)$: (1) C sends $u = w^{r'}$ for $r' \leftarrow_{\mathbb{R}} \mathbb{Z}_q$ to KmS; (2) KmS checks if $u \in G$ and if so returns $v = u^{k_c}$ to C ; (3) C outputs $\text{Obj} = \text{Dec}_{\text{dek}}(e)$.

(Note that C runs the decryption protocol only if c is valid and we define $c = (\text{ObjId}, w, e)$ as valid iff $w \in G$ and $w \neq 1$.)

Key rotation and update: To change client's key from k_c to k'_c , KmS sends $\Delta = k_c/k'_c$ and $y'_c = g^{k'_c}$ to StS. StS replaces y_c with y'_c and replaces each ciphertext $c = (\text{ObjId}, w, e)$ with $c' = (\text{ObjId}, w' = w^\Delta, e)$, provided that $w \in G$. (Element $w \notin G$ indicates an invalid ciphertext which can be removed.)

Figure 4: Updatable Oblivious KMS Scheme

在图1的传统KMS中，密钥轮换操作需要存储服务器StS和KMS服务器KmS之间的交互，其中StS将每个存储的包装发送到KmS以在 k_c 下解包并使用 k'_c 重新包装。这需要在StS和KmS之间传输所有密文值，并将所有 dek 值暴露给KmS。在大型存储系统中，这样的过程可能需要很长时间（特别是在“惰性计算”中，只有当应用程序需要对该对象进行常规解包操作时，StS持有的包装才会更新为 k'_c ）。在此期间，新旧密钥 k_c 和 k'_c 必须存储在KmS中，从而延长了密钥的存在时间和暴露期。

在图4中，我们展示了一个可更新不经意KMS，它从上一节的OKMS方案适配为可更新系统。使用可更新加密[9, 20, 36]技术，我们在安全性和性能方面实现了一些理想的特性。首先，在将密钥 k_c 更改为新密钥 k'_c 后，KMS服务器KmS生成一个短令牌 Δ ，StS可以使用该短令牌更新客户端C的所有密文，从而实现上述安全属性。其次，更新操作是非交互式的：它由StS在本地执行，且StS唯一拥有 Δ 。一旦KmS生成新密钥 k'_c 和相应的更新值 Δ ，KmS可以立即擦除旧密钥 k_c ，从而降低风险，一次只暴露一个密钥。最后，StS的更新操作只需要独立于密文大小地对每个密文进行一次求幂，而过去的可更新加密方案中每个密文至少需要2次求幂（另见脚注3），从而令 k_c 加密的密文能够快速更新。因此，我们获得了一种非常有效的更新过程，它在许多方面比传统KMS实现了更好的安全性：在更新期间， dec 密钥永远不会暴露给StS或KmS；密钥轮换后可以立即删除旧密钥；StS和KmS之间的交互最少（仅传输 Δ ）；一旦StS在本地更新完所有密文， Δ 就可以被StS删除。

图4中的UOKMS方案在一些重要方面不同于图2中的OKMS方案。首先，为了允许快速更新，密文由两部分组成，一个包装和一个对称加密的密文，它从包装中导出加密密钥。在更新时，只更新包装。其次，加密操作是非交互式的，即C（或其他任何人）可以在本地加密数据而无需与KmS交互，前提是它拥有与 k_c 对应的经过认证的“公钥” y_c 的等价密钥（在我们的方案中， $y_c = g^{k_c}$ ）。只有通过与KmS的不经意交互才能解密。作为上述属性的“副作用”，UOKMS方案支持公钥加密，这意味着任何人都可以生成密文，但只有C可以解密它们，从而扩展此类KMS解决方案的用例。注意，解密需要与KmS进行交互，我们认为KmS有方法验证来自C的解密请求。第三，图4中的UOKMS方案是根据特定实例化而不是使用OKMS中的OPRF等通用工具呈现的。实际上，更新操作要求的延展性属性对于通用OPRF是不可能的（但可参阅下面有关弱OPRF的信息）。最后，在非交互式加密的UOKMS中不需要KmS对正确加密的验证，并且可以通过（对称的）身份验证解密操作Dec来验证正确解密。这减少了验证正确指数时对KmS的需要，进一步提升了UOKMS的性能。

UOKMS方案的正确性很容易验证。对于加密操作，令 $w = g^r$ ，其中 r 是随机数，然后从 y_c^r 导出加密密钥 dek ，加密数据并存储 w 。解密时，C在与KmS交互时不经意地计算 w 并从中导出 dek 。这会将原始数据恢复为 $y_c^r = (g^{k_c})^r = (g^r)^{k_c} = w^{k_c}$ 。关于更新操作，分别用 w_t 和 k_t 表示 t 次更新后 w 和 k_c 的值（这里 w_0 表示得到 dek 的时段， k_0 表示那一时段客户端的密钥 k_c ），那么可以归纳地发现，如果 $w_t^{k_t} = w_0 k^0$ （后者是 dek 的推导值），那么对于 $t+1$ 也将如此，即 $(w_{t+1})^{k_{t+1}} = (w_0)^{k_0}$ 。实际上，由于 $w_{t+1} = w_t^{\Delta_{t+1}} = w_t^{k_t/k_{t+1}}$ ，因此 $(w_{t+1})^{k_{t+1}} = (w_t^{k_t/k_{t+1}})^{k_{t+1}} = w_t^{k_t} = w_0^{k_0}$ 。

图4中UOKMS方案的安全性将在第4节中得到证明，且以第3节介绍的安全模型为基础。

在第5节中，我们将展示如何通过一个阈值方案来分配UOKMS的KmS功能，其中包括 Δ 值的分布式生成，以确保只有StS能了解到此内容。

关于弱不经意PRF。图4中的UOKMS方案根据函数 $F_K(w) = H(w^k)$ 导出了对称加密密钥，该函数定义在素数阶 q 的组 G 中的元素上（其中密钥 k 在集合 \mathbb{Z}_q 中随机选择）。函数 F 与图3中的 $OPRF\ dp - op_k(x) = H(X, (h'(x))^k)$ 具有很强的相似性（该函数也被用作图2中OKMS方案的基础），但也存在根本性差异。首先， F 的输入是一组元素（而不是通过dh-op中的哈希函数 H 映射到组中的任意字符串）。更重要的是，对于任意 w ，知道 w^k 的值就能计算一个已知 t 对应的 w^t 的函数。同时，在CDH下很难在独立随机群元素上计算 F ，因此 F 可以被建模为弱PRF（如[40]中所述）。在我们对UOKMS的应用中，我们还利用了 F 可以不经意计算这一点，并利用其同态属性来支持可更新性。我们将把在UC模型中对这种“不经意弱OPRF”函数进行形式化作为未来的工作项目，类似[29]中对OPRF的处理。为了在UOKMS的上下文中使用 F ，我们会直接在第3节中介绍的专门UOKMS安全模型中进行分析。

3 可更新不经意KMS的安全模型

以下介绍可更新不经意KMS的安全模型，它将不经意计算和可更新加密的元素和优势结合在一个模型中。与可更新加密一样，例如[9, 10, 20, 34, 36]，我们考虑随时间进化的密钥，在新的时间段开始时，加密/解密密钥被替换为新密钥。在我们的例子中，这适用于密钥管理服务器KmS持有的客户端密钥 k_c 。目标是在前向安全和妥协后安全的意义下刻画密钥轮换的安全性。也就是说，特定时期的客户端密钥 k_c 的泄露不应该有助于未来或先前时期加密数据的泄露。然而，在后一种情况下，需要满足以下要求。假设密文 e 是使用时段 t 的密钥 k_c 生成的，而未来时段 $t' > t$ 的密钥 k'_c 泄露了；被加密为密文 e 的数据 d 是否还安全？当然，如果攻击者A看到了 e' ，即密文 e 在时段 t' 更新后的版本，那么可以通过解密 $e' A$ 来获得 d 。然而，如果A拥有 k'_c 和 e ，但没有更新后的 e' ，那么 d 的安全就能得到保护。

上述内容说明了可更新加密模型的复杂性，这需要仔细记录攻击者可用的信息：它看到了什么密文，什么时候看到的，获得了什么时候的密钥 k_c ，以及它接收更新信息等。这样做是为了防止攻击者得知任何不能从它请求的信息中平凡地（不可避免地）推导出来的东西。在本节中，我们通过UOKMS安全性的正式模型设定了这些规则和目标，并在第4节中使用它们来证明图4中UOKMS模型的安全性

3.1 正式UOKMS方案

从形式上来说，可更新不经意KMS（UOKMS）方案是一个算法元组，包括 $KGen$, Enc , $UGen$, $UEnc$, 以及协议 Dec , KMS服务器 KmS , 存储服务器 StS , 客户端 C , 满足：

- $KGen$ 是一个密钥生成算法，由 KmS 运行，通过输入的安全参数 ℓ 生成公钥对 (sk, pk) 。
- Enc 是加密算法，可由任意一方运行，在输入密钥 pk 和明文 m 后生成密文 c 。
- $Dec=(Dec.KmS, Dec.C)$ 是交互式解密协议，客户端运行 $Dec.C(pk, c)$, KmS 运行 $Dec.KmS(sk, pk)$ ，其中 $Dec.C$ 输出 m 或 \perp 。
- $UGen$ 是一个更新生成算法，由 KmS 运行，通过输入 (sk, pk) 生成一个新密钥对 (sk', pk') 以及更新令牌 Δ
- $UEnc$ 是密文更新算法，由 StS 运行，对于输入 (c, pk, Δ) 输出更新后的密文 c' 。

一个UOKMS模型必须满足以下正确性。首先，交互式解密必须恢复加密的明文，即对于任何 m ，如果 $(sk, pk) \leftarrow KGen(\ell)$ 且 $c \leftarrow Enc(pk, m)$ ，那么在 $Dec.C(pk, c)$ 与 $Dec.KmS(sk, pk)$ 交互后输出 m 。此外，相同的正确性适用于在后期生成和更新的密钥和密文。也就是说，对于每个 m ，如果 $(sk, pk) \leftarrow KGen(\ell)$ ， $c \leftarrow Enc(pk, m)$ ， $(sk', pk', \Delta) \leftarrow UGen(sk, pk)$ ， $c' \leftarrow UEnc(c, pk, \Delta)$ ，则 $Dec.C(pk', c')$ 与 $Dec.KmS(sk', pk')$ 交互后输出 m 。

关于公共和私有值。我们将UOKMS建模为公钥加密方案，拥有公钥 pk 的任何一方都可以为客户端加密文件，其对应的解密密钥 sk 由KmS持有。假设KmS有办法在使用密钥 sk 进行解密操作之前对客户端进行身份验证，且不需要客户端和KmS之间的秘密信道。假定更新令牌 Δ 通过安全通道从KmS传输到StS。任何其他方都不需要也不应该知道这个值。特别注意，该模型不保证给定 sk_t 和 Δ_{t+1} 时 sk_{t+1} 的保密性或给定 sk_{t+1} 和 Δ_{t+1} 时 sk_t 的保密性。例如，在图4的UOKMS方案中，接收 Δ_{t+1} 后就允许一方从 sk_t 派生 sk_{t+1} 、从 sk_{t+1} 派生 sk_t 。在这种情况下，如果 Δ_{t+1} 泄露，且KmS在时段t遭到攻击而受损，那么时段t+1也会受损，反之亦然（注4：这不是UOKMS方案的必要特性，即可以假定 Δ_{t+1} 能被用于更新密文（和公钥），但能更新相应的密钥。然而，所有现有的与密文无关的可更新加密方案，包括我们的方案，都允许在给定 Δ 的情况下更新 sk ）。

3.2 UOKMS的不经意性和安全性

以下定义形式地说明了KMS的不经意性。

定义3.1 如果对于所有的有效算法A，在输出均为 $(pk, c0, c1)$ 时，使用 $Dec.C(pk, c0)$ 或使用 $Dec.C(pk, c1)$ 与A的交互都无法被区分，且 $c0$ 和 $c1$ 是长度相同的合法密文， pk 是合法公钥（注5：公钥 pk 通常由KmS选择，但在此定义中可以由A选择，模拟一个恶意KmS，但C可以检查公钥和密文的某些属性，例如在运行 Dec 之前检查它们是否包含预期的组元素），那么称该UOKMS方案是不经意的。

如上所述，定义UOKMS和一般可更新加密的安全性需要建立规则：关于对手能接收什么信息、何时接收信息，以及与该信息相关的胜利是如何构成的。在我们的模型中，时间被划分为许多段，每段从新密钥对 (sk, pk) 开始生效到更新令牌 Δ 生成。每个时期，对手A都会收到新的公钥 pk ，并且可以请求查看新的密钥 sk 或更新令牌 Δ ，这取决于与A联系的哪个服务器妥协，可能是服务器KmS或服务器StS。算法A也被授予对密文更新函数UEnc的oracle访问权限，但不允许将其用于微不足道的胜利(trivial win)，例如，将挑战密文更新到它知道密钥的时段。注意，如果A提出要求，A就能得知时段t的密钥 sk_t ，如果A在时段t-1中要求 sk_{t-1} 并在时段t中要求 Δ_t ，那么它也能得知 sk_t 。这表明A在一个时段可得到的信息取决于它在前一个时段知道什么，而UOKMS安全的规则必须反映这一点。

我们通过图5所示的真实-理想实验将这些规则和攻击目标形式化。在每个时段t，攻击者接收 pk_t 并选择破坏KmS以获得 sk_t ，或破坏StS以获得更新令牌 Δ_t ，除非KmS在t-1时段已被破坏（否则攻击者可以从 sk_{t-1} 和 Δ_t 计算出 sk_t ，使得这种情况等同于在时段t破坏KmS和StS）。此外，A获得对oracles Enc、Dec、UEnc的访问权限，具体取决于妥协方。我们的不经意定义的另一个特点是，能够访问不经意解密oracle的攻击者可以在解密调用中解密其选择的任何密文，但每次调用最多只能解密一个挑战密文。通俗来说，通过q次调用解密oracle，A可以解密q条消息，但仅此而已。最后，我们注意到，访问解密oracle的攻击者在不经意设置下为我们的公钥方案提供了CCA安全性。

图5展示了两个实验：实验 $Exp_{uokms}^{real}(\mathfrak{A}, \ell)$ 模拟真实世界对手A与真实UOKMS方案的交互，实验 $Exp_{uokms}^{ideal}(\mathfrak{A}, SIM, \ell)$ 模拟模拟器SIM与“理想的”UOKMS方案的交互。如果两种交互（真实的和理想的）无法区分，我们称UOKMS方案是安全的。

定义3.2 $Adv_{uokms}^{real}(\mathfrak{A}, \ell)$ 表示 $Exp_{uokms}^{real}(\mathfrak{A}, \ell)$ 输出1的概率， $Adv_{uokms}^{ideal}(\mathfrak{A}, SIM, \ell)$ 表示 $Exp_{uokms}^{ideal}(\mathfrak{A}, SIM, \ell)$ 输出1的概率。我们说UOKMS方案是安全的，如果对于所有有效算法A，都存在一个有效算法SIM使得 $|Adv_{uokms}^{real}(\mathfrak{A}, \ell) - Adv_{uokms}^{ideal}(\mathfrak{A}, SIM, \ell)|$ 相对 ℓ 的大小来说是可以忽略不计的。

图5中的真实实验 $Exp_{uokms}^{real}(\mathfrak{A}, \ell)$ 模拟了对手A与UOKMS方案的交互，该方案在 $t = 0, 1, \dots$ 中执行，其中标志位 $corr_t$ 表明A是否损坏KmS(kms)或时段t的存储服务器StS(sts)。在生成初始KMS密钥对(sk_0, pk_0)的初始化之后，我们将 pk_0 交给A，让A与加密、解密和密文更新oracles进行交互。

对从一个时段到下一个时段的进展进行建模时，我们借助“参与方腐败”的oracle Corr，它使用A的决策位 $corr_{t+1}$ 在下一个时段破坏KmS或StS（注6：假设蛮族w.l.o.g，即A在每个时段都恰好腐蚀了各参与方中的一个，特别是StS和KmS在时段t都被破坏时，该真实事件在我们的模型中反映为KmS在两个连续的时段t-1和t中都被破坏，因为这同时向A揭示了 sk_t 和 Δ_t ）。这个oracle触发了一个密钥更新，即一个新的KMS密钥对被创建为 $(sk_{t+1}, pk_{t+1}, \Delta_{t+1}) \leftarrow UGen(sk_t, pk_t)$ ，并且计数器t递增。敌手A随

后收到新的公钥 pk_{t+1} 甚至更多信息，这取决于它破坏的各方：例如，若 $corr_{t+1} = kms$ ，那么 A 也会收到新的密钥 sk_{t+1} ；若 $corr_{t+1} = corr_t$ ，即 A 在两个连续的时段中破坏了同一方，那么 A 也会获得更新令牌 Δ_{t+1} 。关键的是，如果 $corr_t + 1 \neq corr_t$ ，A 无法得到 Δ_{t+1} （实际上，如上所述，在图4的UOKMS方案中，接收更新令牌将允许敌手高效地将对KmS的腐化从时刻t扩展到时刻t+1，反之亦然）。

安全实验假设KmS是被动腐化的，若 $corr_t = kms$ ，我们让 A 得知 sk_t （如果 $corr_{t-1} = kms$ ，也可以告知 Δ_t ），但我不让 A 干扰更新、生成和/或传播已创建的更新令牌和公钥，或执行解密协议（所有现有的可更新加密安全概念都会做出这样的选择，如假设即使对手破坏了存储密钥的实体，密钥更新仍然能诚实地执行）。

我们假设StS的腐化在某种意义上是活跃的：对于 $corr_t = sts$ 的时段，我们不仅让 A 得知了 Δ_t ，而且允许 A 访问密文更新 oracle UEnc，当输入 (t', c) 时，对于 $t' < t$ 和 c 来自时刻 t' 的密文，输出 t 时刻 c 的更新值。也就是说，预言机使用更新令牌 $\Delta_{t+1}, \dots, \Delta_t$ 对（假设的）密文 $c_{t'} = c$ 运行更新算法，并输出更新后的密文 c_t 。这模拟了对手在某个时段向StS注入密文的能力——在StS被腐化时直接修改这些密文，或者将密文发送给客户端，然后客户端将其存储在StS（注7：我们的方案针对公钥加密，因此其他参与方也可以创建密文，这些密文都将进入StS存储），随后由UEnc预言机更新该密文。注意，在 $corr_t = kms$ 的时段不能提供更新协议，因为这将允许 A 使用腐化的KmS密钥解密挑战密文。

我们的 UOKMS 模拟公钥加密，其中敌手 A 可以随意加密任何消息，但加密预言机 Enc 在 UOKMS 安全中的作用是模拟挑战密文的生成。也就是说，在真实情景中，预言机 Enc 基于 A 的输入 m 生成密文 $c = Enc(pk_t, m)$ ，但在理想情景中，相同的密文 c 必须由仅给定 $|m|$ （和标志位 enc）的模拟器算法 SIM 生成，作为输入，同时将明文 m 添加到加密挑战明文的（秘密）列表 L 中。敌手 A 可以使用解密预言机 Dec 解密任何密文（或它选择的任何类似密文的对象）。因为我们的目标是支持不经意解密，所以 A 有效解密得到精确密文的操作对预言机是隐藏的，因此我们必须将每次解密预言机访问都视为尝试解密一些挑战密文。我们在理想情景中对此进行建模，根据 A 的每个 Dec 查询，让 SIM 访问挑战明文表 L 中的某个位置。注意，这在技术上意味着模拟器可以提取 A 试图在不经意解密协议的实例中解密的唯一密文，否则模拟器将不知道它应该访问列表 L 中的哪个明文。还要注意，我们不会在 KmS 被腐化时创建挑战密文，因为敌手得知 KmS 的私钥会使所有此类密文不安全。

$Exp_{uokms}^{real}(\mathcal{A}, \ell)$

Set $t \leftarrow 0$ and $corr_0 \leftarrow sts$. Generate $(sk_0, pk_0) \leftarrow KGen(\ell)$ and give pk_0 to \mathcal{A} . The experiment output is the output of \mathcal{A} after interaction with the following oracles:

Enc: On \mathcal{A} 's input m , if $corr_t = sts$ output $Enc(pk_t, m)$;

Dec: Let \mathcal{A} interact with $Dec.S(sk_t)$;

UEnc: On \mathcal{A} 's input (t', c) , if $corr_t = sts$ and $0 \leq t' < t$ then

set $c_{t'} := c$; for $j = t'+1$ to t set $c_j := UEnc(c_{j-1}, pk_j, \Delta_j)$; output c_t ;

Corr: On \mathcal{A} 's input $corr_{t+1}$, set $(sk_{t+1}, pk_{t+1}, \Delta_{t+1}) \leftarrow UGen(sk_t, pk_t)$;

If $(corr_t, corr_{t+1}) = (kms, kms)$ output $(pk_{t+1}, sk_{t+1}, \Delta_{t+1})$;

If $(corr_t, corr_{t+1}) = (kms, sts)$ output pk_{t+1} ;

If $(corr_t, corr_{t+1}) = (sts, kms)$ output (pk_{t+1}, sk_{t+1}) ;

If $(corr_t, corr_{t+1}) = (sts, sts)$ output (pk_{t+1}, Δ_{t+1}) ;

Increment epoch counter $t := t + 1$.

