支持国密算法的 JavaScript 通用密码库的实现

魏荣[1),2),3)]　郑昉昱[1),2)]　林璟锵[1),2),3)]

[1)](中国科学院信息工程研究所信息安全国家重点实验室，北京 100093)

[2)](中国科学院数据与通信保护研究教育中心，北京 100093)

[3)](中国科学院大学网络空间安全学院，北京 100049)

摘　要　随着近年来 Web 应用的大量普及及其安全问题的频发，用 JavaScript 进行一些密码运算的需求也随之而来。相比传统外插硬件外加驱动的密码计算模式，用 JavaScript 实现密码算法具有跨平台、免安装、兼容性好的优点。我们基于一款用 JavaScript 编写的国际密码算法库，加入了国密 SM2、SM3 和 SM4 算法，并使用固定基的 *comb* 方法对椭圆曲线固定点的标量乘进行了优化，使密钥生成和签名速度提升了一倍以上。在保证运算速度的同时，我们也尽量保持了代码量的最小化，以减小流量消耗和下载时长。我们在 Chrome、Firefox、Opera 和 Maxthon 浏览器中进行了验证和性能评估，在 Firefox 上，SM2 签名算法性能达到了每秒生成 100 对密钥，签名 95 次，验签 40 次，SM3 算法速度达到了 69.75Mbps，SM4 算法速度达到了 110.97Mbps。

关键词　Web 应用；JavaScript；国密算法；标量乘；密码库

中图法分类　　DOI 号：

The Implementation of a General-Purpose Cryptography Library Supporting Domestic Algorithm with JavaScript

Wei Rong[1),2),3)]　Zheng Fangyu[1),2)]　Lin Jingqiang[1),2),3)]

[1)](State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences), Beijing 100093)

[2)](Data Assurance and Communications Security Research Center, Chinese Academy of Sciences, Beijing 100093)

[3)](School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049)

Abstract　With the popularity of Web applications and their frequent occurrence of security problems in recent years, there comes the need for finishing some cryptographic computations with JavaScript. Compared with the traditional pluggable-hardware & driver mode for cryptographic computing, implementing cryptographic algorithm with JavaScript has advantages of cross-platform, installation-free and excellent compatibility. We implemented domestic SM2, SM3 and SM4 algorithms based on a JavaScript-programed cryptography library with international algorithms. In addition, we took advantage of the fixed-base comb method to optimize the fixed-point scalar-multiplication of elliptic curve cryptographic algorithm, thus achieving more than doubled speed of key generation and signature generation than before. Without losing the library's runtime performance, we tried to minimize the code size, so as to reduce the network traffic consumption and the transmission latency. Experiments were carried out in Chrome, Firefox, Opera and Maxthon browsers. In Firefox, SM2 signature algorithm generated 100 pairs of keys, and finished 95 operations of signature generation and 40 operations of signature verification per second. The speed of SM3 and SM4 algorithm reached 69.75 Mbps and 110.97 Mbps, respectively.

———————————————

收稿日期：年-月-日；最终修改稿收到日期：年-月-日 *投稿时不填写此项*. 本论文工作得到自然科学基金“通用计算平台的密钥保护技术研究”（项目编号61772518）和国家重点研发计划网络空间安全重点专项“基于国产密码算法的移动互联网密码服务支撑基础设施关键技术”（项目编号2017YFB0802100）资助. 魏荣，男，1992年生，博士在读，主要研究领域为应用密码学，E-mail: weirong1130@iie.ac.cn. 郑昉昱（**通讯作者**），男，1988年生，博士，助理研究员，主要研究领域为应用密码学、高性能计算、计算机算术，E-mail: zhengfangyu@iie.ac.cn. 林璟锵，男，1978年生，博士，研究员，主要研究领域为数据安全与隐私、应用密码学、密码技术在计算机网络系统的应用，E-mail: linjingqiang@iie.ac.cn.

Key words Web application; JavaScript; domestic cryptographic algorithm; scalar-multiplication; cryptography library

1 引言

Web 应用是指以 B/S（浏览器/服务器，Browser/Server）模式提供服务的网络程序，相对 C/S（客户端/服务器，Client/Server）架构的传统桌面应用而言，其具有跨平台、兼容性好、维护成本低、免安装的优势，因而在近年来迅猛发展，甚至曾一度出现了 Web 应用取代桌面应用的呼声。

当然，在未来相当长的一段时期内，Web 应用都难以完全淘汰桌面应用，因为它受到了本地资源访问限制、网络设施建设等因素的影响，除了在功能丰富度方面无法与桌面程序比拟外，其响应速度也极度依赖网络性能，常常出现内容加载缓慢的问题，而最致命的还是 Web 技术在实现密码运算时天然存在的短板：

