Cryptography Engineering

Libsodium v1.0.12 and v1.0.13 Security Assessment

Contents

1 Executive Summary **2**

2 Introduction **3**
- 2.1 Scope . . . 3
- 2.2 Approach . . . 4
- 2.3 Classification and Severity Rating . . . 4

3 Findings **6**
- 3.1 Summary of Findings . . . 6
- 3.2 Overview of Cryptographic Design . . . 7
- 3.3 Static Analysis Results . . . 11
- 3.4 Dynamic Analysis Results . . . 11
- 3.5 Detailed Findings . . . 12
  - 3.5.1 SD-01: Randomness source on Windows based on unofficial APIs only 12
  - 3.5.2 SD-02: Possible null pointer dereference in key exchange API . . . 12
  - 3.5.3 SD-03: Potential issues with abort() to terminate program on error conditions . . . 13
  - 3.5.4 SD-04: Potential risks with the custom RNG API . . . 13
  - 3.5.5 SD-05: Missing APIs in the libsodium documentation . . . 14
  - 3.5.6 SD-06: Lack of elliptic curves at higher security levels . . . 14
  - 3.5.7 Formal Verification (In progress) . . . 15
  - 3.5.8 Diffs between version 1.0.12 and 1.0.13 . . . 15

4 Conclusions **17**

Executive Summary

Libsodium[1] is an open-source, easy-to-use fork of the C-language NaCl crypto library that is portable, cross-platform and provides many common cryptographic functions. These include public-key encryption, signatures, key derivation, password hashing, message authentication codes, stream ciphers, pseudo-random number generators, random number generation, and support for elliptic curves such as Curve25519. This review was funded by Private Internet Access™ (PIA), although PIA did not direct the review or modify the results.

We performed a security review of the libsodium v1.0.12 and v1.0.13 releases to uncover various types of security weaknesses in its core code. During this limited engagement, our security review included the components of libsodium that handle: authenticated symmetric encryption, public key encryption, hashing, key derivation and key exchange. In addition we analyzed the underlying cryptographic primitives for correctness and security.

Overall our finding is that libsodium is a secure, high-quality library that meets its stated usability and efficiency goals. We did not uncover any major vulnerabilities in the version of libsodium that we reviewed. During our review, we did identify a few *low* severity issues in libsodium related to the usability of the API and software security. In addition, we identified some potential risks with allowing developers to customize the random number generation that developers should take into consideration to prevent introducing vulnerabilities.

[1] `https://github.com/jedisct1/libsodium`

Introduction

Libsodium[1] is an open-source, easy-to-use fork of the C-language NaCl crypto library that is portable, cross-platform and provides many common cryptographic functions. These include public-key encryption, signatures, key derivation, password hashing, message authentication codes, stream ciphers, pseudo-random number generators, random number generation, and support for elliptic curves such as Curve25519.

Libsodium was designed to provide high-level APIs that abstract the implementation details of these common cryptographic operations. Moreover, the well-written API was designed to prevent misuse by application developers. Several programming language bindings exist for libsodium and have been ported to the major operating systems (*e.g.*, Unix/Linux, Windows, Mac OS X). The library has also been ported to support browsers using Emscripten[2] and native client in addition to support for microarchitectures such as ARM and MIPS. Given this extensive support, it is no surprise that libsodium is widely used by several companies, open-source libraries and applications.[3]

2.1 Scope

We performed a security review of the libsodium v1.0.12 and v1.0.13 releases to uncover various types of security weaknesses in its core code. During this limited engagement, our security review included the following components of libsodium:

- **Interfaces for generating random numbers.** We analyzed the interface for each type of platform (Unix/Linux, Windows and Mac OS X), as well as the interface for customizing the random number generator (RNG) if none of the existing options apply.

- **Authenticated symmetric encryption.** This includes the secret-key authenticated encryption (or `crypto_secretbox`) and the authenticated encryption with associated data (AEAD) constructions. It also includes advanced features like precomputation.