$Exp_{uokms}^{ideal}(\mathcal{A}, \text{SIM}, \ell)$

Set $t \leftarrow 0$ and $corr_0 \leftarrow sts$. Initialize an empty challenge plaintext list L . Let a *stateful* algorithm SIM generate pk_0 on input ℓ and give pk_0 to \mathcal{A} . Experiment output is the output of \mathcal{A} after interaction with the following oracles:

Enc: On \mathcal{A} 's input m , if $corr_t = sts$ add m to L and output $SIM(enc, |m|)$;

Dec: Let \mathcal{A} interact with $SIM(dec)$ while letting SIM learn *one* chosen item in L ;

UEnc: On \mathcal{A} 's input (t', c) , if $corr_t = sts$ and $0 \leq t' < t$ then output $SIM(upd, t', c)$;

Corr: On \mathcal{A} 's input $corr_{t+1}$ output $SIM(corr_{t+1})$ and increment epoch counter $t := t + 1$.

Figure 5: Security Experiments for Updatable Oblivious KMS

我们强调， Exp_{uokms}^{real} 安全博弈允许任何模式的腐化，除了在同时段内同时腐化 StS 和 KmS（见注 6）。然而，我们的腐败模型是静态的，因此 A 必须在每个时段开始时决定腐化哪一方（另请参阅下面的讨论）。

先前的可更新加密模型。我们的可更新（不经意）KMS概念与可更新加密（UE）和密钥轮换加密有关，最近的几项工作[9、10、20、34、36]对此进行了研究。UOKMS通过将UE的客户端分成两个独立的实体来扩展UE的概念，KMS服务器保存客户端的解密密钥并生成密钥更新，客户端本身通过交互式解密协议解密从存储服务器检索到的密文。因此，UOKMS模型将可更新加密的概念提升到反映实际大型云存储部署的设置中，其中所有客户端的解密密钥都由专门的密钥管理服务器持有。另一方面，将UOKMS模型中的客户端和KMS压缩为一个实体，给出了UE的准确设置，因此我们的UOKMS方案和安全概念产生了相应的UE方案和概念。

我们的安全模型对应于Lehmann等人的密文独立UE模型[36]。（改进自Everspaugh等人的模型[20]），其中单个更新消息可用于更新任意数量的密文。其中，只有Klooss等人最近的工作[34]解决了CCA安全问题，并让对手访问解密预言机，就像我们在模型中所做的那样。不过，在[34]中显示的两种安全方案中，效率与我们接近的方案不允许对手不受限制地访问密文更新预言机，而我们的模型允许其不受限制地访问Dec和UEnc预言机。[34]的方案允许这种不受限制的oracle访问在很大程度上依赖基于配对的NIZK，使用例如22对用于解密，与我们方案中解密中使用的单个标准组幂相反。然而，我们的UOKMS安全模型专门针对不经意交互式解密的情况，其中解密预言机要求对KMS服务器建模，在盲密文上运行。在这样的环境下，一个标准的CCA概念（其中解密预言机被限制不能解密挑战密文）不适用。因此，我们使用一种“计数方法”来获得安全性，该方法强制第Q次访问解密预言机允许学习明文至多Q条挑战密文中的信息。我们的方案第一次使用不经意解密过程处理可更新加密，并且此设置需要在存在解密预言机的情况下使用“基于计数”的安全概念。

[20、34、36]的UE方案实现了更新不可区分性，即更新到新时段的密文不能有效地链接到前一个时段的原始密文。我们不考虑这个属性，尽管我们的方案可以扩展以支持它，但实现这个属性需要与加密数据的总大小成比例的更新成本，我们认为这在大型存储部署中是不切实际的（见注3）。上述UE方案也考虑了密文完整性，但这个概念专门用于对称密钥加密的情况，而我们的UOKMS模型处理的是公钥加密的情况。

最后应该指出，我们的安全模型是静态的，因为对手必须在每个时段开始时选择是否腐化KMS存储的解密密钥或存储服务器持有的更新令牌（或两者均有）。相比之下，[34, 36]考虑了一种自适应的腐败模型，其中对手可以请求过去任何时期的解密密钥或更新令牌，或两者均有。自适应安全模型更通用，限制更少，但我们只在静态模型中分析我们方案的安全性，因为自适应安全提出了微妙的技术挑战，我们不知道如何克服（注8：我们强调，这只是证明中存在的问题，而不是一个明确的攻击，并且在其他情况下也发现了自适应安全类似的技术问题，例如在主动密码系统中的问题，参见[1, 13, 37]）。从技术上讲，模拟器必须对过去的时段赌博，猜测对手最终会要求过去某个时期的解密密钥（在这种情况下，模拟器需要知道这个时段的密钥），还是要求能够将挑战密文更新到那个时期更新令牌（在这种情况下，模拟器需要在那个时段的密钥中嵌入一个加密挑战）。由于模拟器需要对多项式次数的过去时段进行推算，因此其猜测全部正确的概率可以忽略不计，并且尚不清楚这种策略是否可以生成有效的模拟。因此，我们认为[34、36]的完全自适应模型中的安全性分析仍然是一个未解决的问题。

4 UOKMS模型的安全性分析

图4中的UOKMS模型是信息论不经意的，其解密协议所基于的OPRF协议dh-op也是如此，但该方案的安全性依赖于OMDH-IO计算假设和（接收者）对称加密的非提交属性，定义如下：

具有反向Oracle（OMDH-IO）假设的多点DH。

对任意PPT A，下列概率是可以忽略的：

$$\text{Prob}[\mathcal{A}^{(\cdot)^k, (\cdot)^{1/k}}(g, g^k, g_1, \dots, g_N) = \{(g_{j_s}, g_{j_s}^k)\}_{s=1, \dots, Q+1}]$$

当概率超过Zq中的随机数k，在G = < g >中随机地选取一组元素g₁, ..., g_N，

For any PPT \mathcal{A} the following probability is negligible:

$\text{Prob}[\mathcal{A}^{(1)^k, (\cdot)^{1/k}}(g, g^k, g_1, \dots, g_N) = \{(g_j, g_{j_s}^k\}_{s=1, \dots, Q+1}]$
 with the probability going over random $k \in \mathbb{Z}_q$, random choice
 of group elements g_1, \dots, g_N in $G = \langle g \rangle$, and \mathcal{A} 's randomness,
 and where $(\cdot)^k$ and $(\cdot)^{1/k}$ are exponentiation oracles, and Q is the
 number of \mathcal{A} 's queries to the $(\cdot)^k$ oracle.

接收者不承诺对称密钥加密。我们将对称密钥加密(SKE)的这一属性用于安全分析，以启用图5中安全博弈所需的模拟。通俗来说，它表示在不知道加密密钥的情况下，密文不会提交其底层明文，从而允许模拟器将固定密文“解释”为任何明文的加密。形式上，一个对称加密方案(Enc, Dec)是接收者不承诺(RNC)的，如果：对于任何PPT A，存在PPT SIM，使得A的视图在以下真实和理想博弈中无法区分：(1)在真实博弈中，A与预言机Enc和Reveal交互，其中 $\text{Enc}(i, m)$ 选择随机密钥 k_i 并输出 $e = \text{Enc}(k_i, m)$ ，而Reveal(i)揭示 k_i ；(2)在理想博弈中，A与有状态算法SIM交互，当A发送 (i, m) 作为Enc查询时，SIM必须对输入 $(i, |m|)$ 返回e，而当A发送i作为Reveal查询时，SIM必须对输入 (i, m) 输出 k_i 。

定理4.1 图4中的UOKMS方案是无条件不经意的，如果堆成加密方案Enc是接收者不承诺的，那么它在ROM的OMDH-IO假设下是安全的。

关于证明的注释。定理4.1的证明将在4.1节给出，我们注意到，为了获取协议的定理，OMDH-IO下的逆幕预言机是必须的，因为该协议（在我们的上下文中）提供了敌手A，其在时段t-1腐化了KmS，在时段t腐化了StS，使用一个函数 $(\cdot)^{1/k_t}$ 的预言机。事实上，在时段t，A获得了对UEnc的访问权，它实现了一个取幕预言机 $(\cdot)^{\Delta_t} = (\cdot)^{k_{t-1}/k_t}$ ，加上 k_{t-1} 的知识，A可以对其选择的任何值计算 $(\cdot)^{1/k_t}$ 。SKE Enc的RNC属性同样是必需的。考虑攻击者A的两个查询：(a) 对某些m的Enc查询，以及(b) Dec查询，其中A在接收到的密文 $c = (ObjId, w, e)$ 上运行Dec.C协议。根据图5的UOKMS安全博弈规则，模拟器SIM必须如下进行模拟：(a) 它在消息长度 $|m|$ 上生成c，(b) 在从列表L检索到的输入m上，它模拟协议Dec.S，以将c解密为m。SIM在解密协议中对A的消息u的响应v（见图4）定义了有效KMS密钥为 $k = DL(u, v)$ ， $(ObjId, w, e)$ 的数据加密密钥定义为 $dek = H(w_k)$ 。因此当SIM对于输入 w_k 定义预言机H的输出dek时，它必须满足 $e = \text{Enc}_{dek}(m)$ 。特别地，SIM首先创建仅给定 $|m|$ 的密文e，给定m后，它创建dek，使得 $e = \text{Enc}_{dek}(m)$ 。这表明SKE Enc符合RNC属性。

推论4.2 如果使用CTR或CBC模式实现对称加密，则图4中的UOKMS方案在理想密码模型和ROM中的OMDH-IO假设下是安全的。

因为CTR和CBC加密模式满足理想密码模型中的接收者不承诺属性：假设消息长度 $|m|$ 为块密码E定义了n个块，SIM服务Enc通过设置密文 $e = (IV, e_1, \dots, e_n)$ 来查询输入 $(i, |m|)$ ，其中IV和所有 e_i 都是随机块。当SIM获取 $m = (m_1, \dots, m_n)$ 来提供Reveal(i)查询服务时，值 $(IV, e_1, \dots, e_n, m_1, \dots, m_n)$ 定义了n个输入/输出对，SIM需要为随机密钥k设置 $E(k, \cdot)$ 。对于CTR模式，SIM为所有j设置 $E(k, IV + j) = m_j \oplus e_j$ ，而对于CBC模式，SIM为所有j设置 $E(k, m_j \oplus e_{j-1}) = e_j$ ，其中 $e_0 = IV$ 。无论哪种方式，通过 e_i 的随机性，这会将n个给定点上 $E(k, \cdot)$ 的输出设置为n个随机值。此操作在 $E(k, \cdot)$ 中产生碰撞的概率可以忽略不计，并且由于k的随机性， $E(k, \cdot)$ 中任意点曾被查询过的概率也可以忽略不计。

注释 上述参数可以扩展，从而包括通过encrypt-then-MAC方式进行身份验证的加密，由模拟器选择MAC密钥。

4.1 定理4.1的证明

首先注意，无条件不经意在该UOKMS方案中是直觉的，因为对于任何公钥 p_k 和任何两个有效密文 $c_0 = (ObjId_0, w_0, e_0)$ 和 $c_1 = (ObjId_1, w_1, e_1)$ ，与Dec.C的交互 (p_k, c_b) 在 $b = 0$ 和 $b = 1$ 时是相同的：在任一情况下，C发送 $u = (w_b)^{r'}$ ，其中 $r' \leftarrow_R \mathbb{Z}_q$ ，如果 $wb \in G$ 且 $wb \neq 1$ ，则它是一个随机群元素，因为群阶是质数。为了论证UOKMS的安全性，我们首先展示一个有效的模拟器算法SIM，它可以访问（任何）敌手算法A，与理想UOKMS的 $\text{Exp}_{uokms}^{ideal}$ 交互。然后我们将SIM重写为归约算法 \mathcal{R} ，如果A在区分与真实UOKMS的 $\text{Exp}_{uokms}^{real}$ 的交互及与SIM和 $\text{Exp}_{uokms}^{ideal}$ 的交互时具有 ϵ 优势，即如果

$$\epsilon = |\Pr[1 \leftarrow \text{Exp}_{uokms}^{real}(\mathcal{A}, \ell)] - \Pr[1 \leftarrow \text{Exp}_{uokms}^{ideal}(\mathcal{A}, \text{SIM}, \ell)]|$$

那么归约算法 \mathcal{R} 给 A 权限后，还有相同的概率 ϵ 解决OMDH-IO问题。因此，在OMDH-IO假设下， ϵ 必须可以忽略不计，这样UOKMS方案才是安全的。注意到，在SIM修改的一个步骤中，我们用SKE Enc的RNC属性假定的模拟器替换了真正的对称加密Enc。现在提供证明的细节。

该证明依赖于函数 $H : G \rightarrow \{0, 1\}^\ell$ 的ROM模型，如图4应用于UOKMS。具体来说，我们将H视为外部实体，A需要查询以计算H输出，模拟器SIM和归约算法 \mathcal{R} 拦截 A 对H的调用，我们测量概率 $p_0 = \Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)]$ 和 $p_1 = \Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)]$ ，基于H的随机性。为了简化，我们假设组G对于每个安全参数 ℓ 都是固定的，并且我们为OMDH-IO假设和UOKMS安全都假定了一个非统一的安全模型。为了减少视觉混乱，我们将明文文件表示为m而不是Obj，并且我们在密文中省略了标识符 ObjId。

我们首先描述博弈G，它再现了 A 在真正的安全博弈 $\text{Exp}_{\text{uokms}}^{\text{real}}$ 中看到的相同分布，但它能够帮助理解我们接下来将描述的模拟器SIM。博弈G选择 $k \in Zq$ 并将第一个时段的密钥设置为 $(k_0, y_0) = (k, g^k)$ 。G还选择了G中随机一组元素 g_1, \dots, g_N 的列表，其中N是 A 进行Enc查询次数的上限。对于每个 $i > 0$ ，G根据以下规则选择值：如果 A 在时段i腐化KmS，则G选择随机 $k_i \leftarrow \mathbb{Z}_q$ 并为 $y_i \leftarrow g^{k_i}$ 输出 (k_i, y_i) （如果 A 将KmS腐化了两个时段，也输出 $\Delta_i = k_{i-1}/k_i$ ）。如果 A 在时段i中腐化StS，则G根据 A 在i-1时腐败的一方采取行动：（情况1）如果是StS，则G随机选择 $\Delta_i \leftarrow \mathbb{Z}_q$ ，并输出 $y_i \leftarrow y_{i-1}^{1/\Delta_i}$ ；（情况2）如果是KmS，则G随机选择 $\Delta_{j+1,i} \leftarrow \mathbb{Z}_q$ ，并输出 $y_i \leftarrow y_j^{1/\Delta_{j+1,i}}$ ，其中j是 A 腐化KmS前最后一次腐化StS的时段。令 E_K 表示 A 腐化KmS的时段集合， E_S 为 A 腐化StS的时段集合。上述过程为每个 $i \in E_S$ 定义了值 δ_i ，使得 $y_i = y^{1/\delta_i}$ （因此 $k_i = k/\delta_i$ ），且在 $(i-1) \in E_S$ 时，G可以将 δ_i 计算为 $\delta_{i-1} \cdot \Delta_i$ ；或在 $j \in E_S$ 且 $j+1, \dots, i-1 \subseteq E_K$ 时，G可以将 δ_i 计算为 $\delta_j \cdot \Delta_{j+1,i}$ 。给定这些值，G在时段 $i \in E_S$ 时服务预言机 Enc、Dec 和UEnc（注意， $i \in E_K$ 不允许这些调用）如下：

- G用 $c = (w, \text{Enc}_{dek}(m))$ 回答Enc(m)的第n次调用，其中 $w = (g_n)^{\delta_i}$ ， $dek = H(z)$ ， $z = (g_n)^k$ ；（注意，由于在实际交互中c是分布式的，因此 $z = w^{k/\delta_i}$ ）
- G用 $v = (u^{1/\delta_i})^k$ 回答Dec；
- G用 (w', e) 回答UEnc(t' , c)，其中 $c = (w, e)$ ，当 $t' \in E_S$ 时， $w' = w^{\delta_i/\delta_{t'}}$ ；当 $t' \in E_K$ 时， $w' = (w^{\delta_i \cdot k_{t'}})^{1/k}$