(1) Web 应用的鉴权过程往往需要将用户口令发送到服务端进行比对，尽管客户端会使用加盐哈希来予以一定保护，但仍不能彻底解决窃听、重放等问题；

(2) 虽然一些口令身份鉴别机制可以让 Web 应用不用发送口令信息就能完成密钥协商，例如 EKE（Encrypted Key Exchange）[30]、J-PAKE（Password-Authenticated Key Exchange by Juggling）[14]等，但需要客户端能够完成较为复杂的密码运算——如果以浏览器密码插件的形式实现，则需要插件程序兼容各个系统平台和浏览器内核，需要很高的维护成本；而如果让 JavaScript 来完成这部分工作，则要求浏览器内核对 JavaScript 的解释速度足够快，以免影响用户体验；

(3) 虽然浏览器都内置了 HTTPS（超文本传输安全协议，Hyper Text Transfer Protocol over Secure Socket Layer）功能，但该功能只提供统一的认证、加密和完整性保护服务，开发者无法利用其内置的密码算法来实现自定义的安全协议。

不难看出，让 JavaScript 承担密码运算工作是 Web 应用发展的大势所趋，由于终端设备和浏览器多种多样，性能良莠不齐，难免有人担心作为脚本语言的 JavaScript 无法胜任复杂运算工作（如椭圆曲线标量乘）。不过，随着近年来计算机硬件性能的大幅度提升和浏览器内核的更新换代，JavaScript 的执行速度已经有了很大改观，可以胜任复杂密码运算的工作，尤其是移动互联网时代的到来和 HTML5[31]（超文本标记语言 5，HyperText Markup Language 5）技术的发展，使得 Web 应用在智能移动终端中也大放异彩。让前端脚本承担更多力所能及的计算工作已然时机成熟。

而在我国，由于计算机硬件、操作系统和软件设施建设长期落后国际水平，导致缺乏基本的国产化软硬件平台，国产密码推进工作迟滞。虽然 SM2[18]-[22]、SM3[11]和 SM9[24]-[28]算法已经上升为国际标准，但要得到主流浏览器内核的支持，还有相当长的路要走。目前也只有 360[1]、密信[2]、赢达信[3]等个别企业推出了支持国密算法的浏览器产品，其普及度还很低，而国内为数众多的 Web 应用都还在使用国外密码算法来保障数据安全，其中不乏 MD5、SHA-1、DES、RSA-1024 等已经被建议淘汰或明令禁止的不安全算法。

值得高兴的是，使用 JavaScript 实现密码算法不必拘泥于软硬件平台的限制，这意味着我们可以方便地使用国产密码算法，不用考虑浏览器是否支持的问题。

2 研究问题与现状

我们旨在用 JavaScript 实现一款适用于 Web 应用的通用密码库，可以提供国密 SM2、SM3 和 SM4[17]算法，并保证该库压缩后的大小控制在 50KB 以内。

在 Web 应用中，JavaScript 的性能体现在加载速度和执行速度两方面，而这两者往往处于对立关系。例如，AES（高级加密标准，Advanced Encryption Standard）算法在 OpenSSL 中的实现采用了 T-Table[10]和循环展开的技巧，前者将状态矩阵每一列的列混淆和字节代替操作融合在一张 1KB 的查找表中，只需一次访存就可以完成上述步骤，还可以省略行移位步骤，但该方法需要加密和解密函数各自维护 4KB 大小的查找表，这在本地应用中并不算很大的存储负载，不过在 Web 应用中，从服务端下载 8KB 的查找表会明显增加网络延迟；循环展开方式也存在类似问题，以 AES-128 为例，如果将循

环展开，则会导致主函数代码量增加近十倍，同样会显著增加网络延迟。因此，在实现算法时，应权衡代码量和执行速度之间的利害取舍，对可以通过少许代码预计算出来的常量，尽量放在本地生成，以减小流量消耗，但也要注意，预计算的时间开销不宜过长。

当然，针对 JavaScript 难以进行高性能计算的问题，最理想的解决方案还是尽可能利用设备本地的资源和计算能力，这样不仅可以避免重复下载密码库，还可以省去脚本解释执行的漫长过程，在 Web 技术的发展过程中，也确实产生了很多相关的支撑技术，例如现代浏览器普遍支持的 JIT（即时，Just In Time）编译技术[33]大大弥补了 JavaScript 作为脚本语言的天然速度劣势；此外，还出现了 WebGL[38]、WebAssembly[37]、和 asm.js[4]等更深程度的优化执行技术，它们力求让 JavaScript 的运行速度能够媲美 Native 代码，然而，除了浏览器支持程度不理想外，其较大的编译输出会严重增加下载时间；随着 HTML5 技术的发展，一些主流浏览器也对 Web Worker[32]机制进行了支持，从而改变了长期以来 JavaScript 只能单线程执行的状况，让 Web 应用也开始步入并行计算的时代。

