SQLite3加密原理

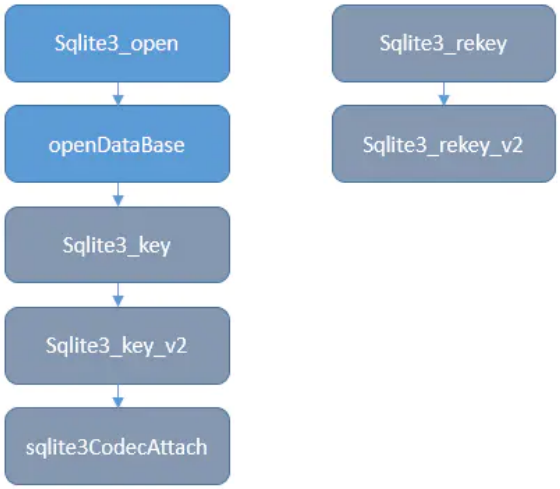
SQLite是一个进程库，实现了一个 自包含的， 无服务器， 零配置， 事务性 的SQL数据库引擎。SQLite是一个开源的项目，由于其小巧，灵活，高效等特点，在终端设备上使用非常广泛。

以下主要介绍一下SQLite3的加密框架。

原生设计

SQLite自身没有加密的实现，但提供了对外的加密接口，通过关键字SQLITE_HAS_CODEC来控制加密模块的开启和关闭。实现这些接口了解加密API的最简单的方法，就是跟着SQLITE_HAS_CODEC查看sqlite3.c，框架就一目了然了。

加密入口



api-1.png

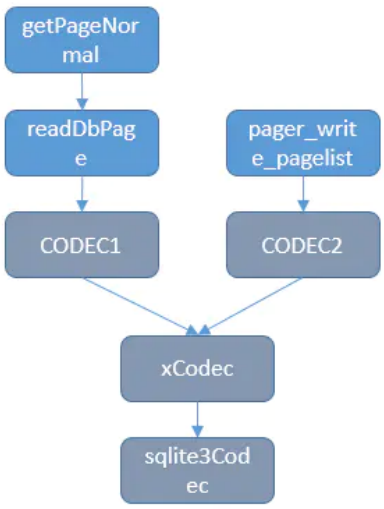
如上图所示为SQLite加密的基础API接口，其中蓝色部分为已有的模块，灰色部门是需要开发者去实现的部分。
sqlite3_key是加密的入口，需要在调用sqlite3_open打开数据库后立刻调用。
sqlite3_key和sqlite3_key_v2本质是一样的，区别是前者默认选择main db，后者可以通过名字选择db文件。
sqlite3_rekey用于修改密码，使用前必须先调用sqlite3_key解密。

page加密

api-2.png

如上图所示为SQLite实现page密码的框架。
SQLIte使用pager来管理页面，pager预留了三个函数指针，分别为xCodec（核心函数，负责page的加解密），xCodeSizeChng（page大小变化的回调），xCodecFree（释放函数），同时预留了指针pCodec用于保存加解密的
上下文。
SQLite已经提供了函数调用的实现，开发者只需实现图片下方的函数即可。

读写加解密



api.png

如上图所示为加解密时API的具体调用。
执行写操作时进行数据的加密，此时会调用CODE2函数，执行读操作是进行数据的解密，此时会调用CODEC1函数，
而最终调用的都是sqlite3Codec这个函数，主要通过传入的参数来控制。

```
//此处为wxSQlite的加解密实现
void* sqlite3Codec(void* pCodecArg, void* data, Pgno nPageNum, int nMode)
{
  Codec* codec = NULL;
  int pageSize;
  if (pCodecArg == NULL)
  {
    return data;
  }
  codec = (Codec*) pCodecArg;
  if (!CodecIsEncrypted(codec))
  {
    return data;
  }

  pageSize = sqlite3BtreeGetPageSize(CodecGetBtree(codec));

  switch(nMode)
  {
    case 0: /* Undo a "case 7" journal file encryption */
    case 2: /* Reload a page */
    case 3: /* Load a page */
      if (CodecHasReadKey(codec))
      {
        CodecDecrypt(codec, nPageNum, (unsigned char*) data, pageSize);
      }
      break;

    case 6: /* Encrypt a page for the main database file */
```