Enc和Dec响应的正确性如下：因为 $k_i = k/\delta_i$ ，至于UEnc，注意到 $k_i = k/\delta_i$ 且从时段 t' 到时段i的隐式更新是 $\Delta_{t',i} = k_{t'}/k_i$ ，这意味着 $\Delta_{t',i} = (k_{t'} \cdot \delta_i) \cdot (1/k)$ 。因此，G再现了与安全博弈 $\text{Exp}_{\text{uokms}}^{\text{real}}$ 完全相同的视图。

模拟器SIM与理想实验 $\text{Exp}_{\text{uokms}}^{\text{ideal}}$ 交互并执行与G相同的算法——包括选择初始密钥k和密钥 k_i ，对于 $i \in E_K$ 更新相关值 δ_i 和 $\Delta_{i,j}$ ；如果 $i \in E_S$ ，定义相应的 δ_i 和 k_i 。为了处理Enc和Dec，SIM求助于对称加密 Enc 的 RNC 属性假定的（有状态）模拟器 SIM_E 。首先，当 A 在时段 $i \in E_S$ 中向Enc预言机发送第t个查询m时，我们将m放在列表L中的t位置，SIM回复 A 为 $c = (w, e)$ ， $w = (g_t)^{\delta_i}$ ，e由 SIM_E 在输入 $(t, |m|)$ 上计算得出。其次，当 A 将u发送给Dec时，SIM回复 $v = (u^{1/\delta_i})^k$ 然后监视 A 对H的查询：如果 A 向H查询 z，使得 $z^{1/k} = g_t$ ，其中 $g_t \in g_1, \dots, g_N$ ，那么SIM要求 $\text{Exp}_{\text{uokms}}^{\text{ideal}}$ 在列表L中的第t个位置显示消息m，将 (t, m) 作为Reveal查询发送给 SIM_E ，并给出dek密钥作为 SIM_E 的响应，定义 $H(z) = dek$ 。根据Enc的RNC属性， SIM_E 生成的(dek, e)在计算上无法与随机dek和 $e = \text{Enc}_{dek}(m)$ 区分开来（特别是此过程会将 H(z) 设置为与随机值无法区分的值）。

A与G、A与SIM（进而与 $\text{Exp}_{\text{uokms}}^{\text{ideal}}$ ）交互的唯一区别是在后一种情况下，如果A在参数 $(g_i)^k$ 上查询H，以获得 $\{g_1, \dots, g_N\}$ 中超过Q个元素，其中Q是A的解密查询的数量：给定Q个解密查询，SIM只能得知列表L中的Q项，因此它可以嵌入正确的消息作为Q个挑战密文的解密，涉及Q个挑战点 $\{g_{j_s}\}_{s=1, \dots, Q}$ ，但SIM无法正确解密第Q+1个密文(w, e)形成的 $w = (g_{j_{Q+1}})^{\delta_i}$ ，以允许A在 $z = (g_{j_{Q+1}})^k = w^{k_i}$ 上查询H。换句话说，如果

$$\epsilon = |\Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)] - \Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)]|$$

则 ϵ 的上限为 A 在 $Q+1$ 个点 $\{g_1, \dots, g_N\}$ 中查询值为 $(g_j)^k$ 的点的概率。但是通过查看 SIM 可以看出，SIM 可以很容易地改成针对 OMDH-IO 问题的归约 \mathfrak{R} ： \mathfrak{R} 沿用 SIM 的算法，只是使用 OMDH-IO 挑战密钥 g^k 作为 y ，点集 (g_1, g_2, \dots, g_N) 作为 OMDH-IO 挑战的一部分，它使用 OMDH-IO 预言机 $(\cdot)^k, (\cdot)^{1/k}$ 而不是直接使用指数 k 。注意，SIM 只使用了 Q 次 $(\cdot)^k$ ，来实现 Q 次 预言机的解密请求，如果 A 以 ϵ 的概率向 H 查询第 $Q+1$ 个参数 $(g_j)^k$ ， \mathfrak{R} 就会以 ϵ 的概率打破 OMDH-IO，因为它能够使用预言机 $(\cdot)^k$ 识别出此类请求。定理 4.1 证毕。

5 阈值OKMS和UOKMS

激发我们工作的密钥管理系统（尤其是存储程序）的特点通常是存储大量数据以及这些数据的价值和其长期存在的性质。此类操作的整体安全性取决于 KMS 客户端密钥的安全性，因此密钥轮换（如 UOKMS 所述）作为限制密钥暴露导致不良影响的一种方式非常重要。然而，首要任务是防止这些密钥泄漏。幸运的是，本文提出的所有方案都有助于通过图 6 中所示的高效阈值 OPRF tdh-op [30] 实现高效的分布式实现。

在我们的应用中，客户端密钥 k_c 在 n 个 KMS 服务器 S_1, \dots, S_n 之间共享，因此需要 $t+1$ 个服务器的协作来计算以 k_c 为密钥的 OPRF 函数，而 t 个服务器的妥协不会向攻击者提供关于 k_c 的任何信息。此外，密钥 k_c 永远不会被重建或存在于一个地方，甚至在生成时也不存在（这也是分布式执行的）。此外，该方案具有主动安全性 [25, 43]，即 n 台服务器之间的共享可以定期刷新，使得攻击者需要在同一时间段内侵入 $t+1$ 台服务器才能破坏密钥。服务器可以被更换并恢复共享，从而根据长期密钥的需要来保护系统的机密性和完整性/可用性。

图 6 中的 tdh-op 函数完全实现了图 2 中 OKMS 方案定义的 OPRF。相比于图 4 的 UOKMS 方案，唯一的区别在于来自客户端的输入（随机组元素，而不是哈希值）。

对效率的注释。tdh-op 中的主要计算成本是对每 $t+1$ 个服务器进行一次取幂，对客户端进行两次取幂，无论 n 和 t 的实际值如何。我们注意到，tdh-op 在图 6 中以简化形式进行了描述，其中假定 C 事先知道重构方 SE 的集合。如果重构集 SE 不是先验已知的（即联系了超过 $t+1$ 个服务器），则每个 S_i 将响应一个 a^{k_i} ，同时 C 将以一次多次取幂为代价计算指数的插值（当 α_i 较小时可以进一步优化，例如 $\alpha_i = i$ ，使用 [44] 中的最新技术）。tdh-op 解决方案的另一个重要特征是可以通过代理服务器（阈值服务器之一，或专用服务器）将服务器值 b_i 聚合到 dh-op 结果中，从而使得阈值实现是对客户透明的。

5.1 分布式更新

虽然阈值解决方案大大提高了 KMS 密钥的安全性，但人们可能仍然希望应用密钥轮换，特别是在考虑到我们的 UOKMS 解决方案的更新效率时。在阈值设置中，这意味着轮换时期开始时，共享密钥 k_c 的服务器需要选择一个新的随机客户端密钥 k'_c 并生成值 $\Delta = k_c / k'_c$ 。但是， Δ 应该只向客户端 C 和存储服务器 StS 公开，这就要求分布式生成 Δ ，其中 t 或更少服务器的子集不会获知有关此值的任何信息。

我们展示了一个给定 (n, t) 的 Shamir 共享密钥 k 的过程。生成 (n, t) 共享新的随机密钥 k' 和更新令牌 $\Delta = k / k'$ 。它使用来自多方计算的两个标准工具：(i) 共享 ρ_1, \dots, ρ_n 的 Shamir 联合生成。 ρ 是 \mathbb{Z}_q 上的均匀随机秘密的取值，例如 [45] 或 [23] 的图 7，以及 (ii) 一个分布式乘法协议，它给定秘密 a 和秘密 b 的共享，生成乘积 $a \cdot b$ ，没有学到任何关于任何秘密的信息，例如 [23]。

分布式更新协议假定 n 个服务器 S_1, \dots, S_n 有一个密钥 k 的共享 (k_1, \dots, k_n) 。为了产生一个新的密钥 k' ，服务器共同产生一个随机秘密 $\rho \in \mathbb{Z}_q$ 的共享 ρ_1, \dots, ρ_n ，并运行分布式乘法以生成新密钥的共享 k'_1, \dots, k'_n ， $k' = \rho \cdot k$ 。最后，每个服务器 S_i 向 C 和/或 StS 发送其共享 ρ_i ，接收者从中重建 ρ 并设置 $\Delta := \rho^{-1} [= k' / k]$ 。

5.2 可验证阈值(U)OKMS

如前所述，能够在加密对象之前验证数据加密密钥dek的正确性是OKMS解决方案的一个重要特征，也是其优于传统基于包装的KM系统的主要优势（图1）。OKMS可验证性[28]要求通过KmS检查正确的OPRF操作，假设客户端拥有与KmS密钥 k_c 相对应的真实公钥 g^{k_c} 。如第2.2节所述，UOKMS的可验证性可以通过正确的对称认证解密直接完成，因此无需检查KmS的不经意操作。然而，在阈值情况下，多个服务器提供解密输入，因此有必要识别行为不端的服务器。因此，在阈值情况下，UOKMS也需要可验证性。

注意，对于图3的单服务器dh-op方案，可以通过简单的非交互式零知识证明来添加可验证性。对于阈值情况，即图6的tdh-op方案，如果我们假设客户端拥有与密钥 $k = k_c$ 的份额 k_i 对应的公钥 g^{k_i} ，那么也可以使用零知识证明来进行验证。然而，这阻止了“代理”（例如，n个服务器中的任何一个）将服务器返回的 b_i 值聚合到OPRF结果中的能力。使用ZK验证时，客户端本身需要做这个聚合。这失去了tdh-op的“客户端透明”属性，它具有重要的实际优势，即客户端（及其软件）不需要知道服务器的实现，无论它是一个单一的服务器部署或多服务器部署。

接下来，我们提出了一种对客户端透明的替代验证程序。客户端只需要拥有密钥 k 的认证公钥 g^k （与服务器数量无关）。我们首先描述图3中单服务器OPRF dh-op情况下的方案，然后将其扩展到阈值情况。

（这直接适用于OKMS，也能够立即适应UOKMS）。该过程让人想起Chaum的不可否认签名协议[15]，但通过分配此处不需要的零知识证明进行了简化。很容易验证完整性保证是无条件的，即，针对无界攻击者。

- 对输入 x ，C设置 $h = H'(x)$ ， $r, c, d \leftarrow_R \mathbb{Z}_q$ ，然后将以下两对值 $a = h^r, b = h^c g^d$ 发送给服务器S。
- S回复 $A = a^k, B = b^k$
- C检查以下等式是否成立：

$$A^{r'} = B^{C'} v^{-dc'}(1)$$

其中 $r' = r^{-1}, c' = c^{-1}, v = g^k$ 。如果等式不成立，它将拒绝；否则C将 $(H'(x))^k$ 的值设为(1)中计算得到的 $A^{r'}$

Key and server initialization.

Key $k \leftarrow_R \mathbb{Z}_q$ is secret shared using Shamir's scheme with parameters n, t ;
Server $S_i, i = 1, \dots, n$, holds share k_i .

Threshold Oblivious Computation of $F_k(x)$.

- On input x , client C picks $r \leftarrow_R \mathbb{Z}_q$ and computes $a := H'(x)^r$; it chooses a subset \mathcal{SE} of $[n]$ of size $t + 1$ and sends to each server $S_i, i \in \mathcal{SE}$, the value a and the subset \mathcal{SE} .
- Upon receiving a from C, server S_i verifies that $a \in G$ and if so it responds with $b_i := a^{\lambda_i \cdot k_i}$ where λ_i is a Lagrange interpolation coefficient for index i and index set \mathcal{SE} .
- When C receives b_i from each server $S_i, i \in \mathcal{SE}$, C outputs as the result of $F_k(x)$ the value $H(x, (\prod_{i \in \mathcal{SE}} b_i)^{1/r})$.

Figure 6: Protocol tdh-op [30]: (n, t)-threshold computation of dh-op from Fig. 3

此过程涉及在两个不同的值上运行dh-op，然后通过客户端的单个多重指数来验证一致性。相对于基础dh-op，该方案的额外计算成本是服务器的一次指数运算和客户端的两次多次指数运算，本质上是未验证情况下的工作量加倍。

我们现在使方案适应阈值OPRF tdh-op。客户端C向每个参与者服务器 S_i 发送相同的值对(a, b)， S_i 回复 $A_i = a^{k_i}, B_i = b^{k_i}$ 。在收集到 $t+1$ 个响应后，C插入指数（一次乘幂）以获得值A、B并检查恒等式(1)。如果它成立，C将 $(H'(x))^k$ 设置为 $A^{r'}$ ，否则它将(1)应用于参与服务器 S_i 接收的每对 A_i, B_i ，使用 $v_i = g^{k_i}$ 而不是v。

正常情况下的计算成本（即对 $v = g^k$ 验证成功时）与单服务器情况下相同，除了指数中的一个额外插值。如果针对 $v = g^k$ 的验证失败，则成本是每个参与服务器的额外乘幂。如前所述，此过程的特殊功能是客户端可以与代理（或网关）交互，客户端的所有操作都与单服务器情况相同。代理会将客户端生成的值(a, b)发送到服务器，并将响应 A_i, B_i 聚合成一个可以用公钥 $v = g^k$ 验证的响应。在发送给客户

端之前，代理可以验证聚合是否正确。如果不是，它需要检查每个服务器发送的各个值并丢弃错误的值——所有这些都是在客户端不知情的情况下完成的，从而产生完全对客户透明的解决方案。

6 实现与表现

我们将分别报告在2.1节的图2中介绍的OKMS和2.2节的图4中介绍的UOKMS方案的实现及其具体表现。

OKMS Client Operations (Single Thread)		
	Wrap	Unwrap
Hash to Curve	58.26	58.26
Generate Blind	1.58	1.58
Apply Blind	68.07	68.07
Create Challenge	83.27	-
Inverse Blind	16.95	16.95
Remove Blind	68.07	68.07
Verify Response	106.67	-
Total Time (μ s)	402.86	212.92
Operations / Second	2,482	4,696

Figure 7: Client operation time and Op/s in OKMS

微基准。所有必要的客户端和服务器操作的实现都是使用OpenSSL 库（版本 1.1.1-pre5），用 C++ 编写的，以提供加密功能。性能测试是在配备 Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz 和15GB内存的机器上进行的。该实现使用优化级别3的gcc编译器编译的。

下表详细说明了每项操作平均超过10,000次试验的运行时间。这些测试仅使用单线程和CPU内核；通过多个CPU内核并发执行这些操作可以改善结果。

在这些测试中，所有椭圆曲线操作均基于NIST P-256。现场操作（针对Shamir和致盲因子）是根据NIST P-256的素数顺序定义的。使用SHA-256和恒定时间简化SWU算法[12]对曲线进行散列（根据图3中定义的OPRF的要求）。

OKMS方案中的客户端操作如图7所示，包括加密（“包装”）和解密（“展开”）。这两个操作的不同之处仅在于为包装操作执行交互式验证，而对于解包（更常见的操作），常规对称密钥验证就足够了。

当(U)OKMS服务器部署在阈值架构中时，某些实体必须“在指数中”执行多项式插值。这可以实现为单个服务器贡献的多重指数以及相应的拉格朗日系数。这种插值操作可以由服务器之一、客户端或专用中间实体以不同的方式执行。

Interpolation Layer Performance (Single Thread)		
(t+1)-of-N	Time (μ s)	Ops / Second
1-of-1	81.68	12,242
3-of-5	155.99	6,410
5-of-9	236.72	4,224
5-of-15	236.66	4,225
6-of-11	280.04	3,570

Figure 8: Interpolation layer performance for various threshold parameters

(U)OKMS Server Operations (Single Thread)		
	Time (μ s)	Ops / Second
Wrap	136.13	7,345
Unwrap	68.07	14,691

Figure 9: (U)OKMS Server performance for wrap and unwrap

UOKMS Operations (Single Thread)		
	Time (μ s)	Operations / Second
Wrap	24.24	41,261
Unwrap	162.33	6,160
Update	68.07	14,691

Figure 10: Client operation time and Op/s in UOKMS

我们观察到多指教时间主要影响插值的成本（计算拉格朗日系数的成本相对较低），总成本取决于阈值而不是服务器总数。我们在图8中报告了插值时间。

图9显示了针对单个服务器测量的服务器操作和此(U)OKMS方案（使用协议tdh-pop实现）的阈值变体。这包括仅在曲线中为每个输入执行取幂（EC标量乘法），输入由客户提供（包装操作成本更高，因为它包括额外的求幂以支持交互式验证5.2节中的程序）。

对于阈值情况，总时间和每秒操作数没有显著差异，因为每个涉及的服务器都并行计算相同的函数。

UOKMS方案（第2.2节）中的客户端性能如图10所示。此设置受益于能够在不涉及服务器的情况下执行包装操作，并且可以进一步受益于用于 g 和 g^k 取幂的预算表。在我们的测试中，预算表提供了超过600%的加速（11.33与68.20 μ s）。我们总结了客户端在UOKMS中每秒可以执行的操作数。

(U)OKMS服务器。为了评估性能和可扩展性，我们使用c4.2xlarge实例类型在亚马逊的弹性计算云(EC2) [3] 上托管我们的(U)OKMS服务器实现。该实例提供了8个虚拟CPU，配置为Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz，具备15GB内存，与上文中微基准的类型一致。

对该服务器的请求是通过HTTP发出的，Web服务器nginx配置了8个工作进程（每个CPU一个）。OKMS功能已作为本地编译模块添加到此Web服务器，该模块使用OpenSSL库（版本1.1.1-pre5）提供加密功能。服务器运行Ubuntu 16.04作为其操作系统。

(U)OKMS Server Throughput (8 CPU cores)		
Scheme	KeepAlive	No KeepAlive
Static Page	65,018	6,462
OPRF (Unwrap)	32,094	6,349

Figure 11: (U)OKMS Server Requests/s on EC2 instance (LAN setting)

吞吐量。为了测量吞吐量，将客户端计算机（也是c4.2xlarge）部署在与服务器相同的Amazon Web Service (AWS) 可用性区域中。我们使用HTTP负载生成工具hey测量每个方案的吞吐量。并发级别配置为80，所有结果均是超过50,000个请求的平均值。所有请求都是针对解包操作的，并使用（带ECDHE-ECDSA-AES256-GCM-SHA384的TLS 1.2）通过HTTPS发送。服务器在NIST P-256曲线上使用带有EC密钥的自签名证书。由于网络延迟几乎可以忽略不计，计算时间在LAN设置中占主导地位，在LAN吞吐量测试期间，CPU内核的利用率接近100%。为了衡量服务器性能的限制，负载生成器不执行客户端的致盲和验证操作。