[1] `https://github.com/jedisct1/libsodium`

[2] Emscripten compiles C/C++ via LLVM bitcode into JavaScript which can run on the web: `https://github.com/kripken/emscripten`.

[3] Companies such as Digital Ocean, Zerocoin Electric Coin Company and Keybase. Open source projects such as ZeroMQ, Macaroons, and MEGA SDK.

- **Authenticated public-key encryption.** This include the `crypto_box` implementation which adds public-key authenticated encryption and message authentication code (MAC) constructions in addition to the `crypto_secretbox` and the one-time secret-key message authentication constructions.

- **Generic hash API.** This is used for computing a fixed-length digest for arbitrary-sized messages based on a secure hash function. It includes the short input hashing API for data structures such as hash tables and bloom filters.

- **Key derivation API.** These are used for deriving keys from passwords or subkeys based on a given master key and key identifier. This includes password hashing and password storage constructions.

- **Key exchange API.** This provides an interface for two parties to securely compute shared secrets using the peer's public key and the communicating party's secret key.

- **Underlying primitives.** This includes the Elliptic Curve Diffie Hellman (or ECDH) implementation, public-key signatures, and stream cipher constructions.

2.2 Approach

The security analysis of the libsodium code base involves a manual and automated analysis to uncover common vulnerabilities in the cryptographic library design and implementation. We first manually analyzed the design for various security weaknesses as well as for several classes of C vulnerabilities which include stack or heap-based buffer overflows, off-by-one errors, use-after-free, possible null pointer dereferences, and other related memory corruption bugs. We then checked the core code for possible vulnerabilities which includes weaknesses in the random number generation, possible timing attacks against various cryptographic algorithms, invalid curve style attacks against the elliptic curves supported and other types of crypto-related weaknesses. Moreover, we performed an automated memory analysis of the code base using both static analysis tools and dynamic analysis tools to uncover security vulnerabilities. We also investigated potential ways for a developer to potentially misuse the API using the library documentation. Finally, we attempted to use formal verification tools developed by Galois, Inc to prove equivalence between algorithm implementations in libsodium and a Galois formal specification, however after significant consultation with Galois, we were unable to complete this task as of the report deadline (see §3.5.7 for details). Future versions of this report may include these results.

2.3 Classification and Severity Rating

We classified the issues found during the security review of libsodium according to the Common Weakness Enumeration (CWE) list. We followed a straightforward severity rating

system of low, medium or high based on best practices for securely coding in C, best practices for cryptographic design, common use cases of libsodium and the potential security impact.

- **Low.** A low severity rating indicates that the issue poses a relatively small or limited risk to applications/users of the library.
- **Medium.** A medium severity rating indicates that the issue could potentially be exploited and an individual user could be compromised.
- **High.** A high severity rating indicates that a large number of users are at significant risk of compromise and the issue should be addressed immediately.

Lastly, we designate the **Info** rating to indicate non-security risks that could negatively impact users/applications that build on the library. This rating should be treated as ways that the open-source project could be improved to better serve users.

Findings

Our review of libsodium began with the v1.0.12 release from March 13th, 2017 and continued with commits since then to the release of v1.0.13 in the Github repository.[1] We attempted to identify weaknesses and vulnerabilities in the cryptographic library by manually reviewing the source, and using static and dynamic memory analysis tools. In this section, we provide a summary of our findings in addition to reporting details on issues that were uncovered during our review.

3.1 Summary of Findings

Overall our finding is that libsodium is indeed a secure, high-quality library that meets its stated usability and efficiency goals. We did not uncover any major vulnerabilities in the version of libsodium that we reviewed. During our review, we did identify several *low* severity issues in the libsodium 1.0.12 release related to the usability of the API and software security. In addition, we identified some potential risks with allowing developers to customize the random number generation that developers should take into consideration to prevent introducing vulnerabilities. We report on the summary of our findings in Table 3.1 and provide further details in Section 3.5.