上述各种技术的演进都是以提升用户体验为导向的，其优先考虑的是页面渲染和响应速度，而非数据安全问题，密码库的开发者可以对其加以利用，以提高密码计算效率，但敏感数据的安全暂时还要依赖浏览器的隔离机制。

考虑到浏览器中的数据安全问题，W3C（万维网联盟，World Wide Web Consortium）于 2017 年正式提出了关于 Web Cryptography API[36]的建议，期望能够以 JavaScript 接口形式为 Web 应用提供基础的密码服务，各主流浏览器也开始加入对上述接口的实现。由于处于更底层位置，这类密码服务的性能远比第三方 JavaScript 密码库更接近 Native 代码[7]，但截至目前，不同浏览器所实现的算法集合或多或少都存在差别，只有极个别浏览器（如 Samsung Internet[44]）完整实现了所有 API[35]。因此，在未来很长一段时期内，Web 开发者依然需要集成自己的 JavaScript 密码库以备不时之需。

可见，即便是应用广泛的国际算法，也未能如 HTTPS 一般成为浏览器的标配功能，而在国外厂商占据主流浏览器市场的今天，国产密码算法要得到浏览器的集成则更是任重道远。2017 年，中科院 DCS 中心研制了通过 Windows CNG[8]接口提供服务的国产商用密码算法库，并基于该库研制了支持商密算法的 Edge 浏览器和 IE 浏览器[41]，上述两款浏览器利用了操作系统的密码服务来为 Web 程序提供更安全的密码功能，这也是未来 Web 密码服务形态的一大发展方向——Web 应用中的密码服务最终是要沉降到更为快速和安全的浏览器内核甚至操作系统中来实现的。但与前文所述的技术类似，上述产品也仅仅提供了基于 Windows 10 系统上两种浏览器的密码服务，并不具备普适性，鉴于我国在操作系统和浏览器领域长期落后的现状，JavaScript 库仍将是国产密码算法入驻浏览器的主要形态。

3 实现方案

本节主要阐述 JavaScript 库中对国密算法的实现和优化方案，并不介绍原算法过程，读者如果想了解算法原理，可以查阅相关国密标准[17]-[23]。

3.1 实现思路

目前，已经有了很多较为著名的 JavaScript 国际密码算法库，例如 clipperz[6]、OpenPGP.js[29]、sjcl[15]、jwcrypto[40]、cryptico[42]、jscrypto[34]和 cryptojs[9]等。综合考虑文件大小、代码架构、密码算法集合和优化程度，我们决定基于 sjcl 库完成对国产密码算法的集成和优化。

sjcl 是由斯坦福大学 Stark 等于 2009 年推出的一款适用于浏览器和 Node JS 平台的 JavaScript 密码库，该库最初围绕 AES 算法进行优化实现，随着版本演进，目前已经能够支持国际上常用的对称、非对称密码算法、哈希算法、消息认证函数、KDF（密钥派生函数，Key Derivation Function）以及随机数发生函数，成为了一款比较全面的密码套件，该库还针对脚本加载速度进行了一定程度的优化，以提升用户体验。

相比上文中其他较流行的密码库，sjcl 重点关注密码原语的优化实现，而非更高层的通信或密码协议，因此非常精简，具有更小的体量和更好的兼容性，这不仅体现在平台兼容性上（它通过了 Mac、Linux 和 Windows 系统下所有主流浏览器的测试），还体现在对旧的 ES5[13]（ECMAScript 5）标准的完美兼容上。

更重要的是，sjcl 库的模块化代码结构对二次开发非常友好，各个模块之间的松散耦合便于开发者根据实际需要选择性地打包，随时去除不必要的

功能，这有利于缩减文件大小，同时也让该库具备了很好的可扩展性，新算法的集成工作比其他同类库更为方便。此外，sjcl 库一直致力于针对 Web 应用场景对密码算法进行特殊优化[15]，相比同类密码库，它在文件大小和运算速度之间达到了较好的平衡[7]。