对于每个方案，我们都在打开和关闭会话KeepAlive的情况下进行了测试。关闭时，必须为每个请求协商新的TCP连接和TLS会话。启用时，连接设置成本会分摊到所有请求上，这与必须解包许多密钥的客户端一致。

图11中的表格详细说明了各种方案和两个KeepAlive配置（全部通过TLS）的每秒请求数(RPS)中观察到的吞吐量。我们将这些方案与静态页面作为基线进行比较。

我们观察到，对于No KeepAlive配置，创建新连接和建立TLS会话的成本占主导地位，导致方案之间的RPS差异很小。对于KeepAlive配置，吞吐量明显更好，在OPRF/T-OPRF情况下达到超过30,000 RPS。因此，我们的(U)OKMS实现可以使用单个服务器处理大量客户端。相比之下，亚马逊的对象存储服务报告的峰值负载为每秒110万个请求[33]。如果需要，可以使用标准技术扩展KMS实施，例如部署更多服务器。KMS中有几十台服务器，每次向亚马逊的服务写入或读取对象时，都可以提供一个唯一的密钥。

硬件安全模块。保护主密钥的最佳做法是将它们保存在硬件安全模块(HSM)[4]中，以防止将它们导出到安全性较差的位置。幸运的是，本文中描述的方法得到了现有商业HSM的支持。实际上，大多数HSM都支持PKCS#11标准[42]。该规范定义了一个名为CKM_ECDH1_DERIVE的API方法，该方法将任意点作为输入并返回该点的x坐标，该点是使用HSM持有的私钥作为标量对输入点进行标量乘法得到的。

我们测试了三个HSM实现，发现它们都支持ECDH1派生方法。返回的x坐标足以执行不经意的密钥推导和验证，但验证（仅在OKMS设置和OPRF的阈值实现中需要）需要检查y坐标的正解和负解，通过导入作为Shamir分享的椭圆曲线私钥，现有的HSM可以用作阈值实现的一部分。由于不经意，可以在不牺牲机密性的情况下在HSM外部对HSM阈值的结果进行插值。

我们注意到使用HSM保存OPRF密钥的两个潜在限制。首先是HSM通常受限于它们支持的曲线。虽然我们评估的所有HSM都支持标准NIST曲线，但没有一个支持Curve25519。第二个限制是性能。虽然高端商业HSM每秒可以实现高达22,000次标量乘法[48]，但这大致相当于单核在多核服务器CPU中才能实现的。

虽然对称密钥包装算法的软件实现比非对称操作快几个数量级，但我们发现HSM通常使用专门的硬件来加速正常情况下较慢的非对称操作。在某些情况下，HSM[48]（包括领先的云提供商[38]使用的HSM）每秒支持的ECC操作数与每秒支持的对称加密操作数相当。

总之，虽然在传统的密钥包装方法（图1）中，人们可以努力保护包装密钥，但例如，在 HSM 中，纯数据加密密钥 (dek) 通过安全性低得多的 TLS 通道（有时在HSM边界之外的多个点结束或可见），并可能暴露给腐败管理员、意外日志等。相比之下，这些漏洞通过不经意的计算方法得以消除，只要OPRF密钥是安全的，就无法了解有关数据的任何信息（除非破坏客户端）。幸运的是，如上所述，在 HSM 中保护这些 OPRF 密钥在今天是实用的，虽然对称操作通常比OPRF的成本更低，但能实现每秒20,000次ECC的HSM几乎不会成为系统的瓶颈（当然，在大型操作中会使用多个HSM）。重要的是，在UOKMS中加密数据不需要与KMS交互，从而进一步提高了性能。此外，与传统系统相比，UOKMS方法提供更高效的密钥轮换。在传统系统中，轮换需要与KMS通信以更新每个密钥（dek 或 kek）。这会减慢轮换过程，导致轮换周期延长并降低安全性。

Updatable Oblivious Key Management for Storage Systems

Stanislaw Jarecki
University of California, Irvine

Hugo Krawczyk
Algorand Foundation

Jason Resch
Independent

ABSTRACT

We introduce Oblivious Key Management Systems (KMS) as a much more secure alternative to traditional wrapping-based KMS that form the backbone of key management in large-scale data storage deployments. The new system, that builds on Oblivious Pseudorandom Functions (OPRF), hides keys and object identifiers from the KMS, offers unconditional security for key transport, provides key verifiability, reduces storage, and more. Further, we show how to provide all these features in a distributed threshold implementation that enhances protection against server compromise.

We extend this system with *updatable encryption* capability that supports key updates (known as key rotation) so that upon the periodic change of OPRF keys by the KMS server, a very efficient update procedure allows a client of the KMS service to *non-interactively* update all its encrypted data to be decryptable only by the new key. This enhances security with forward and post-compromise security, namely, security against future and past compromises, respectively, of the client's OPRF keys held by the KMS. Additionally, and in contrast to traditional KMS, our solution supports public key encryption and dispenses with any interaction with the KMS for data encryption (only decryption by the client requires such communication).

Our solutions build on recent work on updatable encryption but with significant enhancements applicable to the remote KMS setting. In addition to the critical security improvements, our designs are highly efficient and ready for use in practice. We report on experimental implementation and performance.

CCS CONCEPTS

- Security and privacy → Key management.

KEYWORDS

key management, updatable encryption, Oblivious PRF, OPRF

ACM Reference Format:

Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. 2019. Updatable Oblivious Key Management for Storage Systems. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, November 11–15, 2019, London, United Kingdom. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3319535.3363196>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '19, November 11–15, 2019, London, United Kingdom

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6747-9/19/11...\$15.00

<https://doi.org/10.1145/3319535.3363196>

1 INTRODUCTION

The ever expanding cloud storage infrastructure is one of the pillars of modern computing. Yet, the key management systems (KMS) provisioning keys for the protection of the stored data have not changed fundamentally in decades. This setting involves three separate parties: a *client C*, a remote *storage server StS* (e.g., a cloud service) that stores client data in encrypted form, and a *key management server KmS* that stores cryptographic keys for the client. The client uses the services of KmS each time it needs to encrypt or decrypt the data. The idea is that KmS is better equipped to keep keys secret and StS is better equipped to store large amounts of data reliably. Thus, KmS is charged with protecting secrecy and StS with protecting availability.

The typical deployment of such systems in practice (including large cloud-based operations such as AWS [2], Microsoft [39], IBM [27], Google [24]) uses the traditional *wrap-unwrap approach* for managing data encryption keys (dek) as shown in Fig. 1. When client *C* needs to encrypt a data object, it chooses a symmetric key dek with which it encrypts the object, then sends dek to key management server KmS who *wraps* (i.e., encrypts) dek under a client-specific (master) key k_c stored at KmS and returns the result, called a *wrap*, to *C*. Finally, *C* stores wrap and the data encrypted under dek at the storage server StS. When *C* needs to retrieve an object, it gets the corresponding ciphertext from StS, sends the attached wrap to KmS who *unwraps* (i.e., decrypts) it using k_c and sends dek back to *C*, who uses it for decryption.

This key encapsulation mechanism, while effective and widely deployed, presents significant potential vulnerabilities. First, encryption keys dek are exposed in the clear to KmS. Second, the security of dek, hence the security of all encrypted data, relies on the channel between the client and KmS. Such a channel, typically implemented by TLS, is vulnerable to a large class of attacks, from implementation and configuration errors to certification and man-in-the-middle attacks. Third, even in normal operation, the key dek is visible to any middlebox and endpoint where TLS traffic is decrypted. Additionally, KmS can trace objects being encrypted/decrypted via the wrap values. A further shortcoming is the cost of rotating a client key by KmS: Changing the value k_c for a new k_c' requires the client (or StS) sending *each* wrap to KmS for unwrapping under k_c and re-wrapping under k_c' . This is not only a performance issue but a security one too (due to long period of time till all wraps are updated and till k_c can be safely erased).

Oblivious KMS. Our first contribution is a simple approach to key management based on *Oblivious Pseudorandom Functions (OPRF)* [22, 29, 41], that addresses the above vulnerabilities and offers additional features absent in traditional systems. OPRFs are interactive schemes between a server holding a key to a PRF and a client holding an input. At the end of the interaction the client learns the output of the PRF on its input and the server learns nothing (neither the input nor the output of the function). OPRFs have

Parties: key management server KmS, storage server StS, client C (= data owner).
Functions: Symmetric authenticated encryption scheme Enc; wrapping functions Wrap, Unwrap (used to encrypt/decrypt data encryption keys).
Keys: KmS stores a client-specific wrapping key k_c for each client.
Encryption of object (ObjId, Obj) by client C:
(1) C chooses random Enc key dek (data encryption key); (2) C sends (ObjId, dek) to KmS; (3) KmS returns (ObjId, wrap = Wrap $_{k_c}$ (dek)) (Note: KmS authenticates C before using k_c); (4) C sends (ObjId, wrap, Enc $_{\text{dek}}$ (Obj)) to StS for storage.
Decryption of object ObjId by C:
(1) C retrieves (ObjId, wrap, Enc $_{\text{dek}}$ (Obj)) from StS; (2) C sends (ObjId, wrap) to KmS; (3) KmS returns (ObjId, dek = Unwrap $_{k_c}$ (wrap)) (4) C decrypts Enc $_{\text{dek}}$ (Obj) using dek.

Figure 1: Traditional Wrapping-based Key Management

found numerous applications and there are very efficient OPRF implementations, e.g. based on the Diffie-Hellman (DH) problem in regular elliptic curve groups [16, 21, 26, 40, 47] (see Fig. 3).

In our *Oblivious Key Management System* (OKMS) (see Fig. 2), a client C who requires a data encryption key dek for encrypting a data object interacts with the OKMS server in an OPRF protocol. C 's input is an identifier for the data object while the server's input is an OPRF key (typically unique per client and denoted k_c), and C uses the output from the OPRF as the dek¹. In this way, the OKMS server does not learn dek (or even the object identifier). The system does not rely on an external secure channel (e.g., TLS) to transport dek; instead dek is protected by the security properties of the OPRF.²

This addresses two major vulnerabilities of traditional KMS systems: visibility of the dek to the server and potential exposure of this key in transit between client and server. Moreover, using the most efficient DH-based implementations of OPRFs, the protection against these threats is *unconditional*. Even a computationally unbounded server (that knows the OPRF key) or a network eavesdropper cannot learn anything about the dek, or about the object identifier input into the OPRF. Note that in OKMS, the only way for an adversary to decrypt a ciphertext is by impersonating the legitimate client or by learning the OPRF key k_c and the corresponding ObjId value. In contrast, in traditional systems, data encryption keys dek are potentially vulnerable even if the KmS key is well protected (e.g., inside a hardware module) as the dek are transmitted outside the protected zone.

¹ Alternatively, the output of the OPRF can be used as a key-encrypting key (kek) to locally encrypt dek.

² A TLS connection can be used to transport auxiliary information or client credentials but is not needed for transporting data encryption keys.

The OPRF approach supports additional properties that enhance security even further and beyond anything offered by the traditional solutions. First, it provides verifiability, namely, the ability of KmS to prove to C that the returned dek is indeed the value that results from computing the OPRF on the client-provided object identifier. This prevents data loss that occurs if the returned dek is wrong (either due to computing error or to adversarial action); indeed, encrypting data with an incorrect, or irrecoverable, key can lead to irreparable data loss. Second, the DH-based OPRF, hence also the OKMS using it, is amenable to distribution as a multi-server threshold scheme where the OPRF key is protected as long as less than a defined threshold of the servers is corrupted. Finally, the described system can be adapted to also support *updatability*, namely, periodic key rotation of the client master key k_c by KmS with a very efficient (non-interactive) procedure for updating ciphertexts to be decryptable by the new key and not by previous ones. This procedure does not endanger the secrecy of the data and therefore can be performed by the StS. The design of such system is the main technical contribution of our work and is discussed next.

Updatable Oblivious KMS. Traditional wrapping-based key management systems as those described above (and in Fig. 1) require client keys k_c to be *updated periodically* by the server KmS. Such update, known as *key rotation*, is needed to limit the exposure of data upon the exposure of k_c . For traditional wrapping systems, changing k_c with a new k_c' involves *unwrapping and re-wrapping all of a client's ciphertexts* as well as transmitting all these wrap values between the storage server and KMS server. Moreover, an old key k_c cannot be erased until *all* ciphertexts are updated to the new key k_c' , extending the exposure period of k_c significantly.

This need to update clients' keys in storage systems (and other applications) has led to the notion of *updatable encryption* [9] whose goal is to provide more efficient and more secure solutions to this key rotation problem. Many flavors of updatable encryption have been suggested [9, 10, 20, 36]. In this work we investigate this notion in the context of our oblivious KMS approach leading to the design of an *Updatable Oblivious KMS (UOKMS)*.

In UOKMS, upon the rotation of a client's key k_c , server KmS computes a *short update token* Δ as a function of the old and new keys k_c, k_c' , and transmits Δ to client C . Using Δ , C 's storage server StS can transform *all* ciphertexts that were encrypted with keys derived from k_c into ciphertexts decryptable by the new k_c' but not by the old k_c . This operation preserves the security of the data, it is performed locally at the storage server StS *without any interaction with KmS*, and it only modifies a short component of the ciphertext (independent of the length of the encrypted data) making the whole operation highly efficient. Security-wise it protects against future and past compromises of the client's key k_c .

The above UOKMS scheme offers another major performance advantage compared to traditional KMS and our own OKMS scheme: Encryption of data requires *no interaction* with the KMS server, and an interaction is only needed to decrypt data. More generally, our UOKMS supports public key encryption, so everyone can encrypt data for client C , but only C can decrypt it, via an interaction with the KMS server.

Threshold Updatable OKMS. Both OKMS and UOKMS solutions can be implemented via distributed servers so that clients' OPRF

keys are secure for as long as no more than a threshold number of servers are compromised. These systems inherit the high efficiency of Threshold OPRF constructions [30] (also in the case of the OPRF variant used in the UOKMS solution). In the UOKMS setting, the update token Δ is computed distributively among the servers through an efficient multi-party computation. These solutions preserve the verifiability property of OPRFs and they can be implemented in a *client-transparent* way, namely, the client's operations and code are identical regardless of the implementation as a single-server or multi-server. See Section 5.

Formal model and analysis. We formally analyze our UOKMS solution in an Updatable Oblivious KMS security model that shares close similarities with recent models of updatable encryption (or encryption with *key rotation*) [9, 10, 20, 36], but also has some significant differences. One crucial difference comes from the key management setting treated here where the client interacts with *two outsourced remote services* KmS and StS. In particular, this raises potential security vulnerabilities arising from the communication channel between client and KmS. This major concern is absent from previous updatable encryption models that treat the client and KmS essentially as collocated entities. The other aspect that is unique to our solution and formal treatment is the obliviousness of computation on the side of KmS. Yet another difference is that while the typical storage setting does not require public key encryption, we naturally include this setting in our updatable model.

Our updatability model allows attacks on both KmS and StS, including exposure of client keys k_c , update values Δ , and the attacker's ability to see and write ciphertexts into StS. Security is provided against future and past attacks, namely, forward and post-corruption security, with a simulation-based security model. Obviously, the model disallows attack combinations that would lead to trivial wins for the attacker (e.g., decrypting a challenge ciphertext in a period for which it learns the KMS key k_c). The model accommodates the oblivious setting where an attacker that communicates with KmS (and is in possession of C 's credentials) can decrypt *any* q ciphertexts after q interactions with KmS, but all other ciphertexts remain secure. This, together with the attacker's capability to access a ciphertext-update oracle and the use of authenticated encryption, achieves CCA-like security for oblivious and updatable encryption. The security proof for our UOKMS scheme, presented in Section 4, carries in the random oracle model under a strengthened variant of the Gap One-More Diffie-Hellman assumption [5, 32] that we show to hold in the generic group model.

Implementation and performance. In Section 6 we present performance information from our implementation of both OKMS and UOKMS solutions showing the practicality of our techniques, in particular the ability of servers to support a large number of operations and clients per second. In OKMS, client time is approximately 0.4 msec for a wrap and 0.2 msec for an unwrap. For the UOKMS system, performance is even better: a client can sustain over 41000/6000/14000 for wrap/unwrap/update operations per second respectively, with a single-thread and single CPU core, and server operations are only needed for unwrapping. We also demonstrate good throughput and latency results from a prototype implementation of the (U)OKMS Server deployed to an Amazon EC2 instance. We find this implementation capable of answering over

30,000 requests per second in both single-server and multi-server deployments. Finally, we discuss implementation experience managing KmS keys in Hardware Security Modules (HSM).

1.1 Comparison to previous work

We are the first to present a comprehensive updatable solution to the central problem of key management in cloud-based (and other) storage systems that exploits the power of oblivious computation, and the first to develop a security model for such setting. Our motivation and modeling bear similarities with recent models of Updatable Encryption (UE) [9, 10, 20, 34, 36], but also has some significant differences. Most prominent is the use of obliviousness as a way to address potential vulnerabilities arising from a *remote* key management system, as opposed to one that is collocated with the client as was assumed in all the above works on updatable encryption. Other novel features of our solution include unconditional hiding of data encryption keys and object identifiers from KmS, and building a distributed UOKMS service via a threshold implementation. Our updatability solution is ciphertext-independent (namely, the update token is of size independent from the number of ciphertexts and size of data to be updated) as in several prior UE schemes [9, 20, 34, 36]. Among those, our scheme is the most efficient, requiring a single short update value Δ from the KmS server and a single exponentiation per object for the update operation, compared e.g. to two exponentiations *per ciphertext block* in the schemes of [34, 36]. Our UOKMS scheme can be extended to provide ciphertext indistinguishability and unlinkability similarly to e.g. [34, 36], but it would inherit the inefficiency of such solution making it impractical in any large-scale data storage deployment.³ Finally, our model and solution are the first to support public key encryption, including CCA-like security in the setting of oblivious encryption. We elaborate further on the relation to prior updatable encryption work in Section 3.2.

Updatable encryption is closely related to *proxy re-encryption (PRE)*, in particular, the Diffie-Hellman techniques at the center of our implementation directly relate to the PRE scheme of Blaze et al. [7]. Recently, [17, 18] treat forward secrecy and post-corruption security in the context of PRE for which they define evolutionary keys as in our context. However, the requirements of PRE, particularly as set forth in [17], are more stringent than needed in our case. These include generating update values using the delegatee's public key rather than on input its secret key, achieving unidirectionality, supporting general DAG delegation graphs, ensuring ciphertext indistinguishability, and more. As a result, they require more involved and less efficient techniques; in particular, [18] builds on pairing-based constructions and HIBE [8, 14] while [17] uses lattice-based fully-homomorphic techniques from [11, 46]. On the other hand, in spite of their stronger properties, none of these schemes support oblivious computation.

