

基于 FPGA 的 Leighton-Micali 签名方案密钥生成的高速可配置实现

胡 潇, 宋逸峰, 汪文浩, 田 静

南京大学 电子科学与技术工程学院 南京 中国 210023

摘要 由于量子计算机的飞速发展, 现代密码学面临着巨大的挑战。为了实现抗量子计算机攻击的加密, 人们提出了许多新的加密方案, 并对后量子密码学(Post-Quantum Cryptography, PQC)开展了标准化进程。Leighton-Micali 签名(Leighton-Micali signature, LMS)是一种基于哈希的后量子签名方案, 其私钥和公钥尺寸都较小, 且安全性已被充分研究。LMS 被互联网工程小组(Internet Engineering Task Force, IETF)选为 PQC 签名协议的标准方案, 同时被美国国家标准技术局(National Institute of Standards and Technology, NIST)选为一种 PQC 过渡方案。然而, 密钥生成过程中的效率低下, 成为了 LMS 实际应用中的瓶颈。在本文中, 我们首次对 LMS 进行基于 FPGA 的硬件实现与加速。首先, 在不损失安全性的基础上, 我们将 LMS 中的主要哈希函数由 SHA2 替换为 SHA3 函数。其次, 我们设计了一个软硬件协同系统, 将核心的哈希运算用硬件进行实现, 该系统在消耗较少资源的前提下, 可完成 LMS 协议的所有过程: 密钥生成、签名与验证。该系统为物联网(Internet of things, IoT)场景下资源受限的 LMS 应用提供了参考。接着, 我们提出了一个高速的密钥生成架构来加速 LMS。该架构中具有可配置性, 支持 LMS 的所有参数集, 内部的哈希模块根据使用场景进行设计与部署, 且并行度经过精心设计, 以使得架构同时达到低延迟和高硬件利用率。此外, 设计中的控制逻辑被设计为在适应不同参数集的情况下保持一定程度的恒定功率, 以抵御功率分析攻击。该架构使用 Verilog 实现, 并在 Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA 平台上实验。实验结果表明, 与在 Intel(R) Core(TM) i7-6850K 3.60GHz CPU 上启用多线程的目前较优软件实现相比, 本文中的设计在不同参数配置下可实现 55x ~ 2091x 的加速; 与最新的各平台 LMS 工作相比, 本文中设计可实现超过 17x 的加速; 与相近方案的 FPGA 工作相比, 本文中设计可实现约 70x 的加速。

关键词 Leighton-Micali 签名协议; 基于哈希的签名算法; 后量子加密; 硬件实现; 可编程逻辑阵列

中图分类号 TN47 DOI号 10.19363/J.cnki.cn10-1380/tn.2021.11.02

High-Speed and Configurable FPGA implementation of the Key Generation for Leighton-Micali Signature Protocol

HU Xiao, SONG Yifeng, WANG Wenhao, TIAN Jing

Department of Electronic Science and Engineering, Nanjing University, Nanjing 210023, China

Abstract Due to the rapid progress made in quantum computers, modern cryptography is facing great challenges. Many digital signature schemes that have resistance to quantum computing are proposed, and the Post-Quantum Cryptography (PQC) standardization is launched. The Leighton-Micali signature (LMS) is a kind of hash-based signature scheme. It has relatively small private and public keys and its security is well-studied. LMS is selected as a standard scheme for the PQC signature protocols by the Internet Engineering Task Force (IETF), and in the meanwhile, it is recommended as one of the transition schemes by the National Institute of Standards and Technology (NIST) before the PQC standardization finishes. However, the low-efficiency in the key generation process forms the bottleneck in practical applications of LMS. In this article, for the first time, the FPGA-based hardware accelerator is introduced for the LMS scheme. Firstly, we replace the main hash function SHA2 with SHA3 without loss of security. Then, we propose a hardware/software co-design, in which the hash function is offloaded to hardware platform. The co-design can achieve all the procedures of LMS: key generation, signature, and verification. It provides a reference for resource-constrained LMS applications under the Internet of Things (IoT) scenario. Moreover, we propose a high-speed key generation architecture to accelerate LMS. The architecture is delicately devised to be scalable, supporting all the parameter sets for the LMS. The internal hash modules are specifically developed and deployed based on the application scenarios, and the degree of parallelism is carefully designed to achieve low latency and high hardware utilization efficiency. Moreover, the control flow is well managed to accommodate differ-

通讯作者: 田静, 博士, 副研究员, Email: tianjing@nju.edu.cn。

本课题得到国家自然科学基金资助项目(No.61774082)、中央高校基本科研业务费专项资金资助项目(No.021014380065)、江苏省重点科研计划资助项目(No.BE2019003-4)资助。

收稿日期: 2021-04-29; 修改日期: 2021-09-06; 定稿日期: 2021-10-19

ent parameter sets with constant power for the consideration of anti-power analysis attacks. We code our design with Verilog language and implement it on the Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA. The experimental results show that, compared with the superior software implementation running on an Intel(R) Core(TM) i7-6850K 3.60GHz CPU with threading enabled, the new design achieves 55x to 2091x speedups in different parameter configurations; compared with the state-of-the-art works of LMS, the design achieves more than 17x speedup on various platforms; compared with the work of similar scheme on FPGA, about 70x speedup is realized.

Key words leighton-micali signature (LMS); hash-based signatures (HBS); post-quantum cryptography(PQC); hardware implementation; field-programmable gate array (FPGA)

1 引言

由于量子计算的飞速发展, 现代密码学面临着巨大的挑战。目前常用的传统签名方案, 如 RSA^[1], DSA^[2], ECDSA^[3]和 EdDSA^[4]等算法, 都基于某些数学假设如大整数因式分解困难与计算离散对数的困难。然而, Shor 的算法^[5]证明, 这类困难在量子计算机上将会被快速解决。而随着量子计算领域快速发展, 学者预测量子计算机将会在不远的将来被设计出来。因此, 可抵御量子计算机攻击的后量子加密方案正在被积极探索, 标准化团队与学术团体也正在对这些方案积极开展标准化进程。

美国国家标准技术研究所(National Institute of Standards and Technology, NIST)于 2016 年提出后量子标准方案评审标准和要求^[6]后, 向学术界和工业界广泛征集方案。2017 年进行第一轮的评审, 在收到的 82 个方案中选出了 69 个候选方案^[7], 并在 2018 年召开了第一届 PQC 的标准化会议^[8]。随后进行了更全面更进一步的评估, 于 2019 年召开了第二届 PQC 标准化会议, 由第一轮 69 个候选方案中选出了 26 个方案进入第二轮^[7]。2020 年的第三轮评审选出了 7 个入围方案与 8 个候补方案^[9], 第三届的 PQC 标准化会议则会在 2021 年举行, 预计 2022 年年初 NIST 将公布标准初稿并公开征求建议^[9]。

基于哈希的签名方案(Hash-based Signature, HBS)是后量子加密方案中最有潜力的类别之一^[9-10], 其中主要的计算函数是单向哈希函数(Hash)如 SHA2^[11]和 SHA3^[12]。相较于其他的后量子加密方案, 如基于格^[13]、基于同源^[14]、基于编码^[15]的加密方案, 基于哈希的加密方案经过充分的研究, 不需要额外的数学假设, 安全性完全依赖于内部哈希函数的安全性^[10,16-19], 更易于实现^[20], 具有更小的签名尺寸^[16], 且在签名和验证的计算效率上也常常优于其他类别^[21]。

HBS 方案由默克尔树(Merkle Tree)与单次签名(One Time Signature, OTS)构成^[22]: OTS 方案中, 一个私钥用于一条信息的签名, 对应的公钥用于验证

签名的正确性; 而 HBS 的树为一棵二叉默克尔树(以下简称二叉树), 其中树的每个叶子节点都是一个 OTS 公钥的哈希值, 内部节点是用其对应的子节点计算出的哈希值, 树的根节点用于构成 HBS 的公钥。签名时, HBS 的签名由一个 OTS 的叶子节点与该叶子节点到达根节点的路径节点构成。这些路径节点用于帮助签名的验证。

Leighton-Micali 签名(Leighton Micali Signature, LMS)^[23]和扩展的默克尔签名方案(eXtended Merkle Signature Scheme, XMSS)^[24]是两种目前进入因特尔工程任务组(Internet Engineering Task Force, IETF)标准化进程的高效 HBS 方案^[25]。并于 2020 年十月份由 NIST 确定为两种过渡方案标准, 在最终 PQC 标准确定下来之前针对有抗量子签名需求的应用所使用^[16]。这两种方案都是由经典的默克尔算法发展而来的^[22], 是目前最新与最成熟的两种 HBS 方案^[16]。与 XMSS 相比, LMS 具有密钥和签名尺寸更小的优势, 这使得该方案在很多场景下具有更多潜力, 特别是传输带宽受限的场景^[18]。但是, 在相同的安全等级下, 相对于 XMSS, LMS 需要进行更多哈希计算。故而对该方案的核心哈希运算进行加速将极大程度有利于该方案的实际应用。且对于 LMS 方案的三个主要过程: 密钥生成(Key generation)、签名(Signature)和验证(Verification), 密钥生成过程是消耗哈希计算最多的过程, 平均哈希运算次数是后两个过程的两倍^[23]。因此, 加速密钥生成过程可以有效提高整个协议的速度。