我们基于 sjcl 密码库的框架，用 JavaScript 实现了 SM2 签名和加密算法、SM3 摘要算法和 SM4 对称加密算法，支持浏览器和 Node JS 平台，接口与 sjcl 库保持了风格一致，继承了其调用简单的优点；此外，针对最为耗时的 ECC（椭圆曲线密码学，Elliptic Curves Cryptography）算法，我们对其进行了一定程度的优化，将 ECC 密钥生成和签名速度提升了一倍。

3.2 对SM2算法的优化

由于 sjcl 库代码架构的原因，对 SM2 算法的优化实际上也可以惠及库中其他 ECC 算法。

对 ECC 椭圆曲线算法的优化通常集中在最为耗时的点乘运算上，sjcl 库用了常见的固定基窗口方式[15]来加速点乘，但没有对乘法标量进行长度扩充，使其与椭圆曲线基点的阶等长，容易在时序分析下暴露乘法标量（如私钥）长度信息，当然，这只是一个小问题，我们对其进行了修复。

考虑到内存占用和预计算的时间开销，sjcl 库所选的窗口宽度为 4。例如，对 256 位长的标量乘数 t 和点 P，通过 64 次“访存-16 倍点-点加”操作来完成$[t]P$ 运算，步骤如下：

(1) 将 t 表示为 16 进制：

$$t = [t_{63}|t_{62}|\dots|t_0]_{16}.$$

(2) 计算所有 $P_k = [k]P$，其中 $k = 0,1,\cdots,15$；

(3) 按公式(3.1)计算$[t]P$：

$$[t]P = \sum_{i=0}^{63} 16^i \times P_{t_i} \qquad (3.1)$$

还有一种固定基的 *comb* 方法[11]，以另外一种形式来分割乘法标量，通过 64 次“访存-2 倍点-点加”操作来计算$[t]P$，但是乘法标量的预处理和查找表的预计算也相对更繁琐一些。以模长 $l = 256$，窗口宽 $w = 4$ 为例，使用固定基 *comb* 方法计算$[t]P$ 的步骤如下：将 t 表示为二进制，并均分为 4 部分：

(1) $t = [t_{255}\dots t_{192}|t_{191}\dots t_{128}|t_{127}\dots t_{64}|t_{63}\dots t_0]_2$.

(2) 计算所有$[2^{192}s_3 + 2^{128}s_2 + 2^{64}s_1 + s_0]P$，记作 P_k，$k = s_3|s_2|s_1|s_0$，s_0、s_1、s_2 和 s_3 为 0 或 1；

(3) 按公式(3.2)计算$[t]P$：

$$[t]P = \sum_{i=0}^{63} 2^i \times P_{(t_{i+192}|t_{i+128}|t_{i+64}|t_i)} \qquad (3.2)$$

与窗口方法不同，*comb* 方法的预计算无法只通过点加和倍点完成，需要调用点乘方法，因而不能在点乘运行时进行。该方法更适合将预计算结果直接硬编码在程序中，因为与 sjcl 库节省代码量的设计原则不符，加之预计算比窗口方法耗时，最终没有被 sjcl 库采纳。

不过，对于固定点（如椭圆曲线基点）乘法来说，*comb* 方法只需在曲线初始化时进行一次预计算即可，我们为曲线基点编写了单独的点乘方法，使用 *comb* 方法省去了 75%的倍点运算，让基点的点乘运算速度提升了约 110%，从而加速了 ECC 密钥生成和签名过程。在 Maxthon 浏览器中，ECDSA（椭圆曲线数字签名算法，Elliptic Curve Digital Signature Algorithm）和 SM2 签名算法优化前后的性能数据如表 3-1 所示：

表 3-1 ECC 优化前后性能对比

算法	操作	优化前	优化后
ECDSA	生成密钥(对/s)	28	58
	签名(次/s)	27	54
SM2	生成密钥(对/s)	27	56
	签名(次/s)	26	55

另外，SM2 签名算法中，对消息的预处理需要公钥参与，而每次重复计算公钥会造成显著的时间开销，我们在生成和导入密钥对时，将公钥也保存在了私钥对象中，以节省一次不必要的标量乘。

在支持多线程的平台上实现上述两种优化方法时，如果将公式(3.1)和公式(3.2)分派给多个线程去运算，还可以获得成倍的加速效果。例如：在窗口方法中，如果将 $i\in[0,31]$和 $i\in[32,63]$部分的累加工作分派给两个子线程去完成，最后在主线程中合并结果，从理论上来讲，就可以获得近两倍的加速效果。

就固定基 *comb* 方法应该使用多少线程的问题，我们曾用 C 语言在 64 位 PC 平台（4 核处理器）和 ARM 平台（8 核处理器）上分别测试了开启 1、2、4、8 个子线程时的 SM2 点乘运算速度。实验表明，在开启 4 个子线程时，速度达到了最佳——可见，并不是线程数越多，并行度越高，效率就越高，如