Our use of OPRF function can be seen as an "OPRF-as-a-service" application, a term coined in [19]. We borrow the notion of updatable oblivious PRF from that work, but their application was

³ Several prior works, e.g. [34, 36], consider ciphertext unlinkability (over update periods) as a major design goal, but achieve it at the cost of requiring $\tilde{O}(n)$ exponentiations to update a ciphertext of length n . We believe that in most practical settings, linkability would still be possible via metadata, object identifiers, etc., hence not worth the high computational cost it entails.

targeted to password verification protocols, while ours is a general encrypted storage system. (Moreover, the protocol of [19] is significantly less efficient as it uses groups with bilinear maps to obtain the stronger notion of updatable “partially oblivious” PRF, which we do not require.) OPRF’s are also used in “password-protected secret sharing” [28] which can implement distributed password-secured storage but without the ability to update the master encryption key. Moreover, both of these solutions are specialized for password-authenticated clients while UOKMS accommodates any client-to-KMS or client-to-StS authentication mechanisms.

Comparison to U-PHE. The goals of UOKMS bear some similarity to Updatable Password-Hardened Encryption (U-PHE) of [35]. In the U-PHE setting a server S stores encrypted data on behalf of its clients. The encryption and decryption of data *require* S to hold the client’s password and involve an interaction of S with an additional server R , called the *rate limiter*. In particular, an attacker who learns S ’s state (*but not the stored client password*), cannot decrypt client’s data without guessing the client’s password and interacting with the rate limiter R . The solution offers verifiability and updatability similarly to our case, and in terms of our UOKMS model one can think of S as the storage server StS and R as the key management server KmS. However, in contrast to UOKMS, in U-PHE the server S learns both the client’s decrypted message and the client’s password (in particular, one relies on TLS for transmitting the password), while in UOKMS only the client encrypts and decrypts data and neither server learns it. Moreover, the U-PHE decryption protocol is not oblivious, i.e. server R , i.e., KmS, can identify the decrypted ciphertext. Also, as in the case of [19, 28] above, PHE is specialized to the password authentication case, while UOKMS is independent of the means of authentication used by clients, allowing any form of client authentication credentials. Additionally, the U-PHE scheme of [35] is less efficient than our UOKMS, specifically their encryption is interactive while ours is not, their decryption and update are both roughly twice more expensive than ours, and a threshold implementation of the rate-limiter server of [35] would be significantly more expensive than our threshold KmS.

2 UPDATABLE OBLIVIOUS KMS

We present our main scheme, UOKMS (for Updatable Oblivious KMS), that builds on the general approach to Oblivious KMS described in the introduction and recalled next.

2.1 Oblivious Key Management System

Figure 2 specifies the Oblivious KMS (OKMS) protocol that serves as a basis for our Updatable scheme in the next section. OKMS is described and motivated in the Introduction as a much more secure alternative to the wrapping-based approach (Fig. 1) in wide use today in storage systems, particularly in large cloud deployments. When implemented with the DH-based OPRF scheme dh-op from Fig. 3, one obtains an OKMS that is highly efficient (see Sec. 6) and accommodates extensions to verifiability and distributed implementation (Sec. 5). The security of the OKMS scheme and its implementation using dh-op follows from the OPRF properties (in particular as studied in [28, 29]). We do not formally analyze the OKMS scheme but rather do so in Sections 3 and 4 for its extension

Functions: OPRF F and symmetric authenticated encryption scheme Enc.

OPRF Keys: KmS stores a client-specific OPRF key k_c for each client.

Encryption of object Obj by client C: C runs OPRF protocol with KmS where C inputs object identifier ObjId and KmS inputs key k_c . C sets dek = $F_{k_c}(\text{ObjId})$ and stores the pair (ObjId, Enc_{dek}(Obj)) at storage server StS.

Decryption of encrypted object ObjId by client C: As in the encryption case, C interacts with KmS to compute dek = $F_{k_c}(\text{ObjId})$ and decrypts Obj using dek.

Verification of correct computation of dek: Use a verifiable OPRF [28].

Figure 2: Oblivious KMS (OKMS)

Components: G : group of prime order q ; H, H' : hash functions with ranges $\{0, 1\}^\ell$ and G , respectively, where ℓ is a security parameter.

PRF F_k Definition: For key $k \leftarrow_R \mathbb{Z}_q$ and $x \in \{0, 1\}^*$, define

$$F_k(x) = H(x, (H'(x))^k)$$

Oblivious F_k Evaluation between client C and server S

- (1) On input x , C picks $r \leftarrow_R \mathbb{Z}_q$; sends $a = (H'(x))^r$ to S .
- (2) S checks that the received a is in group G and if so it responds with $b = a^k$.
- (3) C outputs $F_k(x) = H(x, b^{1/r})$.

Figure 3: DH-based OPRF function dh-op [29]

to the Updatable OKMS setting presented next. (A model and analysis of OKMS can be obtained by specializing the UOKMS model to a single update period.)

2.2 Updatable OKMS

Key management systems are required, by regulations and best practices, to periodically update client keys k_c (an operation known as *key rotation*). The goal is to limit the negative effects of the compromise of a key k_c to a shorter period of time and to as little data as possible. This is particularly important for keys that protect data stored for long periods of time as it is common in many cloud storage applications (anything from user photos to regulated financial information). Upon the rotation by the KMS server KmS of a key k_c into a new key k_c' , all ciphertexts protected with k_c and held by the storage server StS need to be updated too. The updated ciphertexts should be decryptable by k_c' but not by k_c . The goal is that an attacker that learns k_c but only sees updated ciphertexts should not be able to learn anything about the encrypted data in the new period (while k_c' is unexposed). Similarly if the attacker has seen a ciphertext encrypted using an unexposed k_c and later learns k_c' , it still should not learn anything from that ciphertext. This provides both *forward security* (security against future exposures) and post-compromise security (security against past exposures). Obviously,

Setting: Generator g of group G of prime order q ; symmetric authenticated encryption scheme Enc, Dec with keys of length security parameter ℓ ; hash function $H : G \rightarrow \{0, 1\}^\ell$.

Client keys: KMS server KmS stores a client-specific random key $k_c \in \mathbb{Z}_q$ for each client; storage server StS stores certified public value $y_c = g^{k_c}$ for client C .

Encryption of object Obj: To encrypt Obj under key y_c , pick $r \leftarrow_{\mathbb{R}} \mathbb{Z}_q \setminus \{0\}$, set $w = g^r$ and $\text{dek} = H(y_c^r)$, and output ciphertext triple $c = (\text{ObjId}, w, \text{Enc}_{\text{dek}}(\text{Obj}))$.

Decryption of ciphertext $c = (\text{ObjId}, w, e)$: (1) C sends $u = w^{r'}$ for $r' \leftarrow_{\mathbb{R}} \mathbb{Z}_q$ to KmS; (2) KmS checks if $u \in G$ and if so returns $v = u^{k_c}$ to C ; (3) C outputs $\text{Obj} = \text{Dec}_{\text{dek}}(e)$.

(Note that C runs the decryption protocol only if c is *valid* and we define $c = (\text{ObjId}, w, e)$ as valid iff $w \in G$ and $w \neq 1$.)

Key rotation and update: To change client's key from k_c to k'_c , KmS sends $\Delta = k_c/k'_c$ and $y'_c = g^{k'_c}$ to StS. StS replaces y_c with y'_c and replaces each ciphertext $c = (\text{ObjId}, w, e)$ with $c' = (\text{ObjId}, w' = w^\Delta, e)$, provided that $w \in G$. (Element $w \notin G$ indicates an invalid ciphertext which can be removed.)

Figure 4: Updatable Oblivious KMS Scheme

one also requires that the update process itself does not reveal encrypted information to StS (e.g., decrypting and re-encrypting the data by StS would not be considered secure).

In the traditional wrapping-based KMS of Fig. 1, such key rotation operation requires interaction between the storage server StS and KMS server KmS where StS sends *every* stored wrap to KmS for unwrapping under k_c and re-wrapping using k'_c . This requires the transmission of all wrap values between StS and KmS, and the exposure of all dek values to KmS. In a large storage setting such process can take *very long time* (particularly under the “lazy evaluation” practice where a wrap held by StS is updated to k'_c only when the application requires a regular unwrap operation for that object). During all this time the old and new keys k_c, k'_c must be stored at KmS thus extending the life and exposure period of these keys.

In Fig. 4 we present an *Updatable Oblivious KMS* that adapts the OKMS scheme from the previous section to the updatable setting. Using techniques from *updatable encryption* [9, 20, 36] adapted to the oblivious setting, we achieve some desirable properties, both in terms of security and performance. First, upon the change of key k_c into a new key k'_c , KMS server KmS can produce a *short token* Δ with which *all* the ciphertexts of client C can be updated by StS in a way that achieves the above security properties. Second, the update operation is *non-interactive*: It is performed locally by StS with the sole possession of Δ . Note that once KmS produces a new key k'_c and the corresponding update value Δ , KmS can immediately erase the old key k_c , hence reducing the risk of exposure to only one key at a time. Finally, the update operation at StS only requires a single exponentiation per ciphertext independently of the ciphertext size, compared to at least 2 exponentiations per ciphertext in previous updatable encryption schemes (see also footnote 3), leading to a fast update of all ciphertexts that were encrypted under k_c . Thus, one obtains a very efficient update procedure that achieves better security than in the wrapping-based KMS in many ways: dek keys are never exposed to StS or to KmS during updates; old keys can be erased immediately upon rotation; the interaction between StS and KmS is minimal (only Δ is transmitted); and Δ can be erased by StS as soon as it locally updates all ciphertexts.

The UOKMS scheme from Fig. 4 departs from the OKMS scheme of Fig. 2 in some important ways. First, to allow for fast updates, ciphertexts are composed of two parts, a wrap and a symmetrically encrypted ciphertext that derives the encryption key from wrap. For updates, only wrap is updated. Second, the *encryption operation is non-interactive*, that is, C (or anyone else) can encrypt data locally without interacting with KmS provided that it possesses the equivalent of a certified “public key” y_c corresponding to k_c ($y_c = g^{k_c}$ in our scheme). Decryption is only possible via an oblivious interaction with KmS. As a “side effect” of the above properties, the UOKMS scheme supports public key encryption, meaning that anyone can produce ciphertexts but only C can decrypt them, thus expanding the use cases for such KMS solution. Note that decryption requires interaction with KmS which we assume has the means to authenticate decryption requests from C . Third, the UOKMS scheme from Fig. 4 is presented in terms of a specific instantiation rather than using generic tools like the OPRF in OKMS. Indeed, the malleability properties required for the update operations are not possible with a generic OPRF (but see below about Weak OPRFs). Finally, verifiability of correct encryption by KmS is not needed in UOKMS where encryption is non-interactive, and verification of correct decryption can be done via the (symmetric) authenticated decryption operation Dec. This saves the need to verify the correct exponentiation by KmS, further improving the performance of UOKMS.

Correctness of the UOKMS scheme is easy to validate. For encryption, one sets $w = g^r$ for random r , then derives the encryption key dek from y_c^r , encrypts the data and stores w . For decryption, C computes w^{k_c} obliviously in interaction with KmS and derives dek from this value. This recovers the original data as $y_c^r = (g^{k_c})^r = (g^r)^{k_c} = w^{k_c}$. Regarding the update operation, if we denote by w_t and k_t the values of w and k_c , respectively, after t updates (here w_0 denotes the original value of w computed at the time of deriving dek , and k_0 denotes the client's key k_c as it existed at that time), then one can see inductively that if $w_t^{k_t} = w_0^{k_0}$ (the latter is the value from which dek is derived), then this is also true for $t+1$, namely, $(w_{t+1})^{k_{t+1}} = (w_0)^{k_0}$. Indeed, we have that $w_{t+1} = w_t^{\Delta_{t+1}} = w_t^{k_t/k_{t+1}}$, thus $(w_{t+1})^{k_{t+1}} = (w_t^{k_t/k_{t+1}})^{k_{t+1}} = w_t^{k_t} = w_0^{k_0}$.

Security of the UOKMS scheme from Fig. 4 is proven in Section 4 based on the security model presented in Section 3.

In Section 5 we show how to distribute the KmS functionality of UOKMS through a threshold scheme which includes the distributed generation of the value Δ so only StS can learn it.

On Weak Oblivious PRF. The UOKMS scheme from Fig. 4 derives symmetric encryption keys from a function $F_k(w) = H(w^k)$ defined over elements in a group G of prime order q (where the key k is chosen at random in the set \mathbb{Z}_q). The function F has strong similarities with the OPRF $\text{dh-op}_k(x) = H(x, (H'(x))^k)$ from Fig. 3 that we use as the basis of the OKMS scheme from Fig. 2, as well as some fundamental differences. First, the input to F is a group element (rather than an arbitrary string mapped into the group by the hash function H' in dh-op). But more importantly, knowing w^k for any value w allows to compute the function on w^t for known t . At the same time, computing F on a independently random group element is hard under CDH hence F can be modeled as a Weak PRF (as noted in [40]). In our application for UOKMS we also use the fact that F can be computed obliviously and use its homomorphic properties to support updatability. We leave as a future work item the formalization of such “oblivious Weak OPRF” function in the UC model, similarly to the treatment of OPRFs in [29]. For the purpose of our use of F in the context of UOKMS, we carry the analysis directly in a specialized UOKMS security model that we present in Section 3.

3 SECURITY MODEL FOR UPDATABLE OBLIVIOUS KMS

We introduce the security model for Updatable Oblivious KMS which combines the elements and advantages of oblivious computation and updatable encryption in a single model. As in updatable encryption, e.g. [9, 10, 20, 34, 36], we consider keys that evolve over *epochs*, where at the beginning of a new epoch the encryption/decryption key is replaced with a fresh key. In our case, this applies to client keys k_c held by the Key Management server KmS. The goal is to capture the security of key rotation both in the sense of forward security and post-compromise security. That is, the compromise of a client key k_c from a given epoch should not help in exposing data encrypted either at a later epoch or in a previous epoch. In the latter case, however, one needs to qualify this requirement. Suppose that a ciphertext e is generated using the key k_c from epoch t and later the key k_c' for epoch $t' > t$ is exposed; should the data d encrypted under ciphertext e still be secure? Clearly, if the attacker \mathcal{A} sees e' , the updated version of ciphertext e in epoch t' , then \mathcal{A} can decrypt e' and obtain d . However, if \mathcal{A} possesses k_c' and e but does not have the updated e' then the security of d needs to be fully preserved.

The above illustrates the intricacies of updatable encryption models, which require careful bookkeeping of information available to the attacker: What ciphertexts it sees and when, for what epochs it obtains the secret key k_c , and for which it receives update information, etc. The goal is to prevent the attacker from learning anything that is not trivially (and unavoidably) derivable from the information it requests. In this section, we set these rules and goals through a formal model of UOKMS security, and use it in Section 4 to prove the security of our UOKMS design from Fig. 4.

3.1 Formal UOKMS Scheme

Formally, an Updatable Oblivious KMS (UOKMS) scheme is a tuple of algorithms $KGen$, Enc , $UGen$, $UEnc$, and a protocol Dec , intended for a KMS server KmS , a storage server StS , and a client C , s.t.:

- $KGen$ is a key generation algorithm, run by KmS , which on input a *security parameter* ℓ generates a public key pair (sk, pk) .
- Enc is an encryption algorithm, run by any party, which on input key pk and plaintext m generates ciphertext c .
- $Dec = (Dec.KmS, Dec.C)$ is an interactive decryption protocol between a client running $Dec.C(pk, c)$ and KmS running $Dec.KmS(sk, pk)$, where $Dec.C$ outputs m or \perp .
- $UGen$ is an update generation algorithm, run by KmS , which on input (sk, pk) generates a new key pair (sk', pk') together with an *update token* Δ .
- $UEnc$ is a ciphertext update algorithm, run by StS , which on input (c, pk, Δ) outputs an updated ciphertext c' .

An UOKMS scheme must satisfy the following *correctness* property. First, the interactive decryption must recover the encrypted plaintext, i.e. for any m , if $(sk, pk) \leftarrow KGen(\ell)$ and $c \leftarrow Enc(pk, m)$ then $Dec.C(pk, c)$ outputs m after an interaction with $Dec.KmS(sk, pk)$. Furthermore, the same correctness property applies to keys and ciphertexts produced and updated in later periods. That is, for every m , if $(sk, pk) \leftarrow KGen(\ell)$, $c \leftarrow Enc(pk, m)$, $(sk', pk', \Delta) \leftarrow UGen(sk, pk)$, and $c' \leftarrow UEnc(c, pk, \Delta)$, then $Dec.C(pk', c')$ outputs m after interacting with $Dec.KmS(sk', pk')$.

On public and private values. We model UOKMS as a *public key* encryption scheme where *any party* in possession of the public key pk can encrypt files for the client whose corresponding decryption key sk is held by KmS . It is assumed that KmS has the means to authenticate the client before engaging in a decryption operation using key sk but a secret channel between client and KmS is not needed. The update token Δ is assumed to be transmitted from KmS to StS over a secure channel. No other party needs or should know this value. In particular, the model does not guarantee secrecy of sk_{t+1} given sk_t and Δ_{t+1} or secrecy of sk_t given sk_{t+1} and Δ_{t+1} . For example, in our UOKMS scheme of Figure 4 receiving Δ_{t+1} allows one to derive both sk_{t+1} from sk_t , and sk_t from sk_{t+1} . In this case, if Δ_{t+1} was leaked then a KmS corrupted in epoch t would be effectively also corrupted in epoch $t+1$, and vice versa.⁴

3.2 UOKMS obliviousness and security

The definition below formalizes the notion of KMS obliviousness.

Definition 3.1. We say that a UOKMS scheme is *oblivious* if for all efficient algorithms \mathcal{A} the interaction of \mathcal{A} with $Dec.C(pk, c_0)$ is indistinguishable from interaction with $Dec.C(pk, c_1)$, for any (pk, c_0, c_1) output by \mathcal{A} s.t. c_0, c_1 are valid ciphertexts of the same length and pk is a valid public key.⁵

⁴This is not a necessary feature of a UOKMS scheme, i.e. one could imagine that Δ_{t+1} allows for updating ciphertexts (and the public key), but not for updating the corresponding secret key. However, all existing ciphertext-independent updatable encryption schemes, ours included, allow for updating sk given Δ .

⁵The public key pk , normally chosen by KmS , can be chosen by \mathcal{A} in this definition, modeling a malicious KmS , but C can check some properties of the public key and the ciphertext, e.g. that they contain expected group elements, before running Dec .