Next, we provide a high-level summary of our review. In terms of mitigating common C vulnerabilities, libsodium developers follow best practices with respect to preventing buffer overflows and providing helper routines for handling sensitive data in memory. Specifically, the libsodium memory allocation routines are provided for locking memory pages (`sodium_mlock()`), and wiping sensitive data (`sodium_memzero()`) in a way that is not optimized out by a compiler or linker. `sodium_mlock()` is ideal for preventing swapping sensitive data to disk.

For added security, the library provides guarded heap allocators (`sodium_malloc()`) specifically for storing sensitive data in memory. The allocated memory is placed at the end of a page boundary followed by the guarded page. Any access beyond the end of the memory region will terminate the application. This type of memory allocation comes at a cost (requires up to 4 extra virtual memory pages), slower than the C `malloc()`, and is not ideal for variable-length data.

[1] `https://github.com/jedisct1/libsodium`.

In this same direction, libsodium provides three routines to deal with memory page protections. The `sodium_mprotect_noaccess()` function makes allocated memory inaccessible (cannot read or write) but the data are preserved. The `sodium_mprotect_readonly()` function marks data read only to prevent modifications until a specific operation. The `sodium_mprotect_readwrite()` marks data read and write accessible only after having been protected by either of the former routines.

We reviewed the routines for generating cryptographically strong random numbers on each supported platform. Generally speaking, the randomness used for key generation, nonces and encryption are from strong sources. Weak (or deterministic) sources are clearly marked as such and only used in the test suite to ensure reproducibility. However, we found some potential API risks associated with Windows and provide further details in Section 3.5.1.

We analyzed the underlying core primitives implemented in libsodium to check for various security properties.[2] For instance, we confirmed that each operation was free from branches or array lookups based on secret information. We also confirmed that sensitive data and keys in buffers are securely erased (via `sodium_memzero()`) after specific operations. Similarly, we confirmed that verification operations for password hashing and MACs are done in constant-time using `sodium_memcmp()`. Based on our analysis, we can conclude that libsodium follows cryptographic best practices in terms of resistance to side channels and protecting confidential data.

We checked how libsodium handles user inputs in terms of array bounds checking, null pointer dereferences and error handling. By default, libsodium takes a conservative approach and terminates the program immediately if invalid inputs are supplied by the user. To gracefully recover from these types of errors, the application must intercept the `SIGABRT` signal. In terms of error handling, libsodium routines return `-1` in some cases to indicate invalid inputs, decryption or verification failure. In addition, `errno` is set to indicate the specific error. Lastly, we found only two cases in which it might be possible to dereference a null pointer. This is described in more detail in Section 3.5.2.

In summary, we did not find any critical flaws or vulnerabilities in libsodium and we suggest minor improvements based on our findings described in Section 3.5. Lastly, we also report on the results of static analysis (Section 3.3) and dynamic analysis tools (Section 3.4) on the libsodium core code.

3.2 Overview of Cryptographic Design

Libsodium provides many cryptographic constructions and variants. In this section, we explore the notable cryptographic choices of the library and where it differs from the NaCl

[2]The core primitives include Curve25519, Poly1305 MAC, AES-CTR/GCM, Blake2b, ChaCha20, Salsa, and etc.

Finding Description	ID	Severity
Randomness source on Windows based on unofficial APIs only	SD-01	Low
Possible NULL pointer dereference in key exchange API	SD-02	Low
Potential issues with abort() to terminate program on error conditions	SD-03	Info
Potential risks with the custom RNG API	SD-04	Info
Missing APIs and some API risks not included in libsodium documentation	SD-05	Info
Lack of elliptic curves at higher security levels	SD-06	Info

Table 3.1: Summary of Findings

library. In addition, we mention the associated high-level APIs and the advanced features that are exposed to application developers.

Hash Functions, Short Input Hashing and Message Authentication Codes. The library includes the SHA family of hash functions (SHA-256 and SHA-512) for interoperability, and BLAKE2b (as part of the generic hash function interface) for speed and security.

The library includes secure constructions for keyed-message authentication via HMAC-SHA-256 and HMAC-SHA-512 (with multi-part API for streaming messages). In cases where hashing support for data structures like hash tables are required, libsodium provides a short input hash function based on SipHash [1] that outputs 64-bits. SipHash is optimized for this use case, but is not a cryptographic hash function.