果线程太多，而每个线程的计算量太小，则线程创建和销毁的时间在运行时间中的占比也会升高，变得不可忽略，反而导致性能下降。

不幸的是，由于浏览器中 UI（用户界面，User Interface）渲染线程和 JavaScript 引擎对 DOM（文档对象模型，Document Object Model）树的访问是互斥的，曾经有很长一段时间，为了保证线程安全，JavaScript 都不提供多线程机制。虽然 HTML5 提供了 Web Worker 机制，允许将不访问 DOM 的 JavaScript 代码放在后台线程中运行，但它要求将子线程需要用到的所有对象和函数结构化拷贝到新的上下文环境中，而一次这样的拷贝往往要耗时上百毫秒，其线程本身的开销远远高于 C 语言，可见 Web Worker 机制的主要用途在于确保执行脚本时不阻塞页面渲染，而非高性能并行计算。因此，我们最终决定基于单线程 JavaScript 来完成实现。

3.3 关于SM3和SM4的实现

JavaScript 作为一种弱类型的脚本语言，默认的 *number* 类型为 64 位双精度浮点数，相当于 C 语言中的 *double* 类型。但是在参与位运算时，会默认转化为 32 位整数，也就是说它并不支持 32 位以上的长整数，这不利于实现高性能的密码学大数运算，好在 SM3 和 SM4 算法中的位操作所针对的至多是 32 位长的字，不会受到影响。

sjcl 库的 AES 算法实现采用了 T-Table 优化，而前文也提到过，这种方法需要维护 8KB 大小的查找表，不宜直接使用硬编码方式，Stark 等人将 S 盒、逆向 S 盒以及扩展查找表都放在预计算部分中，在第一次调用 AES 算法时才会动态生成上述查找表，而只硬编码了轮常量。需要注意的是，Web 应用场景下，敌手更容易获得加密耗时，进而发起时序攻击，sjcl 库在使用 T-Table 方案时，在 AES 函数首轮和末轮放弃使用 T-Table 表，转而选择了查找 S 盒，这可以在一定程度上防止基于缓存的时序攻击。

SM4 算法则属于非对称 *Feistel* 结构，其轮函数的处理对象也迥异于 AES 算法的 4×4 状态矩阵，从理论上来看，对 SM4 使用 T-Table 方案所能产生的优化效果虽然远不如 AES，但在编码时，也理应能在每轮处理中节省 7 次位操作。

SM4 算法分组长度 16 字节，需要进行 32 轮变换，将第 i 轮状态记作 4 个 32 位字：(X_i, X_{i+1}, X_{i+2}, X_{i+3})，记第 i 轮轮密钥为 K_i，其中 i=1,2,…,32，则轮变换步骤如下：

(1) $A = (a_0, a_1, a_2, a_3) = X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus K_i$;

(2) $B = (b_0, b_1, b_2, b_3)$
$=(S[a_0], S[a_1], S[a_2], S[a_3])$;

(3) $C = B \oplus (B<<<2) \oplus (B<<<10) \oplus (B<<<18) \oplus (B<<<24)$;

(4) $X_{i+4} = X_i \oplus C$.

步骤(2)中 S 表示 S 盒。而在 T-Table 方案中，可以将步骤(2)和步骤(3)合并如下：

$C = T[a_0] \oplus (T[a_1]<<<24) \oplus (T[a_2]<<<16) \oplus (T[a_3]<<<8)$.

T 表示 T-Table 表，其构造方法如下：

(1) $y = S[x]$，x = 0,1,…,255;

(2) $T[x] = (y \oplus (y<<2), y \oplus (y>>6), y<<<2, y<<<2)$.

我们测试了 SM4 算法 T-Table 方案的优化效果，结果很出乎意料——在 JavaScript 实现中，采用 T-Table 的 SM4 加密速度仅为未优化版本的一半。经过实验与分析，我们发现查找 T-Table 表的耗时超过了查找 S 盒一倍，可见至少在测试浏览器的 JavaScript 解释器下，对 32 位字的访存所消耗的机器周期要多于对字节的访存，这一点与 Native 代码有着明显差异。

除了 T-Table 外，*bit slicing*[5]也是一种常用于对称加密算法的优化技巧，可以实现对大规模数据的并行处理。然而，AES 和 SM4 的 S 盒是基于有限域理论设计的，在 *bit slicing* 实现时会分解为大量位操作[43][48]，这意味着 JavaScript 代码量的急剧增加，因此并不适合在 Web 应用中采用。