As noted above, defining security of UOKMS, and of updatable encryption in general, requires establishing the rules of what information the adversary is entitled to receive and when, and what constitutes a win relative to that information. In our model, time is divided into *epochs* at the beginning of which a new key pair (sk, pk) and an update token Δ are generated. For each epoch the adversary \mathcal{A} receives the new public key pk , and can request to see either the new secret key sk or the update token Δ , which corresponds to \mathcal{A} compromising in that epoch, respectively either server KmS or server StS. Algorithm \mathcal{A} is also given oracle access to the ciphertext-update function UEnc but it is not allowed to use it for trivial wins, e.g., updating challenge ciphertexts to an epoch for which it knows the secret key. Note that \mathcal{A} learns the secret key sk_t of epoch t if \mathcal{A} asks for it, but also if \mathcal{A} asks for sk_{t-1} in epoch $t-1$ and asks for Δ_t in epoch t . This shows that what \mathcal{A} can learn in one epoch may depend on what it knew in the previous epoch, and the UOKMS security game rules must reflect that.

We formalize these rules and the attacker goals via the real-ideal experiments shown in Fig. 5. In each epoch t , the attacker receives pk_t and chooses to either corrupt KmS, hence obtaining sk_t , or to corrupt StS, hence obtaining the update token Δ_t , except if KmS was corrupted in epoch $t-1$ (otherwise the attacker could calculate sk_t from sk_{t-1} and Δ_t , making this case equivalent to corrupting both KmS and StS in epoch t). In addition, \mathcal{A} obtains access to oracles Enc, Dec, UEnc , depending on the compromised party. An aspect of the definition that is specific to our oblivious setting is that the attacker with access to the oblivious decryption oracle can decrypt any ciphertext of its choice in a decryption call, but each call can result in decryption of at most a *single* challenge ciphertext. More generally, with q calls to the decryption oracle, \mathcal{A} can decrypt q messages but nothing more. Finally, we note that the ability of the attacker to access a decryption oracle provides CCA security to our public key scheme in the oblivious setting.

Figure 5 shows two experiments: Experiment $\text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)$ which models an interaction of the real-world adversary \mathcal{A} with the real UOKMS scheme, and experiment $\text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)$ which models an interaction of a simulator SIM with an “ideal” UOKMS scheme. We call a UOKMS scheme secure if the two interactions, real and ideal, are indistinguishable. Formally:

Definition 3.2. Let $\text{Adv}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)$ be the probability that experiment $\text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)$ outputs 1, and let $\text{Adv}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)$ be the probability that experiment $\text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)$ outputs 1. We say that UOKMS scheme is *secure* if for all efficient algorithms \mathcal{A} there exist an efficient algorithm SIM s.t. $|\text{Adv}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell) - \text{Adv}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)|$ is negligible in ℓ .

The real experiment $\text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)$ in Figure 5 models an interaction of adversary \mathcal{A} with a UOKMS scheme which progresses through epochs $t = 0, 1, \dots$ where the flag corr_t designates whether \mathcal{A} corrupts KmS (kms) or the storage server StS (sts) in epoch t . After the initialization which generates the initial KMS key pair $(\text{sk}_0, \text{pk}_0)$ we give pk_0 to \mathcal{A} and let \mathcal{A} interact with the encryption, decryption, and ciphertext update oracles. We model the progress from one epoch to the next via “party corruption” oracle Corr which uses \mathcal{A} 's decision bit corr_{t+1} to corrupt either KmS or StS in the

next epoch⁶. This oracle triggers a key update, i.e. a new KMS key pair is created as $(\text{sk}_{t+1}, \text{pk}_{t+1}, \Delta_{t+1}) \leftarrow \text{UGen}(\text{sk}_t, \text{pk}_t)$, and the epoch counter t is incremented. Adversary \mathcal{A} then receives the new public key pk_{t+1} and possibly more, depending on the parties it corrupts: Namely, if $\text{corr}_{t+1} = \text{kms}$ then \mathcal{A} also gets the new secret key sk_{t+1} , and if $\text{corr}_{t+1} = \text{corr}_t$, i.e. if \mathcal{A} corrupts the same party in the two consecutive epochs, then \mathcal{A} also gets the update token Δ_{t+1} . Crucially, \mathcal{A} does *not* get Δ_{t+1} if $\text{corr}_{t+1} \neq \text{corr}_t$. (Indeed, as mentioned above, in the UOKMS scheme of Figure 4, receiving the update token would allow the adversary to effectively extend the corruption of KmS from epoch t to epoch $t+1$ and vice versa.)

The security experiment assumes that KmS corruptions are passive in the sense that if $\text{corr}_t = \text{kms}$ we let \mathcal{A} learn sk_t (and Δ_t if $\text{corr}_{t-1} = \text{kms}$), but we do not let \mathcal{A} interfere in the update generation and/or the dissemination of the created update token and a public key, or in the execution of the decryption protocol. (All existing Updatable Encryption security notions make such choices, e.g. assuming that even if the adversary compromises the entity that stores the key, the key update is still generated honestly.)

We assume that StS corruptions are active in the sense that for epochs where $\text{corr}_t = \text{sts}$ we not only let \mathcal{A} learn Δ_t , but we also give \mathcal{A} an access to the ciphertext update oracle UEnc which on input (t', c) , for $t' < t$ and c a ciphertext from epoch t' , outputs the updated value of c at epoch t . That is, the oracle runs the update algorithm on (supposed) ciphertext $c_{t'} = c$ using update tokens $\Delta_{t'+1}, \dots, \Delta_t$, and outputs the updated ciphertext c_t . This models the ability of the adversary to inject ciphertexts to StS in some epoch – either by directly modifying these ciphertexts when StS is corrupted, or by sending a ciphertext to the client who then stores it at StS⁷ – and then having this ciphertext updated by the UEnc oracle. Note that the Update protocol is not provided at epochs where $\text{corr}_t = \text{kms}$ since this would allow \mathcal{A} to decrypt challenge ciphertexts using the compromised KmS key.

Our UOKMS models a public key encryption where adversary \mathcal{A} can encrypt any message at will, but the role of the encryption oracle Enc in the UOKMS security game is to model the generation of *challenge ciphertexts*. Namely, in the real game, oracle Enc on \mathcal{A} 's input m generates a ciphertext $c = \text{Enc}(\text{pk}_t, m)$, but in the ideal game the same ciphertext c must be produced by the simulator algorithm SIM given only $|m|$ (and flag enc) as an input while the plaintext m is added to the (secret) list L of encrypted challenge plaintexts. Adversary \mathcal{A} can decrypt any ciphertexts (or indeed any ciphertext-like objects of its choice) using the decryption oracle Dec. Because we aim to support *oblivious* decryption, the precise ciphertext which \mathcal{A} effectively enters into the decryption oracle is hidden from the oracle, hence we must count each decryption oracle access as an attempt to decrypt some challenge ciphertext. We model this in the ideal game by giving SIM access to a *single* location in list L of challenge plaintexts, per each Dec query of \mathcal{A} . Note that this technically implies that the simulator can extract the unique ciphertext which \mathcal{A} attempts to decrypt in this oblivious

⁶ We assume w.l.o.g. that \mathcal{A} corrupts exactly one of these parties in each epoch. In particular, the real-world event when both StS and KmS are corrupted in epoch t is reflected in our model by KmS corrupted in two consecutive epochs $t-1, t$, because this reveals both sk_t and Δ_t to \mathcal{A} . On the other hand, the epoch without corruption strictly weakens the adversary capabilities hence it is subsumed by the other cases.

⁷ Note that our treatment is of a public key encryption, so other parties can potentially create ciphertexts which land in the StS storage.

$\text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)$

Set $t \leftarrow 0$ and $\text{corr}_0 \leftarrow \text{sts}$. Generate $(\text{sk}_0, \text{pk}_0) \leftarrow \text{KGen}(\ell)$ and give pk_0 to \mathcal{A} . The experiment output is the output of \mathcal{A} after interaction with the following oracles:

Enc: On \mathcal{A} 's input m , if $\text{corr}_t = \text{sts}$ output $\text{Enc}(\text{pk}_t, m)$;

Dec: Let \mathcal{A} interact with $\text{Dec.S}(\text{sk}_t)$;

UEnc: On \mathcal{A} 's input (t', c) , if $\text{corr}_t = \text{sts}$ and $0 \leq t' < t$ then

set $c_{t'} := c$; for $j = t'+1$ to t set $c_j := \text{UEnc}(c_{j-1}, \text{pk}_j, \Delta_j)$; output c_t ;

Corr: On \mathcal{A} 's input corr_{t+1} , set $(\text{sk}_{t+1}, \text{pk}_{t+1}, \Delta_{t+1}) \leftarrow \text{UGen}(\text{sk}_t, \text{pk}_t)$;

If $(\text{corr}_t, \text{corr}_{t+1}) = (\text{kms}, \text{kms})$ output $(\text{pk}_{t+1}, \text{sk}_{t+1}, \Delta_{t+1})$;

If $(\text{corr}_t, \text{corr}_{t+1}) = (\text{kms}, \text{sts})$ output pk_{t+1} ;

If $(\text{corr}_t, \text{corr}_{t+1}) = (\text{sts}, \text{kms})$ output $(\text{pk}_{t+1}, \text{sk}_{t+1})$;

If $(\text{corr}_t, \text{corr}_{t+1}) = (\text{sts}, \text{sts})$ output $(\text{pk}_{t+1}, \Delta_{t+1})$;

Increment epoch counter $t := t + 1$.

 $\text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)$

Set $t \leftarrow 0$ and $\text{corr}_0 \leftarrow \text{sts}$. Initialize an empty challenge plaintext list L . Let a *stateful* algorithm SIM generate pk_0 on input ℓ and give pk_0 to \mathcal{A} . Experiment output is the output of \mathcal{A} after interaction with the following oracles:

Enc: On \mathcal{A} 's input m , if $\text{corr}_t = \text{sts}$ add m to L and output $\text{SIM}(\text{enc}, |m|)$;

Dec: Let \mathcal{A} interact with $\text{SIM}(\text{dec})$ while letting SIM learn *one* chosen item in L ;

UEnc: On \mathcal{A} 's input (t', c) , if $\text{corr}_t = \text{sts}$ and $0 \leq t' < t$ then output $\text{SIM}(\text{upd}, t', c)$;

Corr: On \mathcal{A} 's input corr_{t+1} output $\text{SIM}(\text{corr}_{t+1})$ and increment epoch counter $t := t + 1$.

Figure 5: Security Experiments for Updatable Oblivious KMS

decryption protocol instance, or otherwise the simulator wouldn't know which plaintext on list L it should access. Observe also that we do not create challenge ciphertexts in an epoch where KmS is corrupted, because knowledge of KmS's private key makes all such ciphertexts insecure.

We stress that the $\text{Exp}_{\text{uokms}}^{\text{real}}$ security game allows any pattern of corruptions *except* corruption of both StS and KmS in a single epoch (see footnote 6). However, our model of corruptions is *static* in the sense that \mathcal{A} must decide which party to corrupt at the beginning of each epoch. (See also the discussion below.)

Prior Updatable Encryption Models. Our notion of Updatable (Oblivious) KMS is related to *Updatable* Encryption (UE) or Encryption with *Key Rotation*, which was studied in several recent works [9, 10, 20, 34, 36]. UOKMS extends the notion of UE by splitting the UE's client into two separate entities, the KMS server, which holds the client's decryption key and generates key updates, and the client itself, who decrypts the ciphertexts retrieved from the storage server via an interactive decryption protocol with the KMS. The UOKMS model thus lifts the notion of Updatable Encryption to the setting that reflects realistic large cloud storage deployments, where the decryption keys of all clients are held by a specialized Key Management server. On the other hand, collapsing the client and the KMS in the UOKMS model into a single entity gives exactly the setting of UE, hence our UOKMS scheme and security notion give rise to the corresponding UE scheme and notion.

Our security model corresponds to the *ciphertext-independent* UE model of Lehmann et al. [36] (which refines the model of Everspaugh et al. [20]), where a single update message can be used to update any number of ciphertexts. Of these only the recent work of

Klooss et al. [34] addresses CCA security, and lets the adversary access a decryption oracle, as we do in our model. However, of the two schemes shown secure in [34] the one whose efficiency is comparable to ours does not allow the adversary an unrestricted access to the Ciphertext Update oracle, while our model allows unrestricted access to *both* Dec and UEnc oracles. The scheme of [34] which allows such unrestricted oracle access relies heavily on pairing-based NIZKs, using e.g. 22 pairings in decryption, in contrast to a single standard group exponentiation used in decryption in our scheme. However, our model of UOKMS security is specialized to the case of oblivious interactive decryption where the decryption oracle, which models the KMS server, runs on *blinded* ciphertexts. In such setting a standard CCA notion, where the decryption oracle is restricted from decrypting a challenge ciphertext, does not apply. Thus we capture security with a “counting method” which enforces that any Q accesses to the decryption oracle allow for learning plaintext information in at most Q challenge ciphertexts. Ours is the first treatment of Updatable Encryption with *oblivious* decryption procedure, and this setting necessitates a “counting-based” notion of security in the presence of decryption oracle.

The UE schemes of [20, 34, 36] achieve *update indistinguishability*, i.e. a ciphertext updated to the new epoch cannot be efficiently linked to the original from the previous epoch. We do not consider this property, although our scheme can be extended to support it, because achieving this property requires update cost proportional to the *total size* of the encrypted data, which we believe is impractical in large storage deployments (see footnote 3). The above UE

schemes also consider *ciphertext integrity*, but this notion is specialized to the case of symmetric key encryption, while our UOKMS model treats the case of public key encryption.

Finally, we should point out that our security model is static in the sense that an adversary must choose at the beginning of each epoch whether it compromises the decryption key stored by the KMS or the update token held by the storage server (or both). By contrast, [34, 36] consider an adaptive model of corruptions, where an adversary can request either the decryption key or the update token or both for any *past* epoch as well. The adaptive security model is more general and less restrictive, but we analyze the security of our scheme only in the static model because adaptive security presents subtle technical challenges which we do not know how to overcome.⁸ Technically, the simulator would have to make bets about past epochs, guessing whether an adversary will eventually ask for a decryption key for some past epoch (in which case the simulator needs to know this epoch key), or whether an adversary will ask for an update token which allows updating a challenge ciphertext to that epoch (in which case the simulator needs to embed an encryption challenge in that epoch key). Since the simulator needs to make these bets with respect to polynomially-many past epochs, the probability that its guesses are all correct will be negligible, and it is not clear if such strategy can lead to efficient simulation. We thus believe that security analysis in the fully adaptive model of [34, 36] remains an open question.

4 SECURITY ANALYSIS OF THE UOKMS SCHEME

The UOKMS scheme shown in Figure 4 is information-theoretic *oblivious*, as is the OPRF protocol dh-op on which the Decryption protocol in Fig. 4 is based, but the *security* of this scheme relies on the *OMDH-IO* computational assumption and the (*receiver*) *non-committing* property of symmetric encryption, both defined below:

One-More DH with Inverse Oracle (OMDH-IO) Assumption. For any PPT \mathcal{A} the following probability is negligible:

$$\text{Prob}[\mathcal{A}^{(\cdot)^k, (\cdot)^{1/k}}(g, g^k, g_1, \dots, g_N) = \{(g_{j_s}, g_{j_s}^k)\}_{s=1, \dots, Q+1}]$$

with the probability going over random k in \mathbb{Z}_q , random choice of group elements g_1, \dots, g_N in $G = \langle g \rangle$, and \mathcal{A} 's randomness, and where $(\cdot)^k$ and $(\cdot)^{1/k}$ are exponentiation oracles, and Q is the number of \mathcal{A} 's queries to the $(\cdot)^k$ oracle.

Without access to oracle $(\cdot)^{1/k}$, the above is identical to the *One-More DH* (OMDH) assumption [6, 32], which was used e.g. for proving the security of the practical OPRF schemes [28, 29], particularly the one shown in Figure 3 in Section 2.1. Thus, OMDH-IO is a strengthening of OMDH; its security can be proven in the Generic Group Model (GGM) as an extension to the proof of OMDH in that model [31] and with a slight modification of the security bounds. We sketch this adaptation in Appendix A.

Receiver Non-Committing Symmetric-Key Encryption. This property of symmetric-key encryption (SKE) is used in our security analysis to enable the simulation required by the security game in

⁸We stress that this is an issue in the proof only and not an explicit attack, and that similar technical issues were observed regarding adaptive security in other contexts, e.g. in proactive cryptosystems, see e.g. [1, 13, 37].

Fig. 5. Informally, it states that without knowledge of the encryption key, ciphertexts do not commit to their underlying plaintexts, thus allowing the simulator to “explain” a fixed ciphertext as the encryption of any plaintext. Formally, a symmetric encryption scheme (Enc, Dec) is *receiver non-committing* (RNC) if for any PPT \mathcal{A} there exists PPT SIM s.t. \mathcal{A} 's view in the following real and ideal games is indistinguishable: (1) In the real game \mathcal{A} interacts with oracles Enc and Reveal , where $\text{Enc}(i, m)$ picks random key k_i and outputs $e = \text{Enc}(k_i, m)$ while $\text{Reveal}(i)$ reveals k_i ; (2) In the ideal game \mathcal{A} interacts with a *stateful* algorithm SIM , s.t. when \mathcal{A} sends (i, m) as an Enc query, SIM must return e on input $(i, |m|)$, and when \mathcal{A} sends i as a Reveal query, SIM must output k_i on input (i, m) .

THEOREM 4.1. *The UOKMS scheme in Figure 4 is unconditionally oblivious and is secure under the OMDH-IO assumption in ROM if the symmetric encryption scheme Enc is receiver non-committing.*

Notes on the Proof. The proof of Theorem 4.1 is presented in Section 4.1. We note that the inverse exponentiation oracle in OMDH-IO is necessary to obtain the theorem as the protocol (in the context of our model) provides an attacker \mathcal{A} that corrupts KmS in epoch $t - 1$ and StS in period t with an oracle to the function $(\cdot)^{1/k_t}$. Indeed, in epoch t , \mathcal{A} obtains access to UEnc which implements an exponentiation oracle $(\cdot)^{\Delta_t} = (\cdot)^{k_{t-1}/k_t}$, and together with knowledge of k_{t-1} , \mathcal{A} can compute $(\cdot)^{1/k_t}$ on any value of its choice. The RNC property of SKE Enc is likewise necessary. Consider an attacker \mathcal{A} making two queries: (a) an Enc query on some m , and (b) a Dec query where \mathcal{A} runs the Dec.C protocol on the received ciphertext $c = (\text{ObjId}, w, e)$. By the UOKMS security game rules of Fig. 5 the simulator SIM has to simulate this as follows: (a) it produces c on message length $|m|$, and (b) on input m retrieved from list L , it simulates protocol Dec.S so that c decrypts to m . SIM 's response v to \mathcal{A} 's message u in the decryption protocol, see Fig. 4, defines the effective KMS key as $k = \text{DL}(u, v)$, and consequently defines the data encryption key for (ObjId, w, e) as $\text{dek} = H(w^k)$. Thus when SIM defines the output dek of oracle H on input w^k it must satisfy that $e = \text{Enc}_{\text{dek}}(m)$. In particular, SIM first creates ciphertext e given just $|m|$ and then, given m , it creates dek s.t. $e = \text{Enc}_{\text{dek}}(m)$, which implies that SKE Enc satisfies the RNC property.