The Poly1305 [2] Carter-Wegman message authenticator (inherited from NaCl) is used for multiple purposes in libsodium. In addition to being used as part of authenticated encryption constructions, it is also provided as a standalone primitive for one-time authentication of short messages (or `crypto_onetimeauth()`). This interface takes as input a single-use key, message and produces a 128-bit authentication tag.

Password Hashing and Password Storage. The password hashing and password storage APIs and constructions are new and not part of the NaCl library. Specifically, the password hashing primitive is based on memory-hard functions Argon2 and scrypt.[3] Libsodium implements the Argon2i variant which uses data-independent memory access and this is also ideal for password-based key derivation (or `crypto_pwhash()`) in addition to password hashing (or `crypto_pwhash_str()`). The high-level primitives are based on Argon2i by default. In addition, these functions are provided with safe default parameters that require a large amount of memory to execute brute-force attacks. In the documentation, there are also guidelines for picking the parameters (`opslimit` and `memlimit`) to balance between efficiency and security which consider interactive and non-interactive scenarios.

Libsodium provides a similar API specifically for scrypt, which is a widely deployed safe alternative to Argon2 via `crypto_pwhash_scryptsalsa208sha256()` for deriving

[3]Argon2/scrypt are designed to make it expensive to perform large-scale hardware attacks by requiring large amounts of memory and CPU.

keys from passwords and `crypto_pwhash_scryptsalsa208sha256_str()` for password hashing. It is recommended in the documentation that users pre-hash the passwords (via BLAKE2b) prior to applying scrypt if passwords are 65 bytes or longer and potentially hashed via unsalted SHA-256 elsewhere. Given the security implications, we suggest that a separate high-level wrapper could be provided in libsodium that pre-hashes by default for such scenarios to prevent misuse by uninformed users.

Key Derivation. Libsodium recently introduced a key derivation API (or `crypto_kdf_derive_from_key`) in version 1.0.12 specifically for deriving subkeys from a single master key, a subkey identifier (64 bits) and a context string. This API uses BLAKE2b as the underlying hash function and can produce up to 2^{64} keys with key lengths varying from 128 to 512 bits. The required context string ensures that the same master key can be used in two domains of subkeys and prevents potential bugs.

Authenticated Encryption. Libsodium is based on NaCl, and thus inherited a secure secret-key authenticated encryption primitive (or `crypto_secretbox()`) based on the XSalsa20 stream cipher and the Poly1305 message authentication code. Both were originally designed by Bernstein. The XSalsa20 stream cipher increases the nonce size to 192-bits from 64-bits in the original Salsa20 cipher design [3]. One notable improvement in libsodium: the library introduces new wrappers (`crypto_secretbox_easy()` and `crypto_secretbox_detached()`) around the original NaCl `crypto_secretbox()` API to remove the need for message padding and pointer arithmetic prior to encryption. This eliminates potential misuse by developers.

The `crypto_secretbox_easy()` function encrypts a message with a single key and nonce and computes a tag over the resulting ciphertext. The `crypto_secretbox_open_easy()` function verifies the tag and decrypts the ciphertext if successful. In combined mode (or `*_easy()`), the authentication tag and ciphertext are written to the same output buffer (tag is prepended to the encrypted message). In detached mode (or `*_detached()`), the tag and ciphertext are written into separate buffers.

Authenticated Encryption with Associated Data. For AEADs, there are two provably secure constructions to choose from in libsodium: AES-GCM [10] and ChaCha20-Poly1305 [9]. The AES-GCM implementation is hardware-accelerated using AES-NI instructions and resistant to timing attacks. The ChaCha20-Poly1305 implementation combines a stream cipher and is resistant to timing attacks by design. In addition, this particular construction has two additional variants implemented in libsodium: an IETF version [7] and one with an extended nonce (XChaCha20-Poly1305) [4]. One benefit of the XChaCha20-Poly1305 construction is that it enables nonce misuse-resistant schemes. These constructions are not only fast in software and hardware but also maintain interoperability with other popular cryptographic libraries such as OpenSSL. The only exception is that AES-GCM is slower compared to the other possible constructions when AES-NI is unavailable.