基于以上考虑，我们最终采用了原始版 SM4，虽然加密轮数多于 AES，速度也比 T-Table 版 AES 要慢很多，但只需维护 256B 大小的 S 盒即可。由于预计算 S 盒的代码量超过了 256B，直接使用硬编码方式更为合理。

4 性能测试

由于缺乏同类通用密码套件进行对比，我们仅对扩充后的 sjcl 库进行了测试，并对国密算法和参数规模与之相当的对应国际算法进行了性能对比。

4.1 测试环境

我们通过脚本将源代码打包成了 47KB 的密码库文件，并在 64 位 Windows 平台上，使用了三种主流浏览器和一款国产浏览器进行性能测试，测试平台的具体型号、配置和版本如表 4-1 和表 4-2 所

示。

表 4-1 系统/硬件平台

操作系统	Windows 10 64 位
CPU	Intel(R) Core(TM) i7-6820HQ
内存	8GB

表 4-2 软件平台

浏览器	版本
Chrome	74.0.3724.8
Firefox	67.0b6
Opera	58.0.3135.118
Maxthon	5.2.6.1000

作为最具代表性的浏览器，上述几种平台均在不同程度上实现了 Web Crypto API，但目前都不支持国产密码算法。

4.2 测试结果

我们分别对比了 ECDSA 和 SM2 签名算法、SHA-256 和 SM3 哈希算法以及 AES-128 和 SM4 对称加密算法的性能，结果分别见表 4-3、表 4-4 和表 4-5。

由于我们对定点标量乘进行了优化，因此 ECC 算法中涉及基点乘法运算的密钥生成和签名过程要比验签快很多。

表 4-3 ECC 签名算法（单线程）性能

算法	操作	Chrome	Firefox	Opera	Maxthon
ECDSA	生成密钥(对/s)	47	109	47	55
	签名(次/s)	44	95	43	51
	验签(次/s)	17	43	17	19
SM2	生成密钥(对/s)	46	100	45	52
	签名(次/s)	45	95	44	52
	验签(次/s)	17	40	17	19

T-Table 方法旨在节省对称加密算法查找 S 盒之后的混淆和扩散步骤，与 S 盒类似，*Feistel* 结构的算法（如 DES、SM4）加解密过程共用一套 T-Table 查找表，而 AES 则不然，并且在使用该方法对 AES 算法进行优化时，由于解密过程中的逆向字节代替无法融入查找表，所以优化程度较加密过程要低，可以从表 4-5 中看出，AES 算法的加密速度要高于解密速度。

Maxthon 浏览器实际上使用了 Chrome 的 webkit[39]内核并作出了优化，其内置 JavaScript 引擎为 Chrome V8[47]，因此二者性能都很接近使用 Carakan[12]引擎的 Opera 浏览器。相对于 Chrome 而言，FireFox 所使用的 SpiderMonkey[46]引擎显然具有更高的计算性能。

表 4-4 哈希算法性能(Mbps)

算法	Chrome	Firefox	Opera	Maxthon
SHA-256	93.90	82.75	76.29	72.88
SM3	81.38	69.75	72.88	70.76

表 4-5 对称加密算法性能(Mbps)

算法	操作	Chrome	Firefox	Opera	Maxthon
AES-128	加密	87.19	203.45	93.90	135.63
	解密	81.38	174.39	71.81	87.19
SM4	加密	81.38	110.97	67.82	101.72
	解密	81.38	110.97	76.29	110.97

5 结论与展望

我们基于 sjcl 密码库，实现了 JavaScript 版国密 SM2、SM3 和 SM4 算法，形成了一款新的通用密码套件，可以为 Web 应用提供适度安全的前端通用密码服务。此外，我们使用固定基 *comb* 方法对椭圆曲线点乘进行了加速，将 ECC 密钥生成和签名的运算速度提升了一倍以上。

目前存在的问题和下一步工作：

(1) 虽然实现了 SM2 等非对称算法，但使用场景十分有限，不论是安全性还是功能都无法与驱动加硬件 Key 的模式相比，目前至多可以用于 J-PAKE 等协议以及加密、验签工作，在未来工作中，我们将考虑利用拆分或口令派生机制，予以私钥更高等级的保护；

(2) 目前所实现的 SM4 算法系基础版本，在部分平台上比 sjcl 的 AES-128 慢一倍左右，我们也尝试过利用 SIMD.js[45]或 *bit slicing* 来进行优化，但没有得出可行有效的方法，反倒是 AES-128 可以利用 SIMD.js 进行向量化实现，SM4 可能还存在软件优化空间，