需要说明的是, 由于 LMS 方案是一种有状态(Stateful)的 HBS 方案, 不能使用同一个私钥对不同信息进行多次签名, 否则存在被第三方伪造签名的可能性^[23]。同时, 由于 LMS 的签名中包含由内部节点构成的路径, 需要在签名前结束生成密钥的过程。考虑到 HBS 方案的主要应用场景是物联网(Internet of Things, IoT)^[18, 26], 在这些应用场景下, 将会产生大量的签名, 而由于前述的两种特性, 单棵 LMS 树生成的密钥安全时间有限^[23,27]。因此, 加速密钥生成, 对推动 LMS 方案进入实际应用具有重要意义。同时,

考虑到物联网中资源受限的使用场景, 设计速度与面积之间的折衷方案(如软硬协同系统)也将对推进 LMS 进入实际应用产生积极影响。

对于 LMS 相关实现工作, 对于签名和验证的加速与优化工作取得了较多的进展^[18,20,28-30]。但是, 对于密钥生成过程, LMS 方案作为一种较新提出的后量子签名方案, 目前的工作进展较少: 文献[23]中进行了基于多线程 CPU 的实现, 给出了同一参数的不同树高的密钥生成; 文献[31]和文献[32]提供了一种基于 ARM 平台的低功耗 LMS 实现; 其后, 文献[29]与文献[30]中提出了一种基于 CPU 的 LMS 安全启动应用。而对于软硬件协同系统, 目前 LMS 方案没有对应的相关工作。综上, 基于 FPGA 的硬件加速器与软硬件协同系统将填补这部分工作的空缺从而对 LMS 协议的实际应用有较大的推进作用。

本工作中, 我们结合算法特征和架构优化, 首先设计了一种高效的 SHA3 模块, 并基于此模块提出了基于 ZYNQ 架构的软硬协同系统。同时, 对于 LMS 中最耗时的密钥生成过程, 设计了一种高效且可配置的架构。为了同时获得低延迟与高硬件利用率, 该架构的并行度根据参数集进行设计。仿真与 FPGA 实现结果显示, 该架构较现有的 CPU 实现在速度上有 55~2091 倍的速度提升。

具体的工作总结如下:

- 相较于原有的 LMS 方案, 选择使用比 SHA2 哈希函数安全度更高的 SHA3 哈希函数, 进行了软件验证。并对 SHA3 实现了适用于 LMS 方案的架构设计, 以提升哈希模块的硬件利用率。本文中对哈希模块进行了单独的性能评估与比较, 与最新的相关工作相比, 本文中设计的哈希模块具有相当的吞吐率性能。
- 基于 SHA3 架构, 提出一种基于 ZYNQ 开发板的软硬协同系统, 其中核心 SHA3 模块由可编程逻辑实现, 其余操作由 CPU 实现。该软硬协同系统在占用较少资源的基础上, 支持 LMS 的整个协议过程, 为 IoT 场景下资源受限的 LMS 应用提供了参考。
- 针对 LMS 方案中耗时最长的密钥生成过程, 设计了一种高速且可配置的专属加速器。该加速器可以容纳 LMS 原方案中提出的所有参数, 在高并行度与高硬件利用率上取得了最佳的折中。当二叉树深度大幅增加, 只增加非常少的硬件消耗。且本文的架构被设计为在不同参数运行下均可保持功率恒定, 一定程度上可以抵御功率攻击。

- 对于设计的密钥生成架构, 在 Xilinx Zynq UltraScale+ 的开发板上进行了实现, 在使用的硬件资源很少的基础下, 实现了 285M 赫兹的时钟频率。相较于基于 SHA3 的源码在 CPU 上的多线程实现, 本工作实现了 2~4 个数量级的速度提升。

需要说明的是, 本文中关于全硬件密钥生成架构的初步介绍已发表文献[33], 而本文新的贡献点在于: (1)软硬协同系统的设计; (2)哈希模块的性能评估与比较; (3)对于全硬件密钥生成架构, 新增了公钥生成模块的可停止处理, 和可支持的最大树高阈值分析与讨论。

本文结构如下: 第 2 节介绍 LMS 签名方案的主要流程与密钥生成部分的具体算法; 第 3 节具体介绍核心哈希函数的硬件实现、使用硬件进行哈希计算的软硬协同系统的设计以及 LMS 签名方案密钥生成部分的全硬件实现设计; 第 4 节介绍实验过程并对实验结果进行对比与分析; 第 5 节为总结与展望。

2 LMS 签名方案描述

本章节中, 首先简要介绍 LMS 签名方案的主要流程, 然后介绍目前已有的 LMS 参数集, 最后详细介绍 LMS 密钥生成算法以及 SHA3 哈希算法, 即本工作主要关心的算法。

2.1 LMS 方案主要流程

与传统签名方案一致, LMS 方案的三个主要过程为: 密钥生成、签名和验证。对于二叉树树高为 h 的 LMS 方案, 一次密钥生成产生的密钥可用于签名与验证 2^h 个信息。需要说明的是, 对树高为 h 的 LMS 树, 所有节点均有自己的索引值, 范围是 $1 \sim 2^{h+1}-1$ 。对内部节点编号(Node number)规则是, 对于编号 $N < 2^h$ 的内部节点, 它的左右子节点的编号分别为 $2N$ 和 $2N+1$; 对于叶子节点, 叶子编号(Leaf number)依次为 $2^h, 2^h+1, \dots, 2^{h+1}-1$ 。

密钥生成: 密钥生成阶段, 根据 LMS 参数类型 lms_type 确定树高 h , 根据 OTS 类型 ots_type 确定 OTS 内部参数。首先, 以 $0 \sim 2^h$ 为索引生成 2^h 组 OTS 密钥对, 其中的私钥用以构成 LMS 私钥。然后开始构造二叉树以生成 LMS 公钥: 二叉树的 2^h 个叶子节点中, 每个节点都包括叶子编号以及一个 OTS 公钥的信息; 内部节点的值 $T[r]$ 是由其两个子节点的值 $T[2r]$ 和 $T[2r+1]$ 连接后再进行哈希计算得到的。二叉树的根节点的值 $T[1]$ 用以构成 LMS 的公钥, 此公钥可以验证 2^h 个签名。

签名: 单个 OTS 的私钥对某个信息产生的签名,

可由对应的 OTS 公钥进行验证。而在 LMS 方案中, 单次签名过程中使用某个叶子节点对应的 OTS 私钥进行签名。产生的 LMS 签名中, 除了 OTS 签名, 还包括 OTS 类型 ots_type 、叶子节点索引 q 以及从该叶子节点计算到根节点的路径(Array path)。对于树高为 h 的二叉树, 该路径包括 h 个节点的哈希值。如图 1 所示, 对于层数为 3 的二叉树中的叶子节点 $leaf^*$, 路径包括该叶子节点的兄弟节点 $leaf_l$, 通过这两个节点可计算出 $T[5]$, 故往下计算需要兄弟节点 $T[4]$, 同理, 第三层需要的节点值为 $T[3]$ 。

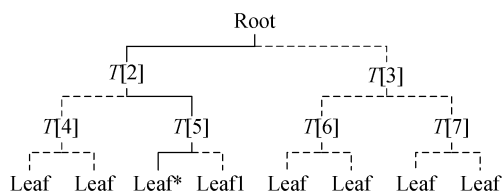


图 1 LMS 签名中的路径

Figure 1 Array path of LMS signature

验证: 对于某个签名, 首先使用其中的 OTS 签名部分计算出 OTS 公钥, 再使用此公钥计算出对应

的叶子节点值。接着根据签名中的路径计算出根节点。计算出的根节点如果与 LMS 公钥相符, 则签名有效, 否则无效。

2.2 OTS&LMS 参数集

首先, LMS 签名协议被提出的同时, 相应的适用参数集也被提出, 包括 OTS 参数与 LMS 参数。

表 1 为 OTS 方案的参数, LMS 中采用 Winternitz OTS(WOTS)^[34]作为其 OTS 方案, 其中的主要参数为 ots_type , H , n , w , p 。其中, ots_type 为 4 个字节长的参数类型标识符, 即通过 ots_type 可知其他 OTS 参数的值; H 指哈希函数的类型, 在本工作中选择高安全度与高灵活性的 SHA3_256 函数; n 指哈希函数输出的字节数, 这个数的选取对整个签名方案的安全性都有较大影响; w 是 Winternitz 系数的位宽, 其选取对安全性几乎没有影响, 它的值描述了签名长度与计算延时的折中: 增加 w 的大小会减少签名的长度但相应的密钥生成的时间会增加; p 是由 n 和 w 决定的参数, 同时决定了密钥生成中的哈希计算的迭代次数; sig_len 指签名的字节数, 一般来说, $sig_len = 4 + n(p + 1)$ 。