Public-key Authenticated Encryption. In the public-key authenticated encryption construction (or `crypto_box()` from NaCl), the scheme is based on X25519 [8] for key exchange, XSalsa20 stream cipher for the encryption, and Poly1305 for the message authentication. Using the curve X25519, users can randomly generate an ephemeral keypair (public key and secret key) or can deterministically generate one from a given seed. The `crypto_box_easy()` routine takes as input the message, a nonce, the recipient's public key and the sender's secret key. This routine computes the shared key from the public and secret key (via scalar multiplication), encrypts the message via the stream cipher with the nonce as input and computes a tag over the ciphertext.[4] The `crypto_box_open_easy()` verifies the tag then decrypts the ciphertext and returns the message on success. Also, libsodium includes an advanced precomputation interface for `crypto_box()` which allows computing the shared key once and then reusing that key structure to encrypt and decrypt several messages between the sender and receiver. With this interface, the user is mainly responsible for securely wiping the memory associated with the shared secret key.

Public-key Signatures. For public-key signatures, libsodium also inherited the `crypto_sign()` API from NaCl. This is based on a Schnorr variant (Ed25519) [**?**] and adds a new multi-part signature API for authenticating multiple messages. This multi-part signature system is based on Ed25519 with message pre-hashing (via SHA-512). As an extension, libsodium provides a utility for converting Ed25519 keys to Curve25519 keys [**?**, Section 2] such that one keypair could be used for both authenticated encryption (via `crypto_box`) and signatures (via `crypto_sign`). However, the library does not support going in the other direction (Curve25519 to Ed25519 keys).[5]

Sealed Public-key Encryption. To support encryption with sender anonymity, libsodium includes a `crypto_box_seal()` API to allow encrypting a message such that even the sender cannot decrypt the message later (by destroying the sender's secret key when generating the ephemeral keypair). The `crypto_box_seal_open()` function then allows a recipient to verify the integrity of the message without identifying the sender.

Unauthenticated Encryption. Also inherited from NaCl, libsodium provides stream ciphers (or `crypto_stream()`) directly to users. However, users are warned in the source (or `crypto_stream.h`) from using these ciphers directly unless generating pseudo-random data from a key or building a larger cryptographic construction or protocol. The ciphers include ChaCha20, Salsa20, Salsa2012, Salsa208, XChaCha20 (new in 1.0.12), Xsalsa20 and AES-128 in counter mode.[6] If reusing a key with this API, it is recommended that the nonces be incremented rather than randomly generated for each new message stream.

[4] Reuses `crypto_secretbox_*()`

[5] We noted that the conversion routine does not include a check to verify that the input point is on the elliptic curve. We determined that this is unlikely to have a security impact, but the library developers have opted to incorporate a check in future versions of the library.

[6] Note that the ciphers are all implemented to resist timing attacks.

Elliptic Curve Diffie-Hellman. Libsodium includes a dedicated interface for Elliptic Curve Diffie-Hellman over Curve25519 [6] that is used in the public-key encryption and key exchange protocol implementations. One important aspect of the scalar multiplication functionality is that it is optimized for different architectures and processors. In terms of security, one difference in libsodium compared to NaCl is that the computed shared secret is rejected if equal to 0 (thereby ensuring that Curve25519 public keys are valid). This validation removes the possibility of an attacker controlling the key without detection by communicating peers.

3.3 Static Analysis Results

We applied the clang static analyzer to find possible bugs in the libsodium 1.0.12 release. The clang static analyzer is a source code level analyzer built on top of the clang/LLVM compiler toolchain. Using the clang static analysis tool, we only found 2 possible null pointer dereference bugs (CWE-476) in the key exchange API when used with Curve25519. These results are discussed in Section 3.5.

We also ran the static analyzer on the libsodium source with commits since the libsodium 1.0.12 release and after the 1.0.13 release. There were no new issues reported by clang and the aforementioned bugs described have been fixed. As indicated in the libsodium documentation, static analysis is part of each release of the project which contributes to the overall quality of the crypto library.