COROLLARY 4.2. *The UOKMS scheme in Figure 4 is secure under the OMDH-IO assumption in the Ideal Cipher Model and ROM if the symmetric encryption is implemented using CTR or CBC modes.*

The corollary follows because CTR and CBC encryption modes satisfy the receiver non-committing property in the Ideal Cipher model: If message length $|m|$ defines n blocks for block cipher E then SIM services Enc query on input $(i, |m|)$ by setting ciphertext $e = (IV, e_1, \dots, e_n)$ where IV and all e_i 's are random blocks. When SIM gets $m = (m_1, \dots, m_n)$ to service $\text{Reveal}(i)$ query, values $(IV, e_1, \dots, e_n, m_1, \dots, m_n)$ define n input/output pairs which SIM needs to set for $E(k, \cdot)$ for random key k . For counter mode CTR, SIM sets $E(k, IV + j) = m_j \oplus e_j$ for all j while for CBC mode, SIM sets $E(k, m_j \oplus e_{j-1}) = e_j$ for all j where $e_0 = IV$. Either way by randomness of e_i 's this sets $E(k, \cdot)$ outputs on n given points to n random values. The probability that this creates collisions in $E(k, \cdot)$ is negligible, and by randomness of k there is a negligible probability that any points of $E(k, \cdot)$ were queried before.

Note. The above argument can be expanded to include authenticated encryption via encrypt-then-mac where the simulator chooses the MAC key.

4.1 Proof of Theorem 4.1

PROOF. Note first that the *unconditional obliviousness* of this UOKMS scheme is immediate, because for any public key pk and any two valid ciphertexts $c_0 = (\text{ObjId}_0, w_0, e_0)$ and $c_1 = (\text{ObjId}_1, w_1, e_1)$, the interaction with Dec.C on (pk, c_b) for $b = 0$ and $b = 1$ is identical: In either case C sends $u = (w_b)^{r'}$ for $r' \leftarrow_R \mathbb{Z}_q$, which is a random group element if $w_b \in G$ and $w_b \neq 1$ because the group order is prime. To argue UOKMS *security* we will first show an efficient simulator algorithm SIM which having access to (any) adversary algorithm \mathcal{A} , interacts with the ideal UOKMS game $\text{Exp}_{\text{uokms}}^{\text{ideal}}$. We will then re-write SIM as a reduction algorithm \mathcal{R} s.t. if \mathcal{A} has ϵ advantage in distinguishing an interaction with the real UOKMS game $\text{Exp}_{\text{uokms}}^{\text{real}}$ and an interaction with SIM and $\text{Exp}_{\text{uokms}}^{\text{ideal}}$, i.e. if

$$\epsilon = |\Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)] - \Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)]|$$

then reduction \mathcal{R} , given access to \mathcal{A} , has the same probability ϵ of solving the OMDH-IO problem. It follows that under the OMDH-IO assumption quantity ϵ must be negligible, which implies that the UOKMS scheme is secure. We note that in one step along these SIM modifications we replace the real symmetric encryption Enc with the simulator assumed by the RNC property of SKE Enc . We provide the details of the proof now.

The proof relies on the ROM model for function $H : G \rightarrow \{0, 1\}^\ell$ used in UOKMS in Figure 4. Specifically, we treat H as an external entity \mathcal{A} needs to query to compute H outputs, simulator SIM and reduction \mathcal{R} intercept \mathcal{A} 's calls to H , and we measure probabilities $p_0 = \Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)]$ and $p_1 = \Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)]$ over the randomness of H . For simplicity of notation we assume that group G is fixed for every security parameter ℓ and we assume a non-uniform security model both for the OMDH-IO assumption and UOKMS security. To reduce visual clutter we denote plaintext files as m instead of Obj and we omit identifiers ObjId in ciphertexts.

We will first describe game G , which reproduces the same distribution \mathcal{A} sees in the real security game $\text{Exp}_{\text{uokms}}^{\text{real}}$, but does it in a way which makes it easier to understand simulator SIM which we will describe next. Game G picks $k \in \mathbb{Z}_q$ and sets the first epoch key as $(k_0, y_0) = (k, g^k)$. Game G also picks a list of random group elements g_1, \dots, g_N in G , where N is the upper-bound on the number of Enc queries \mathcal{A} makes. Then for every $i > 0$, G picks the following values: If \mathcal{A} corrupts KmS in epoch i then G picks random $k_i \leftarrow \mathbb{Z}_q$ and outputs (k_i, y_i) for $y_i \leftarrow g^{k_i}$. (If \mathcal{A} corrupts KmS for two epochs in the row G also outputs $\Delta_i = k_{i-1}/k_i$.) If \mathcal{A} corrupts StS in epoch i then G acts depending on which party \mathcal{A} corrupted in epoch $i-1$: (case 1) If it was StS then G picks random $\Delta_i \leftarrow \mathbb{Z}_q$ and outputs $y_i \leftarrow y_{i-1}^{1/\Delta_i}$; (case 2) If it was KmS then G picks random $\Delta_{j+1,i} \leftarrow \mathbb{Z}_q$ and outputs $y_i \leftarrow y_j^{1/\Delta_{j+1,i}}$ where j was the last epoch when \mathcal{A} corrupted StS before \mathcal{A} corrupted KmS in epoch $i-1$. Let E_K be the set of epochs when \mathcal{A} corrupts KmS and E_S the set of epochs when \mathcal{A} corrupts StS . The above process defines value δ_i for each $i \in E_S$ s.t. $y_i = y^{1/\delta_i}$ (hence $k_i = k/\delta_i$), and G can compute this δ_i as either $\delta_{i-1} \cdot \Delta_i$, if $(i-1) \in E_S$, or as

$\delta_j \cdot \Delta_{j+1,i}$, if $j \in E_S$ and $\{j+1, \dots, i-1\} \subseteq E_K$. Given these values, G services oracles Enc , Dec , and UEnc at epoch $i \in E_S$ (note that these calls are disallowed if $i \in E_K$) as follows:

- G replies to n -th call to $\text{Enc}(m)$ with $c = (w, \text{Enc}_{\text{dek}}(m))$ where $w = (g_n)^{\delta_i}$ and $\text{dek} = H(z)$ for $z = (g_n)^k$; (Note that $z = w^{k/\delta_i}$, hence c is distributed as in the real interaction.)
- G replies to message u to Dec with $v = (u^{1/\delta_i})^k$;
- G replies to $\text{UEnc}(t', c)$ for $c = (w, e)$ with (w', e) for $w' = w^{\delta_i/\delta_{t'}}$ if $t' \in E_S$, and $w' = (w^{\delta_i \cdot k_{t'}})^{1/k}$ if $t' \in E_K$.

The correctness of Enc and Dec responses follows because $k_i = k/\delta_i$, and as for UEnc , note that $k_i = k/\delta_i$, and the implicit update from epoch t' to epoch i is $\Delta_{t',i} = k_{t'}/k_i$, which together implies that $\Delta_{t',i} = (k_t \cdot \delta_i) \cdot (1/k)$. Thus game G reproduces the exact same view as security game $\text{Exp}_{\text{uokms}}^{\text{real}}$.

Simulator SIM interacts with an ideal experiment $\text{Exp}_{\text{uokms}}^{\text{ideal}}$ and executes the same algorithm as game G – including picking the initial key k and keys k_i if $i \in E_K$ and update-related values δ_i and $\Delta_{j,i}$ if $i \in E_S$ as described above (and defining corresponding δ_i 's and k_i 's). For handling oracles Enc and Dec , SIM resorts to the (stateful) simulator SIM_E assumed by the Receiver Non-Committing (RNC) property of the symmetric encryption Enc . First, when \mathcal{A} sends t -th query m to oracle Enc in epoch $i \in E_S$, we put m at position t in list L , and SIM replies to \mathcal{A} with $c = (w, e)$ for $w = (g_t)^{\delta_i}$ and e computed by SIM_E on input $(t, |m|)$. Second, when \mathcal{A} sends u to Dec , SIM replies with $v = (u^{1/\delta_i})^k$ and then monitors \mathcal{A} 's queries to H : If \mathcal{A} makes query z to H s.t. $z^{1/k} = g_t$ for $g_t \in \{g_1, \dots, g_N\}$ then SIM asks $\text{Exp}_{\text{uokms}}^{\text{ideal}}$ to reveal message m at the t -th position in list L , sends (t, m) as the Reveal query to SIM_E , and given key dek as SIM_E 's response, defines $H(z) = \text{dek}$. By the RNC property of Enc , pairs (dek, e) produced by SIM_E are computationally indistinguishable from random dek and $e = \text{Enc}_{\text{dek}}(m)$. (In particular, this process sets $H(z)$ to a value indistinguishable from random.)

The only difference between \mathcal{A} 's interaction with G and \mathcal{A} 's interaction with SIM (which in turn interacts with $\text{Exp}_{\text{uokms}}^{\text{ideal}}$) is if in the latter case \mathcal{A} queries H on arguments $(g_i)^k$ for more than Q elements in $\{g_1, \dots, g_N\}$ where Q is the number of \mathcal{A} 's decryption queries: Given Q decryption queries SIM is allowed to learn only Q items in list L , so it can embed correct messages as decryptions of Q challenge ciphertexts, involving Q challenge points $\{g_j\}_{s=1, \dots, Q}$, but SIM will not be able to decrypt correctly the $(Q+1)$ -st ciphertext (w, e) formed as $w = (g_{j+1})^{\delta_i}$ s.t. \mathcal{A} queries H on $z = (g_{j+1})^k = w^{k_i}$. In other words, if there is ϵ difference between $\Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{real}}(\mathcal{A}, \ell)]$ and $\Pr[1 \leftarrow \text{Exp}_{\text{uokms}}^{\text{ideal}}(\mathcal{A}, \text{SIM}, \ell)]$ then ϵ is upper-bounded by the probability that \mathcal{A} queries H on values $(g_j)^k$ for $Q+1$ points g_j in $\{g_1, \dots, g_N\}$. But by inspection of SIM one can see that SIM can be readily changed to reduction \mathcal{R} against the OMDH-IO problem: \mathcal{R} follows the algorithm of SIM except that uses the OMDH-IO challenge key g^k as y , it gets points (g_1, \dots, g_N) as part of the OMDH-IO challenge, and it uses OMDH-IO oracles $(\cdot)^k, (\cdot)^{1/k}$ instead of using exponent k directly. Note that SIM uses $(\cdot)^k$ only Q times, to service the Q decryption oracle queries, and if \mathcal{A} makes queries to H on $Q+1$ arguments $(g_j)^k$ with probability ϵ , then \mathcal{R} will break OMDH-IO with probability ϵ because \mathcal{R} can

identify such queries with oracle $(\cdot)^{1/k}$. This completes the proof of Theorem 4.1. \square

5 THRESHOLD OKMS AND UOKMS

The key management systems (particularly for storage applications) that motivate our work are often characterized by the large amounts of data they store as well as the value and long-lived nature of this data. The whole security of such an operation depends on the security of the KMS client keys, hence the importance of key rotation (as addressed by UOKMS) as a way to limit the bad effects of key exposure. Yet, the main priority is to prevent these keys from leaking in the first place. Fortunately, all the schemes presented in this paper lend themselves to *efficient distributed implementations* via the very efficient *Threshold OPRF tdh-op* [30] shown in Figure 6.

In our application, client keys k_c are shared among n KMS servers S_1, \dots, S_n , so that the cooperation of $t + 1$ of these is needed to compute the OPRF function with k_c as the key, while the compromise of t servers provides no information to the attacker on k_c . Moreover, the key k_c is never reconstructed or exists in one place, not even at generation (which is also performed distributively). In addition, this scheme enjoys *proactive security* [25, 43], namely, the sharing among the n servers can be refreshed periodically so that the attacker needs to break into $t + 1$ servers during the same time period to be able to compromise the key. Servers can be replaced and shares recovered, protecting secrecy and integrity/availability of the system as needed for long-lived keys.

The tdh-op function from Fig. 6 implements exactly the OPRF as defined in the OKMS scheme from Fig. 2. For the UOKMS scheme of Fig. 4, the only difference is in the input from the client (a random group element rather than a hashed value).

Note on efficiency. The dominant cost of computation in tdh-op is one exponentiation for each of the $t + 1$ servers and two exponentiations for the client regardless of the values n and t . We note that tdh-op is described in a simplified form in Figure 6 where the set of reconstruction parties \mathcal{SE} is assumed to be known by C in advance. If the reconstruction set \mathcal{SE} is not known a-priori (i.e., more than $t + 1$ servers are contacted), each S_i would respond with a^{k_i} and C would compute the interpolation in the exponent at the cost of a single multi-exponentiation (which can be further optimized when the α_i 's are small, e.g., $\alpha_i = i$, using a recent technique from [44]). An additional important feature of the tdh-op solution is that the aggregation of server values b_i into the dh-op result can be done by a proxy server (one of the threshold servers or a special purpose one) so that the threshold implementation is transparent to the client.

5.1 Distributed Updates

While a threshold solution greatly increases the security of the KMS keys, one may still want to apply key rotation, particularly given the efficiency of updates in our UOKMS solution. In the threshold setting this means that at the beginning of a rotation epoch, the servers, that have a sharing of a key k_c , need to choose a new random client key k_c' and generate the value $\Delta = k_c/k_c'$. However, Δ should only be disclosed to the client C and the storage server StS, calling for a distributed generation of Δ where no subset of t or less servers learn anything about this value.

We show a procedure that given (n, t) Shamir sharing of a key k generates (n, t) sharing of a new random key k' and of the update token $\Delta = k/k'$. It uses two standard tools from multi-party computation: (i) The joint generation of a Shamir sharing ρ_1, \dots, ρ_n of a uniformly random secret ρ over \mathbb{Z}_q , e.g., [45] or Fig. 7 of [23], and (ii) a Distributed Multiplication protocol which given the sharings of secret a and secret b generates a sharing of the product $a \cdot b$ without learning anything about either secret, e.g., [23].

The distributed update protocol assumes that n servers S_1, \dots, S_n have a sharing (k_1, \dots, k_n) of a key k . To produce a new key k' the servers jointly generate a sharing ρ_1, \dots, ρ_n of a random secret $\rho \in \mathbb{Z}_q$ and run distributed multiplication to generate shares k'_1, \dots, k'_n of the new key defined as $k' = \rho \cdot k$. Finally, each server S_i sends to C and/or StS its share ρ_i from which the recipient reconstructs ρ and sets $\Delta := \rho^{-1} [= k'/k]$.

5.2 Verifiable Threshold (U)OKMS

As noted earlier, being able to verify the correctness of a data encryption key dek before encrypting an object is an important feature of the OKMS solution and a major advantage over traditional wrapping-based KM systems (Fig. 1). OKMS Verifiability requires checking the correct OPRF operation by the KmS for which Verifiable OPRFs [28] are available, assuming the client possesses the authentic public key g^{k_c} corresponding to KmS key k_c . As indicated in Section 2.2, verifiability in the case of UOKMS can be done directly via correct symmetric authenticated decryption, thus dispensing with the need to check the oblivious operation by KmS. In the threshold case, however, where multiple servers provide input for decryption, it is necessary to identify misbehaving servers. Thus, in the threshold case, verifiability is needed also for UOKMS.

Note that for the single-server dh-op scheme of Fig. 3, verifiability can be added via a simple non-interactive zero-knowledge proof of equality of logarithms. For the threshold case, namely, tdh-op scheme of Fig. 6, if we assume that the client possesses the public keys g^{k_i} corresponding to the shares k_i of key $k = k_c$, then zero-knowledge proofs can be used too for verification. However, this prevents the ability to have a “proxy” (e.g., any one of the n servers) that does the aggregation of the b_i values returned by the servers into the OPRF result. With ZK verification, it is the client itself that needs to do this aggregation. This loses the “client transparency” property of tdh-op that has the important practical advantage that the client (and its software) need not be aware of the implementation of the server, whether it is a single-server deployment or a multi-server one.

Next, we present an alternative verification procedure that is client transparent. The client only needs to have the certified public key g^k for key k (regardless of the number of servers). We first describe the scheme for the case of the single-server OPRF dh-op from Fig. 3 and later extend it to the threshold case. (This works directly for OKMS, the adaptation to UOKMS is immediate.) The procedure is reminiscent of Chaum’s protocol for undeniable signatures [15] but simplified by dispensing of zero-knowledge proofs that are not needed here. It is easy to verify that the integrity guarantee is unconditional, namely, against unbounded attackers.

- On input x , C sets $h = H'(x)$, sets $r, c, d \leftarrow_R \mathbb{Z}_q$, and sends to server S the pair of values $a = h^r, b = h^c g^d$.

Key and server initialization.

Key $k \leftarrow_{\mathbb{R}} \mathbb{Z}_q$ is secret shared using Shamir's scheme with parameters n, t ;
Server $S_i, i = 1, \dots, n$, holds share k_i .

Threshold Oblivious Computation of $F_k(x)$.

- On input x , client C picks $r \leftarrow_{\mathbb{R}} \mathbb{Z}_q$ and computes $a := H'(x)^r$; it chooses a subset \mathcal{SE} of $[n]$ of size $t + 1$ and sends to each server $S_i, i \in \mathcal{SE}$, the value a and the subset \mathcal{SE} .
- Upon receiving a from C , server S_i verifies that $a \in G$ and if so it responds with $b_i := a^{\lambda_i \cdot k_i}$ where λ_i is a Lagrange interpolation coefficient for index i and index set \mathcal{SE} .
- When C receives b_i from each server $S_i, i \in \mathcal{SE}$, C outputs as the result of $F_k(x)$ the value $H(x, (\prod_{i \in \mathcal{SE}} b_i)^{1/r})$.

Figure 6: Protocol tdh-op [30]: (n, t) -threshold computation of dh-op from Fig. 3

- S responds with $A = a^k, B = b^k$.

- C checks that

$$A^{r'} = B^{c'} v^{-dc'} \quad (1)$$

where $r' = r^{-1}, c' = c^{-1}$ (and $v = g^k$). It rejects if the equality does not hold, otherwise C sets the value of $(H'(x))^k$ to $A^{r'}$ which it is already computed for equation (1).