表 1 OTS 参数集

Table 1 Parameter sets of OTS

参数类型 ots_type	哈希函数类型 H	哈希输出 n /byte	折中 w	参数 p	签名长度 sig_len /byte
SHA3_256_N32_W1	SHA3_256	32	1	265	8516
SHA3_256_N32_W2	SHA3_256	32	2	133	4292
SHA3_256_N32_W4	SHA3_256	32	4	67	2180
SHA3_256_N32_W8	SHA3_256	32	8	34	1124

表 2 为 LMS 方案的参数, 主要参数为 lms_type , H , m , h 。其中, lms_type 为 4 个字节长的参数类型标识符, 即通过 lms_type 可知其他 LMS 参数的值; H 指哈希函数的类型, 为保证安全性, LMS 中哈希函数与 LM-OTS 中哈希函数应保持一致, 防止攻击者对其安全性较弱的哈希函数进行攻击^[23], 故本工作中实现的哈希函数均为 SHA3_256 函数, 此后不再赘述; m 指 LMS 树的节点的值的字节长度, 与 LMS 选

取的哈希函数的输出长度一致, 由于 H 不做区分, 故 m 与表 1 中 n 相等, 均为 32, 下文不再做区分; h 指 LMS 的树高。

2.3 LMS 密钥生成

本小节首先介绍 LMS 算法中核心的哈希函数; 然后介绍 WOTS 的密钥对生成算法, 最后介绍 LMS 的密钥对生成算法。此节介绍的算法参考了文献[7]与[23]。

表 2 LMS 参数集

Table 2 Parameter sets of LMS

参数类型 lms_type	哈希函数类型 H	哈希输出 m /byte	树高 h
SHA3_256_M32_H5	SHA3_256	32	5
SHA3_256_N32_H10	SHA3_256	32	10
SHA3_256_N32_H15	SHA3_256	32	15
SHA3_256_N32_H20	SHA3_256	32	20
SHA3_256_N32_H25	SHA3_256	32	25

2.3.1 SHA3_256 函数

对于哈希函数, 本工作选择比原 LMS 方案中 SHA2 函数更新一代的 NIST 第三代标准哈希函数 SHA3 函数。为了与文献[23]中安全度一致, 选择输出长度为 256 比特的 SHA3 函数即 SHA3_256 函数。

SHA3_256 函数的计算流程图由图 2 所示, 首先包含一组打包过程(Pad), 此过程是将不定长的数据打包为 1088 比特整数倍长度的数据方便后续处理, 具体打包规则见文献[7]; 接着将打包后的数据流分别传输到核心函数 f 中。

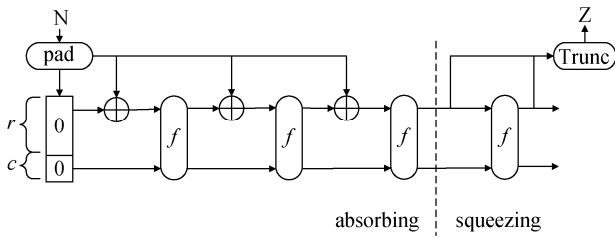


图 2 SHA3 函数

Figure 2 SHA3 function

核心 f 函数(KECCAK 函数)主要由 THETA, RHO, PI, CHI, IOTA 5 步组成。具体算法细节见算法 1, 其中异或 XOR (\oplus), 取反 NOT 和按位与 AND 操作均是位逻辑运算, 旋转 ROT 是逐位循环移位操作, RC 是一个包含 24 个元素的常数序列。

算法 1. SHA3 核心函数——KECCAK.

输入: 状态数组 A ;

输出: 新状态数组 A' ;

1. FOR $i = 0; i < 24; i = i + 1$ DO
2. THETA: ($0 \leq x, y \leq 4, 0 \leq z < 25$)
3. $C[x, y, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$
4. $D[x, z] = C[(x-1), z] \oplus \text{ROT}[C(x+1), 1]$
5. $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$
6. RHO: ($0 \leq x, y \leq 4$)
7. $A[x, y, z] = \text{ROT}(A'[x, y, z], r[x, y])$
8. PI: ($0 \leq x, y \leq 4, 0 \leq z < 25$)
9. $B[y, (2x+3y), z] = A[x, y, z]$
10. CHI: ($0 \leq x, y \leq 4, 0 \leq z < 25$)
11. $A[x, y, z] = B[x, y, z] \oplus$
 $(\text{NOT}(B[(x+1), y, z] \text{ AND } (B[(x+2), y, z])))$

12. IOTA: ($0 \leq z < 25$)
13. $A'[0, 0, z] = A[0, 0, z] \oplus \text{RC}[z]$
14. END FOR

2.3.2 OTS 密钥对生成算法

OTS 密钥对生成算法中包括私钥生成与公钥生成, 以下分别进行介绍。

OTS 私钥生成算法如算法 2 所示: 输入随机值 I 和 $seed$, 叶子节点索引值 q 和参数类型 ots_type , 输出一个 OTS 私钥。需要生成序列数为 p 的 x 序列。使用初始 $seed$ 值与参数 $I, q, 0\text{xff}$ 和索引值 i 进行一次哈希计算, 即可得到一个 $x[i]$ 。注意此处的 0xff 是一个 1 字节长的常数, 索引值 i 是一个 2 字节长的无符号数。故一次 OTS 私钥计算过程将调用 p 次哈希计算。生成的 x 序列与 I, q 和 ots_type 值组成一个 OTS 私钥。

算法 2. OTS 私钥生成流程(OTS_pri_key_gen).

输入: 16 字节长的随机数 I , 32 字节长的随机数 $seed$, 4 字节长的参数类型 ots_type , 4 字节长的叶子节点索引 q ;

输出: OTS 私钥

$\text{priv_key} = \{ots_type \| I \| q \| x[0] \| \dots \| x[p]\};$

1. 根据 ots_type 与表 1 设置参数 p .
2. FOR $i = 0; i < p; i = i + 1$ DO
3. $x[i] = H(I \| q \| i \| 0\text{xff} \| seed)$;
4. END FOR

OTS 公钥生成算法如算法 3 所示: 输入随机值 I , 参数类型 ots_type , 一个 OTS 私钥, 输出一个 OTS 公钥。首先取出私钥中的 x 序列值, 再对其进行哈希计算。对于每个索引值为 i 的 $x[i]$, 进行 $2^w - 1$ 次哈希的迭代计算, 每次计算输入为上一次哈希输出值 temp (初始 $\text{temp} = x[i]$) 与 I, q, i 和迭代数的索引值 j 的组合值, 第 $2^w - 1$ 次哈希的输出值为 $y[i]$ 。注意此处的索引值 i 是 2 字节长的无符号数, 内部循环 j 是 1 字节长的无符号数。对于长度为 p 的 x 序列, 进行 p 组迭代, 得到序列长度为 p 的 y 序列。将 y 序列与 $I, q, D_PBLC = 0\text{x8181}$ 组合, 再进行一次哈希计算, 得到 OTS 公钥的主要组成部分 K 。

算法 3. OTS 公钥生成流程.

输入: 16 字节长的随机数 I , 32 字节长的随机数 $seed$, OTS 私钥 priv_key ;

输出: OTS 公钥 $\{ots_type||I||q||K\}$;

1. 根据 ots_type 与表 1 设置参数 p 与 w .
 2. 从 $priv_key$ 中得到 x 序列, 随机数 I , 和参数 q .
 3. FOR $i = 0; i < p; i = i + 1$ DO
 4. $temp = x[i]$;
 5. FOR $j = 0; j < 2^w - 1; j = j + 1$ DO
 6. $temp = H(I||q||i||j||temp)$.
 7. END FOR
 8. $y[i] = temp$;
 9. END FOR
 10. $K = H(I||q||D_PBLC||y[0]||\dots||y[p])$.
-

2.3.3 LMS 密钥生成算法

LMS 密钥算法中包括私钥生成与公钥生成, 以下分别进行介绍。

算法 4.LMS 私钥生成流程.

输入: 参数类型 lms_type 与 ots_type ;
输出: 树高为 h 的 LMS 签名方案的私钥;

1. 根据 lms_type 和表 2 确定树高 h .
 2. 设置 I 为一个 16 字节长的随机数.
 2. 设置 $seed$ 为一个 32 字节长的随机数.
 3. FOR $q = 0; q < 2^h; q = q + 1$ DO
 4. $PTS_PRIV[q] =$
 $OTS_pri_key_gen(I, q, ots_type, seed)$
 5. END FOR
 6. 设置 $q = 0$.
-

LMS 私钥生成算法如算法 4 所示: 输入参数类型 ots_type , 树的深度 h , 输出 LMS 的全部私钥值。 $seed$ 和 I 分别是长度为 32 比特和 16 比特的初始随机值, 一棵二叉树使用的 $seed$ 和 I 在初始化后不再改变; h 指 LMS 二叉树的深度, 对于二叉树深度为 h 的 LMS 签名协议, 私钥生成阶段需要生成 2^h 个 OTS 私钥, 即调用 2^h 次 OTS 私钥生成算法。 q 为叶子节点的索引值。

算法 5.LMS 公钥生成流程.