希望下一步工作中能够找到有效方法来提升其性能；

(3) 在移动互联网应用中，HTML5+Native 的架构大行其道，它用 Web 代码来实现需要跨平台的核心功能，极大地简化了开发工作，还让 HTML5 可以方便地访问移动终端 Native 资源，如传感器、相机、通信录等，这在传统 PC 上是很难实现的，因此可以很大程度上解决 Web 应用功能受限的问题，而这也给了我们新的启发——至少在移动端上，能否利用对 Native 的访问来扩展非对称密码算法的应用场景，实现移动软件 Key 的效果？能否利用 Native 资源为移动 Web 应用构造高质量的随机数发生器？我们相信这些设想都值得在下一步工作中去尝试。

参 考 文 献

[1] 360 安全浏览器, https://browser.360.cn/se/ver/gmzb.html 2019,8,19

[2] 密信浏览器(MeSince Browser), 支持国密算法SSL证书https加密，显示不同的证书展示界面(V1/V2/V3/V4), https://www.mesince.com/zh-cn/browser 2019,8,19

[3] 速龙安全浏览器_国密安全浏览器 - 赢达信, https://www.win-trust.cn/products/21 2019,8,19

[4] asm.js, http://asmjs.org/ 2019,8,19

[5] Biham E . A fast new DES implementation in software[C]// International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 1997.

[6] Clipperz, http://www.clipperz.com/ 2019,8,13

[7] Comparing Performance of JavaScript Cryptography Libraries, https://medium.com/@encryb/comparing-performance-of-javascript-cryptography-libraries-42fb138116f3 2019,8,9

[8] Cryptography API: Next Generation - Windows applications | Microsoft Docs, https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal 2019,8,18

[9] CryptoJS, https://cryptojs.gitbook.io/docs/ 2019,8,13

[10] Daemen, J., Rijmen, V.: AES Proposal: Rijndael. NIST AES Algorithm Submission (September 1999), available online: http://csrc.nist.gov/archive/aes/ rijndael/Rijndael-ammended.pdf.

[11] Darrel Hankerson, Alfred Menezes, Scott Vanstone. Guide to Elliptic Curve Cryptography. New York: Springer-Verlag, 2004.

[12] Dev.Opera — Carakan, https://dev.opera.com/blog/carakan/ 2019,8,19

[13] ECMAScript – Wikipedia, https://en.wikipedia.org/wiki/ECMAScript#5th_Edition 2019,8,19

[14] EKE: Hao, F. and S. Shahandashti, "The SPEKE Protocol Revisited", Security Standardisation Research, pp. 26-38.

[15] E. Stark, M. Hamburg, and D. Boneh, Symmetric Cryptography in JavaScript: //Proceedings of the Annual Computer Security Applications Conference, Honolulu, Hawaii, America, 2009:7-11.

[16] E.F. Brickell, D.M. Gordon, K.S. McCurley, D.B. Wilson, Fast exponentiation with precomputation, in: Advances in Cryptology – Proceedings of CRYPTO'92, vol. 658, Springer-Verlag (1992), pp. 200-207.

[17] GM/T 0002-2012, SM4 block cipher algorithm [S] (in Chinese) (GM/T 0002-2012, SM4 分组密码算 [S]).

[18] GM/T 0003.1-2012, Public key cryptographic algorithm SM2 based on elliptic curves-Part 1:General [S] (in Chinese) (GM/T 0003.1-2012, SM2 椭圆曲线公钥密码算法第 1 部分：总则 [S]).

[19] GM/T 0003.2-2012, Public key cryptographic algorithm SM2 based on elliptic curves-Part 2:Digital signature algorithm [S] (in Chinese) (GM/T 0003.2-2012 SM2 椭圆曲线公钥密码算法第 2 部分：数字签名算法 [S]).

[20] GM/T 0003.3-2012 Public key cryptographic algorithm SM2 based on elliptic curves-Part 3:Key exchange protocol [S] (in Chinese) (GM/T 0003.3-2012 SM2 椭圆曲线公钥密码算法第 3 部分：密钥交换协议 [S]).

[21] GM/T 0003.4-2012 Public key cryptographic algorithm SM2 based on elliptic curves-Part 4:Public key encryption algorithm [S] (in Chinese) (GM/T 0003.4-2012, SM2 椭圆曲线公钥密码算法第 4 部分：公钥加密算法 [S]).

[22] GM/T 0003.5-2012, Public key cryptographic algorithm SM2 based on elliptic curves-Part 5:Parameter definition [S] (in Chinese) (GM/T 0003.5-2012, SM2 椭圆曲线公钥密码算法第 5 部分：参数定义 [S]).