3.4 Dynamic Analysis Results

We analyzed the memory handling in libsodium 1.0.12 using dynamic analysis tools on a Ubuntu 16.04 LTS system. In particular, we used the address sanitizer tool (a memory corruption detector for C/C++ code) to find possible memory corruption bugs in libsodium crypto modules. We simultaneously applied valgrind as well to detect possible memory leaks that might arise while using the crypto API.

As a result of applying both tools, we could not find any memory corruption related issues in the core library. One reason for this is that libsodium dynamically allocates memory mainly in the password hashing component.[7] In this component, the libsodium developers follow best practices and always release memory even on error conditions. In addition, the developers execute dynamic analysis tools before each minor release and this process helps catch potential memory corruption bugs as new features are added to the code base.

[7]We note here that `sodium_malloc` also allocates memory on the heap. However, this is not used in the core library and meant for users of the library.

3.5 Detailed Findings

In this section, we describe the detailed findings from the security review of libsodium.

3.5.1 SD-01: Randomness source on Windows based on unofficial APIs only

On Windows-based platforms, the main source of entropy for cryptographic operations in libsodium comes from the `RtlGenRandom()` function. This involves loading the `ADVAPI32.DLL` dynamic library and then retrieving the `SystemFunction036` function (also known as `RtlGenRandom()`). While this API provides a direct and low-overhead API to obtain randomness, libsodium developers made an explicit choice *not* to support the traditional `CryptGenRandom()` due to the memory overhead of Window's Crypto API.

Libsodium depends only on this unofficial API and if Microsoft decides to remove it at any point in the future, then there is no fallback option. We recommend adding an alternative method using *e.g.*, `CryptGenRandom()` despite the increased memory overhead. This could be optionally enabled at compile time for users that prefer to use `CryptGenRandom()` with libsodium on Windows.

3.5.2 SD-02: Possible null pointer dereference in key exchange API

Using the clang static analyzer, we found possible null pointer dereferences in `crypto_kx_client_session_keys()` and `crypto_kx_server_session_keys()` of the new key exchange API. The key exchange protocol allows two parties to securely derive a set of shared keys. The purpose of the `crypto_kx_server_session_keys()` function is to compute one or two session keys (or shared secrets) using the client's public and private key, and the server's public key. The input into this routine are pointers to output buffers `rx` and `tx`. If only one session key is required, then the user can simply set either `rx` or `tx` to `NULL`. In the expected use of this API, there is no possibility of a null pointer dereference. This is not the case if the user were to accidentally set both `rx` to `tx` to null. As a result, there is a null pointer dereference on line 100 in the code shown below.

```
70 int
71 crypto_kx_server_session_keys(unsigned char rx[crypto_kx_SESSIONKEYBYTES],
72                               unsigned char tx[crypto_kx_SESSIONKEYBYTES],
73                               const unsigned char server_pk[
                                     crypto_kx_PUBLICKEYBYTES],
74                               const unsigned char server_sk[
                                     crypto_kx_SECRETKEYBYTES],
75                               const unsigned char client_pk[
                                     crypto_kx_PUBLICKEYBYTES])
76 {
77     crypto_generichash_state h;
78     unsigned char            q[crypto_scalarmult_BYTES];
```

```
79      unsigned char               keys[2 * crypto_kx_SESSIONKEYBYTES];
80      int                         i;
81
82      if (rx == NULL) {
83          rx = tx;
84      }
85      if (tx == NULL) {
86          tx = rx;
87      }
88      if (crypto_scalarmult(q, server_sk, client_pk) != 0) {
89          return -1;
90      }
91      COMPILER_ASSERT(sizeof keys <= crypto_generichash_BYTES_MAX);
92      crypto_generichash_init(&h, NULL, 0U, sizeof keys);
93      crypto_generichash_update(&h, q, crypto_scalarmult_BYTES);
94      sodium_memzero(q, sizeof q);
95      crypto_generichash_update(&h, client_pk, crypto_kx_PUBLICKEYBYTES);
96      crypto_generichash_update(&h, server_pk, crypto_kx_PUBLICKEYBYTES);
97      crypto_generichash_final(&h, keys, sizeof keys);
98      sodium_memzero(&h, sizeof h);
99      for (i = 0; i < crypto_kx_SESSIONKEYBYTES; i++) {
100         tx[i] = keys[i];    // null derefence occurs here if rx = tx = NULL
101         rx[i] = keys[i + crypto_kx_SESSIONKEYBYTES];
102     }
103     sodium_memzero(keys, sizeof keys);
104
105     return 0;
106 }
```