- 输入: OTS 公钥中的哈希数组 K ;
输出: LMS 公钥 $\{lms_type||ots_type||I||T[1]\}$;
1. FOR $r = 2^{(h+1)} - 1; r > 1; r = r - 1$ DO
-

2. IF $r \geq 2^h$ THEN

3. $T[r] = H(I||r||D_LEAF||K[r - 2^h])$.

4. ELSE

5. $T[r] = H\left(\begin{matrix} I||r||D_INTR|| \\ T[2*r]||T[2*r+1] \end{matrix}\right)$.

6. END IF

7. END FOR

LMS 公钥生成算法如算法 5 所示: 输入 2^h 个 OTS 公钥值的 K , 输出 LMS 的公钥值。首先, 参数 $D_LEAF=0x8282$, $D_INTR=0x8383$; 假设对于每个节点的索引值为 r , 且对于每个索引值为 r 的内部节点, 其子节点的索引值为 $2r$ 和 $2r+1$ 。对于输入的 2^h 个 K , 索引值范围为 $2^h \sim 2^{h+1}$ 。当 $r \geq 2^h$ 时, 此时计算叶子节点的值, 将该 K 值与 I, r, D_LEAF 组合后进行一次哈希计算; 当 $r < 2^h$ 时, 此时计算内部节点的值, 将该叶子节点值与其兄弟节点的值组合, 再与 I, r, D_INTR 组合后进行一次哈希计算, 得到这两个节点的父节点的值。层层计算后, 最终得到的根节点值 $T[1]$ 。将 $T[1]$ 与参数类型 ots_type , OTS 类型 ots_type 和随机值 I 组合后, 即构成 LMS 公钥。该公钥可用于验证 2^h 个签名。

3 本文工作

本文实现了 LMS 协议的软硬协同系统与针对密钥生成过程的全硬件架构设计。本章首先介绍核心 SHA3 函数的相关设计, 接着介绍软硬协同系统, 最后介绍密钥生成架构。

3.1 SHA3 函数模块设计

哈希函数是整个 LMS 密钥生成过程中最核心的计算模块, 其算法过程已经在章节 2.2.1 中详细描述, 在这里介绍几个本工作中实际使用到的模块。

3.1.1 KECCAK 模块

SHA3_256 中核心的 KECCAK 模块如图 3 所示。同算法 1, 该模块的主要子模块包括 THETA, IOTA,

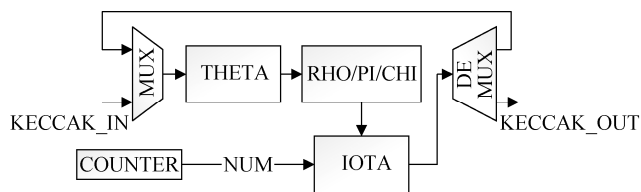


图 3 KECCAK 模块

Figure 3 KECCAK module

RHO, PI, CHI。主要运算均为逻辑运算, 移位, 逻辑与, 异或等, 子模块中的具体逻辑运算如算法 1 中所示。KECCAK 模块被设计为一次性可最多处理 1600 比特的输入且在 27 个时钟周期内完成计算。

3.1.2 HASH_X 和 HASH_LS 模块

由算法 2、3、5 可知, 在密钥生成过程中, 大致

需要两种哈希函数, 一种用于处理定长数据, 另一种处理不定长数据。具体的各哈希函数输入长度如表 3 所示。

为了简化不同哈希模块中 PAD 模块的复杂度并减少硬件资源消耗, 设计中使用了两种哈希模块: HASH_X 和 HASH_LS。

表 3 哈希输入数据的长度
Table 3 Inputs' data lengths of HASH function

算法	哈希输入数据(字节/比特)	总长度(字节/比特)
2	$I, q, i, 0\text{xff}, \text{seed}$ 16/128, 4/32, 2/16, 1/8, 32/256	55/440
3 (计算 $y[i]$)	I, q, i, j, temp 16/128, 4/32, 2/16, 1/8, 32/256	55/440
3 (计算 K)	$I, q, D_PBL, y[0], \dots, y[p]$ 16/128, 4/32, 32/256, \dots , 32/256	$20 + 32 \times p / (20 + 32 \times p) \times 8$
5 (叶子节点)	$I, r, D_LEAF, K[r-2^n]$ 16/128, 4/32, 2/16, 32/256	54/432
5 (内部节点)	$I, r, D_INTR, T[2r], T[2r+1]$ 16/128, 4/32, 2/16, 32/256, 32/256	86/688

HASH_X(图 4)用于处理字节数为 X 且 $X \leq 1088$ 的定长输入, 包含一个 PAD 模块与一个 KECCAK 模块。PAD 模块按照 NIST 标准将输入打包成长度为 1088 整数倍比特数的数据。LMS 过程中, 部分计算模块中的哈希输入一直为定长, 用 HASH_X 模块实现将降低这些哈希内部 PAD 模块与控制逻辑的硬件复杂度。由表 3 可知, 需要处理的定长数据为 440 比特、432 比特和 688 比特, 故实际使用到的 HASH_X 为 HASH_440, HASH_432 和 HASH_688。

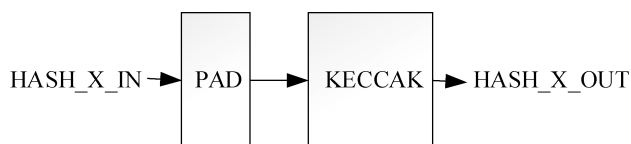


图 4 HASH_X 模块
Figure 4 HASH_X module

HASH_LS(图 5)用于处理字节数超过 1088 比特的不定长数据。输入数据首先被填充(PAD)成为长度为 1088 比特整数倍的数据, 接着被缓存 FIFO (Buffer FIFO)进行缓存和切割为 1088 比特长度的数据流 (STREAM), 再进行后处理。多路选择器(MUX)与多路分配器(DEMUX)用于处理当前 KECCAK 的输入与输出: KECCAK 首轮输入为初始值(INITIAL_VALUE)与第一个 1088 比特数的数据的异或值, 其余时刻输入为上一轮 KECCAK 的 1600 比特输出状态(STATE)与当前输入的 1088 比特数据的异或值; 迭代数据达到数据流的个数时, HASH_LS 模块输出当前的 KECCAK 状态值, 作为整个 HASH_LS 的最终输出值。

需要说明的是, HASH_X 模块消耗的时钟周期数(Clock cycles)等同于一个 KECCAK 模块消耗的时

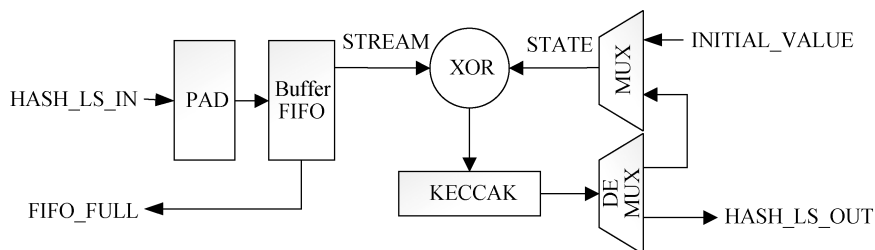


图 5 HASH_LS 模块
Figure 5 HASH_LS module

钟周期数, 即 27 个时钟周期。而对于 HASH_LS, 输入数据的长度为 $X(X > 1088)$ 时, 内部的 KECCAK 模块需要进行 $\lceil X/1088 \rceil$ 次计算, 故 HASH_LS 至少需要 $27 \times \lceil X/1088 \rceil$ 个时钟周期数。

3.1.3 HASH_CTR 和 AXI_HASH 模块

针对软硬件协同系统设计的哈希模块需要包含软硬件通信逻辑。在本设计中为 HASH_CTR 和 AXI_HASH 模块。HASH_CTR 包含 GP 接口的 AXI 协议状态机, 接收 PS 端的开始信号和 32bits 配置数据。AXI_HASH 包含 SHA3_256 核心运算模块, 外部与 FIFO 相连, 由 FIFO 中取数据进行 hash 运算。

3.2 软硬协同系统设计

面向低功耗应用场景, 使用软硬件协同方案可将原本在 ARM 核上执行的 hash 函数卸载至 FPGA 上, 以降低 ARM 核的负载, 为 ARM 核保留更多的运算资源执行其他功能。软硬协同系统的思路为核心哈希运算在 FPGA 上运行, 其余控制逻辑在软件端执行。本节主要介绍软硬协同系统的顶层设计与软硬件通信逻辑。

3.2.1 系统顶层

如图 6 所示为软硬件系统的顶层设计, 采用 Xilinx 公司 ZYNQ 架构的 FPGA 开发板, PS (Programming System) 端为基于板上的 ARM 平台的软件逻辑, PL (Programmable Logic) 端为基于 FPGA 的可编程逻辑的硬件系统。LMS 主体控制逻辑由软件在 ARM 处理器上实现, 核心运算的 HASH 模块在 FPGA 上运行, 两部分之间通过 AXI4 协议的 GP (Global Purpose) 接口与 HP (High Performance) 接口进行数据交换。ARM 核执行 HASH 模块的驱动函数, 控制信号由 GP 接口传至 PL 侧控制寄存器中, 数据由 HP 接口传至 PL 侧 FIFO 缓存。HASH_CTR 模块处理控制寄存器中的值, 判断是否启动 AXI_HASH 运算模块, 并配置哈希运算参数。运算结束后 AXI_HASH 模块将完成信号置“1”, 由 HASH_CTR 模块采集到该信号之后, 将 HASH 运算的输出结果与哈希完成的标识值由 GP 口传输至 PS 侧 DDR 中, ARM 端监测 DDR 中标识地址的值, 当监测到完成标识值时, 读取 DDR 中存放哈希结果地址中的值, 完成一次哈希运算。