This procedure involves running dh-op on two different values and then verifying consistency via a single multi-exponentiation by the client. The additional computational cost with respect to the base dh-op is a single exponentiation for the server and two multi-exponentiations for the client, essentially doubling the work for the non-verified case.

We now adapt the scheme to the threshold OPRF tdh-op. The client C sends the same pair of values (a, b) to each participant server S_i who responds with $A_i = a^{k_i}, B_i = b^{k_i}$. Upon gathering $t + 1$ responses, C interpolates in the exponent (one multi-exponentiation) to obtain values A, B and checks the identity (1). If it holds, C sets $(H'(x))^k$ to $A^{r'}$, else it applies the check (1) to each pair A_i, B_i received by participating server S_i using $v_i = g^{k_i}$ instead of v .

The computational cost in the normal case, where the verification against $v = g^k$ succeeds, is the same as in the single-server case except for one additional interpolation in the exponent. If verification against $v = g^k$ fails then the cost is an additional multi-exponentiation per each participating server. As said, the special feature of this procedure is that the client can interact with a proxy (or gateway) in a way that all operations by the client are identical to the single-server case. The proxy will send the values (a, b) generated by the client to the servers and will aggregate the responses A^i, B^i into a single response that can be verified with the public key g^k . Before sending to the client, the proxy can verify if the aggregation verifies correctly. If not, it needs to check the individual values sent by each server and discard the bad ones – all of this is done without any awareness by the client. Thus resulting in fully client-transparent solution.

6 IMPLEMENTATION AND PERFORMANCE

We report on implementation and performance of the OKMS and UOKMS schemes from Section 2.1 (Fig. 2) and Section 2.2 (Fig. 4), respectively.

OKMS Client Operations (Single Thread)		
	Wrap	Unwrap
Hash to Curve	58.26	58.26
Generate Blind	1.58	1.58
Apply Blind	68.07	68.07
Create Challenge	83.27	-
Inverse Blind	16.95	16.95
Remove Blind	68.07	68.07
Verify Response	106.67	-
Total Time (μs)	402.86	212.92
Operations / Second	2,482	4,696

Figure 7: Client operation time and Op/s in OKMS

Microbenchmarks. Implementations of all necessary client and server operations were written in C++ using the OpenSSL library (version 1.1.1-pre5) to provide cryptographic functionality. Performance tests were conducted on a machine with an Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz having 15 GB of memory. The implementation was compiled with the gcc compiler with optimization level 3.

The following tables detail the run times of each operation averaged over 10,000 trials. These tests used only a single thread and CPU core; results could be improved by performing these operations concurrently across multiple CPU cores.

In these tests, all elliptic curve operations were based on NIST P-256. Field operations (for Shamir and blinding factors) were defined over the prime order of NIST P-256. Hashing to the curve (as required by the OPRF defined in Fig. 3) was performed using SHA-256 and the constant time *Simplified SWU* algorithm [12].

Client operations in the OKMS scheme are shown in Fig. 7 for both encryption (“wrapping”) and decryption (“unwrapping”). The two operations differ only in that interactive verifiability is performed for wrap operations, while for unwrapping (the more common operation) regular symmetric key verification suffices.

When the (U)OKMS servers are deployed in a threshold architecture then some entity must perform polynomial interpolation “in the exponent”. This can be implemented as a multi-exponentiation of the individual server’s contributions together with the corresponding Lagrange coefficients. This interpolation operation could variously be performed by one of the servers, by the client, or by a dedicated intermediate entity.

Interpolation Layer Performance (Single Thread)		
(t+1)-of-N	Time (μ s)	Ops / Second
1-of-1	81.68	12,242
3-of-5	155.99	6,410
5-of-9	236.72	4,224
5-of-15	236.66	4,225
6-of-11	280.04	3,570

Figure 8: Interpolation layer performance for various threshold parameters

(U)OKMS Server Operations (Single Thread)		
	Time (μ s)	Ops / Second
Wrap	136.13	7,345
Unwrap	68.07	14,691

Figure 9: (U)OKMS Server performance for wrap and unwrap

UOKMS Operations (Single Thread)		
	Time (μ s)	Operations / Second
Wrap	24.24	41,261
Unwrap	162.33	6,160
Update	68.07	14,691

Figure 10: Client operation time and Op/s in UOKMS

We observe that the multi-exponentiation time dominates the cost of the interpolation (computing the Lagrange coefficients is correspondingly cheap), and we find that the total cost depends on the threshold rather than the total number of servers. We report interpolation times in Fig. 8.

Server operations, measured for the single server and threshold variants of this (U)OKMS scheme (implemented with protocol *tdh-pop*) are shown in Fig. 9. This includes only performing an exponentiation (EC scalar multiply) in the curve for each input provided by the client (the wrap operation is more costly as it includes an additional exponentiation to support the interactive verification procedure from Section 5.2).

Total time and operations per second is not significantly different for the threshold case, as each of the involved servers computes the same function in parallel.

Client performance in the UOKMS scheme (Sec. 2.2) is shown in Fig. 10. This setting benefits from being able to perform wrap operations without server involvement, and can further benefit from precomputation tables for exponentiation of g and g^k . In our testing, precomputation provided more than a 600% speed up (11.33 vs. 68.20 μ s). We summarize the number of operations per second the client can perform in the UOKMS.

(U)OKMS Server. To evaluate performance and scalability we hosted our (U)OKMS server implementation on Amazon’s Elastic Compute Cloud (EC2)[3] using a *c4.2xlarge* instance type. This instance type provides 8 virtual CPUS with an Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz having 15 GB of memory and was the

(U)OKMS Server Throughput (8 CPU cores)		
Scheme	KeepAlive	No KeepAlive
Static Page	65,018	6,462
OPRF (Unwrap)	32,094	6,349

Figure 11: (U)OKMS Server Requests/s on EC2 instance (LAN setting)

same instance type used to obtain the microbenchmark numbers above.

Requests to this server were issued over HTTP and the web server, *nginx*, was configured with 8 worker processes (one per CPU). OKMS functionality was added to this web server as a natively compiled module which used the OpenSSL library (version 1.1.1-pre5) to provide cryptographic functionality. The server ran Ubuntu 16.04 as its operating system.

Throughput. To measure throughput a client machine (also *c4.2xlarge*) was deployed in the same Amazon Web Service (AWS) availability zone as the server. We used the HTTP load generating tool *hey* to measure the throughput for each scheme. *hey* was configured with a concurrency level of 80 and all results were averaged over 50,000 requests. All requests were for an unwrap operation and were sent over HTTPS using (TLS 1.2 with ECDHE-ECDSA-AES256-GCM-SHA384). The server used a self-signed certificate with an EC key on the NIST P-256 curve. Computation time dominates in the LAN setting due to almost negligible network latency, with the CPU cores reaching near 100% utilization during the LAN throughput tests. To gauge the limits of the server performance, client-side operations of blinding and verification were not performed by the load generator.

For each scheme, we tested with session KeepAlive on and off. When off, a new TCP connection and TLS session must be negotiated for each request. When on, the connection setup costs are amortized over all requests, which is in line with a client that must unwrap many keys.

The table in Figure 11 details the observed throughput in requests per second (RPS) for the various schemes and two KeepAlive configurations (all over TLS). We compare these schemes to a static page as a baseline.

We observe that for the *No KeepAlive* configuration, the cost of creating the new connection and establishing the TLS session dominates resulting in very little difference in RPS between the schemes. For the *KeepAlive* configuration, throughput is significantly better, achieving over 30,000 RPS for the OPRF/T-OPRF case. Thus our (U)OKMS implementation can handle a large number of clients with a single server. For comparison, Amazon’s object storage service reported a peak load of 1.1 million requests per second [33]. If needed, the KMS implementation can be scaled with standard techniques, such as deploying a greater number of servers. With a few dozen servers in the KMS, a unique key could be supplied each time an object is written to or read from Amazon’s service.

Hardware Security Modules. A best practice for securing master keys is to keep them within Hardware Security Modules (HSMs)[4] to prevent their export to less secure locations. Fortunately, the

methods described in this paper are supported by existing commercial HSMs. Indeed, most HSMs support the PKCS#11 standard[42]. This specification defines an API method called CKM_ECDH1_DERIVE that takes an arbitrary point as an input and returns the x-coordinate of the point resulting from a scalar multiplication of the input point using an HSM-held private key as the scalar.

We tested three HSM implementations and found all supported the ECDH1 derive method. The returned x-coordinate is sufficient to perform an oblivious key derivation, and verification, but verification (only needed in the OKMS setting and for threshold implementations of the OPRF) requires that both positive and negative solutions for the y-coordinate be checked. By importing an elliptic curve private key computed as a Shamir share, existing HSMs can be used as part of a threshold implementation. Due to obliviousness, interpolating the result from a threshold of HSMs can be done external to the HSM without sacrificing confidentiality.

We note two potential limitations of using an HSM to hold the OPRF key. The first is that HSMs are often limited in the curves they support. While all of the HSMs we evaluated supported standard NIST curves, none supported Curve25519. The second limitation is performance. While high end commercial HSMs can achieve up to 22,000 scalar multiplications per second[48], this is roughly equivalent to what a single core can achieve in a multi-core server CPU.

While software implementations of symmetric key wrapping algorithms can be several orders of magnitude faster than asymmetric operations, we found HSMs often employ specialized hardware to accelerate the normally slower asymmetric operations. In some cases, HSMs[48] including those used by leading cloud providers[38], the number of supported ECC operations per second is comparable to that of supported symmetric encryption operations per second.

In conclusion, while in the traditional key-wrapping approach (Fig. 1) one can secure wrapping keys diligently e.g., in HSMs, the plain data encryption keys (dek) travel over much less secure TLS channels (sometimes ending or visible in multiple points outside the HSM boundary), and are potentially exposed to rogue administrators, accidental logging, etc. In contrast, these vulnerabilities are eliminated by the oblivious computation approach where as long as the OPRF key is secure, nothing can be learned about the data (other than by corrupting the client). Fortunately, securing these OPRF keys in HSMs is practical today as noted above, and while symmetric operations are less expensive than OPRF ones in general, HSMs with 20,000 EC op/sec can hardly be the system's bottleneck (of course, in large operations multiple HSMs will be used). Importantly, in UOKMS encrypting data does not necessitate of interaction with the KMS, further increasing performance. Additionally, the UOKMS approach offers much more efficient key rotation than traditional systems where rotation requires communication with the KMS for each key (dek or kek) to be updated. This slows down the rotation process, resulting in longer rotation periods and reduced security.

ACKNOWLEDGMENTS

We thank Anja Lehmann for very helpful discussions related to security notions of Updatable Encryption schemes. Our implementation experience and reporting has benefited enormously from the work of Martin Schmatz, Navaneeth Rameshan, and Mark Seaborn. We thank the CCS reviewers who helped improving the presentation of the paper.

REFERENCES

- [1] J. F. Almansa, I. Damg  rd, and J. B. Nielsen. Simplified threshold RSA with adaptive and proactive security. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 593–611, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [2] Amazon Web Services. Aws key management service cryptographic details, 2016. <https://d1.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf>.
- [3] Amazon Web Services. Aws elastic compute cloud, 2018. <https://aws.amazon.com/ec2/>.
- [4] E. Barker and W. Barker. Recommendation for key management, part 2: Best practices for key management organizations (2nd draft). Technical report, National Institute of Standards and Technology, 2018.
- [5] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003.
- [6] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The One-More-RSA-Inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, 2003.
- [7] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, Heidelberg, May / June 1998.
- [8] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In M. Franklin, editor, *CRYPTO 2004*, volume 3122 of *LNCS*, pages 41–55. Springer, Heidelberg, Aug. 2004.
- [9] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic prfs and their applications. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part I*, pages 410–428, 2013.
- [10] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic prfs and their applications. *IACR Cryptology ePrint Archive*, 2015:220, 2015.
- [11] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Heidelberg, Aug. 2011.
- [12] E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. *Cryptography ePrint Archive*, Report 2009/340, 2009. <http://eprint.iacr.org/2009/340>.
- [13] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. In M. Wiener, editor, *Advances in Cryptology - CRYPTO'99*, pages 98–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [14] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In E. Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003.
- [15] D. Chaum. Zero-knowledge undeniable signatures. In I. Damg  rd, editor, *EUROCRYPT'90*, volume 473 of *LNCS*, pages 458–464. Springer, Heidelberg, May 1991.
- [16] D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, Aug. 1993.
- [17] A. Davidson, A. Deo, E. Lee, and K. Martin. Strong post-compromise secure proxy re-encryption. In *Information Security and Privacy (ACISP) 2019*, 2019.
- [18] D. Derler, S. Krenn, T. Lorisser, S. Ramacher, D. Slamanig, and C. Striecks. Revisiting proxy re-encryption: Forward secrecy, improved security, and applications. In M. Abdalla and R. Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 219–250. Springer, Heidelberg, Mar. 2018.
- [19] A. Everspaugh, R. Chatterjee, S. Scott, A. Juels, and T. Ristenpart. The pythia PRF service. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 547–562, Washington, D.C., 2015. USENIX Association.
- [20] A. Everspaugh, K. G. Paterson, T. Ristenpart, and S. Scott. Key rotation for authenticated encryption. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017. Proceedings, Part III*, pages 98–129, 2017.
- [21] W. Ford and B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 176–180, Gaithersburg, MD, USA, June 4–16, 2000. IEEE Computer Society.

- [22] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In J. Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, Feb. 2005.
- [23] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In B. A. Coan and Y. Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998.
- [24] Google Cloud. Google cloud key management service, 2018. <https://cloud.google.com/kms/>.
- [25] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Heidelberg, Aug. 1995.
- [26] B. A. Huberman, M. K. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *EC*, 1999.
- [27] IBM. Ibm key protect, 2018. <https://console.bluemix.net/catalog/services/key-protect>.
- [28] S. Jarecki, A. Kiayias, and H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, Dec. 2014.
- [29] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. Highly-efficient and composable password-protected secret sharing (or: how to protect your bitcoin wallet online). In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 276–291. IEEE, 2016.
- [30] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In D. Gollmann, A. Miyaji, and H. Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017.
- [31] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. Cryptology ePrint Archive, Report 2017/363, 2017. <http://eprint.iacr.org/2017/363>.
- [32] S. Jarecki and X. Liu. Fast secure computation of set intersection. In J. A. Garay and R. D. Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 418–435. Springer, Heidelberg, Sept. 2010.
- [33] Jeff Barr. Amazon S3 – Two Trillion Objects, 1.1 Million Requests / Second, 2013. <https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/>.
- [34] M. Kloß, A. Lehmann, and A. Rupp. (R)CCA secure updatable encryption with integrity protection. In *Eurocrypt 2019*, 2019.
- [35] R. Lai, C. Egger, M. Reinert, S. Chow, M. Maffei, and D. Schröder. Simple password-hardened encryption services. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [36] A. Lehmann and B. Tackmann. Updatable encryption with post-compromise security. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, pages 685–716, 2018.
- [37] A. Y. Lindell. Adaptively secure two-party computation with erasures. In M. Fischlin, editor, *Topics in Cryptology - CT-RSA 2009*, pages 117–132, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [38] Microsoft. How many cryptographic operations are supported per second with dedicated hsm?, 2019. <https://docs.microsoft.com/en-us/azure/dedicated-hsm/faqs#performance-and-scale>.
- [39] Microsoft Azure. Azure key vault, 2018. <https://docs.microsoft.com/en-us/azure/key-vault/key-vault-overview>.
- [40] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.
- [41] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467. IEEE Computer Society Press, Oct. 1997.
- [42] OASIS Open. PKCS #11 Cryptographic Token Interface Base Specification Version 2.40, 2015. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html>.
- [43] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *10th ACM PODC*, pages 51–59. ACM, Aug. 1991.
- [44] A. Patel and M. Yung. Fully dynamic password protected secret sharing, 2017. manuscript.
- [45] T. P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract) (rump session). In D. W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 522–526. Springer, Heidelberg, Apr. 1991.
- [46] Y. Polyakov, K. Rohloff, G. Sahu, and V. Vaikuntanathan. Fast proxy re-encryption for publish/subscribe systems. *ACM Transactions on Privacy and Security (TOPS)*, 20, 2017.
- [47] K. Sakurai and Y. Yamane. Blind decoding, blind undeniable signatures, and their applications to privacy protection. In *Proceedings of the First International Workshop on Information Hiding*, pages 257–264, London, UK, UK, 1996. Springer-Verlag.
- [48] Thales. SafeNet Luna Network HSM, 2019. <https://safenet.gemalto.com/resources/data-protection/luna-sa-network-attached-hsm-product-brief/>.

A PROOF OF THE OMDH-IO ASSUMPTION IN THE GENERIC GROUP MODEL

We sketch the steps for adapting the GGM proof of OMDH from [31] to the OMDH-IO case. As argued in [31], it suffices to show OMDH security for $N = Q + 1$, in which case the upper-bound on a probability that a GGM adversary solves the OMDH problem in a group of prime order q while making r group operations and Q queries to the exponentiation oracle $(\cdot)^k$ is $(Q(2Q + r)^2)/q$. In a GGM proof, see Theorem 6 in Appendix A in [31], every group element the adversary obtains is represented with a (random string assigned to) a polynomial in unknowns (u_1, \dots, u_N, k) for $u_i = \text{DL}(g, g_i)$. Group multiplications or divisions correspond to, respectively, adding or subtracting such polynomials, and querying oracle $(\cdot)^k$ on a group element corresponds to multiplying the corresponding polynomial by k . The proof argues that the only way the adversary can win is either if some two different polynomials it creates have equal values on random inputs (u_1, \dots, u_N, k) , or that the group elements it outputs correspond to polynomials $k \cdot u_1, \dots, k \cdot u_N$. The latter case is easily seen as impossible for an adversary which can has only $Q = N - 1$ accesses to the “multiply-a-polynomial-by- k ” oracle $(\cdot)^k$, while the upper-bound on the probability of the first case comes from the fact that there are at most $2Q + r$ polynomials, each one has at most degree Q in k (and linear in variables u_1, \dots, u_N), and the fact that a non-zero Q -degree polynomial can have at most Q roots, hence each pair of different polynomials can evaluate to the same value on random exponent (u_1, \dots, u_N, k) with probability at most Q/q . If the GGM adversary in addition makes t queries to the inverse-exponentiation oracle $(\cdot)^{1/k}$, each query multiplies the corresponding polynomial by k^{-1} , and the resulting polynomials, after multiplying all of them by k^t , can be thought of as polynomials of degree at most $Q + t$ instead of Q . Thus by the same argument, the upper-bound on the probability of GGM adversary to solve all $N = Q + 1$ challenges is bounded by $(Q + t)(2Q + t + r)^2/q$. Note that $r >> \max(Q, t)$ in typical applications, including our UOKMS scheme, hence this bound can be approximated as $(Q + t)r^2/q$.