Note that the same issue exists in `crypto_kx_client_session_keys()` on line 62.

3.5.3 SD-03: Potential issues with abort() to terminate program on error conditions

We found several places in libsodium in which the program is terminated via `abort()` due to the user specifying a larger than expected buffer length or insufficient randomness generated, for example. Given that bindings are important to the libsodium ecosystem (exposing the C API to PHP, Go, Java and many other programming languages), it may not be possible or convenient to handle the termination of a program in this fashion. Program termination prevents the application from gracefully recovering from such errors. One option would be to set a unique error code via `errno` for such error conditions (similar to other cases in libsodium) and propagate the error accordingly. This could be provided as a compile time option for users that would prefer such error handling.

3.5.4 SD-04: Potential risks with the custom RNG API

The interface to customize the random number generator provides a means to change how libsodium obtains randomness for a given platform. For some embedded platforms, using

this API may be the only option for secure randomness. Libsodium ships a secure pseudo-random number generator (PRNG) construction which obtains a small amount of entropy from `/dev/random` or `/dev/urandom` (if available) to seed the Salsa20 stream cipher. Moreover, the default implementation includes `random_stir()` to occasionally reseed the PRNG. To use this PRNG, the user simply calls `randombytes_set_implementation(&randombytes_salsa20_implementation)` prior to initializing libsodium's core via `sodium_init()`.

The RNG implementation is comprised of 6 function pointers and only 3 are required.

```
1 typedef struct randombytes_implementation {
2         const char *(*implementation_name)(void);
3         uint32_t (*random)(void);
4         void (*stir)(void);
5         uint32_t (*uniform)(const uint32_t upper_bound);
6         void (*buf)(void * const buf, const size_t size);
7         int (*close)(void);
8 } randombytes_implementation;
```

There are known issues with this approach. It is not thread-safe and a `randombytes_stir()` implementation is optional. Generally speaking, users could abuse this API in ways that would result in a weak PRNG for sensitive cryptographic operations. As such, we recommend limiting this interface to prevent such misuse. One possible approach would be to provide a generic PRNG implementation (similar to `randombytes_salsa20_implementation()`) but that takes as input a stream cipher (or hash function). Thus, users would be able to swap Salsa20 with ChaCha20 for example.

3.5.5 SD-05: Missing APIs in the libsodium documentation

Libsodium ships with extensive documentation regarding how to use the library API and the rationale for the design. Extending such a document with each library release is a heroic effort for the libsodium project. Unfortunately, there are some inconsistencies between what is in the version 1.0.12 release and what is presently documented. For instance, some of the helper functions designed to simplify key and nonce generation primitives have been recently added to the library but have not been documented. We recommend that libsodium developers take a pass over the documentation to ensure consistency with the latest release.[8]

3.5.6 SD-06: Lack of elliptic curves at higher security levels

The main curve supported by the library is Curve25519 which is equivalent to the 128-bit security level. Currently, libsodium does not offer any curve support at the 224 and 256-bit security levels. While it is our understanding that Ed448-Goldilocks is on the roadmap for a future release (for 224-bit security), we recommend adding one additional curve at the 256-bit security level as well. This would provide sufficient curve options at different security levels for application developers.

[8]Project source docs can be found at `https://github.com/jedisct1/libsodium-doc`.