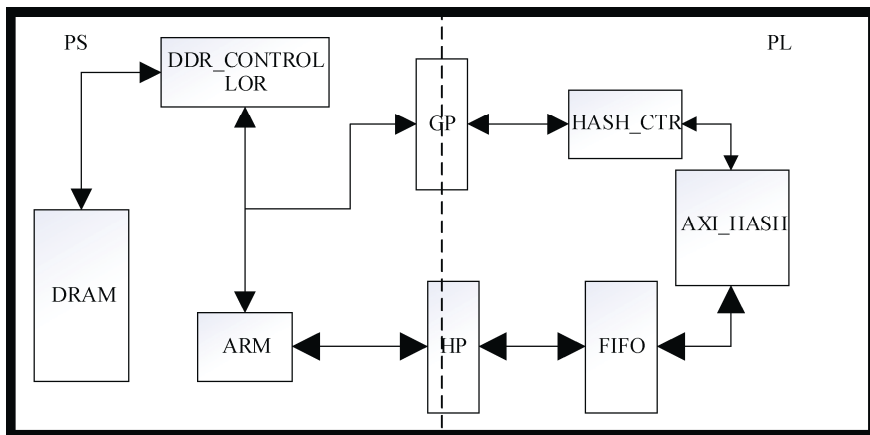


图 6 软硬协调系统顶层

Figure 6 Overall Design of Hardware-software Co-design

3.2.2 接口设计

为实现 PS 端与 PL 端之间的通信, 我们使用 DDR 中一段固定地址作为通信媒介。使用 Linux 系统函数 `mmap()`, 将 DDR 内存中一段物理地址映射到系统中的虚拟地址, 使得系统与 PL 侧逻辑都能对该段地址内存做读写操作, 从而使得 PS 端与 PL 端能够相互通信。

图 7 为软硬件协同的控制信号通信逻辑图。PS 端(C 语言实现)将 PL 端挂在总线上的 AXI 接口的物理地址通过 `mmap()` 函数映射到系统虚拟地址, 并将

地址分别分配给 `enable` 与 `valid` 两变量。`Enable` 和 `valid` 的初始值分别为 `True` 和 `False`。当 `valid` 为 `FALSE` 时, PS 端将进入循环等待 PL 端完成哈希运算, PL 端完成哈希运算后将 `valid` 置为 `True`。PS 端将跳出循环, 并重置 `enable` 和 `valid` 信号。

3.3 密钥生成全硬件实现

针对 LMS 中耗时最长的密钥生成过程, 本工作的硬件设计思路为首先考虑 OTS 密钥生成的架构设计, 再考虑 LMS 中的树构造, 即叶子节点与内部节点的计算。

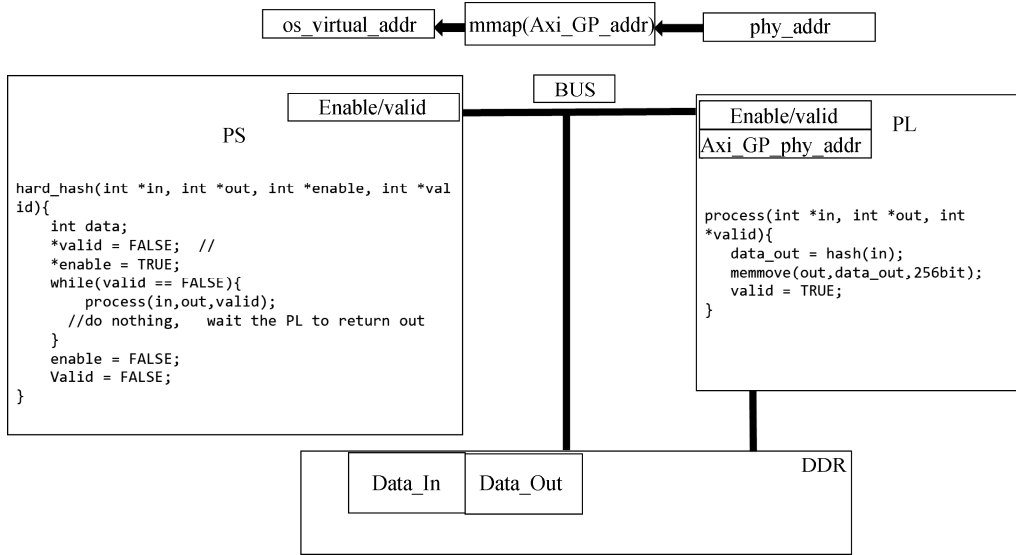


图 7 PS 端与 PL 端通信逻辑

Figure 7 Communication and control logic between PS and PL

3.3.1 OTS 生成 y 架构 (OTS_Gen_y)

由表 3 可知, 进行算法 2 与算法 3 中生成 y 序列的计算中, 哈希的输入内容发生变化但长度不变, 均为 55 字节即 440 比特, 故硬件上可将计算单元进行复用。因此, 将算法 2 与算法 3 中计算 y 序

列的部分合并设计, 并与算法 3 中最后进行输入不定长的哈希计算的部分拆分开。本小节首先介绍用于实现算法 2 生成 OTS 私钥与算法 3 中生成 y 序列的模块, 我们称之为 OTS_Gen_y 模块, 如图 8 所示。

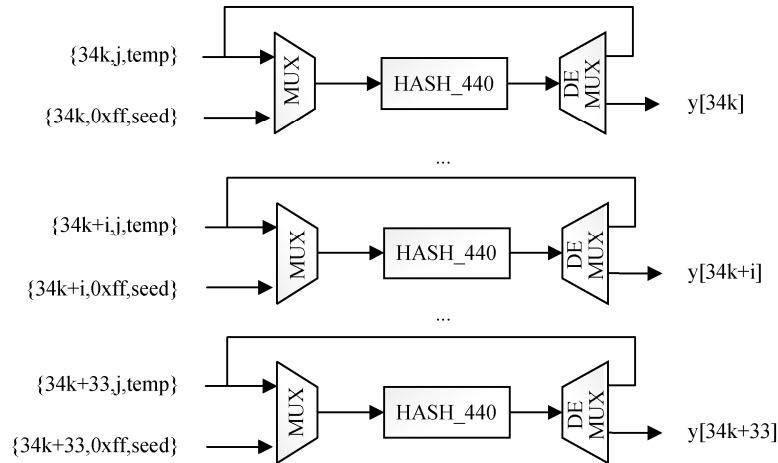


图 8 OTS_Gen_y 模块

Figure 8 OTS_Gen_y module

首先, 在计算 OTS 私钥与 y 序列的过程中, 哈希的输入长度均为 55 字节即 440 比特, 故选择 HASH_440 模块为核心处理单元; 其次, 计算最后不定长哈希的输入中的 y 序列时, 所需要的最小的迭代组数为 $p=34$, 且 p 的其余值均为小于 34 的倍数的临近值。故本工作在 HASH_440 的处理上选择了 34 并行。这样设计可以使得各并行模块中的数据实现完全同步处理, 同时实现最大的硬件利用率。

对于每个 HASH_440 模块, 首轮计算时, 将初始

随机值与各模块索引值(index)输入到 HASH_440 中, 此时 HASH_440 的输出结果为一个 OTS 私钥; 在随后的计算中, 将索引值 index、迭代次数 j 与上一轮 HASH_440 的输出(temp)再次输入到 HASH_440 模块中。这样迭代 2^w-1 次后, 索引值为 index 的 HASH_440 模块生成的结果对应于算法 3 中的 y 序列的 $y[index]$ 。

对于 $p=34$ 的参数集, 上述操作执行 1 轮; 对于 $p>34$ 的参数集, 这 34 并行模块中的运算将被执行

$\lceil p/34 \rceil$ 轮。设这个执行轮数用变量 k 表示, 则在第 k 次执行时, 第 i 个 HASH_440 的模块的输入索引值为 $index=34k+i$, 如图 8 所示。

需要注意的是: ①以上述方式, 单个 HASH_440 的模块可以被复用到 OTS 私钥生成与公钥生成的两个阶段。且由于 LMS 中一次密钥生成过程生成的密钥可用于进行 2^h 次有效签名与验证, HASH_440 模块的复用将不会对协议的整体效率产生影响。②在 $p>34$ 的参数集中, 在最后一次迭代时, 部分 HASH_440 模块本来不需要进行计算, 但在本工作的设计中, 每一轮计算的每个哈希模块都被激活以进行运算。如 $p=133$ 时, 需要执行 $k=4$ 轮计算, 而在第 4 轮计算时, 34 个模块中的第 32、33、34 模块仍在进行计算, 但输入输出的结果均为无效值。这样设计的好处是, 哈希计算时尽量实现恒定功率, 以此可以一定程度上抵御功率攻击。