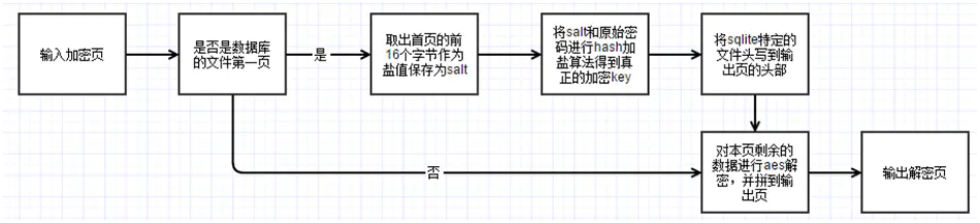
```
      if (CodecHasWriteKey(codec))
      {
        unsigned char* pageBuffer = CodecGetPageBuffer(codec);
        memcpy(pageBuffer, data, pageSize);
        data = pageBuffer;
        CodecEncrypt(codec, nPageNum, (unsigned char*) data, pageSize, 1);
      }
      break;

    case 7: /* Encrypt a page for the journal file */
      /* Under normal circumstances, the readkey is the same as the writekey.  However,
         when the database is being rekeyed, the readkey is not the same as the writekey.
         The rollback journal must be written using the original key for the
         database file because it is, by nature, a rollback journal.
         Therefore, for case 7, when the rollback is being written, always encrypt using
         the database's readkey, which is guaranteed to be the same key that was used to
         read the original data.
      */
      if (CodecHasReadKey(codec))
      {
        unsigned char* pageBuffer = CodecGetPageBuffer(codec);
        memcpy(pageBuffer, data, pageSize);
        data = pageBuffer;
        CodecEncrypt(codec, nPageNum, (unsigned char*) data, pageSize, 0);
      }
      break;
  }
  return data;
}
```

现有方案

下面介绍一下主要的几种SQLite加密的实现：

- SEE SQLite官方实现的版本，有加密的功能，不过要收费。
- wxSQlite wxWidgets 中实现了加密的功能，开源免费，比较轻量的实现。
- SQLCipher 目前最主流的方案，开源免费，安全性比较好，可以自由选择加密方式，但体积较大。
- SQLiteCrypto 主要使用了AES-256和SHA256来加密，也是收费的版本。

SQLiteCipher**实现原理**



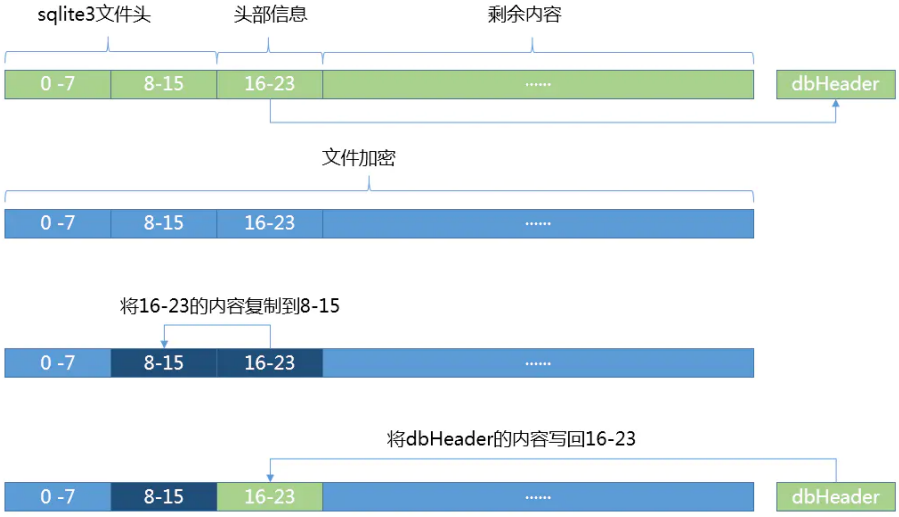
sqlcipher.png

上图为sqlCipher的实现原理，加密流程为以下步骤：

- 传入密钥
- 通过Rand_bytes算法生成16个字节的salt，并存储在数据库第一页的头部（SQLite3的db文件，头部前16个字节固定为SQLite 3 format，所以可以利用文件头来存储一些数据）。
- 通过PKCS5_PBKDF2_HMAC_SHA1算法将密钥和salt一起加密并多次迭代，生成AES加密所用的key；此处是对key的加密，即使原始的密码泄露，也无法解密数据。
- 通过AES对称加密算法对每一页的文件内容（有效的内容，不包含文件头和reserved字段）进行加密。
- 加密时，文件执行过AES加密后，对文件内容，通过Hmac算法，获取文件校验码，填充在page尾部（SQLite3提供了reserved字段，自动在page尾部预留一段空间）。
- 解密时，先调用Hmac算法获取文件标识码，与page尾部的数据进行对比，如果数据一致，则证明文件没有被篡改过，不然证明文件已经被篡改，则抛出异常。