3.5.7 Formal Verification (In progress)

Libsodium originally started as a fork of NaCl with the same core cryptographic algorithms and a compatible API. However, it has been improved and extended to support additional features. For example, libsodium provides optimized implementations of ChaCha20, Salsa20 and Poly1305 for specific processors (*e.g.*, SSE3/AVX2). One potential problem is that subtle changes in libsodium might impact correctness and passing known answer tests alone do not provide sufficient guarantees that the cryptographic algorithms are correct on all inputs.

To this end, we attempted to use the existing formal verification tools developed by Galois, Inc to prove equivalence between certain implementations in libsodium and its formal specification. The tools consists of Cryptol, a domain-specific language for specifying cryptographic algorithms.[9] For example, Cryptol provides a set of example specifications for stream ciphers such as Salsa20 and ChaCha20. It also includes the software analysis workbench (SAW) which is a powerful tool for extracting formal models from cryptographic implementations and analyzing those models using automated reasoning tools. One such reasoning tool is ABC developed at UC Berkeley which is a system for synthesis and verification.[10] This includes the Microsoft Z3 theorem prover [5].

Unfortunately, due to incompatibilities between libsodium and SAW we were unable to complete this task by the report deadline. We are currently consulting with Galois, Inc. to address these issues and will include results in a future version of this report, if possible.

3.5.8 Diffs between version 1.0.12 and 1.0.13

This report originally focused on the security review of libsodium 1.0.12. However, version 1.0.13 was released recently on July 13th, 2017 and we further analyzed the differences to date. There are approximately 98 commits between the 1.0.12 and 1.0.13 version and we provide a summary of the notable differences (excluding documentation improvements, fixes to various software build issues and readability of the source code):

- Fixed the `crypto_pwhash_argon2i_MEMLIMIT_MAX` constant which was incorrectly defined for 32-bit platforms.
- Added an AVX2 optimized implementation of the Argon2 round function.
- Added the Argon2id variant of Argon2 in addition to a high-level API that integrates both Argon2i and Argon2id. The password hashing API (`crypto_pwhash_str_verify()`) can work with either Argon2i and Argon2id variants by checking for the appropriate prefix in the hashed password.
- Added an XChaCha version for the `crypto_box_seal()` primitive. This version is implemented as `crypto_box_curve25519xchacha20poly1305_seal*()` and does not include a high-level API.

[9] `http://cryptol.net/files/cryptol_whitepaper.pdf`
[10] `https://people.eecs.berkeley.edu/~alanmi/abc/`

To the best of our knowledge, we could not identify any new vulnerabilities as a result of the commits during this period.

Conclusions

Our review of libsodium did not uncover any critical flaws or vulnerabilities in the core library. Libsodium developers have followed best practices in terms of cryptographic design and secure development in C. The library provides the right level of abstraction for application developers that need a secure, easy-to-use API for cryptography. Moreover, the library provides secure defaults for every cryptographic primitive with production-level quality. To maintain support for a variety of applications and protocols, we recommend that the libsodium project remain focused on providing simple cryptographic functions for common problems to users. As new cryptographic primitives, features and high-level APIs are added to the library, it is important that the libsodium developers continue to highlight the intended use cases and provide example code to prevent misuse.

Bibliography

[1] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[2] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.

[3] Daniel J Bernstein. Salsa20 specification. *eSTREAM Project algorithm description,* `http://www.ecrypt.eu.org/stream/salsa20pf.html`, 2005.

[4] Daniel J Bernstein. Extending the Salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop*, volume 2011, 2011.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC7748, January 2016.

[7] Adam Langley and Yoav Nir. ChaCha20 and Poly1305 for IETF protocols. 2015.

[8] Y Nir and S Josefsson. Curve25519 and Curve448 for the Internet Key Exchange protocol version 2 (IKEv2) key agreement. Technical report, 2016.

[9] Gordon Procter. A security analysis of the composition of ChaCha20 and Poly1305. *IACR Cryptology ePrint Archive*, 2014:613, 2014.

[10] Joseph Salowey, Abhijit Choudhury, and David McGrew. AES Galois Counter Mode (GCM) cipher suites for TLS. Technical report, 2008.