3.3.2 LMS 公钥生成架构

前一部分已经包含 LMS 私钥生成的过程, 这里介绍 LMS 公钥生成的模块, 也是本工作架构中的顶层模块。如图 9 所示, LMS 公钥生成模块将 LMS 公钥生成分为叶子节点生成(LMS_LEAF_GEN)和树生成(LMS_TREE_GEN)两部分。

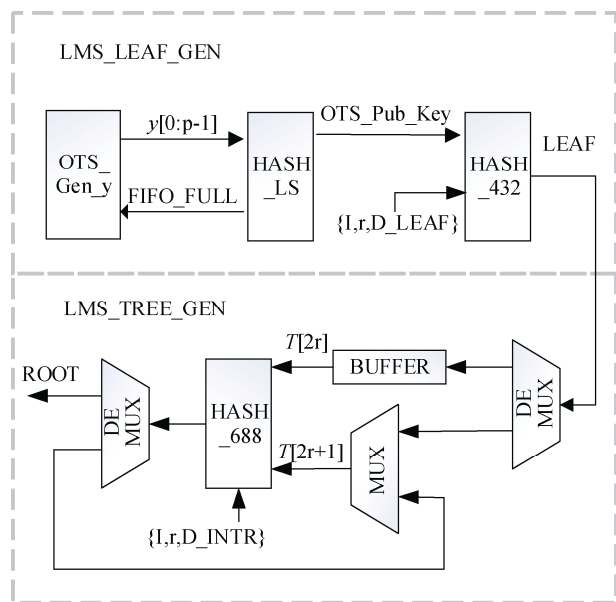


图 9 密钥生成顶层设计

Figure 9 Top design of key generation

叶子节点生成(LMS_LEAF_GEN): 首先, 当 OTS_Gen_y 产生 y 序列后, 序列中 p 个 256 比特的数据被填充、打包成单位为 1088 比特的数据流并输入到 HASH_LS 模块中, 用以计算 OTS 公钥值中的 K 。

当 HASH_LS 内部的 Buffer FIFO 满时, 向 OTS_Gen_y 模块输入 FIFO 满信号(FIFO_FULL), 用以暂停 OTS_Gen_y 模块的输出, 注意为尽可能保持功率稳定, OTS_Gen_y 内部的运算不会停止, 而是在 FIFO_FULL 无效之前, 进行数据不变的运算。同时, 一旦 OTS_Gen_y 开始输出有效数据, HASH_LS 模块开始进行运算, 而当所有的 y 都输入到 HASH_LS 模块, HASH_LS 模块计算产生有效的 K 值后, K 被用于组成 OTS 的公钥值(OTS_Pub_Key)。由表 3 可知, 该公钥值与叶子节点的索引值、D_LEAF 等参数组合成长度为 432 比特的数据, 输入到 HASH_432 模块用以计算叶子节点的值。

树生成(LMS_TREE_GEN): 考虑到叶子节点的生成是间歇的, 在构造内部节点的过程中采用深度优先的思路, 即对于当前输入的叶子节点, 不断计算内部节点知道不可再计算父节点为止。

首先使用一组缓存的寄存器(BUFFER), 此组寄存器的尺寸为 $h \times 256$ 比特。为方便叙述, 将寄存器的每个单位称为 BUFFER[i], 其中 BUFFER[i]都是 256 比特且 $0 < i < h$ 。BUFFER 用以存储暂时没有兄弟节点的叶子节点和内部节点, 并对存有有效数据的索引进行标记, 即当第 i 层存有有效数据时, 该层对应的标记 flag[i]=1。每当一个叶子节点值输入, 判断该节点的兄弟节点是否已经存在于寄存器中, 若不是, 则将此节点存在 BUFFER[$h-1$]中; 若是, 则从 BUFFER[$h-1$]中取出其兄弟节点的值使用 HASH_688 模块计算这两个叶子节点的父节点, 同时使原节点的标记 flag[$h-1$]无效。对计算出的父节点在 $h-2$ 层再次进行上述操作, 直到找不到对应的兄弟节点。当最后一个叶子节点输入, 只需再经过 h 个 HASH_688 的计算延时, 即可得到最终的根节点, 即完成 LMS 公钥的计算。在此设计中, 只需改变寄存器 BUFFER 的深度, 即可改变此架构所允许的 LMS 最大树深度。

由于本文的初步工作^[33]中没有对此架构支持的极限树高进行说明, 在此进行补充分析与说明。当树高 h 小于一定阈值时, 不会出现叶子节点数据输入到树生成模块中但未被计算到的情况。考虑到叶子节点的计算需要 2^w 个 HASH_440 模块延时(1 次计算 OTS 私钥, 2^w-1 次计算 OTS 公钥), 1 个 HASH_LS 模块延时和一个 HASH_432 延时; 而内部节点的计算最多进行 h 次 HASH_688 计算, 且进行 h 次计算时是在计算最终的根节点 $T[1]$, 即计算其他内部节点时最多需要 $h-1$ 次 HASH_688 模块的延时。由章节 3.1.2 可知, HASH_440、HASH_432、HASH_688 消耗的时

钟周期数均为 27, HASH_LS 消耗的时钟周期数为 $\lceil p \times 256 / 1088 \rceil \times 27$ 。时钟周期数分析见表 4。如果出现叶子节点的输入快于内部节点计算, 则有 $h-1 > 2^w + \lceil p \times 256 / 1088 \rceil + 1$ 即 $h > 2^w + \lceil p \times 256 / 1088 \rceil + 2$ 。结合表 1, 可能出现上述情况的 h 最小需满足有 $h > 34$ (此时 $p=67, w=4$)。但根据表 2, 目前 h 的推荐最大值为 25。故在推荐参数集中, 本设计中的树生成模块可以在不遗漏数据的基础上完成最快计算。

表 4 叶子节点与内部节点计算的时钟数
Table 4 Computing clock cycles of leaf nodes and internal nodes

叶子节点计算时钟数	内部节点计算时钟数(最大)
$2^w \times 27 + \lceil p \times 256 / 1088 \rceil \times 27 + 27$	$(h-1) \times 27$

3.4 安全性分析

本文工作中将原 LMS 方案的核心计算函数即 SHA2 函数替换成了 SHA3 函数, 对于这样的替换, 需要对安全性进行分析。

3.4.1 SHA2 vs SHA3

从哈希的安全性角度, 在 NIST 相关标准文件^[12]中提到, SHA3 具有 SHA2 函数的所有安全性能, 包括 128 比特安全强度的抗碰撞性(Collision resistance)和 256 比特安全强度的抗原像攻击性(Preimage resistance)。而在抗 2 级原像攻击性(2^{nd} preimage resistance)方面, SHA3 函数的安全强度为 256 比特, SHA2 函数的安全强度为 $256-L(M)$ 比特, 其中 M 是输入数据比特数, $L(M)$ 是一个关于 M 的非负正相关函数。也就是说, 输入数据越长, SHA3 的此项安全性性能越优于 SHA2 函数, $L(M)$ 具体计算公式见(12)。

3.4.2 LMS 安全性

对于包括 LMS 在内的 HBS 方案, 协议的安全性完全取决于其中采用的哈希函数的安全性^[10,16-19]。理论上来说, OTS 中采用的哈希函数和 LMS 树的内部节点采用的哈希函数可以不同, 但为了防止出现更弱哈希函数被攻击的情况, 更建议采用相同的哈希函数^[23], 如 LMS 原方案及现有的相关工作均采用 SHA2 函数。因此, LMS 方案的安全性完全取决于内部采用的单一哈希函数的性质。而在 IETF 公开的 LMS 文献中^[23]提到, 当前的参数设置可支持多种哈希函数如 SHA256(即 SHA2)和 SHA3 函数, 故在安全等级保证不变的前提下, 替换哈希函数并不

会产生多余的安全性损失。

4 性能评估

性能评估方面, 软硬协同系统硬件端和设计的硬件架构部分用 verilog 进行实现, 综合与实现过程采用了 Xilinx Vivado 2019.2 版本的 EDA 工具, ARM 和 FPGA 平台选择 Xilinx Zynq UltraScale+ MPSoC ZCU104 开发板。

4.1 软硬协同系统

对于软硬协同系统, 硬件端资源(表 5)消耗 5013 个查找 LUT(Loop up table, LUT)和 9838 个触发器(Flip flop, FF), 分别占到 FPGA 芯片总资源的 2.17%和 2.13%。需要说明的是, 在不同参数写的硬件端资源消耗是一致的(包括不定长哈希模块与相关接口配置)。

表 5 软硬件协同系统资源消耗
Table 5 Resource consumption of the Hardware-software Co-design

资源	数量	总共	占比(%)
LUTs	5013	230400	2.17
FFs	9838	460800	2.13

在 $w=8$ 且 $h=5$ 的参数下, 软硬件协同系统需要 9 s 左右完成密钥生成。其中主要的时间消耗在 FPGA 与 ARM 的通信过程。对于时间性能, 由于目前只在硬件端部署了一个哈希模块, 比起核心计算, 软硬件通信过程消耗了更多时钟数, 总体计算时间消耗较长; 且随着 w, h 增大, 软硬件协同系统生成密钥所需的计算时长将增加得更快。因此, 目前本文给出一个较小尺寸的参数下的时间性能, 由于这是针对 LMS 软硬协同系统的首次探索, 该性能可以为后续工作提供物联网应用场景下实现 LMS 的时间性能参考。