PS：以上为默认的算法，sqlCipher没有固定算法，用户可以自己设置。

wxSQLite**实现原理**



wxsqlite.png

上图为wxSQLite加密部分的实现原理，大致实现与sqlCipher类似，区别是没有添加salt，没有生成HMAC码。

具体差异

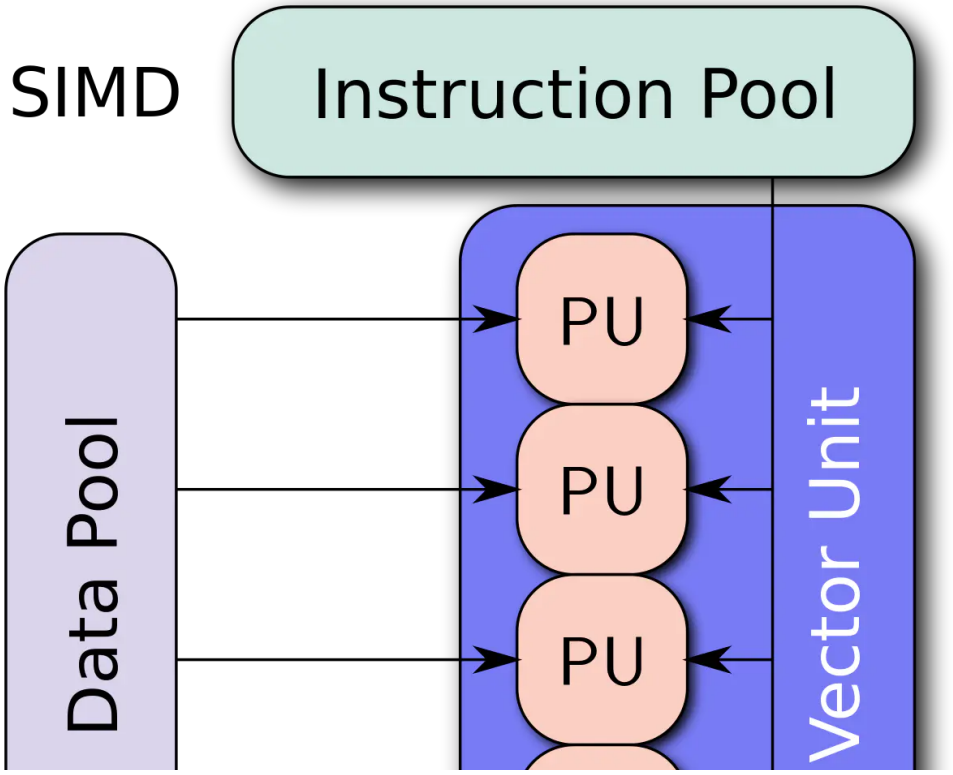
	wxSqlite	SQLCipher
Key加密	128bit: MD5+RC4 256bit: SHA256	Openssl: PKCS5_PBKDF2_HMAC_SHA1 CommonCrypto: CCKeyDerivationPBKDF Libtomcrypt: pkcs_5_alg2
salt	无	Openssl: RAND_bytes CommonCrypto: SecRandomCopyBytes Libtomcrypt: fortuna_read
hmac	无	Openssl: Hmac_sha1 CommonCrypto: kCCHmacAlgSHA1 Libtomcrypt: sha-1
Page加密	AES-256-CBC	Openssl:AES-256-CBC(默认，可替换)
DB文件头部	前24个字节	前16个字节（存储salt）
代码增量	252KB（代码）	14.6MB（加密算法库代码）

compare.png

硬件加密

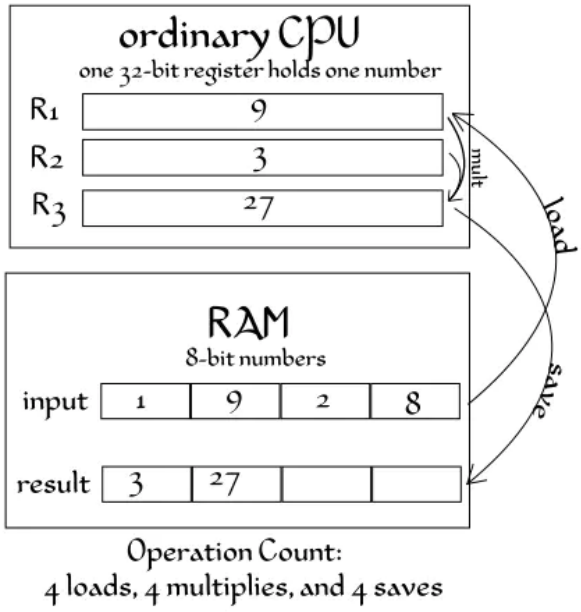
OpenSSL的加密库提供了硬件加速的功能，在执行加密算法的时候，可以通过汇编代码，直接操作寄存器，来优化加密的速度。本质上是使用了SIMD的技术。
单指令流多数据流（英语：Single Instruction Multiple Data，缩写：SIMD）是一种采用一个控制器来控制多个处理器，同时对一组数据（又称"数据向量"）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。

下图为架构图

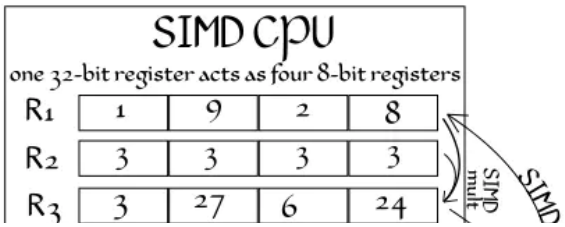


SIMD2.svg.png

下面两张图可以表明SIMD和普通调用CPU,GPU的区别



Non-SIMD_cpu_diagram1.svg.png



SIMD_cpu_diagram1.svg.png

结束语

文章写得比较简单，大部分直接贴图了，如果有什么问题，欢迎一起探讨。