[23] GM/T 0004-2012, SM3 cryptographic hash algorithm [S] (in Chinese) (GM/T 0004-2012, SM3 密码杂凑算法 [S]).

[24] GM/T 0044.1-2016, Identity-based cryptographic algorithms SM9-Part 1:General [S] (in Chinese) (GM/T 0044.1-2016, SM9 标识密码算法 第 1 部分：总则 [S]).

[25] GM/T 0044.2-2016, Identity-based cryptographic algorithms SM9-Part 2: Digital signature algorithm [S] (in Chinese) (GM/T 0044.2-2016, SM9 标识密码算法 第 2 部分：数字签名算法 [S]).

[26] GM/T 0044.3-2016, Identity-based cryptographic algorithms SM9-Part 3: Key exchange protocol [S] (in Chinese) (GM/T 0044.3-2016, SM9 标识密码算法 第 3 部分：密钥交换协议 [S]).

[27] GM/T 0044.4-2016, Identity-based cryptographic algorithms SM9-Part 4: Key encapsulation mechanism and public key encryption algorithm [S] (in Chinese) (GM/T 0044.4-2016, SM9 标识密码算法 第 4 部分：密钥封装机制和公钥加密算法 [S]).

[28] GM/T 0044.5-2016, Identity-based cryptographic algorithms SM9-Part 5: Parameter definition [S] (in Chinese) (GM/T 0044.5-2016, SM9 标识密码算法 第 5 部分：参数定义 [S]).

[29] Hanewinkel, H., http://www.hanewin.net/encrypt/ 2019,8,13

[30] Hao, F. and P. Ryan, "J-PAKE: Authenticated Key Exchange Wit

hout PKI", Transactions on Computational Science XI, pp. 192-206.
[31] HTML5 Introduction, https://www.w3schools.com/html/html5_intro.asp 2019,8,19
[32] HTML5 Web Workers, https://www.w3schools.com/html/html5_webworkers.asp 2019,8,19
[33] Just-in-time compilation. In Wikipedia, https://en.wikipedia.org/w/index.php?title=Just-in-time_compilation&oldid=910416478 2019,8,19
[34] Kriskowal/jscrypto, https://github.com/kriskowal/jscrypto 2019,8,13
[35] Web Crypto API, https://developer.mozilla.org/zh-CN/docs/Web/API/Web_Crypto_API 2019,8,10
[36] Web Cryptography API, https://www.w3.org/TR/WebCryptoAPI/ 2019,8,9.
[37] WebAssembly, https://webassembly.org/ 2019,8,19
[38] WebGL Overview - The Khronos Group Inc, https://www.khronos.org/webgl/ 2019,08,19
[39] Webkit, https://webkit.org/ 2019,8,19
[40] Welcome to JWCrypto's documentation, https://jwcrypto.readthedocs.io/en/latest/ 2019,8,13
[41] Windows系统的国产密码算法模块, http://info.dacas.cn/thread.aspx?ID=3039 2019,8,11
[42] Wwwtyro/cryptico, https://github.com/wwwtyro/cryptico 2019,8,13
[43] Rebeiro C , Selvakumar D , Devi A S L . Bitslice Implementation of AES[C]// International Conference on Cryptology and Network Security. Springer, Berlin, Heidelberg, 2006.
[44] Samsung Internet | Apps – The Official Samsung Galaxy Site, https://www.samsung.com/global/galaxy/apps/samsung-internet/ 2019,8,19
[45] SIMD in JavaScript | 01.org, https://01.org/node/1495 2019,8,19
[46] SpiderMonkey - Mozilla | MDN, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey 2019,8,19
[47] V8 JavaScript engine, https://v8.dev/ 2019,8,19
[48] Zhang J., Ma M., Wang P. (2018) Fast Implementation for SM4 Cipher Algorithm Based on Bit-Slice Technology. In: Qiu M. (eds) Smart Computing and Communication. SmartCom 2018.

Wei Rong, born in 1992, male, Ph.D. candidate, interested in Applied Cryptography.
Email: weirong1130@iie.ac.cn.

Zheng Fangyu, born in 1988, male, Ph.D., assistant research fellow, interested in Applied Cryptography, High-performance Computing, Computer Arithmetic.
Email: zhengfangyu@iie.ac.cn

Lin Jingqiang, born in 1978, male, Ph.D., research fellow, interested in Data Security and Privacy, Applied Cryptography, and the Application of Cryptography in Computer Network System.
Email: linjingqiang@iie.ac.cn.