4.2 密钥生成架构

对于密钥生成的硬件架构, 需要说明的是, HASH_LS 模块中使用的 FIFO 尺寸为 64×1088 比特; 树生成模块中的 BUFFER 尺寸为 32×256 比特, 即允许的最大树高为 32。为了直观地体现本工作的优势, 将我们所设计的架构对表 1 所示的所有参数集与表 2 中的部分参数集进行了仿真与延时计算, 并在 Xilinx Zynq UltraScale+ ZCU104 的开发板上进行了实现。

4.2.1 哈希计算模块

首先, 在实现结果中, 在最大工作频率下, 几种不同尺寸的哈希模块的资源占用情况如表 6 所示。在查找表与触发器的资源占用上, HASH_X 模块与 HASH_LS 模块出现了明显的数量差异。同时, 由于 HASH_X 模块不需要进行数据缓存, 不需要额外占用块存储器(Block RAM, BRAM)。因此, 对哈希模块进行分开设计达到了节省硬件资源的结果。

表 6 哈希模块资源消耗

Table 6 Resource consumption of HASH modules

模块	资源		
	LUTs	FFs	BRAMs
HASH_440	4661	2300	0
HASH_432	2943	2296	0
HASH_688	2927	2551	0
HASH_LS	6502	6175	17

(注: 此处的 HASH_LS 模块中的 Buffer FIFO 尺寸为 64×1088 比特)

为了更直观地说明本工作中哈希模块的高效性, 对哈希进行最大时钟频率下吞吐率的计算, 并选取相关工作的结果进行对比, 如表 7 所示。由表可见, 本工作中哈希模块的吞吐率与相关工作可达到相当的吞吐率。且由于本工作中将根据 LMS 方案内部计算的特性, 对哈希模块进行相匹配的设计与部署, 可增加整个方案的资源利用率。

4.2.2 密钥生成架构

对于整个密钥生成架构, 在最大时钟频率达到 285MHz 的基础上, 资源消耗情况如表 8 所示。由表可知, 本文中提出的硬件架构在硬件资源上, 分别

消耗了 82.75% 的查找表、27.49% 的触发器和 5.45% 的块存储器资源。

表 7 哈希模块吞吐率对比

Table 7 Comparisons of throughputs of HASH modules

工作 ^①	FPGA 型号	BRAM 存储	吞吐率(Mbps)
[35]	Virtex-6	否	8830
[36]	Virtex-7	否	13963
		是	14876
本工作	UltraScale+	否 ^②	14103
		是	13297

(注: ① 此处选取的工作都是基于核心为 KECCAK-1600 的 SHA3 函数。② 本工作的是否使用 BRAM 存储分别对应 HASH_LS 模块和 HASH_X 模块。)

表 8 密钥生成资源消耗

Table 8 Resource consumption of the key generation design

资源	数量	总共	占比(%)
LUTs	190660	230400	82.75
FFs	126656	460800	27.49
BRAMs	17	312	5.45

目前还没有关于 LMS 协议密钥生成的 FPGA 硬件实现。所以, 本文选取已有的基于其他平台的工作来进行对比, 结果如表 9 所示。需要注意的是, 由于文献[29]和文献[30]的 CPU 实现结果已经优于文献[23]中结果, 在此不再对比文献[23]中性能。由表可知, 本工作中的密钥生成架构相较于目前最新的工作实现了超过一个量级的加速比。

表 9 与以前工作的密钥生成延时对比

Table 9 Latency comparison of key generation with previous works

树高 h	折中 w	延时(ms)			加速比
		ARM ^{①②③} /[31-32]	CPU ^{④⑤} /[29-30]	FPGA ^⑥ /本工作	
5	8	5602	-	0.748	7489.3×
	4	702	-	0.097	7237.1×
10	8	179268	-	23.90	7500.7×
	4	22470	-	3.059	7345.5×
15	8	/	13720	764.8	17.94×
	4	/	2519	97.83	25.74×

(注: ① ARM 平台为 STM32F4DISCOVERY 开发板, 其中具有 32 位 ARM Cortex-M4 的 FPU 内核, 1-Mbyte 闪存 ROM 和 192-Kbyte RAM。② 此处 ARM 平台的计算延时根据原参考文献中时钟数与 Cortex-M4 的最大工作频率 168MHz 计算得出。③ 基于 SHA2 函数。④ CPU 型号为 Intel Xeon CPU @ 2.20GHz, 具有双核和 7.68GB RAM。⑤ 基于 SHA2 函数。⑥ 基于 SHA3 函数。)

由于以前的工作可对比的参数不足且没有核心计算替换为 SHA3 的 LMS 实现, 我们还进行了软件评估, 并将本工作的 FPGA 实现与软件实现进行对

比。对于软件实现, 将标准参考文档^[23]提供的开源代码^[37]中的 SHA2_256 哈希函数换成 SHA3_256 函数, 并在 Intel(R) Core(TM) i7-6850K 3.60GHz CPU 进行

了允许多线程的实现。本工作与该 CPU 的延时对比如表 10 所示。

表 10 CPU 与 FPGA 实现的延时对比
Table 10 Latency comparison of implementation on CPU and FPGA

树高 h	折中 w	CPU 延时(ms)	FPGA 延时(ms)	加速比
5	8	1450	0.748	1938×
	4	180	0.097	1855×
	2	100	0.103	971×
	1	110	0.200	550×
10	8	46320	23.90	1938×
	4	591	3.059	193×
	2	325	3.229	101×
	1	366	6.689	55×
15	8	1599110	764.8	2091×
	4	19575	97.83	200×
	2	10422	103.2	101×
	1	11889	202.8	59×

可以看出, 本工作的 FPGA 实现相比于目前较优的 CPU 实现, 在不同的参数下, 具有 55~2091 倍

的速度提升。当 $h=5$ 且 $w<8$ 的时候, 速度提升浮动较大, 经过分析, 我们认为原因是, 软件实现中虽然计算量减小但 CPU 中一些必要的控制没有随之减少, 故 FPGA 实现的速度提升倍数尤其大; 而当 h 或 w 更大的时候, 相同的 w 对应的速度提升倍数趋于稳定, 更具有参考意义。

4.2.3 与 XMSS 的比较和讨论

由于 LMS 和 XMSS 方案的相似性, 在 LMS 现有可比较工作不足的情况下, 与相近的 XMSS 方案的相关工作比较也具有一定的参考意义。对于 XMSS 方案, 目前已经出现较多的实现与优化工作^[18,26-27,31-32,38-39], 但相似地, 这些工作较多集中于签名和验证过程。我们整理了目前文献中最新的 XMSS 密钥生成过程的性能数据^[27,31-32,38], 在相同安全等级、相似参数配置的基础上与本文架构相比较, 结果如表 11 所示。可以看出, 在现有参数对比下, 相较于 XMSS 的软件实现, 本工作中架构实现了超过两个量级的加速比; 相较于 XMSS 的 FPGA 实现, 本工作中架构实现了约 70 倍的加速比。从这项对比中, 进一步说明本工作所实现的硬件加速器的高效性。

表 11 与 XMSS 的密钥生成延时对比
Table 11 Latency comparison of key generation with XMSS

工作	方案	平台	哈希类型	树高 h	折中 w	延时
[31]	XMSS	ARM ^①	SHA2	5	4	1.45 s
[32]				10	4	22.5 s
[38]	XMSS	FPGA ^②	SHA2	10	4	1.68 s
[27]	XMSS	CPU ^③	SHA2	20	8	25 h
本 工 作	LMS	FPGA	SHA3	5	4	0.748 ms
				10	4	23.90 ms
				20	8	~2.45 s ^④

(注: ① ARM 配置与延时计算同表 9, 数据来自原文中与 LMS 最相近的 XMSS_SIMPLE 方案。② FPGA 型号为 Xilinx Artix-7 系列。③ CPU 型号为 AMD Geode LX800, 具有 255MB 的 SDRAM。④ 此数据根据表 10 中 $h=15$ 的测试数据预估得出。)

5 总结

本工作首先在不损失安全性的前提下, 对 LMS 中核心的 SHA2 函数替换成 SHA3 函数, 为 LMS 的灵活配置方面进行了探索; 接着, 在本文中首次提出了整个协议的软硬协同系统; 该软硬协同系统在占用较少资源的基础上, 可在整个 LMS 过程进行使用并通过验证, 为后续 LMS 进入物联网场景下的实际应用提供了参考。同时, 针对最耗时的 LMS 密钥生成的过程, 提出了一种快速、可扩展的基于 FPGA 的硬件加速器。在该加速器设计中, 许多优化方案被用于增加并行性和可伸缩性, 并减少延迟。架构被设

计为在允许不同的参数集的情况下, 仍然保持恒定功率。该结构在 FPGA 上的实现结果表明, 新的硬件实现在速度性能上大大优于传统的 CPU 实现。

未来, 我们将关注 LMS 软硬件协同系统中密钥生成模块全卸载至 FPGA 上的加速方案, 以及 LMS 中签名与验证的硬件加速等。

致 谢 感谢国家自然科学基金资助项目(No.61774082)、中央高校基本科研业务费专项资金资助项目(No.021014380065)、江苏省重点科研计划资助项目(No.BE2019003-4)对本文的资助。

参考文献

- [1] Rivest R L, Shamir A, Adleman L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems[J]. *Communications of the ACM*, 1978, 21(2): 120-126.
- [2] ElGamal T. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms[J]. *IEEE Transactions on Information Theory*, 1985, 31(4): 469-472.
- [3] Simplicio Jr M A, Cominetti E L, Patil H K, et al. Privacy-preserving linkage/revocation of VANET certificates without LAs[J]. *IACR Cryptol. ePrint Arch.*, 2018, 2018: 788.
- [4] Bernstein D J, Duif N, Lange T, et al. High-Speed High-Security Signatures[J]. *Journal of Cryptographic Engineering*, 2012, 2(2): 77-89.
- [5] Shor P W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer[J]. *SIAM Review*, 1999, 41(2): 303-332.
- [6] Announcing Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms[J]. *The Federal Register / FIND*, 2016, 81(244).
- [7] Alagic G, Alperin-Sheriff J, Apon D, et al. Status Report on the First round of the NIST Post-Quantum Cryptography Standardization Process[R]. National Institute of Standards and Technology, 2019.
- [8] NIST. Workshop on Cybersecurity in a Post-Quantum World[DB]. National Institute of Standards and Technology, Gaithersburg, Maryland, April 2-3, 2015. <https://csrc.nist.gov/Events/2015/Workshop-on-Cybersecurity-in-a-Post-Quantum-World>.
- [9] Alagic G, Alperin-Sheriff J, Apon D, et al. Status Report on the First round of the NIST Post-Quantum Cryptography Standardization Process[R]. National Institute of Standards and Technology, 2019.
- [10] Wang W, Stöttinger M. Post-Quantum Secure Architectures for Automotive Hardware Secure Modules[J]. *IACR Cryptol. ePrint Arch.*, 2020, 2020: 26.
- [11] Function N H. Description of SHA-256, SHA-384 AND SHA-512[J]. *ACM Trans. Program. Lang. Syst*, 2004, 9: 9.
- [12] Dworkin M J. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions[R]. National Institute of Standards and Technology, 2015.
- [13] Fouque P A, Hoffstein J, Kirchner P, et al. Falcon: Fast-fourier Lattice-based Compact Signatures over NTRU[J]. *Submission to the NIST's Post-quantum Cryptography Standardization Process*, 2018, 36.
- [14] Azarderakhsh R, Campagna M, Costello C, et al. Supersingular Isogeny Key Encapsulation[J]. *Submission to the NIST Post-Quantum Standardization project*, 2017.
- [15] Overbeck R, Sendrier N. Code-Based Cryptography[M]. Post-Quantum Cryptography. Berlin, Heidelberg: Springer Berlin Heidelberg, 95-145.
- [16] Cooper D A, Apon D C, Dang Q H, et al. Recommendation for Stateful Hash-Based Signature Schemes[J]. *NIST Special Publication*, 2020, 800: 208.
- [17] Shafieinejad M, Esfahani N N. A Scalable Post-Quantum Hash-Based Group Signature[J]. *Designs, Codes and Cryptography*, 2021, 89(5): 1061-1090.
- [18] Regnath E, Steinhorst S. AMSA: Adaptive Merkle Signature Architecture[C]. *2020 Design, Automation & Test in Europe Conference & Exhibition*, 2020: 1532-1537.
- [19] Sikeridis D, Kampanakis P, Devetsikiotis M. Post-Quantum Authentication In TLS 1.3: A Performance Study[C]. *2020 Network and Distributed System Security Symposium*, 2020: 71.
- [20] Kampanakis P, Sikeridis D. Two Post-quantum Signature Use-cases: Non-issues, Challenges and Potential Solutions[C]. *The 7th ETSI/IQC Quantum Safe Cryptography Workshop*, 2019: 3.
- [21] Cho J Y, Sergeev A. Post-quantum MACsec Key Agreement for Ethernet Networks[C]. *The 15th International Conference on Availability, Reliability and Security*, 2020: 1-6.
- [22] Merkle R C. Secrecy, Authentication, and Public Key Systems[M]. Stanford university, 1979.
- [23] McGrew D, Curcio M, Fluhrer S. RFC 8554—Leighton-Micali Hash-based Signatures[R]. IETF, Tech. Rep., apr 2019. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc8554>, 2019.
- [24] Huelsing A, Butin D, Gazdag S, et al. XMSS: EXTended Merkle Signature Scheme[R]. RFC Editor, 2018.
- [25] IETF. rfcs. Website, <https://www.ietf.org/standards/rfcs/>.
- [26] Suhail S, Hussain R, Khan A, et al. On the Role of Hash-Based Signatures In Quantum-Safe Internet of Things: Current Solutions and Future Directions[J]. *IEEE Internet of Things Journal*, 2021, 8(1): 1-17.
- [27] Gazdag, S.L., Friedl, M., Loebenberger, D. Post-Quantum Software Updates[J]. *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik – Informatik für Gesellschaft*, 2019: 459-472.
- [28] Banegas G, Zandberg K, Herrmann A, et al. Quantum-Resistant Security for Software Updates on Low-Power Networked Embedded Devices[EB/OL]. 2021
- [29] Kampanakis P, Panburana P, Curcio M, et al. Post-Quantum Hash-Based Signatures for Secure Boot[J]. *IACR Cryptol. ePrint Arch.*, 2020, 2020: 1584.
- [30] Kampanakis P, Panburana P, Curcio M, et al. Post-Quantum LMS and SPHINCS+ Hash-Based Signatures for UEFI Secure Boot[J]. *IACR Cryptol. ePrint Arch.*, 2021, 2021: 41.
- [31] Kampanakis P, Fluhrer S. LMS vs XMSS: Comparison of two Hash-Based Signature Standards[J]. *IACR Cryptology ePrint Archive: Report 2017/349*, 2017.
- [32] Campos F, Kohlstadt T, Reith S, et al. LMS vs XMSS: Comparison of Stateful Hash-Based Signature Schemes on ARM Cortex-M4[C]. *International Conference on Cryptology in Africa*, 2020: 258-277.
- [33] Song Y F, Hu X, Wang W H, et al. High-Speed and Scalable FPGA Implementation of the Key Generation for the Leighton-Micali Signature Protocol[C]. *2021 IEEE International Symposium on Circuits and Systems*, 2021: 1-5.
- [34] Winternitz R S. A Secure One-Way Hash Function Built from DES[C]. *1984 IEEE Symposium on Security and Privacy*, 1984: 88.
- [35] Neue T, Rao M, Toal D, et al. Efficient and High Speed Fpga Bump in the Wire Implementation for Data Integrity and Confidentiality Services in the IoT[M]. *Sensors for Everyday Life. Springer, Cham*, 2017: 259-285.

- [36] Kundi D E S, Khalid A, Aziz A, et al. Resource-Shared Crypto-Coprocessor of AES Enc/Dec with SHA-3[J]. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020, 67(12): 4869-4882.
- [37] Martin C. hash-sigs, Website, <https://github.com/cisco/hash-sigs>.

- [38] Thoma J P, Güneysu T. A Configurable Hardware Implementation of XMSS[J]. *IACR Cryptol. ePrint Arch.*, 2021, 2021: 352.
- [39] Blaauwendraad B, Erkin Z, Schwabe P, et al. Postquantum Hash-based Signatures for Multi-chain Blockchain Technologies[D]. *Master Thesis*, 2019.



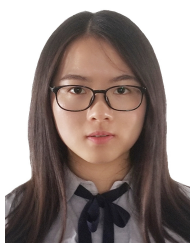
胡潇 于 2019 年在西安交通大学信息工程专业获得学士学位。现在南京大学信息与通信工程专业攻读博士学位。研究领域为密码学中的硬件加速设计。研究兴趣包括: 格密码、同态加密。Email: huxiao@smail.nju.edu.cn



宋逸峰 于 2018 年在南京大学物理学专业获得学士学位。现在南京大学集成电路工程专业攻读硕士学位。研究领域为密码学与区块链系统中的硬件加速。研究兴趣包括: 区块链、密码学。Email: 141120091@smail.nju.edu.cn



汪文浩 于 2020 年在南京大学空间科学技术专业获得学士学位。现在南京大学信息与通信工程专业攻读博士学位。研究领域为机器人与深度强化学习。研究兴趣包括: 计算机视觉和密码学。Email: 161210024@smail.nju.edu.cn



田静 于 2020 年在南京大学信息与通信专业获得博士学位。现任南京大学电子与信息工程学院副研究员。研究领域为用于数字信号处理和密码工程的超大规模集成电路设计。研究兴趣包括: 集成电路设计、区块链、后量子加密、同态加密。Email: tianjing@nju.edu.cn